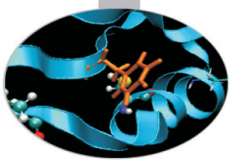
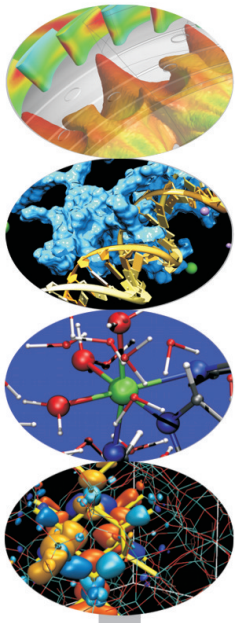
CINECA SCAI
SuperComputing Applications and Innovation



Debugging and Optimization of Scientific Applications

P. Lanucara V. Ruggiero

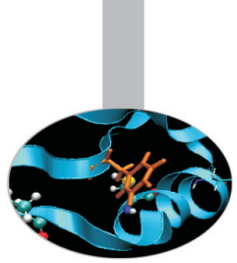
CINECA Rome - SCAI Department



Rome, 19-21 April 2017

CINECA

AGENDA



19th April 2017

9.00-9.30 Registration
9.30-10.30 Architectures
10.30-13.00 Cache and Memory System
14.00-15.00 Pipelines
15.00-17.00 Profilers

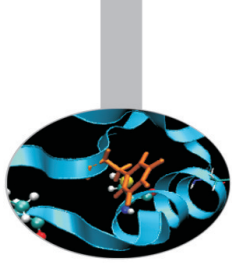
20th april 2017

9.30-13.00 Compilers
14.00-15.30 Scientific Libraries
15.00-17.00 Makefile

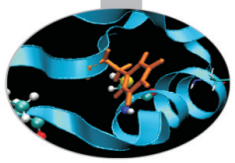
21st april 2017

9.30-13.00 Debugging
14.00-17.00 Final hands-on

Outline



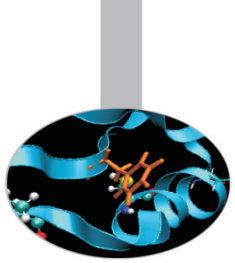
What is the best performance which can be achieved?



Matrix multiplication (time in seconds)

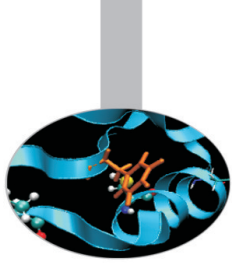
Precision	single	double
Incorrect Loop	7500	7300
Without optimization	206	246
With optimization (-fast)	84	181
Optimized code	23	44
Using ACML Library (serial)	6.7	13.2
Using ACML Library (2 threads)	3.3	6.7
Using ACML Library (4 threads)	1.7	3.5
Using ACML Library (8 threads)	0.9	1.8

Let's start!



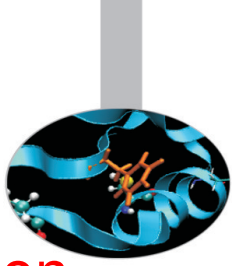
- Write the main loop of the code and verify the obtained performances.
- Use the Fortran and/or the C code.
  - What performances have been obtained?
  - There are differences between Fortran and C codes?
  - How change the perfomances using different compilers?
  - And using different compilers' options?
  - Do you have a different performances changing the order of the loops?
  - Can I rewrite the loop in a more efficient mode?

Matrix multiplication



- Fortran and C code
- Columns rows product $C_{i,j} = A_{i,k} B_{k,j}$
- Time:
  - Fortran: date_and_time (> 0.001")
  - C: clock (>0.05")
- Square matrices of size n
  - Required memory (double precision) $\approx (3 * n * n) * 8$
  - Number of total operations $\approx 2 * n * n * n$
    - We must access n elements of the two original matrices for each element of the destination matrix.
    - n products and n sums for each element of the destination matrix.
  - Total Flops = $2 * n^3 / Time$
- Always verify the results :-)

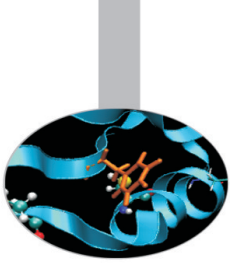
Measure of performances



- Estimate the number of computational operations at execution N_{Flop}
  - 1 FLOP= 1 Floating point OPeration (addition or multiplication).
  - Division, square root, trigonometric functions require much more work and hence take more time.
- Estimate execution time T_{es}
- The number of floating operation per second is the most widely unit used to estimate the computer performances:

  $Perf = \frac{N_{Flop}}{T_{es}}$
  - The minimum count is 1 Floating-pointing Operation per second (FLOPS)
  - We usually use the multiples:
    - 1 MFLOPS= 10^6 FLOPS
    - 1 GFLOPS= 10^9 FLOPS
    - 1 TFLOPS= 10^{12} FLOPS

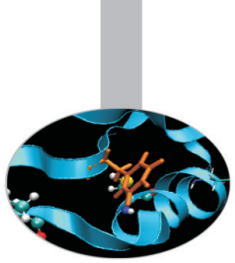
Exercises



```
https://hpc-forge.cineca.it/files/CoursesDev/public/2017/
Debugging_and_Optimization_of_Scientific_Applications/Roma/Intro_exercises.tar


tar xvf Intro_exercises.tar
```

- Directory tree
  - src/eser_?/fortran
  - src/eser_?/c

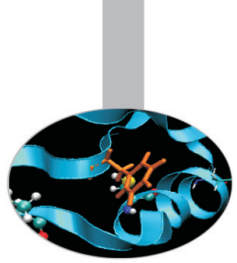
Matrix multiplication



- Go to src/eser_1
- Write the main loop (columns rows product) to Fortran(mm.f90) or/and C(mm.c) code.
- Run the matrix multiplication code
- N=1024

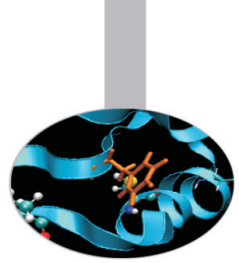
Language	time	Mflops
Fortran		
C		

Makefile



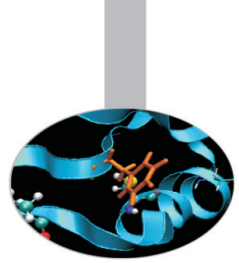
- To compile
  - make
- To clean
  - make clean
- To change the compiler options
  - make "FC=ifort"
  - make "CC=icc"
  - make "OPT=fast"
- To compile using single precision arithmetic
  - make "FC=ifort -DSINGLEPRECISION"
- To compile using double precision arithmetic
  - make "FC=ifort"

cat /proc/cpuinfo



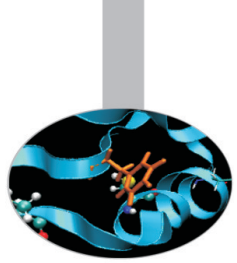
```
processor       : 0
vendor_id       : GenuineIntel
cpu family      : 6
model           : 37
model name      : Intel(R) Core(TM) i3 CPU       M 330  @ 2.13GHz
stepping        : 2
cpu MHz         : 933.000
cache size      : 3072 KB
physical id     : 0
siblings        : 4
core id         : 0
cpu cores       : 2
apicid          : 0
initial apicid  : 0
fpu             : yes
fpu_exception   : yes
cpuid level     : 11
wp              : yes
wp              : yes
flags           : fpu vme de pse tsc msr pae mce cx8
bogomips        : 4256.27
clflush size    : 64
cache_alignment : 64
address sizes   : 36 bits physical, 48 bits virtual
...
```

lscpu



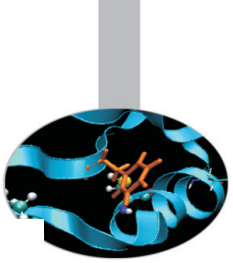
```
Architecture:          x86_64
CPU op-mode(s):        32-bit, 64-bit
Byte Order:            Little Endian
CPU(s):                4
On-line CPU(s) list:   0-3
Thread(s) per core:    2
Core(s) per socket:    2
CPU socket(s):         1
NUMA node(s):          1
Vendor ID:             GenuineIntel
CPU family:            6
Model:                 37
Stepping:              2
CPU MHz:               933.000
BogoMIPS:              4255.78
Virtualization:        VT-x
L1d cache:             32K
L1i cache:             32K
L2 cache:              256K
L3 cache:              3072K
NUMA node0 CPU(s):     0-3
```

Matrix multiplication



What do you think about the obtained results?

Software development



Problem

Solution method

Algorithms

Programming

Source code

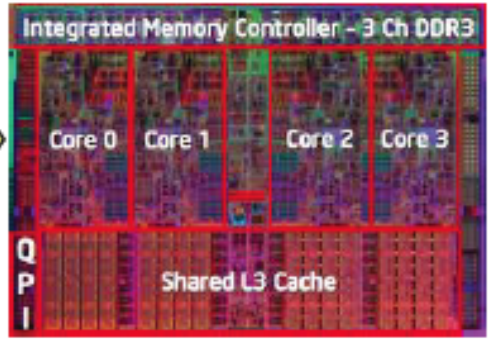
```
#include <stdio.h>
#define SIZE 1000
main(int argc, char** argv) {
  int A[SIZE], B[SIZE], C[SIZE];
  int i;

  for (i=0; i<SIZE; i++) {
    B[i] = i;
    C[i] = SIZE-i;
  }

  /* Add B and C */
  for (i=0; i<SIZE; i++) {
    A[i] = B[i]*C[i];
  }
```

Compiling

Compiled and optimized code

```
.globl main
        .type   main,@function
main:
        pushl %ebp
        movl %esp,%ebp
        subl $12004,%esp
        pushl %edi
        pushl %esi
        pushl %ebx
        nop
        movl $0,-12004(%ebp)
.L2:
        cmpl $999,-12004(%ebp)
        jle .L6
        jmp .L3
        .align 4
.L6:
        movl -12004(%ebp),%eax
        movl %eax,%edx
        leal 0(,%edx,4),%eax
        leal -4000(%ebp),%edx
        movl -12004(%ebp),%ecx
        movl %ecx,%ebx
        leal 0(,%ebx,4),%ecx
        leal -8000(%ebp),%ebx
        movl -12004(%ebp),%esi
        movl %esi,%edi
        leal 0(,%edi,4),%esi
        leal -12000(%ebp),%edi
        movl (%ecx,%ebx),%ecx
        imull (%esi,%edi),%ecx
        movl %ecx,(%eax,%edx)
.L4:
        incl -12004(%ebp)
        jmp .L2
        .align 4
.L3:
        leal -12016(%ebp),%esp
        popl %ebx

.Lfe1:
        .size   main,.Lfe1-main
        .ident  "GCC: (GNU) 2.8.1"
```

Execution

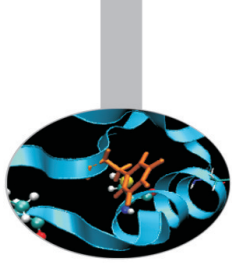


Result

Output

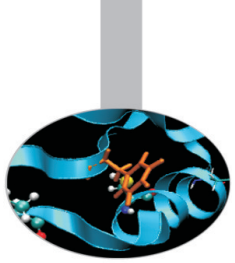
```
Hierarchical code: Plummer model

nbody      dtime       eps       theta     usequad     dtout       tstop
 1024    0.03125    0.0250       1.00       false    0.2500      2.0000

 tnow        Te+U        T/U       nttot      nbavg      ncavg     cputime
0.000     -0.2627     -0.4943      203185         84        114        0.00

          cm pos      0.0000     -0.0000      0.0000
          cm vel     -0.0000      0.0000      0.0000
          am vec      0.0057      0.0156     -0.0222

 tnow        Te+U        T/U       nttot      nbavg      ncavg     cputime
0.031     -0.2627     -0.4940      202260         81        116        0.01

          cm pos      0.0000     -0.0000      0.0000
          cm vel      0.0000     -0.0000      0.0000
          am vec      0.0057      0.0156     -0.0222

Time is:    3.5 seconds
```

Solution Method



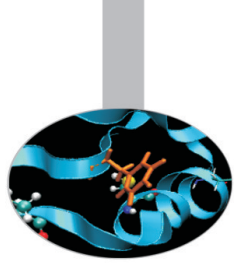
- A problem can typically be solved in many different ways
  - we have to choose a correct and efficient solution method
- A solution may include many different stages of computation using different algorithms
  - Example : first solve a linear equation system, then do a matrix multiplication and after that a FFT
- Each stage in the solution may operate on the same data
  - the data representation should be well suited for all the stages of the computation
  - different stages in the solution may have conflicting requirements on how data is represented

Algorithms



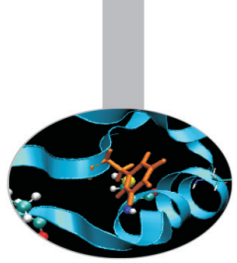
- A specific problem can typically be solved using a number of different algorithms
- The algorithm has to
  - be correct
  - give the required numerical accuracy
  - be efficient, both with respect to execution time and use of memory
- The choice of the numerical algorithm significantly affects the performances.
  - efficient algorithm $\rightarrow$ good performances
  - inefficient algorithm $\rightarrow$ bad performances
- Good performances are related to the choice of the algorithm.
- Golden rule
  - Before writing code choose an efficient algorithm: otherwise, the code must be rewritten!!!!

Programming



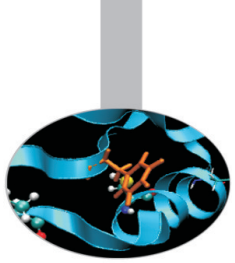
- We program in high level languages
  - C,C++,Fortran,Java Python
- To achieve best performances, languages which are compiled to executable machine code are preferred (C,C++,Fortran,..)
  - the differences in efficiency between these depend mainly on how well developed the compiler is, not so much on the languages themselves
- Interpreted languages, and languages based on byte code are in general less efficient (Python, Java, JavaScript, PHP, ...)
  - the program code is not directly executed by the processor, but goes instead through a second step of interpretation
- High-level code is translated into machine code by a compiler
- the compiler transforms the program into an equivalent but more efficient program
- the compiler must ensure that the optimized program implements exactly the same computation as the original program

Compiler optimization



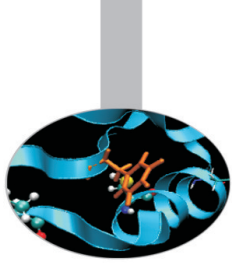
- The compiler analyzes the code and tries to apply a number of optimization techniques to improve the performance
  - it tries to recognize code which can be replaced with equivalent, but more efficient code
- Modern compilers are very good at low-level optimization
  - register allocation, instruction reordering, dead code removal, common subexpression elimination, function inlining , loop unrolling, vectorization, ...
- To enable the compiler to analyze and optimize the code the programmer should:
  - avoid using programming language constructs which are known to be inefficient
  - avoid programming constructs that are difficult for the compiler to optimize (optimization blockers)
  - avoid unnecessarily complicated constructs or tricky code, which makes the compiler analysis difficult
  - write simple and well-structured code, which is easy for the compiler to analyze and optimize

Execution



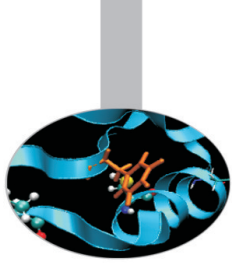
- Modern processors are very complex systems
  - superscalar, superpipelined architecture
  - out of order instruction execution
  - multi-level cache with pre-fetching and write-combining
  - branch prediction and speculative instruction execution
  - vector operations
- It is very difficult to understand exactly how instruction are executed by the processor
- Difficult to understand how different alternative program solutions will affect performance
  - programmers often have a weak understanding of what happens when a program is executed

What to optimize

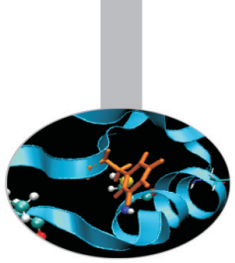


- Find out where the program spends most of its time
  - it is unnecessary to optimize code that is seldom executed
- The 90/10 rule
  - a program spends 90% of its time in 10% of the code
  - look for optimizations in this 10% of the code
- Use tools to find out where a program spends its time
  - profilers
  - hardware counters

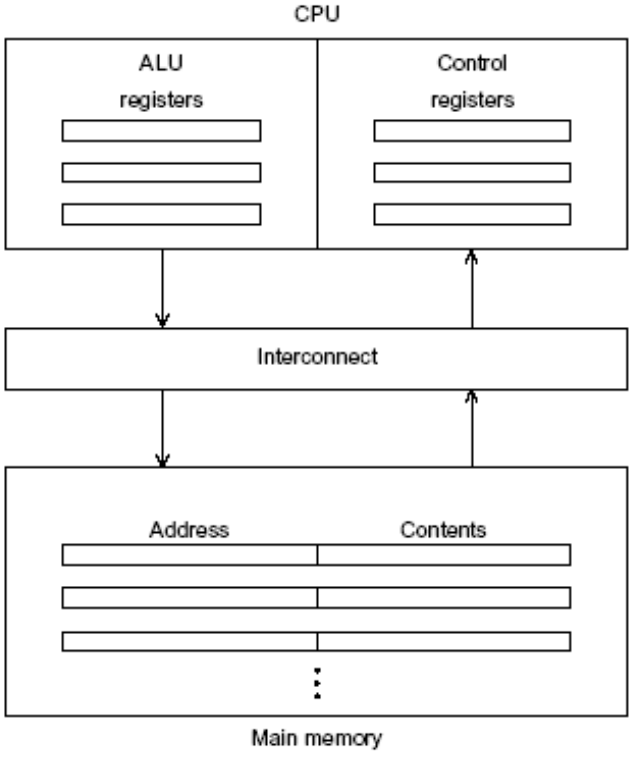
Outline



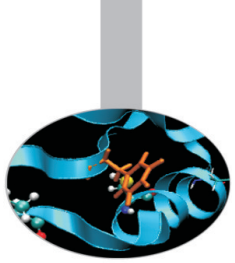
Von Neuman achitecture



- Central processing unit (CPU)
  - Arithmetic Logic Unit (perfoms all arithmetic computations)
  - Control Unit
  - Registers (fast memory)
- Interconnession (Bus)
- Random Access Memory (RAM)
  - Adress to access the memory locations
  - Data contents (istrunctions, data)

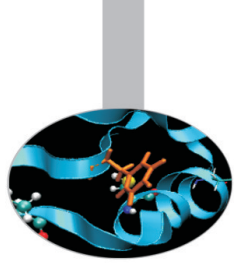


Von Neumann Architecture



- Data are transferred from memory to CPU (fetch or read instrunction)
- Data are transferred from CPU to memory (written to memory o stored)
- Von Neumann Architectures carry out instrunctions one after another, in a single linear sequence, and they spend a lot of time moving data to and from memory. This slows the computer
- The difficulty of overcoming the disparity between processor speeds and data access speeds is called Von Neumann bottleneck.
- The modern CPU are able to perform the instructions at least one hundred times faster than the time required to recover data from the RAM (fetch instruction).

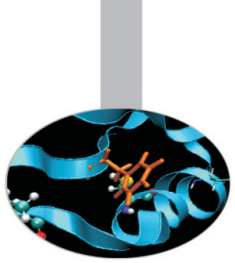
The evolution of computing systems



The solution for the von Neumann bottleneck are:

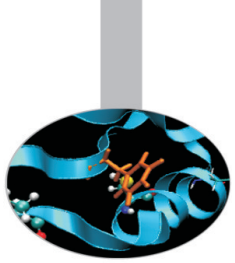
- Caching
  Very fast memories that are physically located on the chip of the processor.
  There are multi levels of cache (first, second and third).
- Virtual memory
  The RAM works as a cache to store large amounts of data.
- Instruction level parallelism
  A family of processor and compiler design techniques that speed up execution by causing individual machine operations to execute in parallel (pipelining ,multiple issue).

Flynn's classification

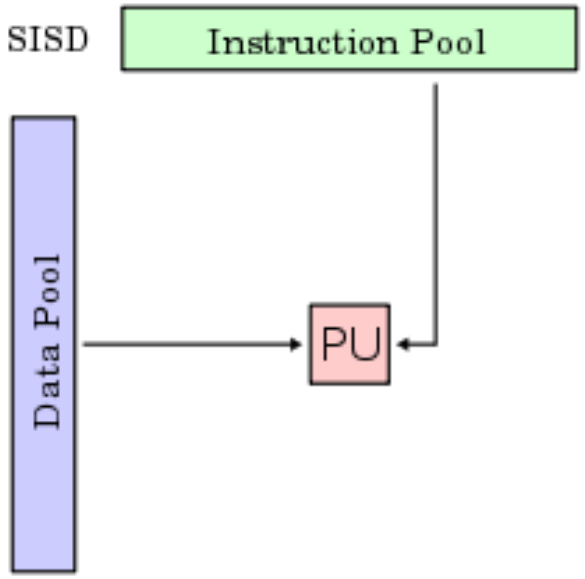


- A classification of computer architectures based on instructions and data streams.
  - SISD:single instruction, single data. Traditional Von Neumann architecture, scalar uniprocessor system.
  - SIMD:single instruction, multiple data. Vector architectures, Vector processors, GPU.
  - MISD:multiple instruction, single data. Does not exist.
  - MIMD:multiple instruction, multiple data. Different processors/cores may be executing different instructions on different pieces of data or run multiple independent programs at the same time.
- The modern architectures are a mixed of these classes.

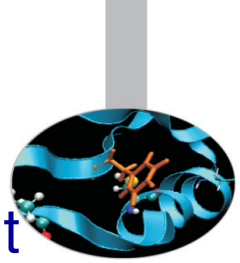
SISD



- Computers with only one executive unit and one memory. At one time, one instruction operates on one data.
- Good performances can be achieved increasing the bus data and the levels of memory or by pipelining and multiple issues.

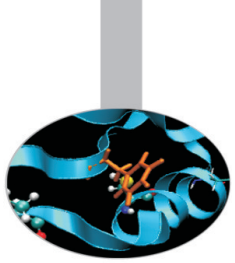


SIMD



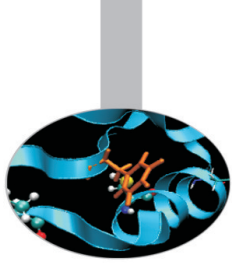
- The same instruction is executed synchronously on different sets of data.
  - Model for synchronous computation.
- Vector processors.
  - Many ALUs
  - vector registers
  - vector Load/Store Units
  - vector instructions
  - interleaved memory
  - OpenMP, MPI
- Graphical Processing Unit
  - GPU fully programmable
  - many ALUs
  - many Load/Store units
  - many SFUs
  - many thousands of parallel threads
  - CUDA

MIMD



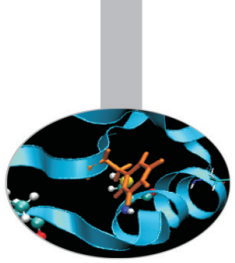
- Multiple autonomous processors simultaneously executing different instructions on different data
  - Model for asynchronous computation
- Cluster
  - a large number of compute nodes (hundreds, thousands)
  - Many nodes with many multicore processors.
  - Shared memory on a node
  - Distributed memory between nodes
  - Multi-level memory hierarchies
  - OpenMP, MPI, MPI+OpenMP

Bandwidth



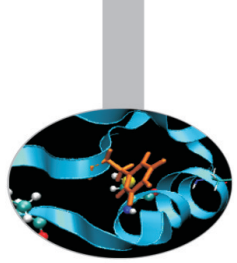
- It is the rate at which data can be read from or stored into the memory by a processor.
- It is expressed in units of bytes/second (Mb/s, Gb/s, etc..)
- A = B * C
  - Read B data from the memory
  - Read C data from the memory
  - Calculate B * C product
  - Save the result in memory (A variable)
- 1 floating-point operation $\rightarrow$ 3 memory accesses

Stream



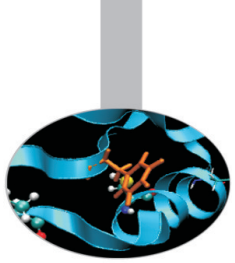
- It is a simple, synthetic benchmark designed to measure sustainable memory bandwidth (in MB/s) and a corresponding computation rate for four simple vector kernels
  - Copy a $\rightarrow$ c (copy)
  - Copy a*b $\rightarrow$ c (scale)
  - Sum a+b $\rightarrow$ c (add)
  - Sum a+b*c $\rightarrow$ d (triad)
- http://www.cs.virginia.edu/stream/ref.html

www.top500.org

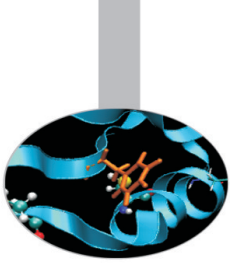


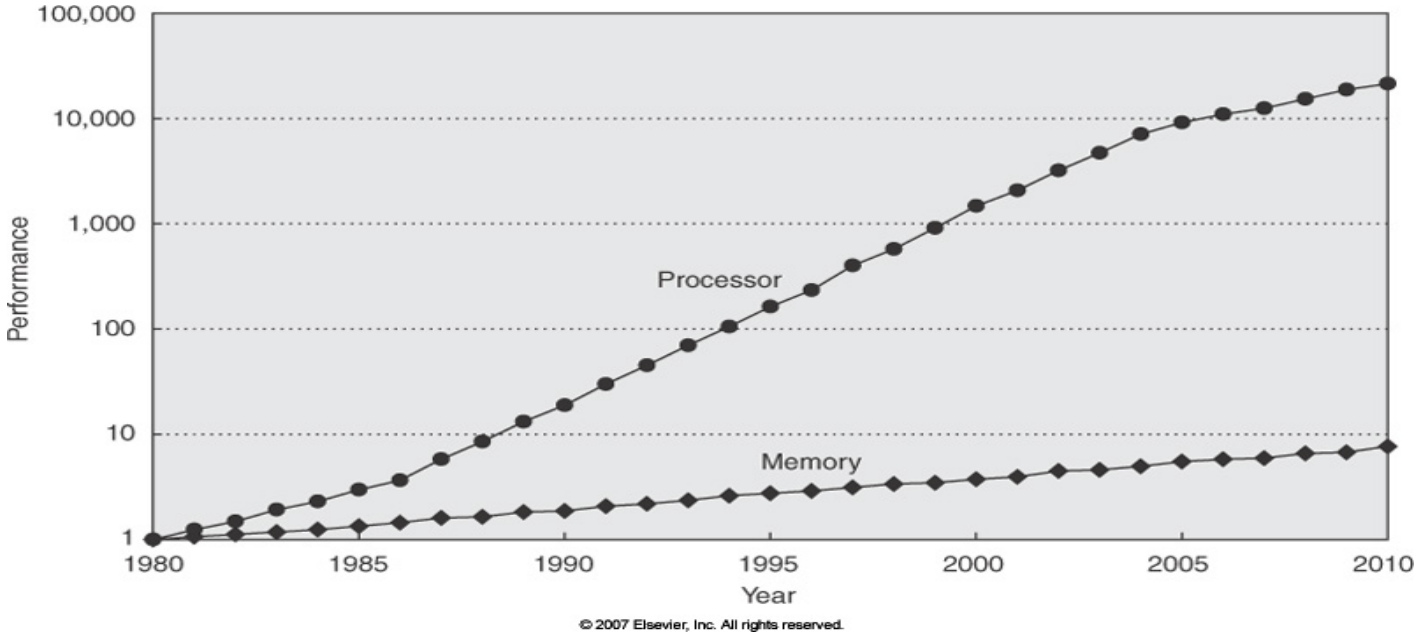
RANK	SITE	SYSTEM	CORES	RMAX (TFLOP/S)	RPEAK (TFLOP/S)	POWER (KW)
1	National Supercomputing Center in Wuxi China	**Sunway TaihuLight** - Sunway MPP, Sunway SW26010 260C 1.45GHz, Sunway NRCPC	10,649,600	93,014.6	125,435.9	15,371
2	National Super Computer Center in Guangzhou China	**Tianhe-2 (MilkyWay-2)** - TH-IVB-FEP Cluster, Intel Xeon E5-2692 12C 2.200GHz, TH Express-2, Intel Xeon Phi 31S1P NUDT	3,120,000	33,862.7	54,902.4	17,808
3	DOE/SC/Oak Ridge National Laboratory United States	**Titan** - Cray XK7 , Opteron 6274 16C 2.200GHz, Cray Gemini interconnect, NVIDIA K20x Cray Inc.	560,640	17,590.0	27,112.5	8,209
4	DOE/NNSA/LLNL United States	**Sequoia** - BlueGene/Q, Power BQC 16C 1.60 GHz, Custom IBM	1,572,864	17,173.2	20,132.7	7,890
5	RIKEN Advanced Institute for Computational Science (AICS) Japan	K computer, SPARC64 VIIIfx 2.0GHz, Tofu interconnect Fujitsu	705,024	10,510.0	11,280.4	12,660
6	DOE/SC/Argonne National Laboratory United States	**Mira** - BlueGene/Q, Power BQC 16C 1.60GHz, Custom IBM	786,432	8,586.6	10,066.3	3,945
7	DOE/NNSA/LANL/SNL United States	**Trinity** - Cray XC40, Xeon E5-2698v3 16C 2.3GHz, Aries interconnect Cray Inc.	301,056	8,100.9	11,078.9	
8	Swiss National Supercomputing Centre (CSCS) Switzerland	**Piz Daint** - Cray XC30, Xeon E5-2670 8C 2.600GHz, Aries interconnect , NVIDIA K20x Cray Inc.	115,984	6,271.0	7,788.9	2,325
9	HLRS - Höchstleistungsrechenzentrum Stuttgart Germany	**Hazel Hen** - Cray XC40, Xeon E5-2680v3 12C 2.5GHz, Aries interconnect Cray Inc.	185,088	5,640.2	7,403.5	
10	King Abdullah University of Science and Technology Saudi Arabia	**Shaheen II** - Cray XC40, Xeon E5-2698v3 16C 2.3GHz, Aries interconnect Cray Inc.	196,608	5,537.0	7,235.2	2,834

Outline



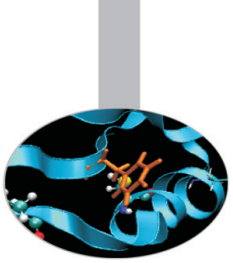
Memory system

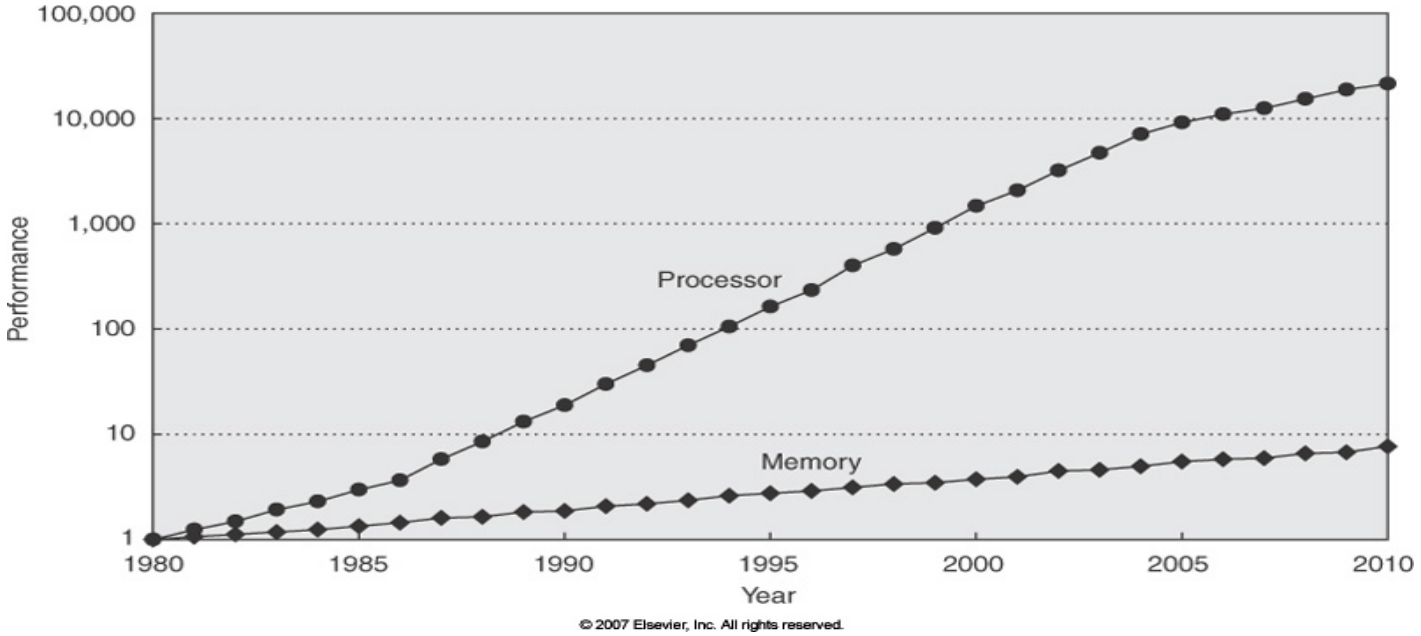




- CPU power computing doubles every 18 months
- Access rate to RAM doubles every 120 months
- Reducing the cost of the operations is useless if the loading data is slow

Memory system

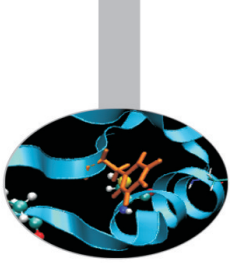


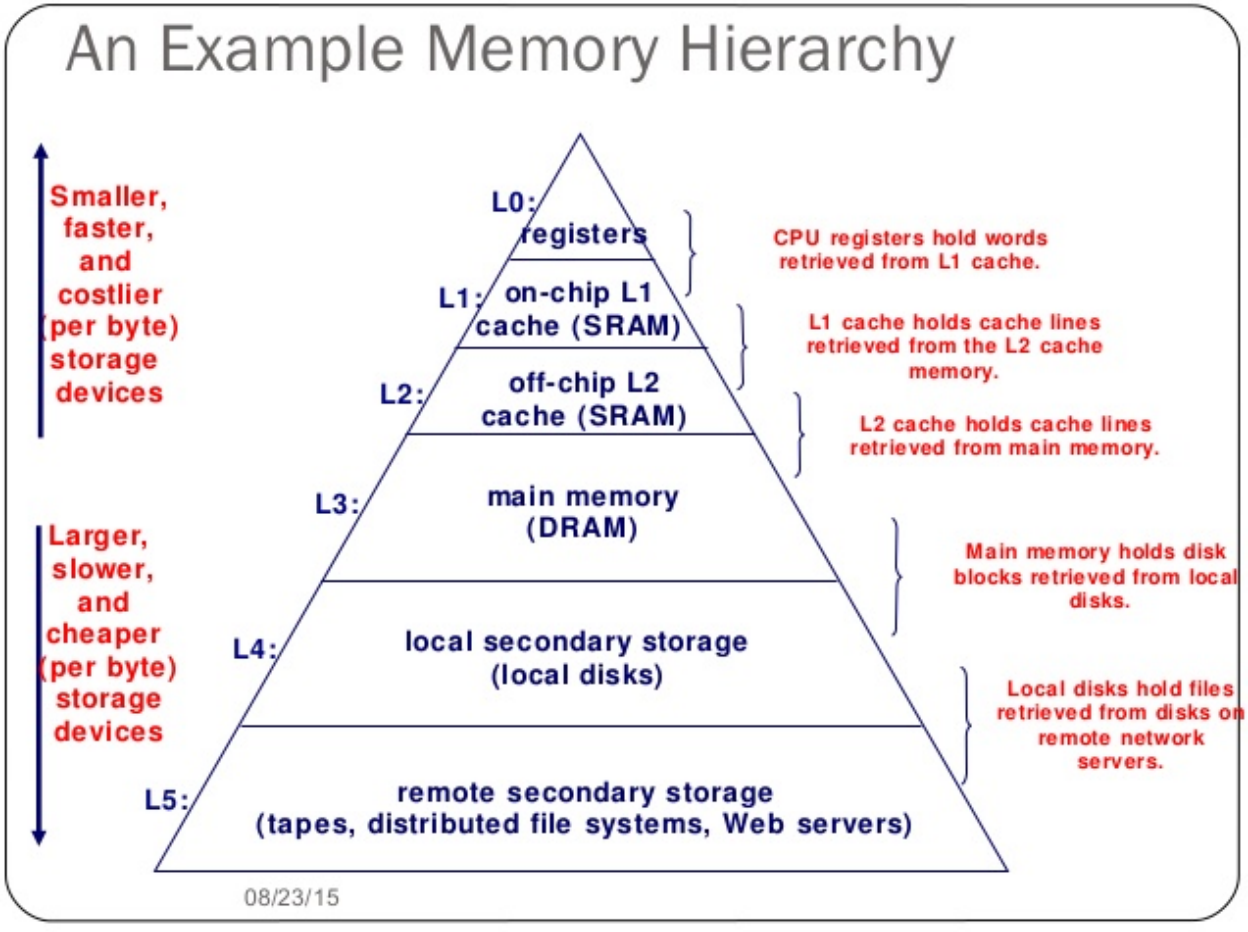


- CPU power computing doubles every 18 months
- Access rate to RAM doubles every 120 months
- Reducing the cost of the operations is useless if the loading data is slow

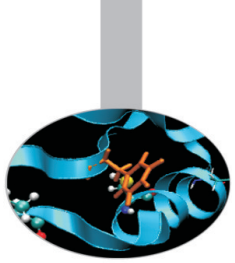
- Solution: intermediate fast memory layers
- A Hierarchical Memory System
- The hierarchy is transparent to the application but the performances are strongly enhanced

The Memory Hierarchy



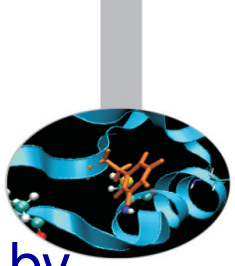


SRAM vs. DRAM



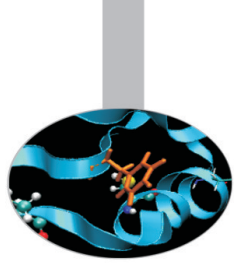
- Dynamic RAM (DRAM) main memory
  - one transistor cell
  - cheap
  - it needs to be periodically refreshed
    - data are not available during refreshing
- Static RAM (SRAM) cache memory
  - cell requires 6-7 transistor
  - expensive
  - it does not need to be refreshed
    - data are always available.
- DRAM has better price/performance than SRAM
  - also higher densities, need less power and dissipate less heat
- SRAM provides higher speed
  - used for high-performance memories (registers, cache memory)

Clock cycle



- The speed of a computer processor, or CPU, is determined by the clock cycle, which is the amount of time between two pulses of an oscillator.
- Generally speaking, the higher number of pulses per second, the faster the computer processor will be able to process information
- The clock speed is measured in Hz, typically either megahertz (MHz) or gigahertz (GHz). For example, a 4GHz processor performs 4,000,000,000 clock cycles per second.
- Computer processors can execute one or more instructions per clock cycle, depending on the type of processor.
- Early computer processors and slower CPUs can only execute one instruction per clock cycle, but modern processors can execute multiple instructions per clock cycle.

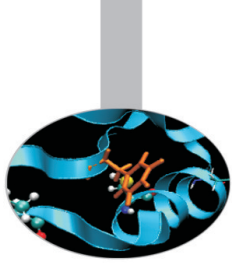
Clock cycles cost



- Here is the cost of the main operations:
  - add: 1 CPU cycle
  - sub: 1 CPU cycle
  - mul: 1 CPU cycle
  - div : $\cong$ 20 CPU cycles
- As we can see, a division is very expensive compared to a multiplication
- It is much better to compute the inverse number and multiply by it!
- Be careful, when we multiply by inverse we lose some precision in the calculation
- FMA

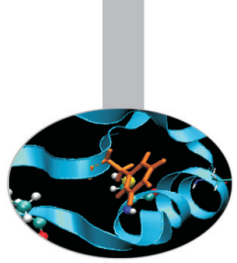
```
d=a*b+c
```

The Memory Hierarchy



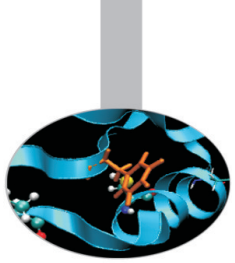
- From small, fast and expensive to large, slow and cheap
- Access times increase as we go down in the memory hierarchy
- Typical access times (Intel Nehalem)
  - register — immediately (0 clock cycles)
  - L1 — 3 clock cycles
  - L2 — 13 clock cycles
  - L3 — 30 clock cycles
  - memory — 100 clock cycles
  - disk — 100000 - 1000000 clock cycles

The Cache



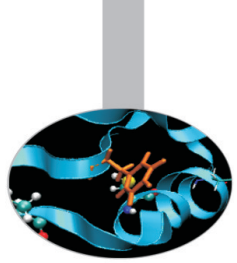
Why this hierarchy?

The Cache



Why this hierarchy?
It is not necessary that all data are available at the same time.
What is the solution?

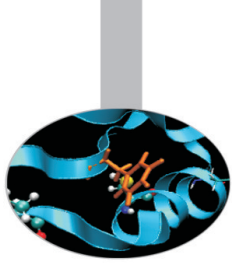
The Cache



Why this hierarchy?
It is not necessary that all data are available at the same time.
What is the solution?

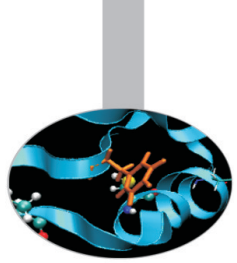
- The cache is divided in one (or more) levels of intermediate memory, rather fast but small sized (kB $\div$ MB)
- Basic principle: we always work with a subset of data.
  - data needed $\rightarrow$ fast memory access
  - data not needed (for now) $\rightarrow$ slower memory levels
- Limitations
  - Random access without reusing
  - Never large enough . . .
  - faster, hotter and . . . expensive $\rightarrow$ intermediate levels hierarchy.

The cache



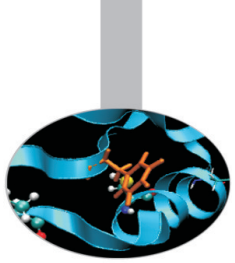
- CPU accesses higher level cache:
- The cache controller finds if the required element is present in cache:
  - Yes: data in cache is used
  - No: new data is loaded in cache; if cache is full, a replacement policy is used to replace (a subset of) the current data with the new data
- The data replacement between main memory and cache is performed in data chunks, called cache lines or cache blocks.
- block = The smallest unit of information that can be transferred between two memory levels (between two cache levels or between RAM and cache)
  - consists of a number of consecutive memory locations
  - typical cache block size is 64 bytes

Replacement: locality principles



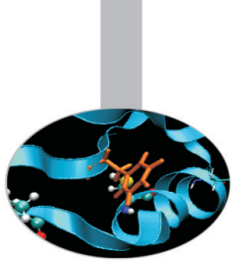
- Spatial locality
  - High probability to access memory cell with contiguous address within a short space of time (sequential instructions; data arranged in matrix and vectors sequentially accessed, etc.)

Replacement: locality principles



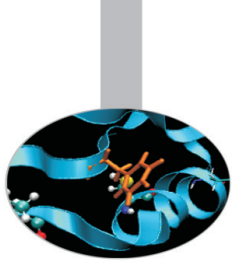
- Spatial locality
  - High probability to access memory cell with contiguous address within a short space of time (sequential instructions; data arranged in matrix and vectors sequentially accessed, etc.)
  - Possible advantage: we read more data than we need (complete block) in hopes of next request

Replacement: locality principles



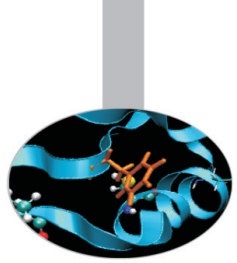
- Spatial locality
  - High probability to access memory cell with contiguous address within a short space of time (sequential instructions; data arranged in matrix and vectors sequentially accessed, etc.)
  - Possible advantage: we read more data than we need (complete block) in hopes of next request
- Temporal locality
  - High probability to access memory cell that was recently accessed within a short space of time (instructions within body of cycle frequently and sequentially accessed, etc.)

Replacement: locality principles



- Spatial locality
  - High probability to access memory cell with contiguous address within a short space of time (sequential instructions; data arranged in matrix and vectors sequentially accessed, etc.)
  - Possible advantage: we read more data than we need (complete block) in hopes of next request
- Temporal locality
  - High probability to access memory cell that was recently accessed within a short space of time (instructions within body of cycle frequently and sequentially accessed, etc.)
  - We take advantage replacing the least recently used blocks

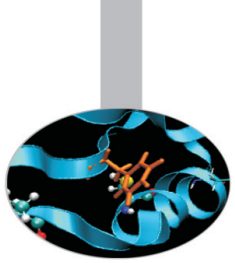
Replacement: locality principles



- Spatial locality
  - High probability to access memory cell with contiguous address within a short space of time (sequential instructions; data arranged in matrix and vectors sequentially accessed, etc.)
  - Possible advantage: we read more data than we need (complete block) in hopes of next request
- Temporal locality
  - High probability to access memory cell that was recently accessed within a short space of time (instructions within body of cycle frequently and sequentially accessed, etc.)
  - We take advantage replacing the least recently used blocks

Data required from CPU are stored in the cache with contiguous memory cells as long as possible

Example: Locality?

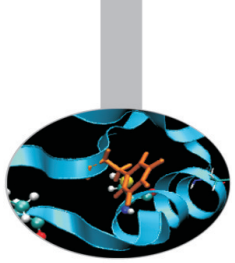


```
sum = 0;
for ( i = 0; i < n; i ++)
sum += a[ i ];
return sum;
```

- Data:
  - Temporal: sum referenced in each iteration
  - Spatial: array a[ ] accessed consecutively
- Instructions:
  - Temporal: loops cycle through the same instructions
  - Spatial: instructions referenced in sequence

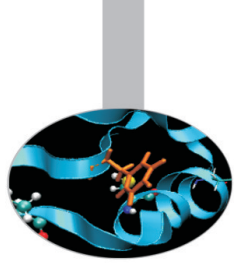
Being able to assess the locality of code is a crucial skill for a performance programmer

Cache:Some definition



- Hit: The requested data from CPU is stored in cache
- Miss: The requested data from CPU is not stored in cache
- Hit rate: The percentage of all accesses that are satisfied by the data in the cache.
- Miss rate:The number of misses stated as a fraction of attempted accesses (miss rate = 1-hit rate).
- Hit time: Memory access time for cache hit (including time to determine if hit or miss)
- Miss penalty: Time to replace a block from lower level, including time to replace in CPU (mean value is used)
- Miss time: = miss penalty + hit time, time needed to retrieve the data from a lower level if cache miss is occurred.

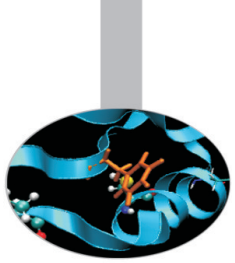
Cache: access cost



Level	access cost
L1	1 clock cycle
L2	7 clock cycles
RAM	36 clock cycles

- 100 accesses with 100% cache hit: $\rightarrow$ t=100
- 100 accesses with 5% cache miss in L1: $\rightarrow$ t=130
- 100 accesses with 10% cache miss in L1 $\rightarrow$ t=160
- 100 accesses with 10% cache miss in L2 $\rightarrow$ t=450
- 100 accesses with 100% cache miss in L2 $\rightarrow$ t=3600

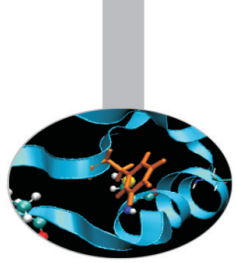
Cache miss in all levels



1. search two data, A and B
2. search A in the first level cache (L1) O(1) cycles
3. search A in the second level cache (L2) O(10) cycles
4. copy A from RAM to L2 to L1 to registers O(10) cycles
5. search B in the first level cache (L1) O(1) cycles
6. search B in the second level cache (L2) O(10) cycles
7. copy B from RAM to L2 to L1 to registers O(10) cycles
8. run command

O(100)overhead cycles !!!

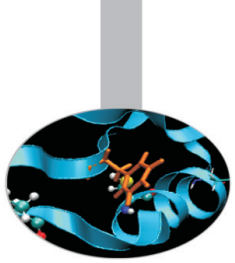
Cache hit in all levels



- search two data, A and B
- search A in the first level cache(L1) O(1) cycles
- search B in the first level cache(L1) O(1) cycles
- run command

O(1) overhead cycles

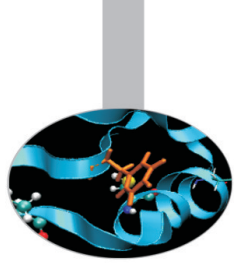
Performance estimate: an example



```
float sum = 0.0f;
  for (i = 0; i < n; i++)
    sum = sum + x[i]*y[i];
```

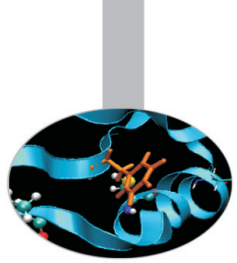
- At each iteration, one sum and one multiplication floating-point are performed
- The number of the operations performed is $2 \times n$

Execution time T_{es}



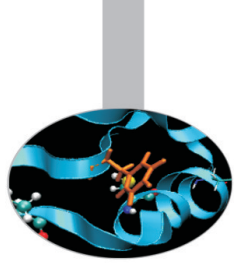
- $T_{es} = N_{flop} * t_{flop}$
- $N_{flop} \rightarrow$ Algorithm

Execution time T_{es}



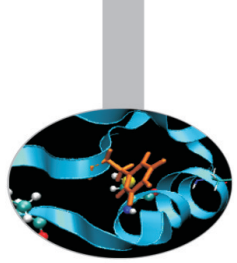
- $T_{es} = N_{flop} * t_{flop}$
- $N_{flop} \rightarrow$ Algorithm
- $t_{flop} \rightarrow$ Hardware

Execution time T_{es}



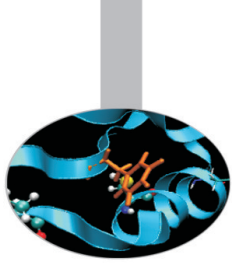
- $T_{es} = N_{flop} * t_{flop}$
- $N_{flop} \rightarrow$ Algorithm
- $t_{flop} \rightarrow$ Hardware
- consider only execution time
- What are we neglecting?

Execution time T_{es}



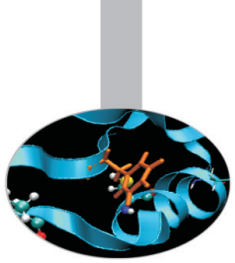
- $T_{es} = N_{flop} * t_{flop}$
- $N_{flop} \rightarrow$ Algorithm
- $t_{flop} \rightarrow$ Hardware
- consider only execution time
- What are we neglecting?
- t_{mem} The required time to access data in memory.

Therefore . . .



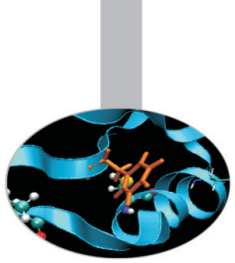
- $T_{es} = N_{flop} * t_{flop} + N_{mem} * t_{mem}$
- $t_{mem} \rightarrow$ Hardware
- How N_{mem} affects the performances?

N_{mem} Effect



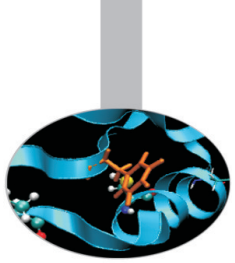
- $Perf = \frac{N_{Flop}}{T_{es}}$
- for $N_{mem} = 0 \rightarrow Perf^* = \frac{1}{t_{flop}}$
- for $N_{mem} > 0 \rightarrow Perf = \frac{Perf^*}{1 + \frac{N_{mem} * t_{mem}}{N_{flop} * t_{flop}}}$
- Performance decay factor
- $\frac{N_{mem}}{N_{flop}} * \frac{t_{mem}}{t_{flop}}$
- how to achieve the peak performance?

N_{mem} Effect



- $Perf = \frac{N_{Flop}}{T_{es}}$
- for $N_{mem} = 0 \rightarrow Perf^* = \frac{1}{t_{flop}}$
- for $N_{mem} > 0 \rightarrow Perf = \frac{Perf^*}{1 + \frac{N_{mem} * t_{mem}}{N_{flop} * t_{flop}}}$
- Performance decay factor
- $\frac{N_{mem}}{N_{flop}} * \frac{t_{mem}}{t_{flop}}$
- how to achieve the peak performance?
- Minimize the memory accesses.

Spatial locality: access order

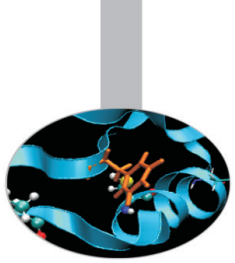


- Matrix multiplication in double precision 512X512
- Measured MFlops on Jazz (Intel(R) Xeon(R) CPU X5660 2.80GHz)
- gfortran compiler with -O0 optimization

index order	Fortran	C
i,j,k	109	128
i,k,j	90	177
j,k,i	173	96
j,i,k	110	127
k,j,i	172	96
k,i,j	90	177

The efficiency of the access order depends more on the data location in memory, rather than on the language.

Array in memory

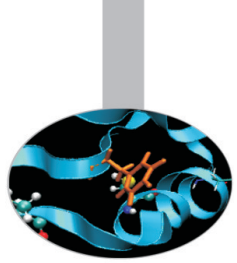


- Memory $\rightarrow$ elementary locations sequentially aligned
- A matrix, a_{ij} element : i row index, j column index
- Matrix representation is by arrays

- How are the array elements stored in memory?

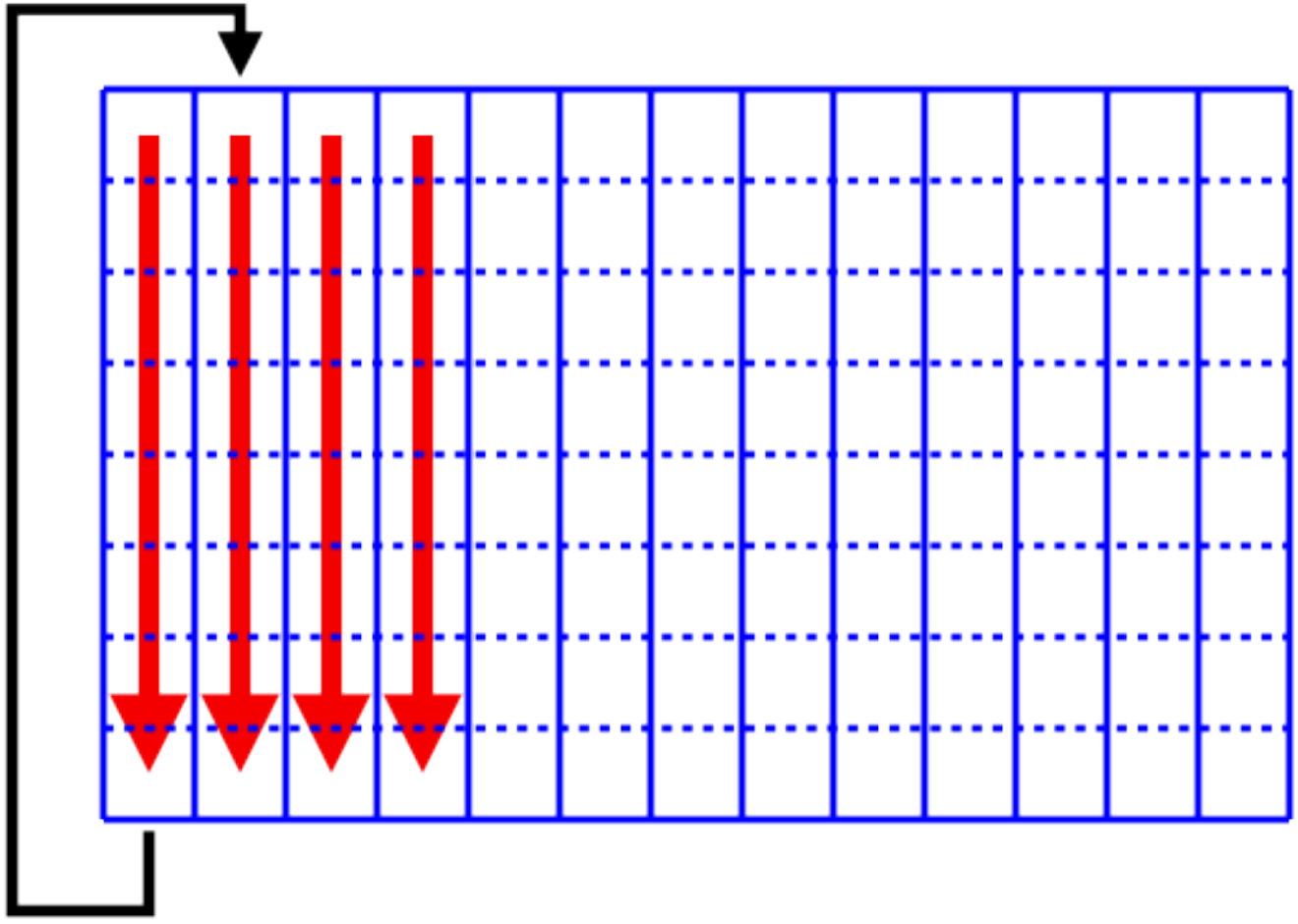
- C: sequentially access starting from the last index, then the previous index . . .
  a[1][1] a[1][2] a[1][3] a[1][4] . . .
  a[1][n] a[2][n] . . . a[n][n]
- Fortran: sequentially access starting from the first index, then the second index . . .
  a(1,1) a(2,1) a(3,1) a(4,1) . . .
  a(n,1) a(n,2) . . . a(n,n)

The stride

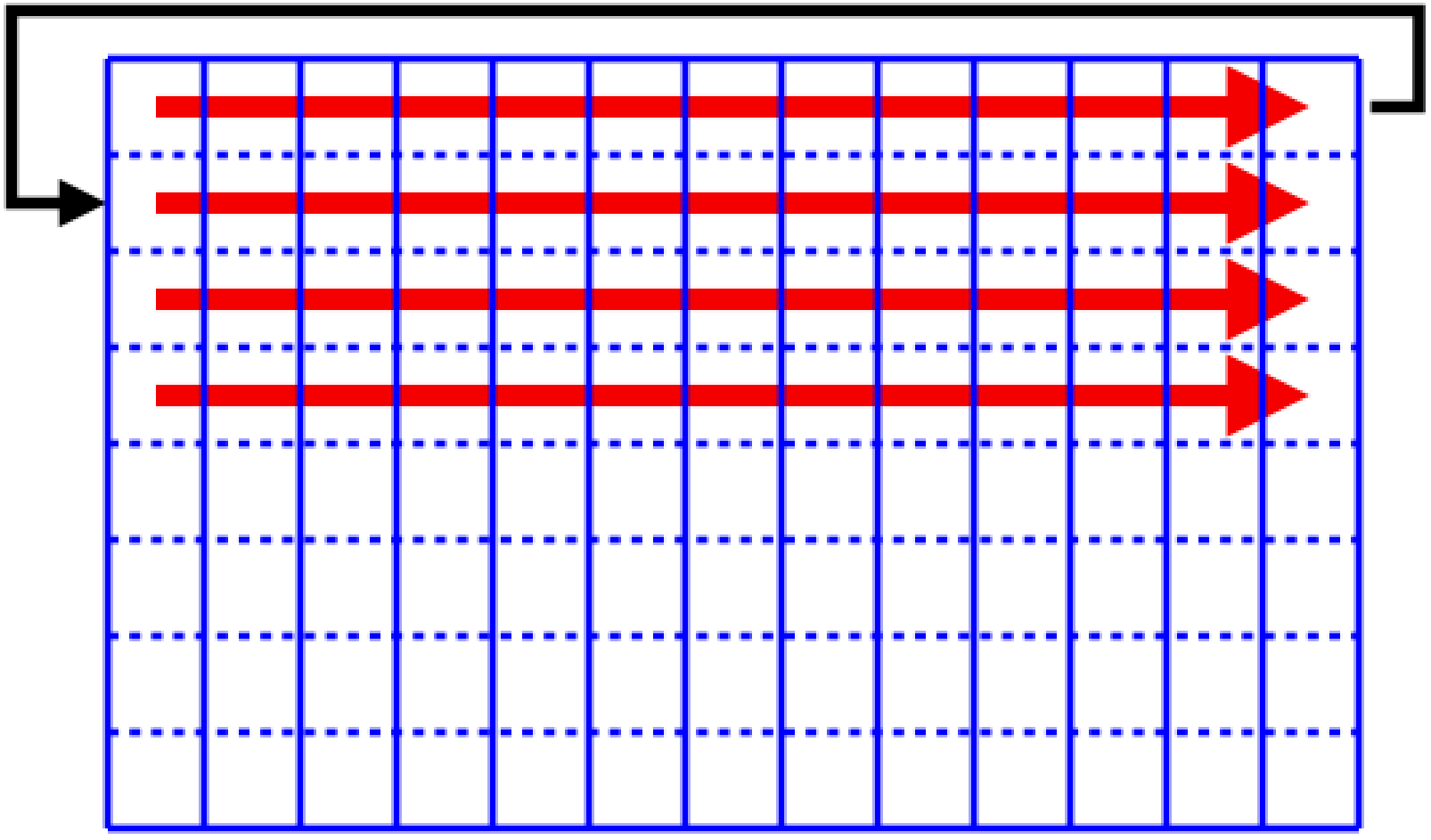


- The distance between successively accessed data
  - stride=1 $\rightarrow$ I take advantage of the spatial locality
  - stride » 1 $\rightarrow$ I don't take advantage of the spatial locality
- Golden rule
  - Always access arrays, if possible, with unit stride.

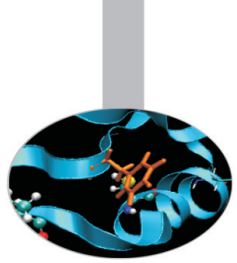
Fortran memory ordering



C memory ordering

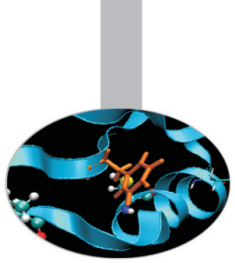


Best access order



- Calculate multiplication matrix-vector:
  - Fortran: d(i) = a(i) + b(i,j)*c(j)
  - C: d[i] = a[i] + b [i][j]*c[j];
- Fortran
  - do j=1,n
    do i=1,n
    d(i) = a(i) + b(i,j)*c(j)
    end do
    end do
- C
  - for(i=0;i<n,i++1)
    for(j=0;i<n,j++1)
    d[i] = a[i] + b [i][j]*c[j];

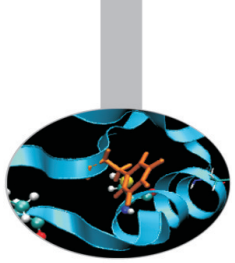
Spatial locality:linear system



Solving triangular system

- $Lx = b$
- Where:
  - L $n \times n$ lower triangular matrix
  - x n unknowns vector
  - b n right hand side vector
- we can solve this system by:
  - forward substitution
  - partitioning matrix

Spatial locality:linear system

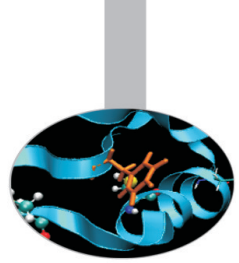


Solving triangular system

- $Lx = b$
- Where:
  - L $n \times n$ lower triangular matrix
  - x n unknowns vector
  - b n right hand side vector
- we can solve this system by:
  - forward substitution
  - partitioning matrix

What is faster?
Why?

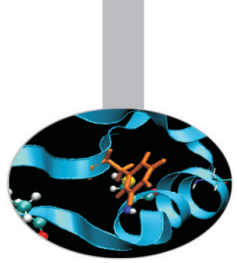
Forward substitution



Solution:

```
. . .
do i = 1, n
     do j = 1, i-1
        b(i) = b(i) - L(i,j) b(j)
    enddo
    b(i) = b(i)/L(i,i)
enddo
. . .
```

Forward substitution



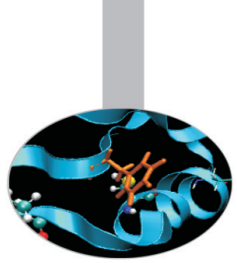
Solution:

```
...
do i = 1, n
     do j = 1, i-1
        b(i) = b(i) - L(i,j) b(j)
    enddo
    b(i) = b(i)/L(i,i)
enddo
...
```

```
[vruggie1@fen07 TRI]$ ./a.out
```

```
time for solution    8.0586
```

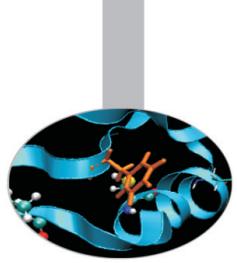
Matrix partitioning



Solution:

```
. . .
do j = 1, n
    b(j) = b(j)/L(j,j)
     do i = j+1,n
         b(i) = b(i) - L(i,j)*b(j)
    enddo
enddo
. . .
```

Matrix partitioning



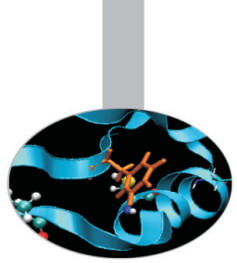
Solution:

```
...
do j = 1, n
    b(j) = b(j)/L(j,j)
     do i = j+1,n
        b(i) = b(i) - L(i,j)*b(j)
    enddo
enddo
...
```

```
[vruggie1@fen07 TRI]$ ./a.out
```

```
time for solution   2.5586
```

What is the difference?



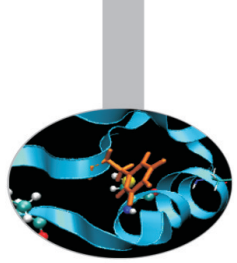
- Forward substitution

```
do i = 1, n
     do j = 1, i-1
        b(i) = b(i) - L(i,j) b(j)
     enddo
     b(i) = b(i)/L(i,i)
enddo
```

- Matrix partitioning

```
do j = 1, n
     b(j) = b(j)/L(j,j)
     do i = j+1,n
        b(i) = b(i) - L(i,j)*b(j)
     enddo
enddo
```

What is the difference?



- Forward substitution

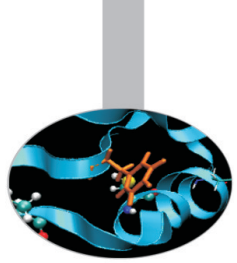
```
do i = 1, n
     do j = 1, i-1
        b(i) = b(i) - L(i,j) b(j)
     enddo
     b(i) = b(i)/L(i,i)
enddo
```

- Matrix partitioning

```
do j = 1, n
     b(j) = b(j)/L(j,j)
     do i = j+1,n
        b(i) = b(i) - L(i,j)*b(j)
     enddo
enddo
```

- Same number of operations, but very different elapsed times the difference is a factor of 3
- Why?

Let us clarify…



This matrix is stored:

A	D	G	L
B	E	H	M
C	F	I	N

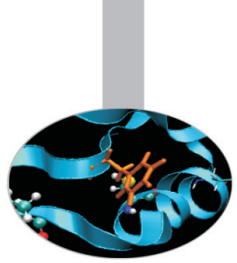
In C:

A	D	G	L	B	E	H	M	C	F	I	N

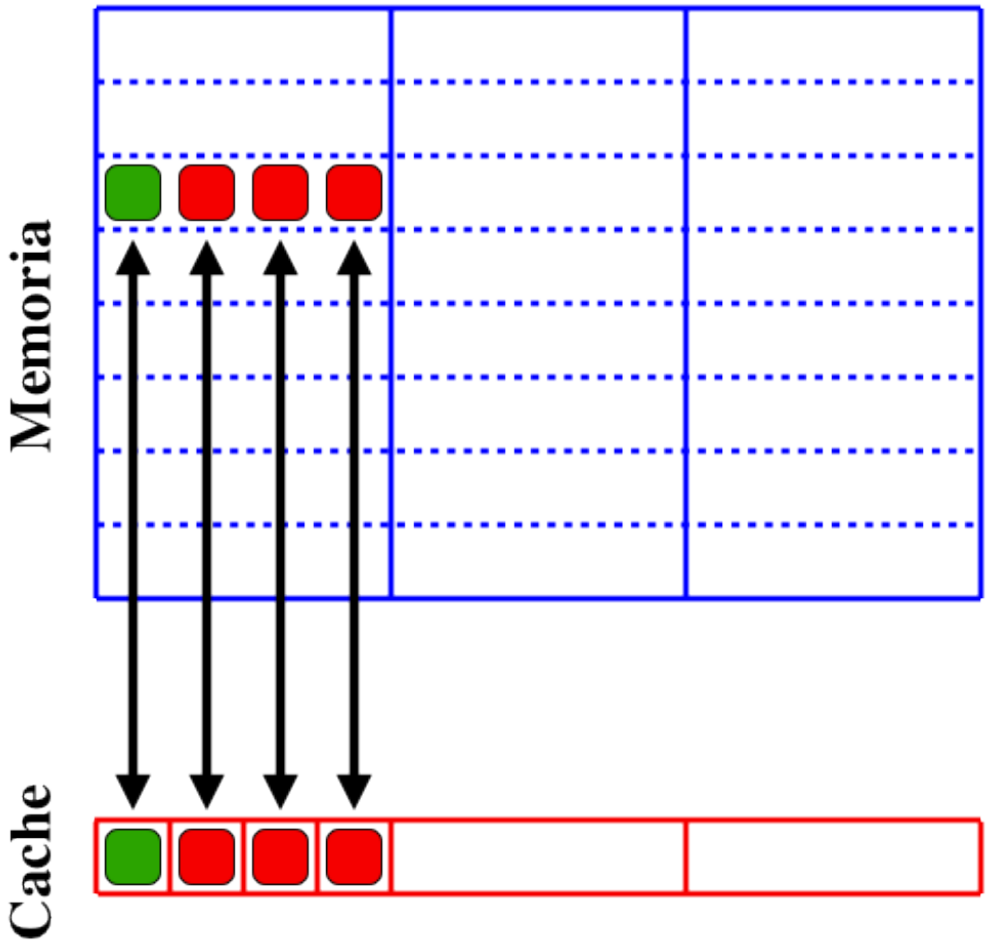
In Fortran:

A	B	C	D	E	F	G	H	I	L	M	N

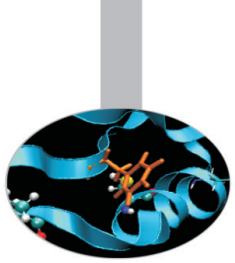
Spatial locality: cache lines



- The cache is structured as a sequence of blocks (lines)
- The memory is divided in blocks with the same size of the cache line
- When data are required the system loads from memory the entire cache line that contains the data.



Dimension and data reuse

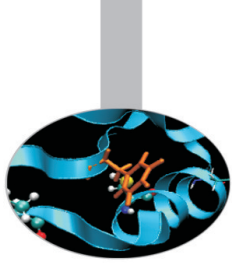


- Multiplication matrix-matrix in double precision
- Versions with different calls to BLAS library
- Performance in MFlops on Intel(R) Xeon(R) CPU X5660 2.80GHz

Dimension	1 DGEMM	*N* DGEMV	N^2 DDOT
500	5820	3400	217
1000	8420	5330	227
2000	12150	2960	136
3000	12160	2930	186

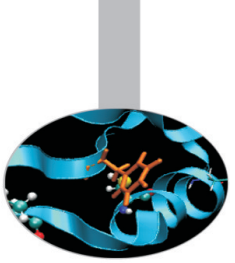
Same number of operations but the use of cache memory is changed!!!

Cache reuse



```
...
d=0.0
do I=1,n
j=index(I)
d = d + sqrt(x(j)*x(j) + y(j)*y(j) + z(j)*z(j))
...
```

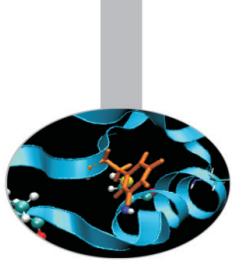
Cache reuse



```
...
d=0.0
do I=1,n
j=index(I)
d = d + sqrt(x(j)*x(j) + y(j)*y(j) + z(j)*z(j))
...
```

Can I change the code to obtain best performances?

Cache reuse

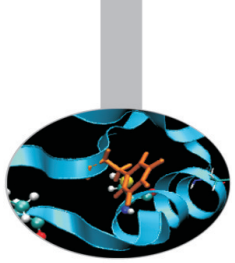


```
...
d=0.0
do I=1,n
j=index(I)
d = d + sqrt(x(j)*x(j) + y(j)*y(j) + z(j)*z(j))
...
```

Can I change the code to obtain best performances?

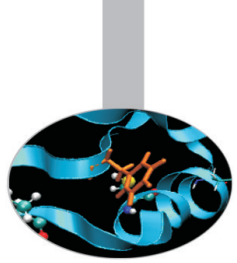
```
...
d=0.0
do I=1,n
j=index(I)
d = d + sqrt(r(1,j)*r(1,j) + r(2,j)*r(2,j) + r(3,j)*r(3,j))
...
```

Registers



- Registers are memory locations inside CPUs
- small amount of them (typically, less than 128), but with zero latency
- All the operations performed by computing units
  - take the operands from registers
  - return results into registers
- transfers memory $\leftrightarrow$ registers are different operations
- Compiler uses registers
  - to store intermediate values when computing expressions
  - too complex expressions or too large loop bodies force the so called “register spilling”
  - to keep close to CPU values to be reused
  - but only for scalar variables, not for array elements

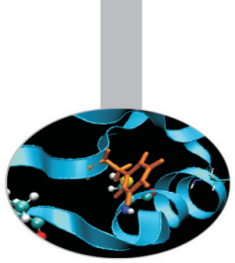
Cache blocking



- Data are processed in chunks fitting into the cache memory
- Cache data are reused when working for the single block
- Compiler can do it for simple loops, but only at high optimization levels
- Example: matrix transpose

```
do jj = 1, n , step
  do ii = 1, n, step
    do j= jj,jj+step-1,1
      do i=ii,ii+step-1,1
        a(i,j)=b(j,i)
      end do
    end do
  end do
end do
```

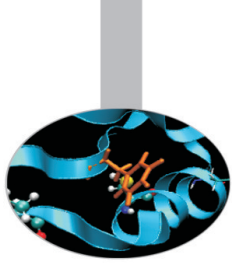
Cache: capacity miss and trashing



Cache may be affected by

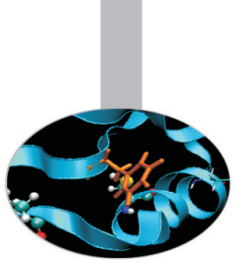
- capacity miss
  - only a few lines are really used (reduced effective cache size)
  - processing rate is reduced
- trashing:
  - a cache line is thrown away even when data need to be reused because new data are loaded
  - slower than not having cache at all!
  - It may occur when different instruction/data flows refer to the same cache lines
  - It depends on how the memory is mapped to the cache

Cache mapping

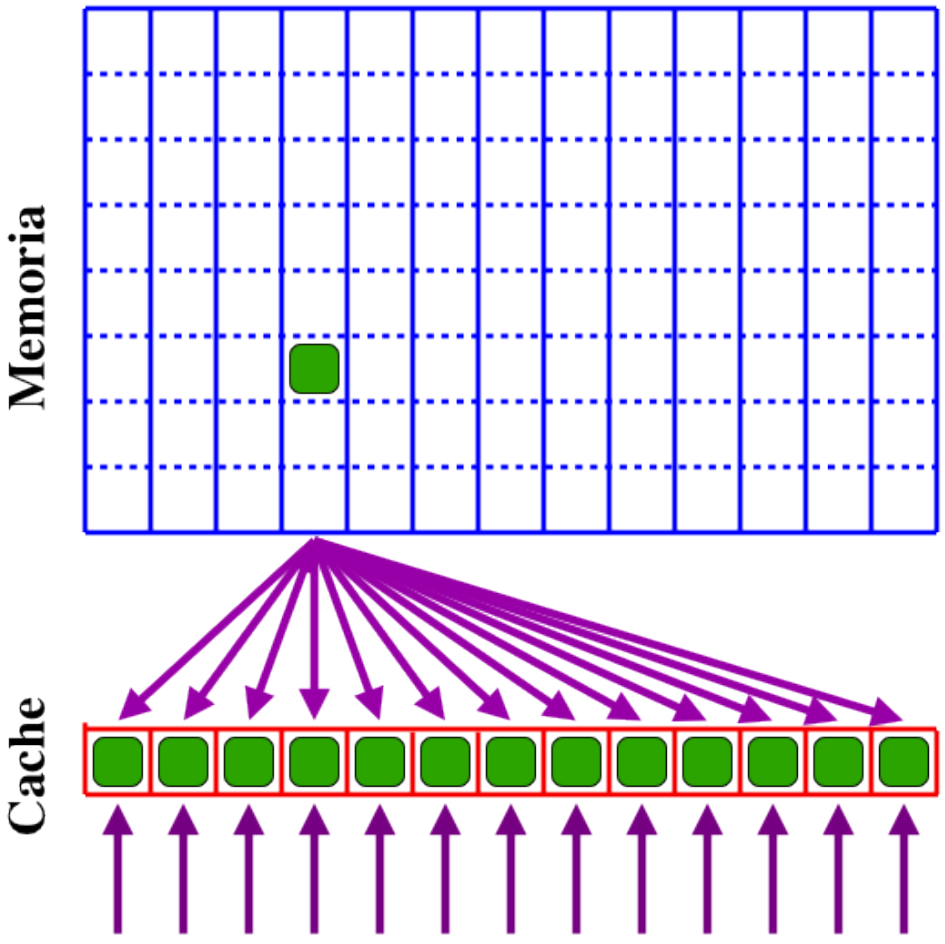


- A cache mapping defines where memory locations will be placed in cache
  - in which cache line a memory addresses will be placed
  - we can think of the memory as being divided into blocks of the size of a cache line
  - the cache mapping is a simple hash function from addresses to cache sets
- Cache is much smaller than main memory
  - more than one of the memory blocks can be mapped to the same cache line
- Each cache line is identified by a tag
  - determines which memory addresses the cache line holds
  - based on the tag and the valid bit, we can find out if a particular address is in the cache (hit) or not (miss)

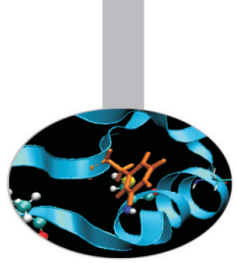
Fully associative cache



- A cache where data from any address can be stored in any cache location.

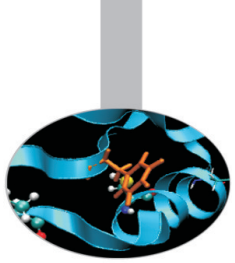


Fully associative cache

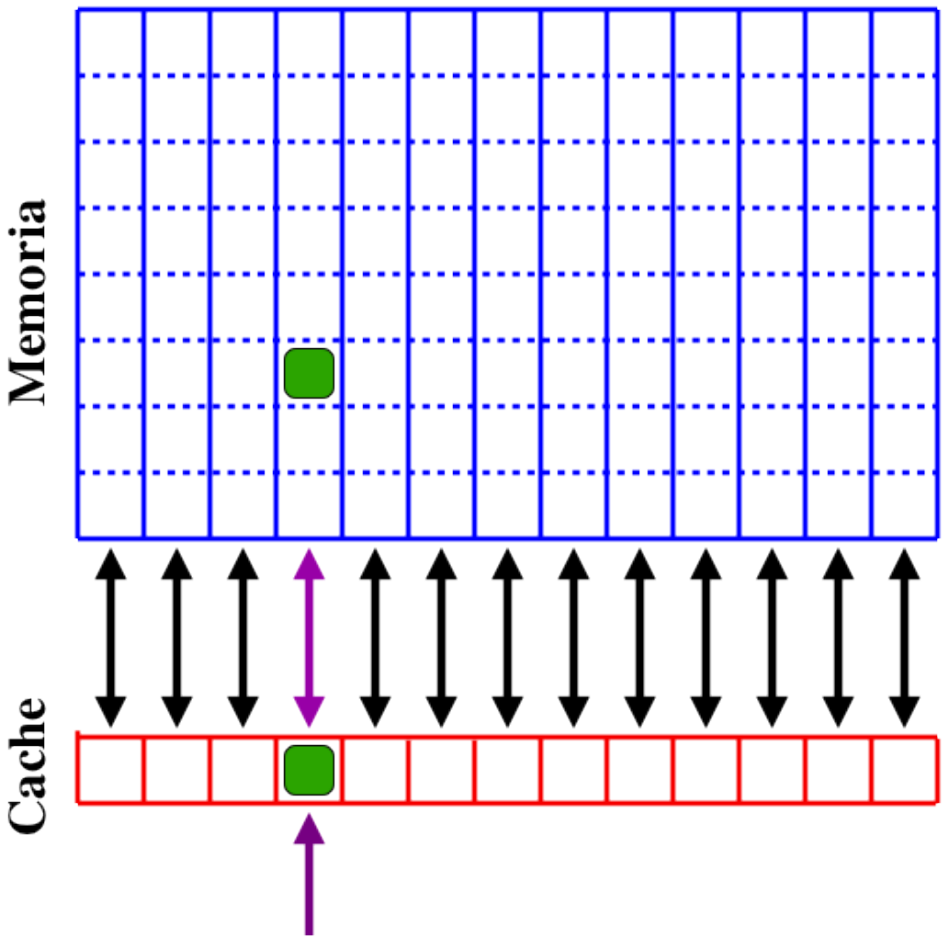


- Pros:
  - full cache exploitation
  - independent of the patterns of memory access
- Cons:
  - complex circuits to get a fast identify of hits
  - substitution algorithm: demanding, Least Recently Used (LRU) or not very efficient First In First Out (FIFO)
  - expensive and small sized

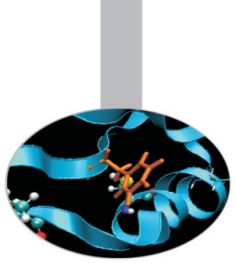
Direct mapped cache



- Each main memory block can be mapped to only one slot. (linear congruence)

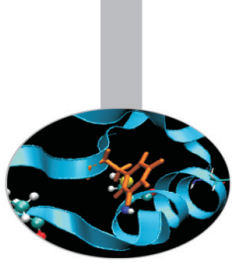


Direct mapped cache

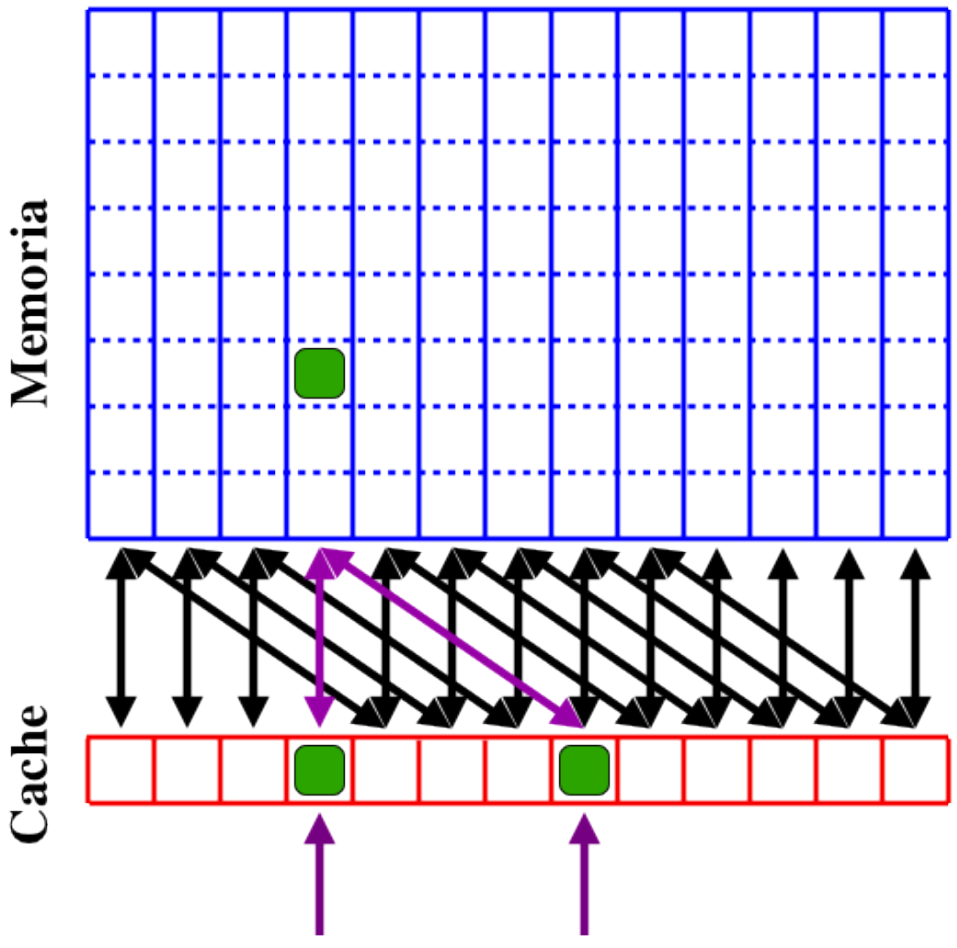


- Pros:
  - easy check of hit (a few bit of address identify the checked line)
  - substitution algorithm is straightforward
  - arbitrarily sized cache
- Cons:
  - strongly dependent on memory access patterns
  - affected by capacity miss
  - affected by cache trashing

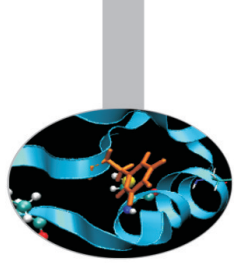
N-way set associative cache



- Each memory block may be mapped to any line among the possible cache lines

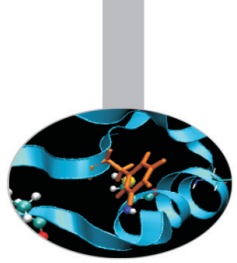


N-way set associative cache



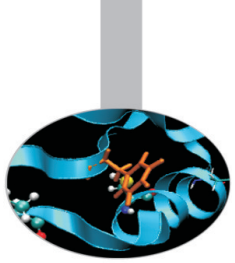
- Pros:
  - is an intermediate choice
    - N=1 $\rightarrow$ direct mapped
    - N= number of cache lines $\rightarrow$ fully associative
  - allows for compromising between circuital complexity and performances (cost and programmability)
  - allows for achieving cache with reasonable sizes
- Cons:
  - strongly conditioned by the memory pattern access
  - partially affected by capacity miss
  - partially affected by cache trashing

Cache: typical situation



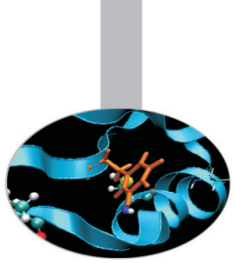
- Cache L1: 4÷8 way set associative
- Cache L2÷3: 2÷4 way set associative o direct mapped
- Capacity miss and trashing must be considered
  - strategies are the same
  - optimization of placement of data in memory
  - optimization of pattern of memory accesses
- L1 cache works with virtual addresses
  - programmer has the full control
- L2÷3 caches work with physical addresses
  - performances depend on physical allocated memory
  - performances may vary when repeating the execution
  - control at operating system level

Cache Trashing



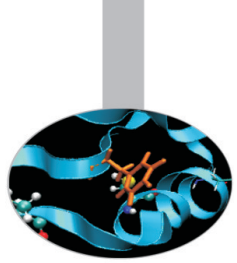
- Problems when accessing data in memory
- A cache line is replaced even if its content is needed after a short time
- It occurs when two or more data flows need a same small subset of cache lines
- The number of load and store is unchanged
- Transaction on memory bus gets increased
- A typical case is given by flows requiring data with relative strides of 2 power

No trashing: C(i)= A(i)+B(i)



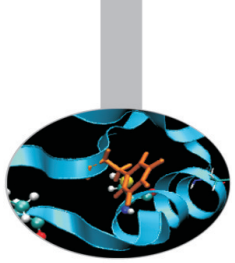
- Iteration i=1
  1. Search for A(1) in L1 cache → cache miss
  2. Get A(1) from RAM memory
  3. Copy from A(1) to A(8) into L1
  4. Copy A(1) into a register un registro
  5. Search for B(1) in L1 cache → cache miss
  6. Get B(1) from RAM memory
  7. Copy from B(1) to B(8) in L1
  8. Copy B(1) into a register
  9. Execute summation
- Iteration i=2
  1. Search for A(2) into L1 cache → cache hit
  2. Copy A(2) into a register
  3. Search for B(2) in L1 cache → cache hit
  4. Copy B(2) into a register
  5. Execute summation
- Iteration i=3

Trashing: C(i)= A(i)+B(i)



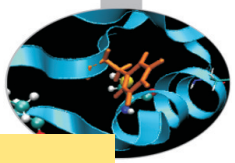
- Iteration i=1
  1. Search for A(1) in the L1 cache $\rightarrow$ cache miss
  2. Get A(1) from RAM memory
  3. Copy from A(1) to A(8) into L1
  4. Copy A(1) into a register
  5. Search for B(1) in L1 cache $\rightarrow$ cache miss
  6. Get B(1) from RAM memory
  7. Throw away cache line A(1)-A(8)
  8. Copy from B(1) to B(8) into L1
  9. Copy B(1) into a register
  10. Execute summation

Trashing: C(i)= A(i)+B(i)



- Iteration i=2
  1. Search for A(2) in the L1 cache → cache miss
  2. Get A(2) from RAM memory
  3. Throw away cache line B(1)-B(8)
  4. Copy from A(1) to A(8) into L1 cache
  5. Copy A(2) into a register
  6. Search for B(2) in L1 cache → cache miss
  7. Get B(2) from RAM memory
  8. Throw away cache line A(1)-A(8)
  9. Copy from B(1) to B(8) into L1
  10. Copy B(2) into a register
  11. Execute summation
- Iteration i=3

How to identify it?



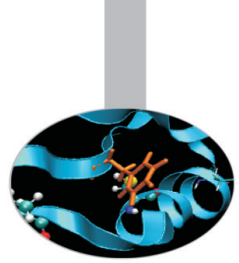
- Effects depending on the size of data set

```
...
integer,parameter  :: offset=..
integer,parameter  :: N1=6400
integer,parameter  :: N=N1+offset
....
real(8)     :: x(N,N),y(N,N),z(N,N)
...
do j=1,N1
    do i=1,N1
      z(i,j)=x(i,j)+y(i,j)
    end do
end do
...
```

offset	time
0	0.361
3	0.250
400	0.252
403	0.253

Solution is padding

Cache padding



a(1)		b(1)	
1		1025	

↑

1 mod 1024=1 1025 mod 1024= 1

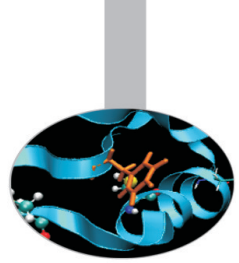
In the cache:

a(1)		1024

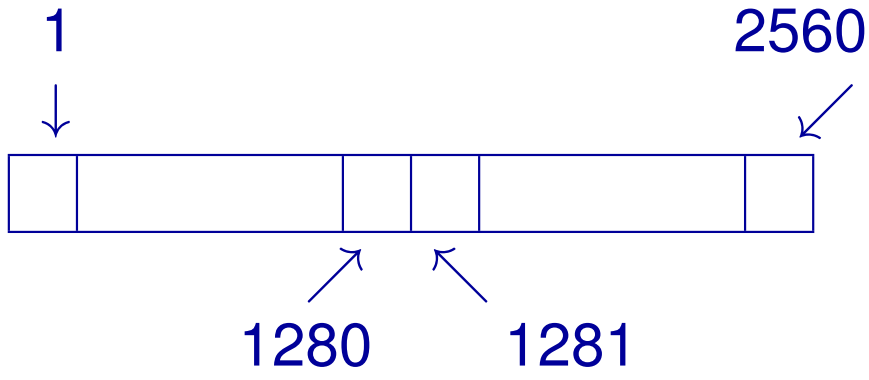
trashing

b(1)		1024

Cache padding



```
integer offset=
(cache line size)/SIZE(REAL)
real, dimension=
(1024+offset)     :: a,b
common/my_comm /a,b
do i=1, 1024
a(i)=b(i) + 1.0
enddo
```



1 mod 1024 =1

1281 mod 1024 = 257

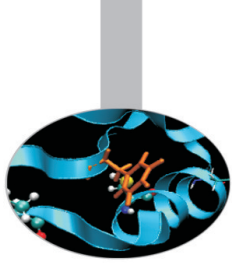
offset→ staggered matrixes

cache → no more problems

- Array padding : when two or more arrays share cache lines, then padding the array with additional leading elements may help to move the start of the array access up

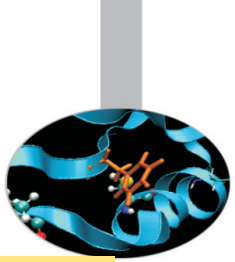
Don't use matrix dimension that are powers of two

Misaligned accesses



- Bus transactions get doubled
- On some architectures:
  - may cause run-time errors
  - emulated in software
- A problem when dealing with
  - structured types (TYPE and struct)
  - local variables
  - “common”
- Solutions
  - order variables with decreasing order
  - compiler options (if available. . .)
  - different common
  - insert dummy variables into common

Misaligned Accesses



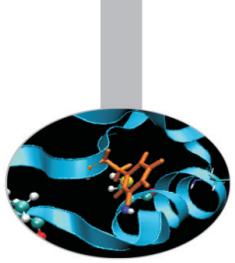
```
      parameter (nd=1000000)
      real*8 a(1:nd), b(1:nd)
      integer c
      common /data1/ a,c,b
....
      do j = 1, 300
         do i = 1, nd
            somma1 = somma1 + (a(i)-b(i))
         enddo
      enddo
```

Different performances for:

```
      common /data1/ a,c,b
      common /data1/ b,c,a
      common /data1/ a,b,c
```

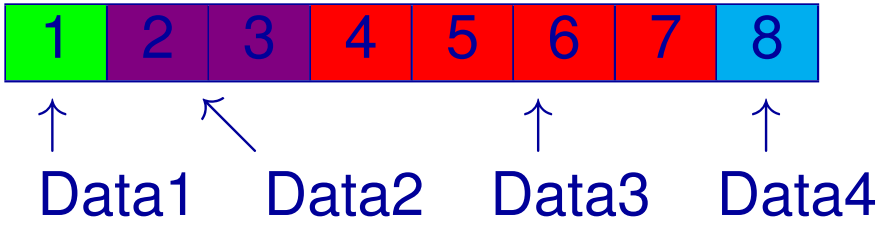
It depends on the architecure and on the compiler which usually warns and tries to fix the problem (align common)

Memory alignment



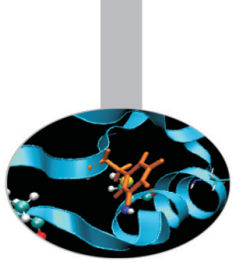
In order to optmize cache using memory alignment is important. When we read memory data in word 4 bytes chunk at time (32 bit systems) The memory addresses must be powers of 4 to be aligned in memory.

```
struct MixedData{
char Data1;
short Data2;
int Data3
char Data4
                }
```



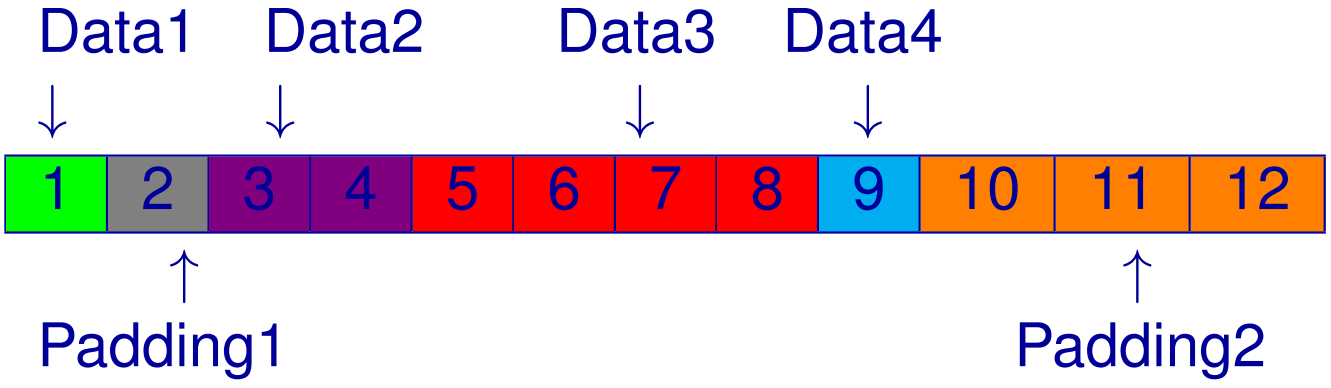
To have Data3 value two reading from memory need.

Memory alignment

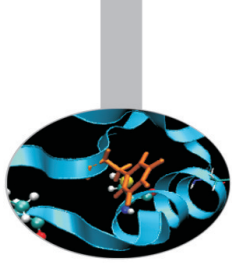


With alignment:

```
struct MixedData{
char Data1;
char Padding1[1];
short Data2;
int Data3
char Data4
char Padding2[3];
                 }
```

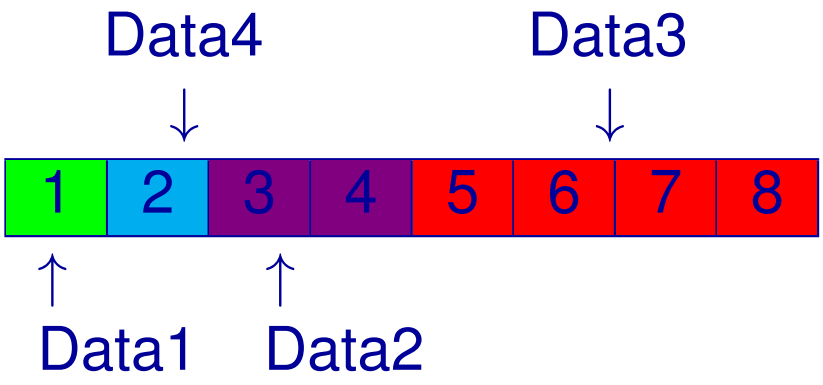


Memory alignment

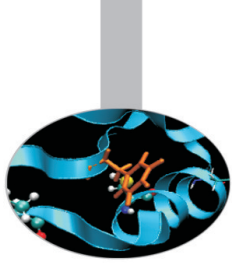


Old struct costs 8 bytes, new struct (with padding) costs 12 bytes.
We can align data exchanging their order.

```
struct MixedData{
char Data1;
char Data4
short Data2;
int Data3
                }
```

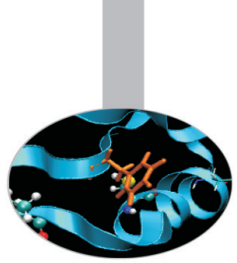


How to detect the problem?



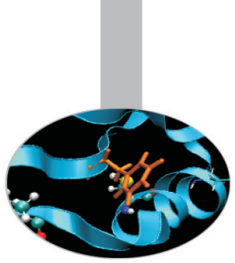
- Processors have hardware counters
- Devised for high clock CPUs
  - necessary to debug processors
  - useful to measure performances
  - crucial to ascertain unexpected behaviors
- Each architecture measures different events
- Of course, vendor dependent
  - IBM: HPCT
  - INTEL: Vtune
- Multi-platform measuring tools exist
  - Valgrind, Oprofile
  - PAPI
  - Likwid
  - . . .

Cache is a memory



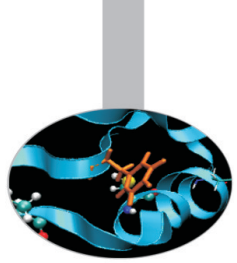
- Its state is persistent until a cache-miss requires a change
- Its state is hidden for the programmer:
  - does not affect code semantics (i.e., the results)
  - affects the performances
- The same routine called under different code sections may show completely different performances because of the cache state at the moment
- Code modularity tends to make the programmer forget it
- It may be important to study the issue in a context larger than the single routine

Valgrind



- Software Open Source useful for Debugging/Profiling of programs running under Linux OS, sources not required (black-box analysis), and different tools available:
  - Memcheck (detect memory leaks, . . .)
  - Cachegrind (cache profiler)
  - Callgrind (callgraph)
  - Massif (heap profiler)
  - Etc.
- http://valgrind.org

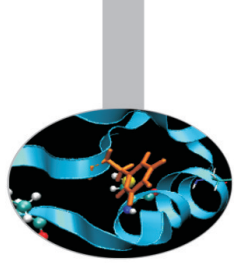
Cachegrind



```
valgrind --tool=cachegrind <nome_eseguibile>
```

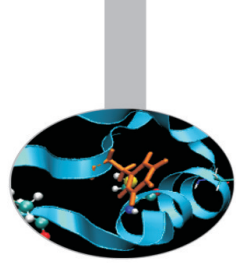
- Simulation of program-cache hierarchy interaction
  - two independent first level cache (L1)
    - instruction (I1)
    - data cache (D1)
  - a last level cache, L2 or L3(LL)
- Provides statistics
  - I cache reads (Ir executed instructions), I1 cache read misses (I1mr), LL cache instruction read misses (ILmr)
  - D cache reads, Dr,D1mr,DLlmr
  - D cache writes, Dw,D1mw,DLlmw
- Optionally provides branches and mispredicted branches

Cachegrind:example I



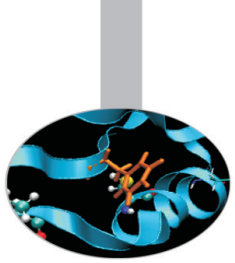
```
==14924== I   refs:      7,562,066,817
==14924== I1  misses:            2,288
==14924== LLi misses:            1,913
==14924== I1  miss rate:          0.00%
==14924== LLi miss rate:          0.00%
==14924==
==14924== D   refs:      2,027,086,734  (1,752,826,448 rd   + 274,260,286 wr)
==14924== D1  misses:       16,946,127  (   16,846,652 rd   +      99,475 wr)
==14924== LLd misses:          101,362  (        2,116 rd   +      99,246 wr)
==14924== D1  miss rate:           0.8% (          0.9%     +         0.0%  )
==14924== LLd miss rate:           0.0% (          0.0%     +         0.0%  )
==14924==
==14924== LL refs:          16,948,415  (   16,848,940 rd   +      99,475 wr)
==14924== LL misses:           103,275  (        4,029 rd   +      99,246 wr)
==14924== LL miss rate:            0.0% (          0.0%     +         0.0%  )
```

Cachegrind:example II



```
==15572== I   refs:      7,562,066,871
==15572== I1  misses:            2,288
==15572== LLi misses:            1,913
==15572== I1  miss rate:          0.00%
==15572== LLi miss rate:          0.00%
==15572==
==15572== D   refs:      2,027,086,744  (1,752,826,453 rd   + 274,260,291 wr)
==15572== D1  misses:      151,360,463  (  151,260,988 rd   +      99,475 wr)
==15572== LLd misses:          101,362  (        2,116 rd   +      99,246 wr)
==15572== D1  miss rate:           7.4% (          8.6%     +         0.0%  )
==15572== LLd miss rate:           0.0% (          0.0%     +         0.0%  )
==15572==
==15572== LL refs:         151,362,751  (  151,263,276 rd   +      99,475 wr)
==15572== LL misses:           103,275  (        4,029 rd   +      99,246 wr)
==15572== LL miss rate:            0.0% (          0.0%     +         0.0%  )
```

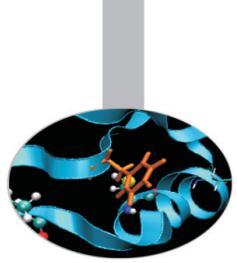
Cachegrind:cg_annotate



- Cachegrind automatically produces the file cachegrind.out.<pid>
- In addition to the previous information, more detailed statistics for each function is made available

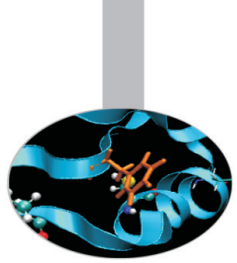
```
cg_annotate cachegrind.out.<pid>
```

Cachegrind:options



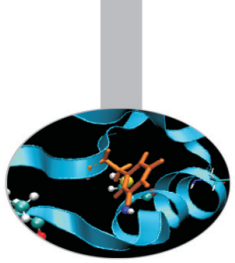
- —I1=<size>,<associativity>,<line size>
- —D1=<size>,<associativity>,<line size>
- —LL=<size>,<associativity>,<line size>
- —cache-sim=no|yes [yes]
- —branch-sim=no|yes [no]
- —cachegrind-out-file=<file>

Exercises: summary



- Matrix multiplication: loop order
- Matrix multiplication: blocking
- Matrix multiplication:blocking and padding
- Measuring cache performances

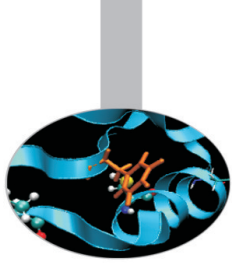
Order of the loops



- *eser*_2 (fortran/mm.f90 or c/mm.c)
- Set N=512, measure the performances changing the order of the loops
- Use fortran and/or c codes
- Use the different compilers without optmizations (-O0)

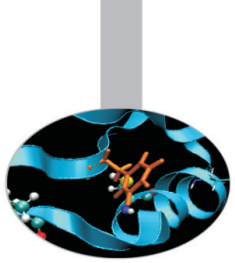
Indici	Tempi	C
i,j,k		
i,k,j		
j,k,i		
j,i,k		
k,i,j		
k,j,i		

src/eser_3/fortran/mm.f90



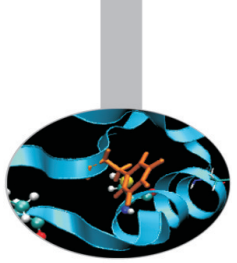
```
1   ...
2       integer, parameter :: n=1024  ! size of the matrix
3       integer, parameter :: step=4
4       integer, parameter :: npad=0
5   ...
6       real(my_kind) a(1:n+npad,1:n) ! matrix
7       real(my_kind) b(1:n+npad,1:n) ! matrix
8       real(my_kind) c(1:n+npad,1:n) ! matrix (destination)
9
10          do jj = 1, n, step
11            do kk = 1, n, step
12               do ii = 1, n, step
13                  do j = jj, jj+step-1
14                     do k = kk, kk+step-1
15                        do i = ii, ii+step-1
16                           c(i,j) = c(i,j) + a(i,k)*b(k,j)
17                        enddo
18  ...
```

src/eser_3/c/mm.c



```
1  #define nn (1024)
2  #define step (4)
3  #define npad (0)
4
5  double a[nn+npad][nn+npad];        /** matrici**/
6  double b[nn+npad][nn+npad];
7  double c[nn+npad][nn+npad];
8  ...
9   for (ii = 0; ii < nn; ii= ii+step)
10    for (kk = 0; kk < nn; kk = kk+step)
11       for (jj = 0; jj < nn; jj = jj+step)
12          for ( i = ii; i < ii+step; i++ )
13             for ( k = kk; k < kk+step; k++ )
14                for ( j = jj; j < jj+step; j++ )
15                   c[i][j] = c[i][j] + a[i][k]*b[k][j];
16 ...
```

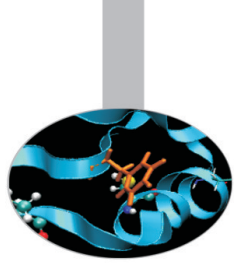
Blocking



- *eser*_3 (fortran/mm.f90 or c/mm.c)
- Set N=1024 and measure the performances changing the value of the step variable.
- Use Fortran and/or c codes
- Use the different compilers with optimization -O3

Step	Fortran	C
4		
8		
16		
32		
64		
128		
256		

Blocking and Padding



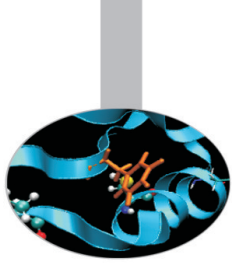
- *eser*_3 (fortran/mm.f90 or c/mm.c)
- Set N=1024 and npad=9 and measure the performances changing the value of step variable.
- Use Fortran and/or c codes
- Use the different compilers with optimization -O3

Step	Fortran	C
4		
8		
16		
32		
64		
128		
256		

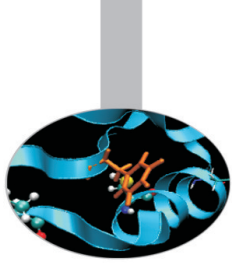
The performances of the cache

- valgrind
  - Use the cachegrind tool of valgrind to evaluate the performances of the cache memory changing the order of the loops.

Outline

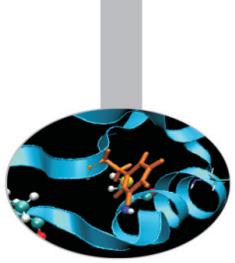


CPU: internal parallelism?



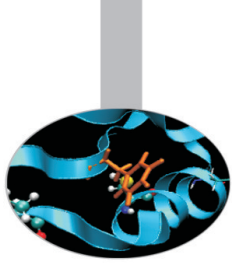
- CPU are entirely parallel
  - pipelining
  - superscalar execution
  - units SIMD MMX, SSE, SSE2, SSE3, SSE4, AVX
- To achieve performances comparable to the peak performance:
  - give a large amount of instructions
  - give the operands of the instructions

Pipelining:Definitions



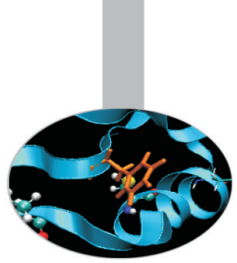
- Pipelining is an implementation technique where multiple operations on a number of instructions are overlapped in execution.
- An instruction execution pipeline involves a number of steps, where each step completes a part of an instruction.
- Each step is called a pipe stage or a pipe segment.
- The stages or steps are connected one to the next to form a pipe – instructions enter at one end and progress through the stage and exit at the other end.
- Throughput of an instruction pipeline is determined by how often an instruction exists the pipeline.
- The time to move an instruction one step down the line is is equal to the machine cycle and is determined by the stage with the longest processing delay

Latency and Throughput



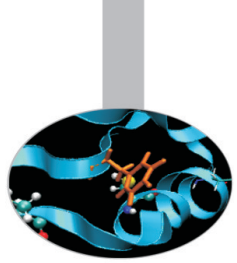
- Latency : the number of clocks to complete an instruction when all of its inputs are ready
- Throughput : the number of clocks to wait before starting an identical instruction
  - Identical instructions are those that use the same execution unit

Basic Performance Issues In Pipelining



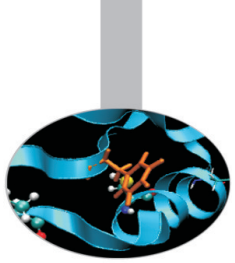
- Pipelining increases the CPU instruction throughput: The number of instructions completed per unit time. Under ideal condition instruction throughput is one instruction per machine cycle, or CPI = 1
- Pipelining does not reduce the execution time of an individual instruction: The time needed to complete all processing steps of an instruction (also called instruction completion latency).
- It usually slightly increases the execution time of each instruction over unpipelined implementations due to the increased control overhead of the pipeline and pipeline stage registers delays.

The pipeline



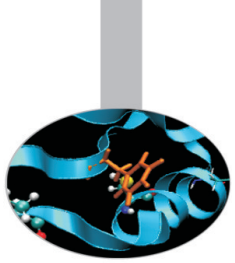
- Pipeline, channel or tube for carrying oil
- An operation is split in independent stages and different stages are executed simultaneously
  - fetch (get, catch) gets the instruction from memory and the pointer of Program Counter is increased to point to the next instruction
  - decode instruction gets interpreted
  - execute send messages which represent commands for execution
- Parallelism with different operation stages
- Processors significantly exploit pipelining to increase the computing rate

CPU cycle

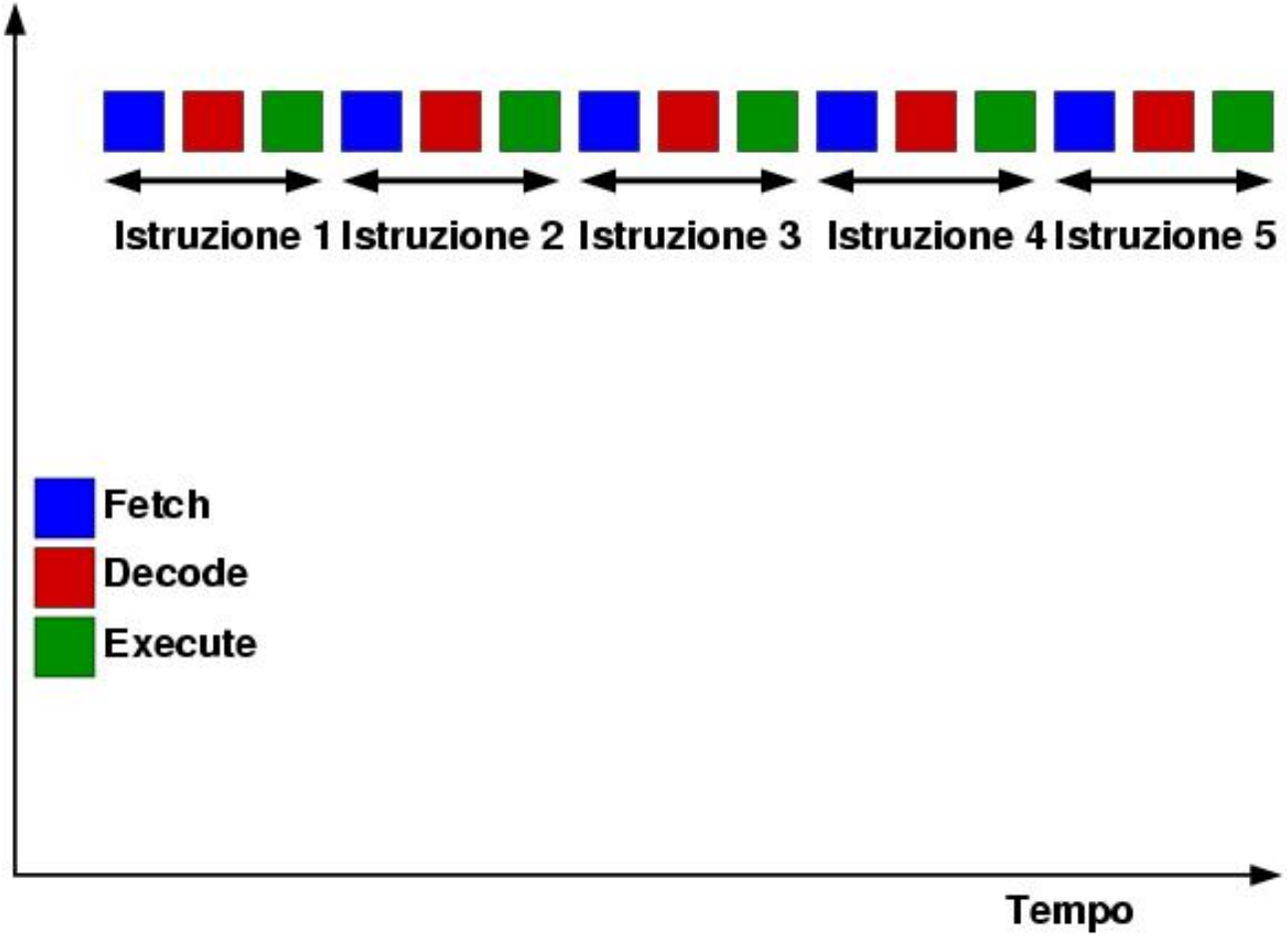


- The time to move an instruction one step through the pipeline is called a machine cycle
- CPI (clock Cycles Per Instruction)
  - the number of clock cycles needed to execute an instruction
  - varies for different instructions
  - its inverse is IPC (Instructions Per Cycle)

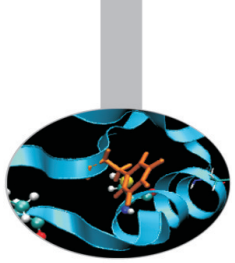
NON pipelined computing units



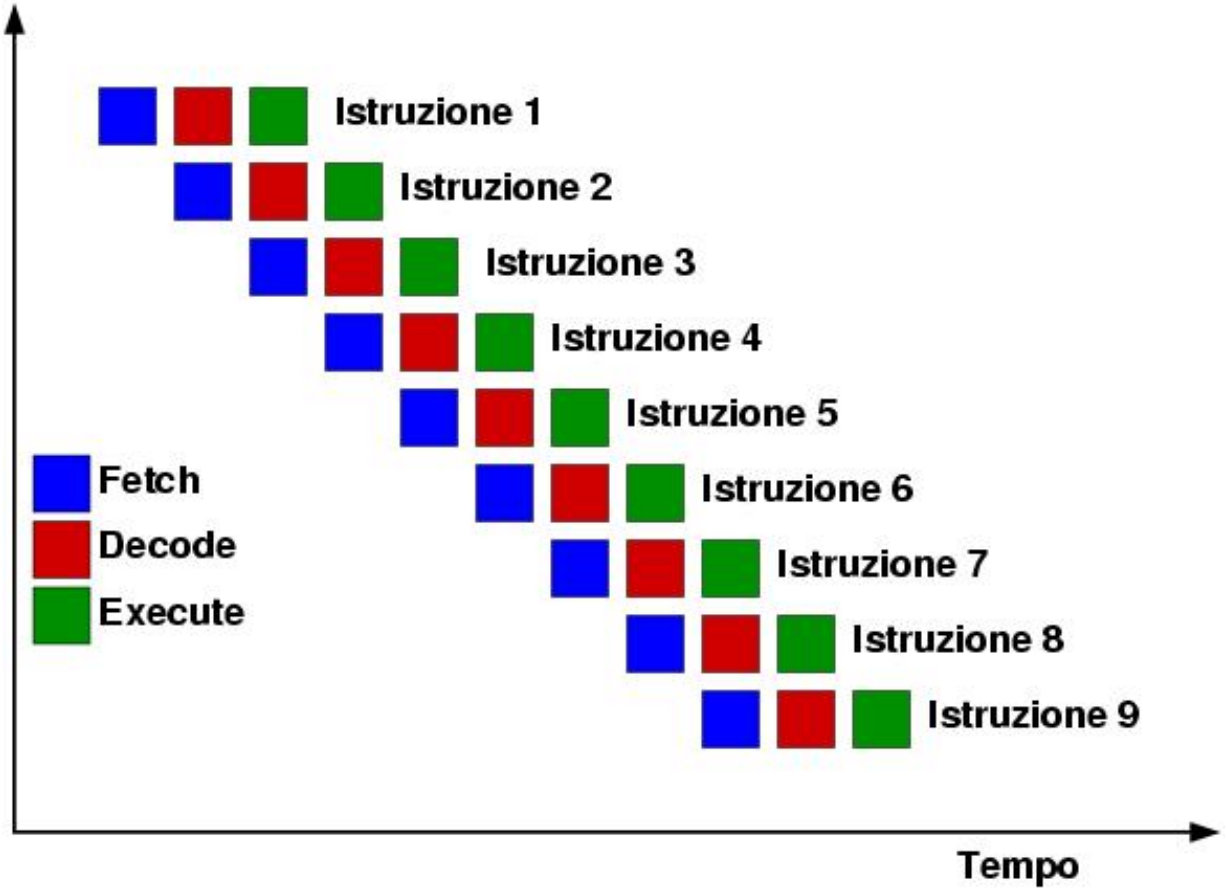
- Each instruction is completed after three cycles



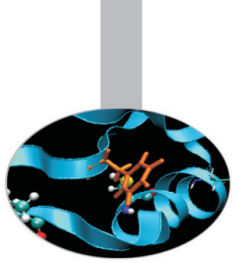
Pipelined units



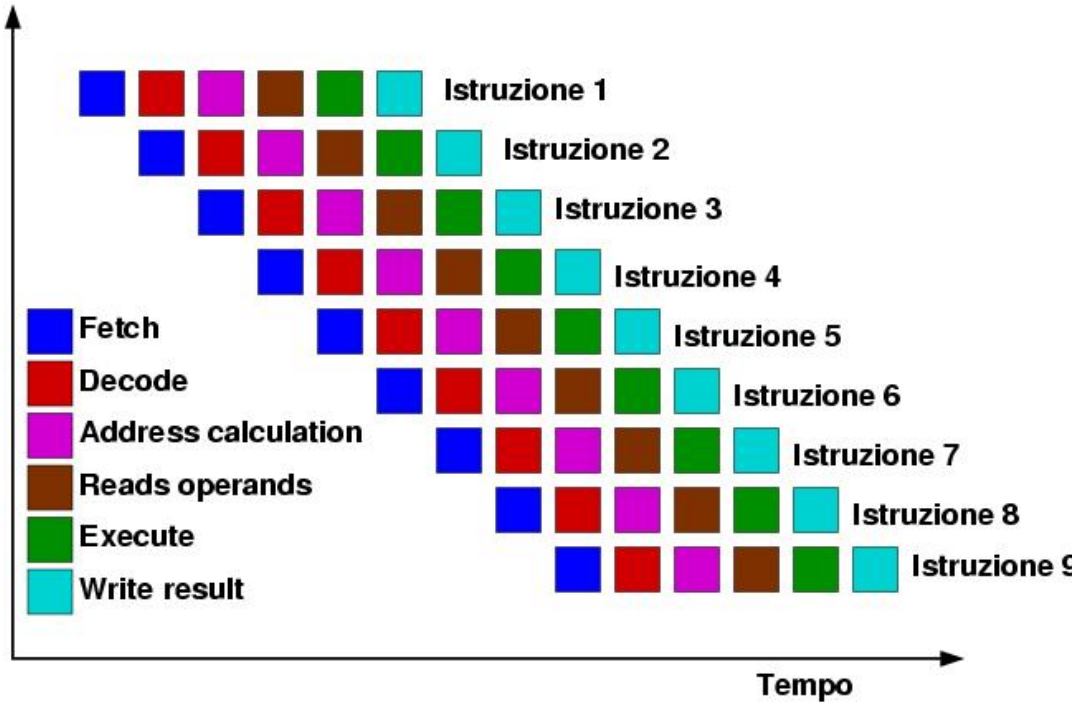
- After 3 clock cycles, the pipeline is full
- A result per cycle when the pipeline is completely filled
- To fill it 3 independent instructions are needed (including the operands)



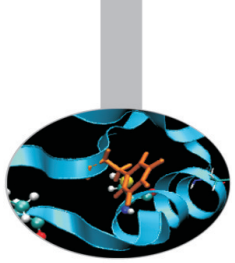
Superpipelined computing units



- After 6 clock cycles, the pipeline is full
- A result per cycle when the pipeline is completely filled
- To fill it 6 independent instructions are needed (including the operands)
- It is possible to halve the clock rate, i.e. doubling the frequency

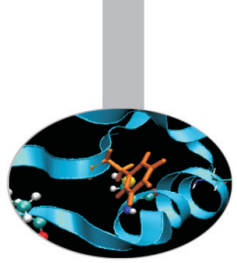


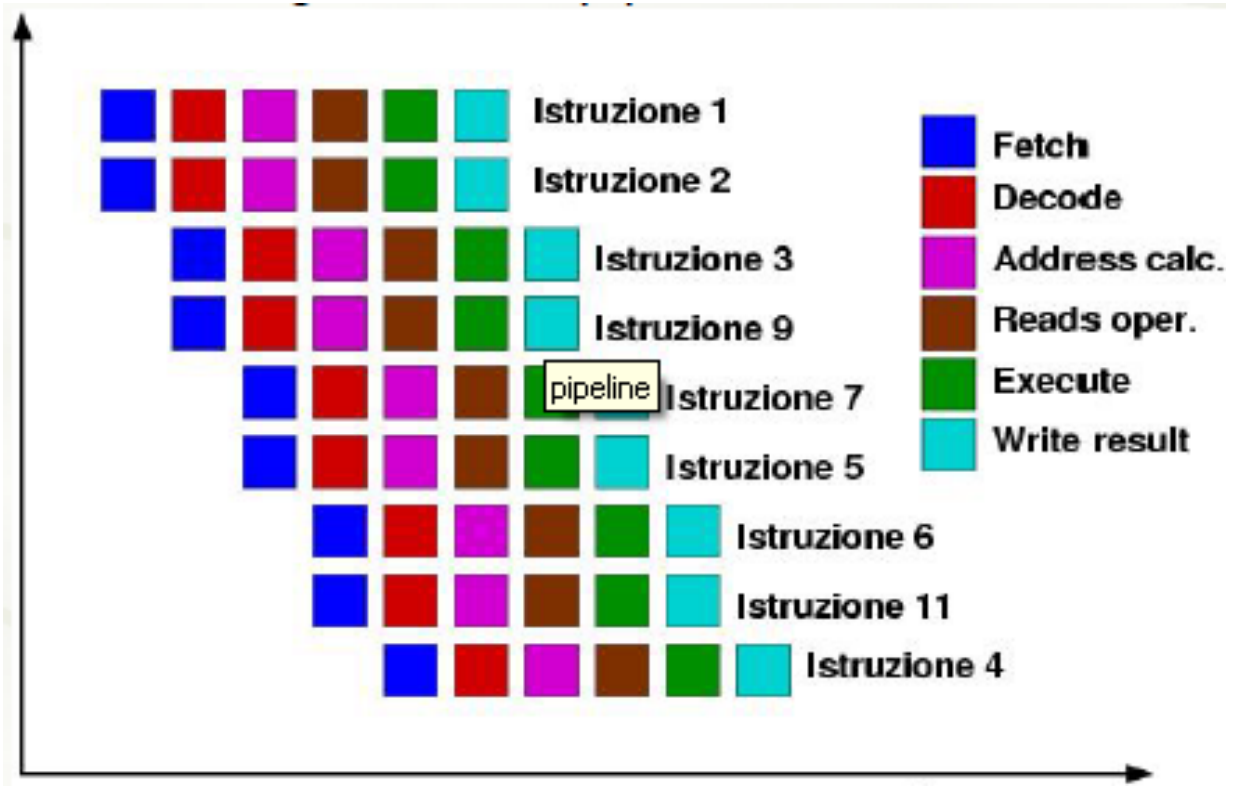
Out of order execution



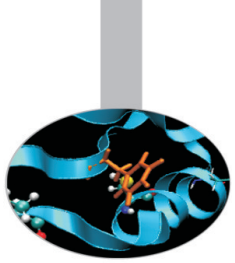
- Dynamically reorder the instructions
  - move up instructions having operands which are available
  - postpone instructions having operands still not available
  - reorder reads/write from/into memory
  - always considering the free functional units
- Exploit significantly:
  - register renaming (physical vs architectural registers)
  - branch prediction
  - combination of multiple read and write from/to memory
- Crucial to get high performance on present CPUs
- The code should not hide the reordering possibilities

Out of order execution



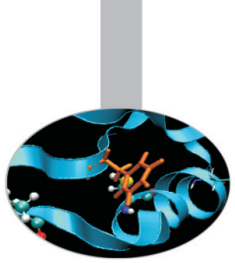


Superscalar execution



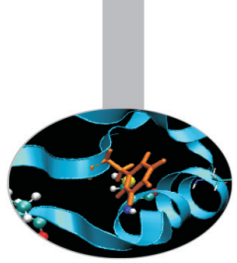
- CPUs have different independent units
  - functional differentiation
  - functional replication
- Independent operations are executed at the same time
  - integer operations
  - floating point operations
  - skipping memory
  - memory accesses
- Instruction Parallelism
- Hiding latencies
- Processors exploit superscalarity to increase the computing power for a fixed clock rate

How to exploit internal parallelism?



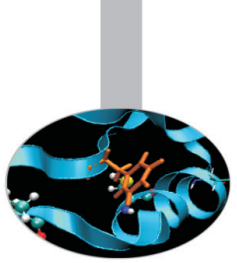
- Processors run at maximum speed (high instruction per cycle rate (IPC)) when
  1. There is a good mix of instructions (with low latencies) to keep the functional units busy
  2. Operands are available quickly from registers or D-cache
  3. The FP to memory operation ratio is high (FP : MEM > 1)
  4. Number of data dependences is low
  5. Branches are easy to predict
- The processor can only improve #1 to a certain level with out-of-order scheduling and partly #2 with hardware prefetching
- Compiler optimizations effectively target #1-3
- The programmer can help improve #1-5

Strategies



- loop unrolling $\rightarrow$ unroll the loop
- loop merging $\rightarrow$ merge loops into a single loop
- loop splitting $\rightarrow$ decompose complex loops
- function inlining $\rightarrow$ avoid breaking instruction flow

Loop unrolling

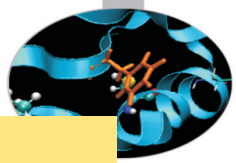


- Repeat the body of a loop k times and go through the loop with a step length k
- k is called the unrolling factor

```
do j = 1, nj                    -> do j = 1, nj
do i = 1, ni                    -> do i = 1, ni, 2
 a(i,j)=a(i,j)+c*b(i,j)         ->  a(i  ,j)=a(i  ,j)+c*b(i  ,j)
                                ->  a(i+1,j)=a(i+1,j)+c*b(i+1,j)
```

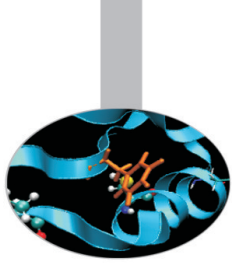
  - The unrolled version of the loop has increased code size, but in turn, will execute fewer overhead instructions.
  - The same number of operations, but the loop index is incremented half of the times
  - The performance of this loop depends upon both the trace cache and L1 cache state.
  - In general the unrolled version runs faster because fewer overhead instrunctions are executed.
  - It is not valid when data dependences exist.

Reduction & Unroll



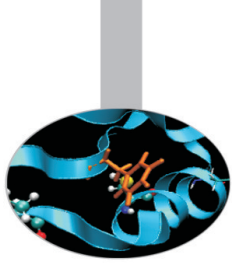
```
    do j = i, nj ! normal case 1)
       do i = i, ni
          somma = somma + a(i,j)
       end do
    end do
......
    do j = i, nj !reduction to 4 elements.. 2)
       do i = i, ni, 4
          somma_1 = somma_1 + a(i+0,j)
          somma_2 = somma_2 + a(i+1,j)
          somma_3 = somma_3 + a(i+2,j)
          somma_4 = somma_4 + a(i+3,j)
       end do
    end do
    somma = somma_1 + somma_2 + somma_3 + somma_4
f77 -native -O2 (-O4)
time 1) ---> 4.49785 (2.94240)
time 2) ---> 3.54803 (2.75964)
```

What inhibits loop unrolling?



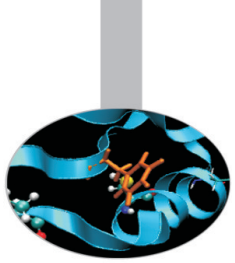
- Conditional jumps (`if` ...)
- Calls to intrinsic functions and library (sin, exp, ...)
- I/0 operations in the loop

Compiler options



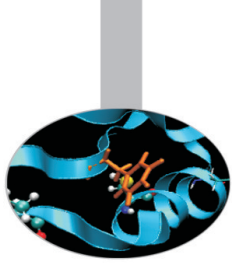
- Can I know how compiler works?

Compiler options



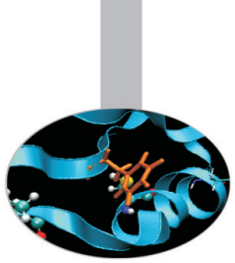
- Can I know how compiler works?
- See reference documentation for the compiler.

Compiler options



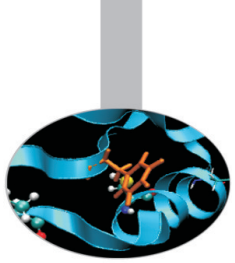
- Can I know how compiler works?
- See reference documentation for the compiler.
- Use, for example, the intel compiler with flag **-qopt-report**.

Esercises:summary



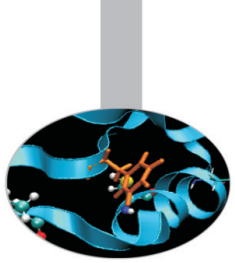
- Matrix multiplication:unrolling
- Matrix multiplicatin:unrolling and padding

src/eser_4/fortran/mm.f90



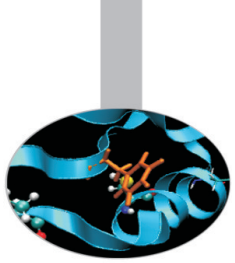
```
1   ...
2       integer, parameter :: n=1024  ! size of the matrix
3       integer, parameter :: step=4
4       integer, parameter :: npad=0
5   ...
6       real(my_kind) a(1:n+npad,1:n) ! matrix
7       real(my_kind) b(1:n+npad,1:n) ! matrix
8       real(my_kind) c(1:n+npad,1:n) ! matrix (destination)
9
10       do j = 1, n, 2
11            do k = 1, n
12               do i = 1, n
13                  c(i,j+0) = c(i,j+0) + a(i,k)*b(k,j+0)
14                  c(i,j+1) = c(i,j+1) + a(i,k)*b(k,j+1)
15               enddo
16            enddo
17       enddo
18  ...
```

src/eser_4/c/mm.c



```
1  #define nn (1024)
2  #define step (4)
3  #define npad (0)
4
5  double a[nn+npad][nn+npad];        /** matrici**/
6  double b[nn+npad][nn+npad];
7  double c[nn+npad][nn+npad];
8  ...
9  for (i = 0; i < nn; i+=2)
10    for (k = 0; k < nn; k++)
11       for (j = 0; j < nn; j++) {
12           c[i+0][j] = c[i+0][j] + a[i+0][k]*b[k][j];
13           c[i+1][j] = c[i+1][j] + a[i+1][k]*b[k][j];
14 }
15 ...
```

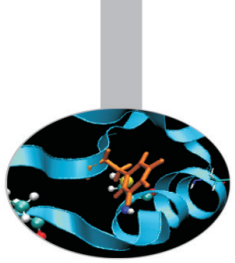
Unrolling the external loop



- *eser*_4 (fortran/mm.f90 or c/mm.c)
- Set N=1024 and measure the performances changing the size of the unrolling of the external loop.
- Use Fortran and/or c codes
- Use the different compilers with optimization -O3

Unrolling	Fortran	C
2		
4		
8		
16		

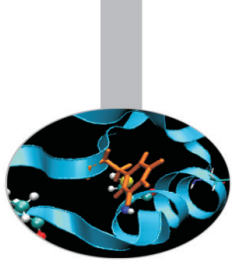
Unrolling and padding of the external loop



- *eser*_4 (fortran/mm.f90 or c/mm.c)
- Set N=1024 and npad=9 and measure the performances exchanging the size of the unrolling of the external loop.
- Use Fortran and/or c codes
- Use gnu complier with optimization -O3

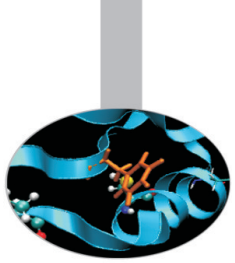
Unrolling	Fortran	C
2		
4		
8		
16		

Last effort ...

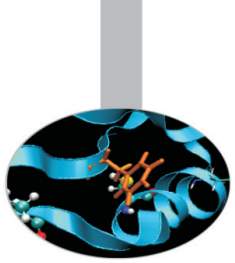


- What is the best performance achieved using:
  - blocking
  - unrolling of the external loop
  - padding
  - . . . other optimizations..
- with N=2048?

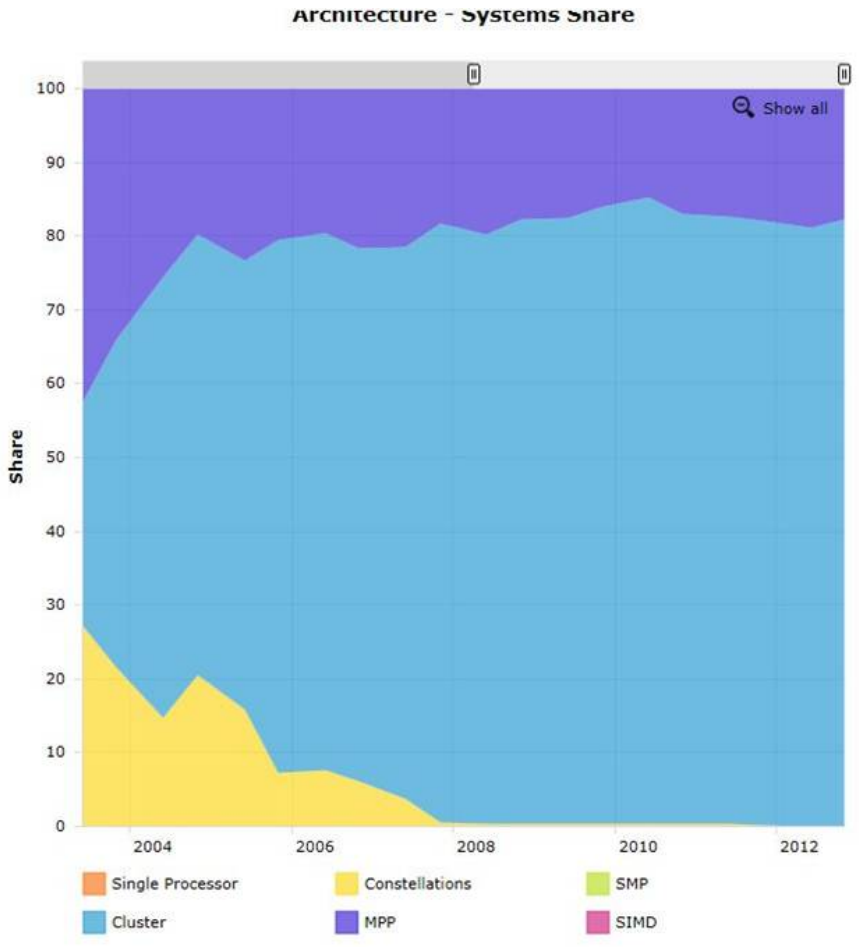
Outline



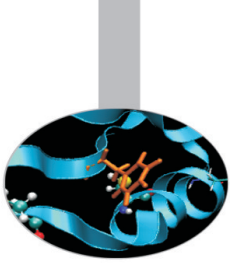
Outline

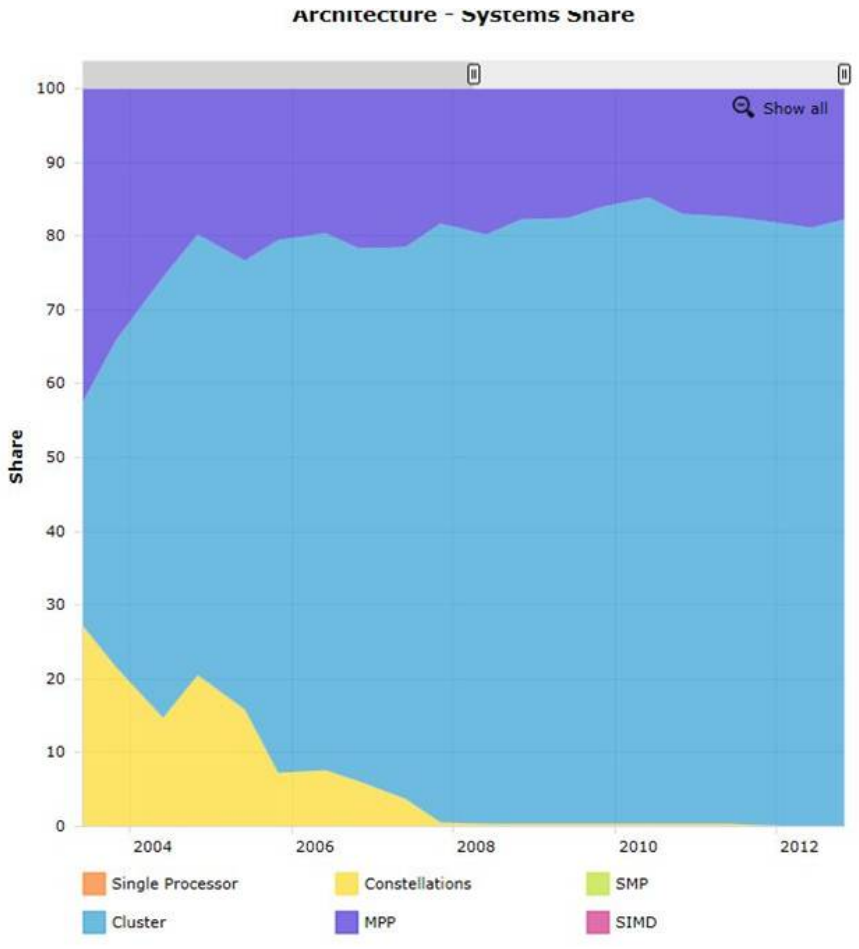


Architectural trend (Top500 list)



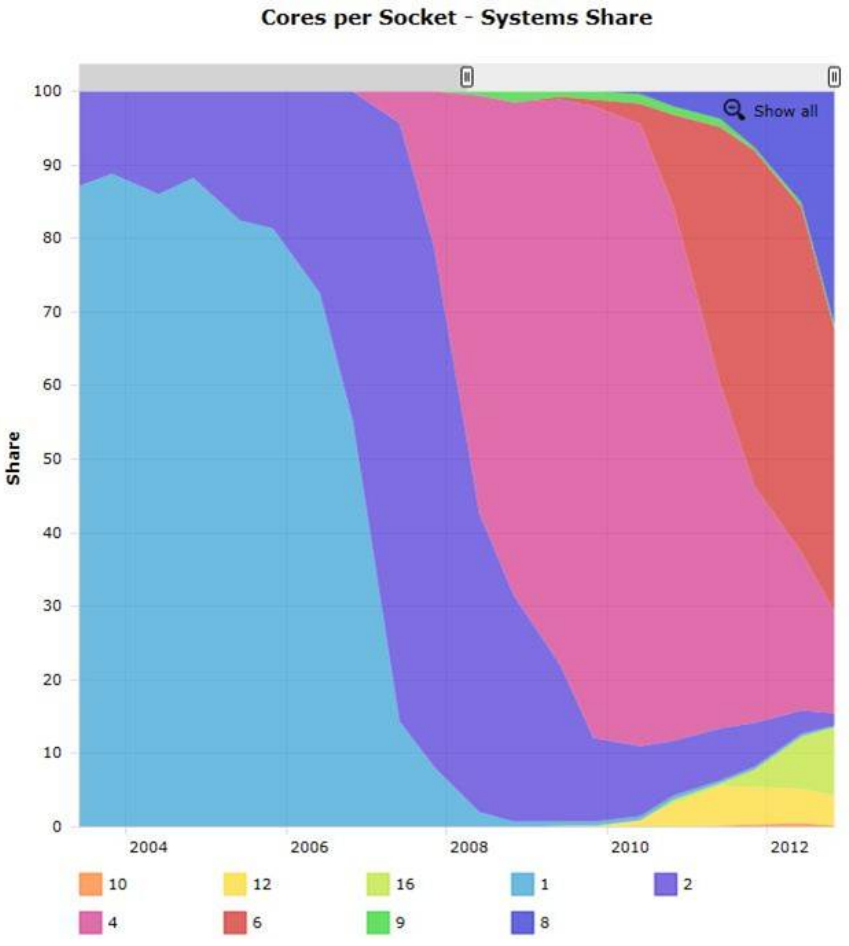
Architectural trend (Top500 list)



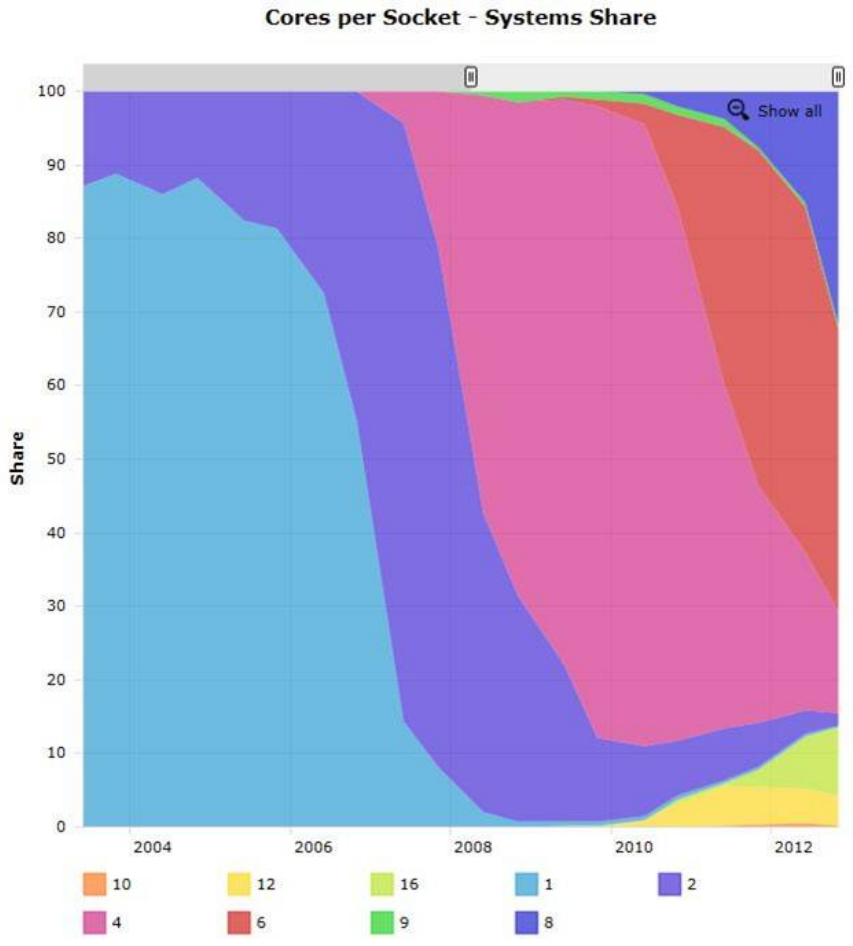


clusters dominates High Performance Computing marketplace

microprocessors trend

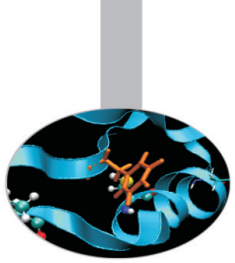


microprocessors trend



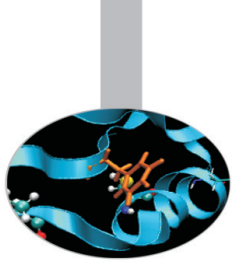
organizing the logic: "multicore" per "socket" chips

Why performance monitoring is important?



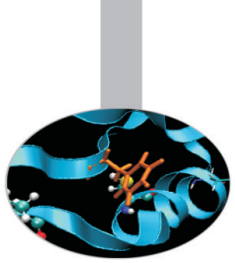
- Increasing number of parallel and "hybrid" architectures with:

Why performance monitoring is important?



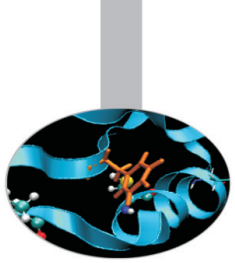
- Increasing number of parallel and "hybrid" architectures with:
  - reduced memory "bandwidth"

Why performance monitoring is important?



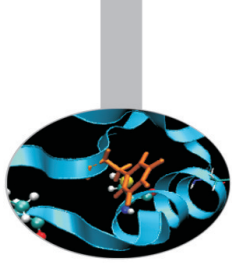
- Increasing number of parallel and "hybrid" architectures with:
  - reduced memory "bandwidth"
  - reduced amount of memory per "core"

Why performance monitoring is important?



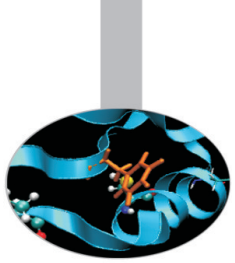
- Increasing number of parallel and "hybrid" architectures with:
  - reduced memory "bandwidth"
  - reduced amount of memory per "core"
  - more complex memory hierarchies

Why performance monitoring is important?



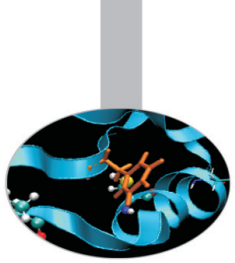
- Increasing number of parallel and "hybrid" architectures with:
  - reduced memory "bandwidth"
  - reduced amount of memory per "core"
  - more complex memory hierarchies
- Programming is hard and requires special skills

Why performance monitoring is important?



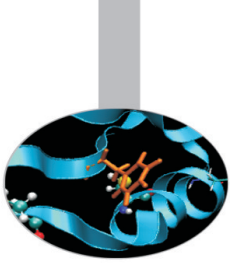
- Increasing number of parallel and "hybrid" architectures with:
  - reduced memory "bandwidth"
  - reduced amount of memory per "core"
  - more complex memory hierarchies
- Programming is hard and requires special skills
- Huge performance improvement is hard"

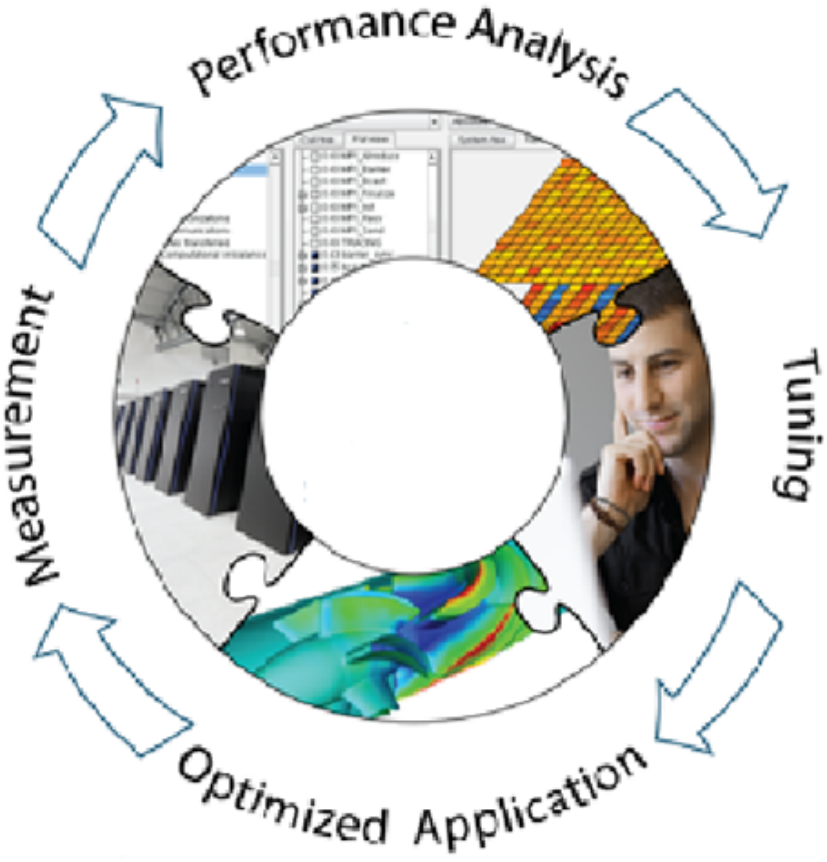
Why performance monitoring is important?



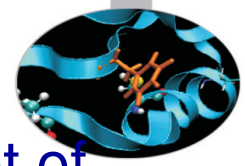
- Increasing number of parallel and "hybrid" architectures with:
  - reduced memory "bandwidth"
  - reduced amount of memory per "core"
  - more complex memory hierarchies
- Programming is hard and requires special skills
- Huge performance improvement is hard"
- Performance analysis tools are became extremely important for understanding program behavior. Computer architects need such tools to evaluate how well programs will perform on new architectures and drive a further optimization, parallelization, re-design....

Performance Lyfe Cycle



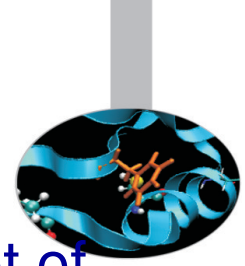


Profiling



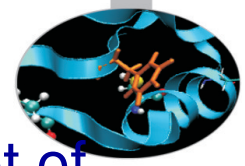
- A standard serial or parallel application is composed of a lot of functions, routines,....
- Code optimization and parallelization is hard. If you want to tune performances is crucial:

Profiling



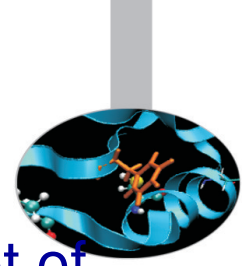
- A standard serial or parallel application is composed of a lot of functions, routines,....
- Code optimization and parallelization is hard. If you want to tune performances is crucial:
  - to see how much time is actually spent in a specific part of the entire application

Profiling



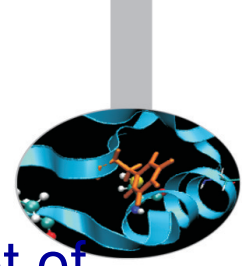
- A standard serial or parallel application is composed of a lot of functions, routines,....
- Code optimization and parallelization is hard. If you want to tune performances is crucial:
  - to see how much time is actually spent in a specific part of the entire application
  - to find the "call-" and "dependency-" graph for the application

Profiling



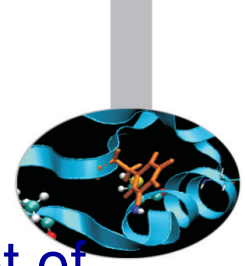
- A standard serial or parallel application is composed of a lot of functions, routines,....
- Code optimization and parallelization is hard. If you want to tune performances is crucial:
  - to see how much time is actually spent in a specific part of the entire application
  - to find the "call-" and "dependency-" graph for the application
  - to find out application "bottlenecks" and "critical paths"

Profiling



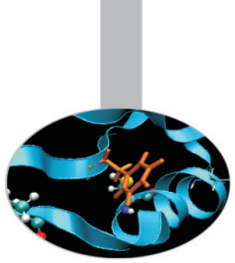
- A standard serial or parallel application is composed of a lot of functions, routines,....
- Code optimization and parallelization is hard. If you want to tune performances is crucial:
  - to see how much time is actually spent in a specific part of the entire application
  - to find the "call-" and "dependency-" graph for the application
  - to find out application "bottlenecks" and "critical paths"
- Depending on software complexity and dimensionality (e.g. number of lines of code) is not so easy to have a clear idea about aforementioned statements.

Profiling



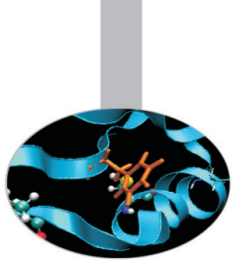
- A standard serial or parallel application is composed of a lot of functions, routines,....
- Code optimization and parallelization is hard. If you want to tune performances is crucial:
  - to see how much time is actually spent in a specific part of the entire application
  - to find the "call-" and "dependency-" graph for the application
  - to find out application "bottlenecks" and "critical paths"
- Depending on software complexity and dimensionality (e.g. number of lines of code) is not so easy to have a clear idea about aforementioned statements.
- The main idea is to start froma simple "Profiling" of our application. What does "Profiling" mean? essentially, it refers to obtaining dynamic information from a controlled program execution.

Profiling



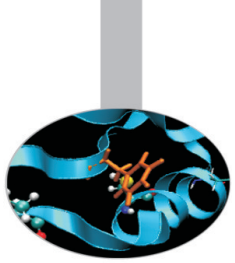
- There are a wide variety of Profiling tools. They can broadly be divided in different groups, depending on:

Profiling



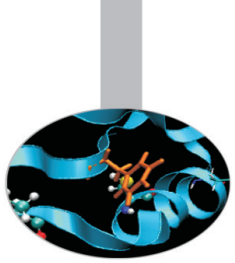
- There are a wide variety of Profiling tools. They can broadly be divided in different groups, depending on:
  - ease of use or not

Profiling



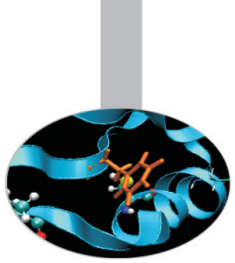
- There are a wide variety of Profiling tools. They can broadly be divided in different groups, depending on:
  - ease of use or not
  - proprietary vs public domain

Profiling



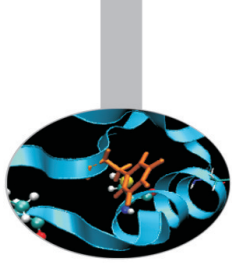
- There are a wide variety of Profiling tools. They can broadly be divided in different groups, depending on:
  - ease of use or not
  - proprietary vs public domain
  - intrusive or not intrusive

Profiling



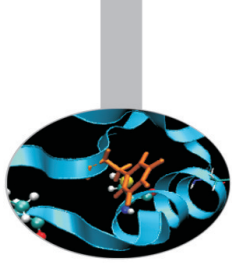
- There are a wide variety of Profiling tools. They can broadly be divided in different groups, depending on:
  - ease of use or not
  - proprietary vs public domain
  - intrusive or not intrusive
  - ....

Profiling

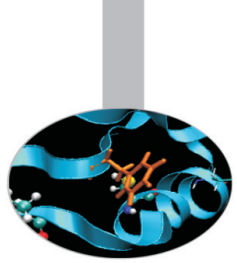


- There are a wide variety of Profiling tools. They can broadly be divided in different groups, depending on:
  - ease of use or not
  - proprietary vs public domain
  - intrusive or not intrusive
  - ....
- let‘s start the tour: from simplest to the most complex tool. The main idea will be to collect all the informations that can be used to increase the performances of our application.

Outline

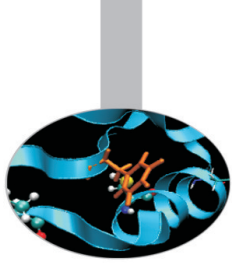


time: main characteristics



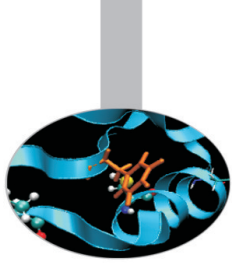
- You can call him from anywhere *Unix /Linux* machine.

time: main characteristics



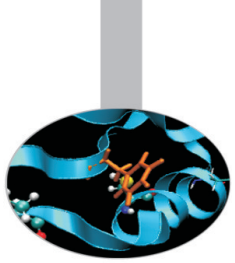
- You can call him from anywhere *Unix /Linux* machine.
- It returns the program total time of execution and other useful informations.

time: main characteristics



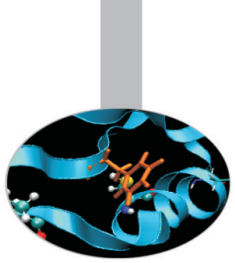
- You can call him from anywhere *Unix /Linux* machine.
- It returns the program total time of execution and other useful informations.
- There is no need to change anything. No compilation overhead, no source code modification. (non intrusive).

time: main characteristics



- You can call him from anywhere *Unix /Linux* machine.
- It returns the program total time of execution and other useful informations.
- There is no need to change anything. No compilation overhead, no source code modification. (non intrusive).
- time <name_executable>

time: main characteristics

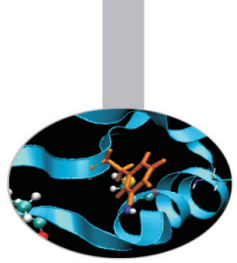


- You can call him from anywhere *Unix /Linux* machine.
- It returns the program total time of execution and other useful informations.
- There is no need to change anything. No compilation overhead, no source code modification. (non intrusive).
- time <name_executable>

Here's a typical output from this command:

```
[planucar@node165 TIMERS]$ /usr/bin/time  ./a.out <realloc.in
 real maxsize (Kbytes)=   750000.00000000000
12.69user 4.76system 0:17.45elapsed 100%CPU (0avgtext+0avgdata 751088maxresident)k
0inputs+0outputs (0major+161115minor)pagefaults 0swaps
```

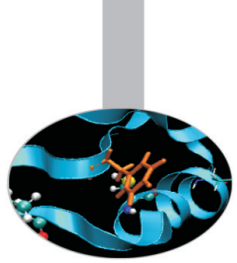
time: output



12.6u

1. (*User time*) CPU time (in seconds) that the program spent to run.

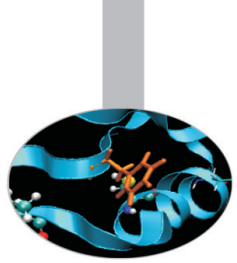
time: output



12.6u 4.76s

1. (*User time*) CPU time (in seconds) that the program spent to run.
2. (*System time*) CPU time (in seconds) that the program/process spent in doing system calls during its execution.

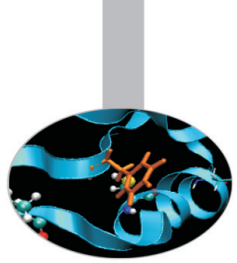
time: output



12.6u 4.76s 0:17.45

1. (*User time*) CPU time (in seconds) that the program spent to run.
2. (*System time*) CPU time (in seconds) that the program/process spent in doing system calls during its execution.
3. (*Elapsed time*) The time (h:m:s) that elapses while the program runs ("elapsed time").

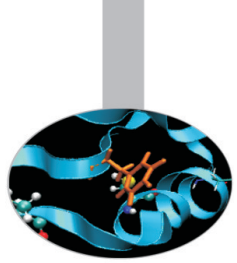
time: output



12.6u 4.76s 0:17.45 100%

1. (*User time*) CPU time (in seconds) that the program spent to run.
2. (*System time*) CPU time (in seconds) that the program/process spent in doing system calls during its execution.
3. (*Elapsed time*) The time (h:m:s) that elapses while the program runs ("elapsed time").
4. The percentage of total CPU used in the process/program.

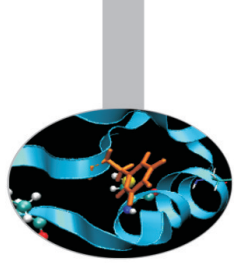
time: output



```
12.6u 4.76s 0:17.45 100% 0avgtext+0avgdata
751088maxresident)k
```

1. (*User time*) CPU time (in seconds) that the program spent to run.
2. (*System time*) CPU time (in seconds) that the program/process spent in doing system calls during its execution.
3. (*Elapsed time*) The time (h:m:s) that elapses while the program runs ("elapsed time").
4. The percentage of total CPU used in the process/program.
5. Parameters related to the set size of the process (in Kbytes).

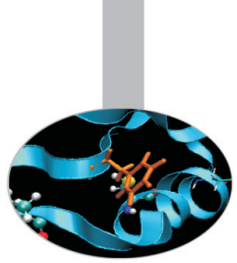
time: output



```
12.6u 4.76s 0:17.45 100% 0avgtext+0avgdata
751088maxresident)k 0inputs+0outputs
```

1. (*User time*) CPU time (in seconds) that the program spent to run.
2. (*System time*) CPU time (in seconds) that the program/process spent in doing system calls during its execution.
3. (*Elapsed time*) The time (h:m:s) that elapses while the program runs ("elapsed time").
4. The percentage of total CPU used in the process/program.
5. Parameters related to the set size of the process (in Kbytes).
6. Input/output parameters (integer value).

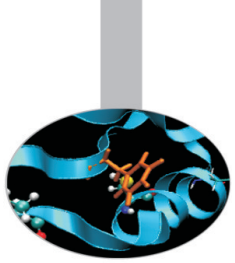
time: output



12.6u 4.76s 0:17.45 100% 0avgtext+0avgdata
751088maxresident)k 0inputs+0outputs 0major+161115minor

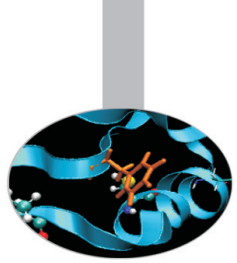
1. (*User time*) CPU time (in seconds) that the program spent to run.
2. (*System time*) CPU time (in seconds) that the program/process spent in doing system calls during its execution.
3. (*Elapsed time*) The time (h:m:s) that elapses while the program runs ("elapsed time").
4. The percentage of total CPU used in the process/program.
5. Parameters related to the set size of the process (in Kbytes).
6. Input/output parameters (integer value).
7. "Page-faults" usage (integer value).

time-output



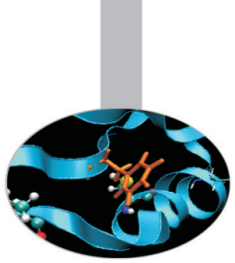
- The output of time command contains potentially useful informations:

time-output



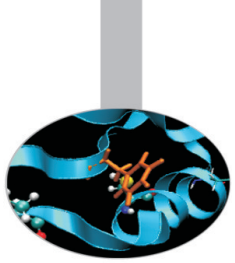
- The output of time command contains potentially useful informations:
  - (*The "user" time is comparable with "sys" time*)

time-output



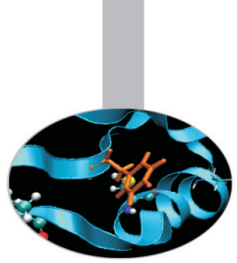
- The output of time command contains potentially useful informations:
  - (*The "user" time is comparable with "sys" time*)
  - (*The percentage of CPU usage is 100%*)

time-output



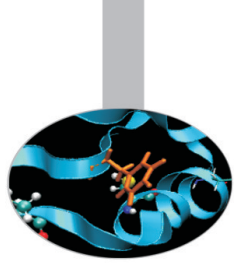
- The output of time command contains potentially useful informations:
  - (*The "user" time is comparable with "sys" time*)
  - (*The percentage of CPU usage is 100%*)
  - (*There is no I/O*)

time-output



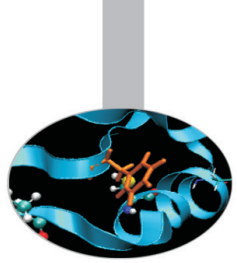
- The output of time command contains potentially useful informations:
  - (*The "user" time is comparable with "sys" time*)
  - (*The percentage of CPU usage is 100%*)
  - (*There is no I/O*)
  - (*There are (almost) no "page-faults"*)

time-output



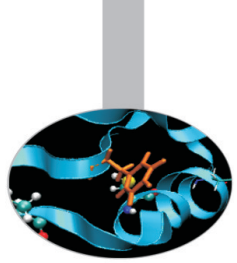
- The output of time command contains potentially useful informations:
  - (*The "user" time is comparable with "sys" time*)
  - (*The percentage of CPU usage is 100%*)
  - (*There is no I/O*)
  - (*There are (almost) no "page-faults"*)
  - (*The Maximum resident set size of the program.*)

time-output



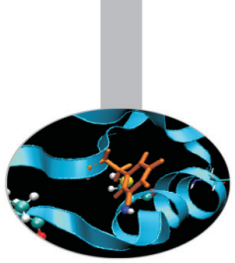
- The output of time command contains potentially useful informations:
  - (*The "user" time is comparable with "sys" time*)
  - (*The percentage of CPU usage is 100%*)
  - (*There is no I/O*)
  - (*There are (almost) no "page-faults"*)
  - (*The Maximum resident set size of the program.*)
  - Be careful, on some machine can be consistently given as too large by a factor four!

Page-faults



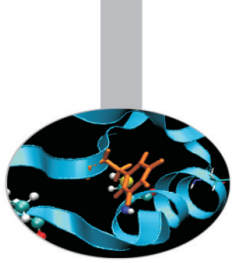
- The higher the size of the problem, the more the number of "page-faults" (at least 8 millions). What happens?

Page-faults



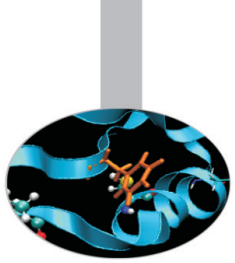
- The higher the size of the problem, the more the number of "page-faults" (at least 8 millions). What happens?
- A "page-fault" is a type of signal, called trap, raised by the hardware when a running program accesses a memory page that is mapped into the virtual address space, but not loaded in physical memory.

Page-faults



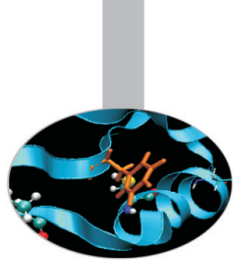
- The higher the size of the problem, the more the number of "page-faults" (at least 8 millions). What happens?
- A "page-fault" is a type of signal, called trap, raised by the hardware when a running program accesses a memory page that is mapped into the virtual address space, but not loaded in physical memory.
- When handling a page fault, the operating system tries to make the required page accessible at the location in physical memory, moving another non-free page from memory to disk to save space.

Page-faults



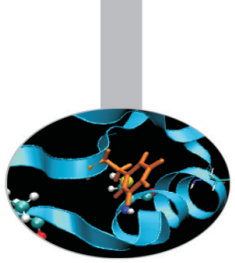
- The higher the size of the problem, the more the number of "page-faults" (at least 8 millions). What happens?
- A "page-fault" is a type of signal, called trap, raised by the hardware when a running program accesses a memory page that is mapped into the virtual address space, but not loaded in physical memory.
- When handling a page fault, the operating system tries to make the required page accessible at the location in physical memory, moving another non-free page from memory to disk to save space.
- This operation is really time-consuming and may slow-down the execution of our program.

time: output analysis



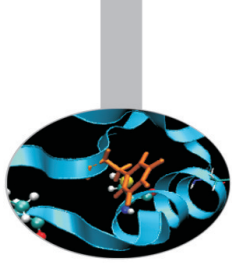
- For this example *System time* $\sim$ *User time*.

time: output analysis



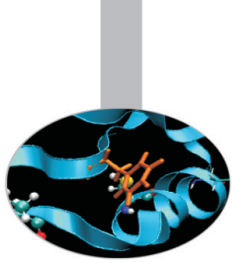
- For this example *System time* $\sim$ *User time*.
- it is not good, because it‘*s* may be due to page-faults activity or inefficient memory usage. In this case, a lot of system calls are done.

time: output analysis



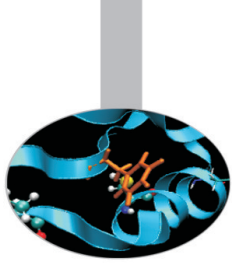
- For this example *System time* $\sim$ *User time*.
- it is not good, because it's may be due to page-faults activity or inefficient memory usage. In this case, a lot of system calls are done.
  The program need to "allocate" and "deallocate" a lot of matrices during its executioni: this is highly forbidden.

time: output analysis



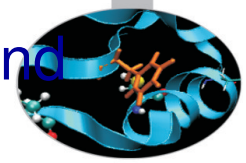
- For this example *System time* $\sim$ *User time*.
- it is not good, because it's may be due to page-faults activity or inefficient memory usage. In this case, a lot of system calls are done.
  The program need to "allocate" and "deallocate" a lot of matrices during its executioni: this is highly forbidden.
- *System time* + *User time* $\sim$ *Elapsed time*

time: output analysis



- For this example *System time* $\sim$ *User time*.
- it is not good, because it's may be due to page-faults activity or inefficient memory usage. In this case, a lot of system calls are done.
  The program need to "allocate" and "deallocate" a lot of matrices during its executioni: this is highly forbidden.
- *System time* + *User time* $\sim$ *Elapsed time*
  there is only one running process on machine.

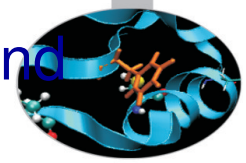
time: output analysis



Changing the program structure (e.g. avoiding allocations and deallocations during program lifetime) lead to significant performance improvement:

```
1.57user 0.10system 0:01.67elapsed 100%CPU (0avgtext+0avgdata 375944maxresident)k
0inputs+0outputs (0major+1080minor)pagefaults 0swaps
```

time: output analysis

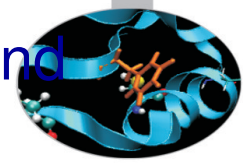


Changing the program structure (e.g. avoiding allocations and deallocations during program lifetime) lead to significant performance improvement:

```
1.57user 0.10system 0:01.67elapsed 100%CPU (0avgtext+0avgdata 375944maxresident)k
0inputs+0outputs (0major+1080minor)pagefaults 0swaps
```

now, things are beginning to make sense. Infact: *System time* << *User time*.

time: output analysis



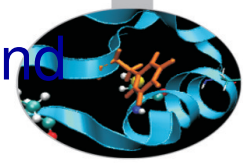
Changing the program structure (e.g. avoiding allocations and deallocations during program lifetime) lead to significant performance improvement:

```
1.57user 0.10system 0:01.67elapsed 100%CPU (0avgtext+0avgdata 375944maxresident)k
0inputs+0outputs (0major+1080minor)pagefaults 0swaps
```

now, things are beginning to make sense. Infact: *System time* << *User time*.

In the end, time command is a good tool to track useful informations in a "quick-and-dirty" way. Furthermore, it is non-intrusive.

time: output analysis



Changing the program structure (e.g. avoiding allocations and deallocations during program lifetime) lead to significant performance improvement:

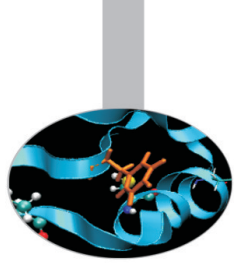
```
1.57user 0.10system 0:01.67elapsed 100%CPU (0avgtext+0avgdata 375944maxresident)k
0inputs+0outputs (0major+1080minor)pagefaults 0swaps
```

now, things are beginning to make sense. Infact: *System time* << *User time*.

In the end, time command is a good tool to track useful informations in a "quick-and-dirty" way. Furthermore, it is non-intrusive.
A major limitation of the time command: is quite difficult (or impossible) to extract some interesting features from a real-world application. For example, running COSMO meteorological application up 1 hour simulation 48 ("cores") of a standard multiprocessors machine (like Galileo):

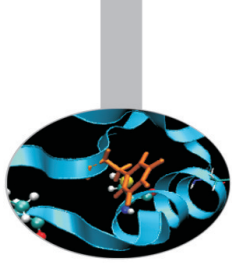
```
12973.38user 1915.82system 20:55.80elapsed 1185%CPU (0avgtext+0avgdata 2597648maxresident)k
19608inputs+10649880outputs (147major+223489935minor)pagefaults 0swaps
```

time: output analysis



At a first glance, the program should have worked. The first number (*1185 %CPU*):

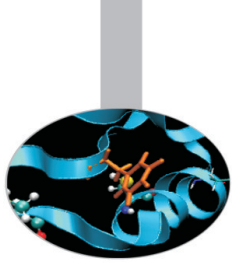
time: output analysis



At a first glance, the program should have worked. The first number (*1185 %CPU*):

- is the percentage of CPU usage. It is something much higher than 100% (Iit is not surprising that, because we are using 48 cores of our supercomputer).

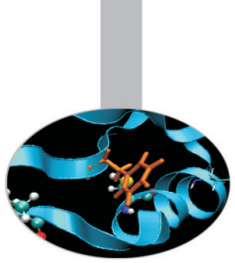
time: output analysis



At a first glance, the program should have worked. The first number (*1185 %CPU*):

- is the percentage of CPU usage. It is something much higher than 100% (Iit is not surprising that, because we are using 48 cores of our supercomputer).
- Letting *System time* to zero, the *User time* is equal to the *elapsed time* times a number proportional to the CPU percentage.

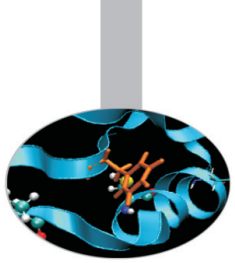
time: output analysis



At a first glance, the program should have worked. The first number (*1185 %CPU*):

- is the percentage of CPU usage. It is something much higher than 100% (Iit is not surprising that, because we are using 48 cores of our supercomputer).
- Letting *System time* to zero, the *User time* is equal to the *elapsed time* times a number proportional to the CPU percentage.
- This number is lower than the number of used cores

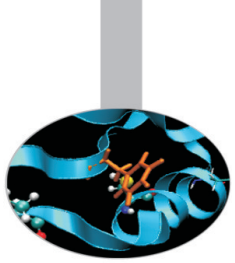
time: output analysis



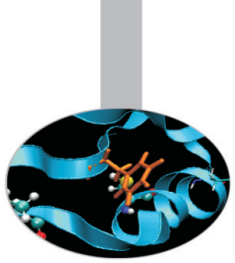
At a first glance, the program should have worked. The first number (*1185 %CPU*):

- is the percentage of CPU usage. It is something much higher than 100% (Iit is not surprising that, because we are using 48 cores of our supercomputer).
- Letting *System time* to zero, the *User time* is equal to the *elapsed time* times a number proportional to the CPU percentage.
- This number is lower than the number of used cores
- The resulting parallel efficiency is not really satisfactory.

Outline

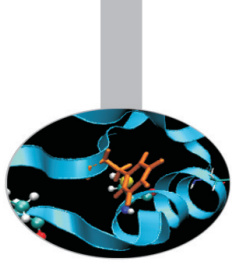


Top



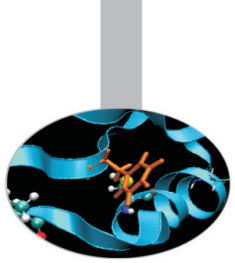
The time command returns useful informations of the time spent in the execution of a given process (program). Nevertheless, this information is static (available only when the program completes) without giving us any additional informations about its "behaviour" (over the time).

Top



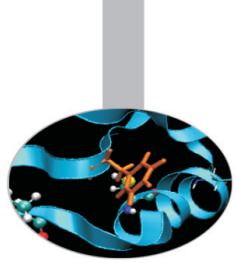
The time command returns useful informations of the time spent in the execution of a given process (program).
Nevertheless, this information is static (available only when the program completes) without giving us any additional informations about its "behaviour" (over the time).
Besides, time command does not return any other kind of information about computing and network resources (cores, I/O, network) status and activity related to our application.

Top



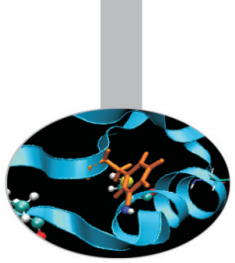
The time command returns useful informations of the time spent in the execution of a given process (program).
Nevertheless, this information is static (available only when the program completes) without giving us any additional informations about its "behaviour" (over the time).
Besides, time command does not return any other kind of information about computing and network resources (cores, I/O, network) status and activity related to our application.
In the end, Top is a quite simple command returning these and other informations.
Command Sintax:
top [options ...]

Top command output

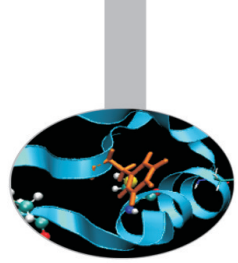


```
top - 14:57:46 up 19 days, 23:19, 38 users,  load average: 4.38, 1.68, 0.73
Tasks: 449 total,   3 running, 442 sleeping,   3 stopped,   1 zombie
Cpu(s): 39.3%us,  0.9%sy,  0.0%ni, 59.7%id,  0.0%wa,  0.0%hi,  0.1%si,  0.0%st
Mem:  24725848k total, 11623572k used, 13102276k free,   124732k buffers
Swap: 15999960k total,    96420k used, 15903540k free,  8921564k cached

  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
21524 lanucara  20   0 2407m 1.5g 4880 R 860.9  6.3   0:26.85 mm_mkl
21450 fferre    20   0  115m 6752 1640 R 99.0  0.0   0:27.21 parseBlastout.p
21485 lanucara  20   0 17400 1572  976 R  0.7  0.0   0:00.04 top
  416 root      20   0     0    0    0 S  0.3  0.0  14:55.00 rpciod/0
  424 root      20   0     0    0    0 S  0.3  0.0   0:27.90 rpciod/8
  442 root      15  -5     0    0    0 S  0.3  0.0   2:59.49 kslowd001
  450 root      20   0     0    0    0 S  0.3  0.0  22:58.02 nfsiod
 8430 paoletti  20   0  114m 2116 1040 S  0.3  0.0   0:01.43 sshd
 9522 nobody    20   0  167m  13m 1020 S  0.3  0.1  14:54.15 gmond
20338 tbiagini  20   0  114m 1920  872 S  0.3  0.0   0:00.04 sshd
26365 lanucara  20   0  149m 3384 2088 S  0.3  0.0   0:01.82 xterm
26395 lanucara  20   0 17396 1568  972 S  0.3  0.0   0:29.53 top
    1 root      20   0 21444 1112  932 S  0.0  0.0   0:05.37 init
    2 root      20   0     0    0    0 S  0.0  0.0   0:00.45 kthreadd
    3 root      RT   0     0    0    0 S  0.0  0.0   0:08.27 migration/0
    4 root      20   0     0    0    0 S  0.0  0.0   0:05.73 ksoftirqd/0
```

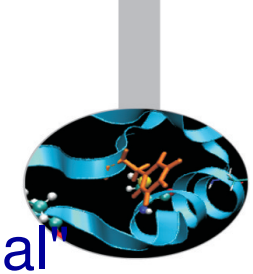
Outline



gprof overview

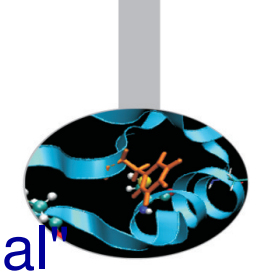


gprof overview



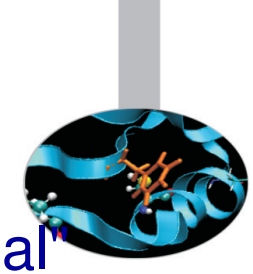
- time and top are nice tools to return "large-grain" and "global" informations of our application.

gprof overview



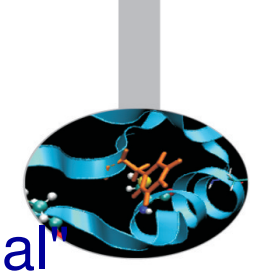
- time and top are nice tools to return "large-grain" and "global" informations of our application.
- Clearly, this kind of analysis is often applied to quite simple benchmarks while we need something more effective dealing with "real-world" applications (like COSMO code for example).

gprof overview



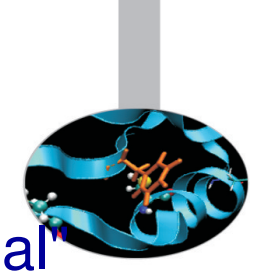
- time and top are nice tools to return "large-grain" and "global" informations of our application.
- Clearly, this kind of analysis is often applied to quite simple benchmarks while we need something more effective dealing with "real-world" applications (like COSMO code for example).
- At a first glance, what we need is a tool simple, "portable" across different computing machines and returning the main informations relevant to our application.

gprof overview



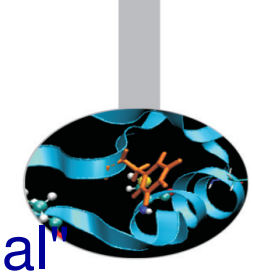
- time and top are nice tools to return "large-grain" and "global" informations of our application.
- Clearly, this kind of analysis is often applied to quite simple benchmarks while we need something more effective dealing with "real-world" applications (like COSMO code for example).
- At a first glance, what we need is a tool simple, "portable" across different computing machines and returning the main informations relevant to our application.
- gprof, is part of the GNU toolchain, so the "portability" is fairly satisfied.
- Main features:

gprof overview



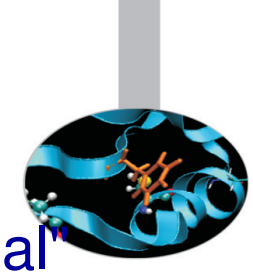
- time and top are nice tools to return "large-grain" and "global" informations of our application.
- Clearly, this kind of analysis is often applied to quite simple benchmarks while we need something more effective dealing with "real-world" applications (like COSMO code for example).
- At a first glance, what we need is a tool simple, "portable" across different computing machines and returning the main informations relevant to our application.
- gprof, is part of the GNU toolchain, so the "portability" is fairly satisfied.
- Main features:
  - a little bit intrusive (but not too much!)

gprof overview



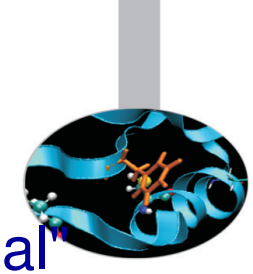
- time and top are nice tools to return "large-grain" and "global" informations of our application.
- Clearly, this kind of analysis is often applied to quite simple benchmarks while we need something more effective dealing with "real-world" applications (like COSMO code for example).
- At a first glance, what we need is a tool simple, "portable" across different computing machines and returning the main informations relevant to our application.
- gprof, is part of the GNU toolchain, so the "portability" is fairly satisfied.
- Main features:
  - a little bit intrusive (but not too much!)
  - it returns "subroutine" and "functions" related informations

gprof overview



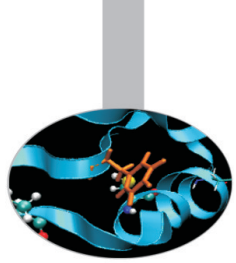
- time and top are nice tools to return "large-grain" and "global" informations of our application.
- Clearly, this kind of analysis is often applied to quite simple benchmarks while we need something more effective dealing with "real-world" applications (like COSMO code for example).
- At a first glance, what we need is a tool simple, "portable" across different computing machines and returning the main informations relevant to our application.
- gprof, is part of the GNU toolchain, so the "portability" is fairly satisfied.
- Main features:
  - a little bit intrusive (but not too much!)
  - it returns "subroutine" and "functions" related informations
  - it returns "call-graph" related informations

gprof overview

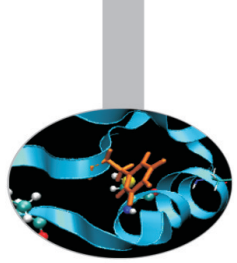


- time and top are nice tools to return "large-grain" and "global" informations of our application.
- Clearly, this kind of analysis is often applied to quite simple benchmarks while we need something more effective dealing with "real-world" applications (like COSMO code for example).
- At a first glance, what we need is a tool simple, "portable" across different computing machines and returning the main informations relevant to our application.
- gprof, is part of the GNU toolchain, so the "portability" is fairly satisfied.
- Main features:
  - a little bit intrusive (but not too much!)
  - it returns "subroutine" and "functions" related informations
  - it returns "call-graph" related informations
  - is based on top of "Sampling" and "Instrumentation" concepts

Gprof "Sampling"

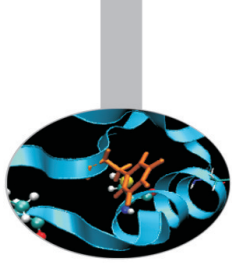


Gprof "Sampling"



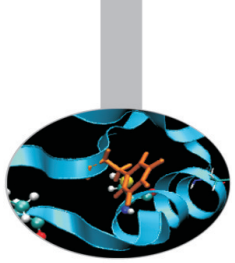
- The "Sampling" technique is used by Gprof (and in general by Profiling tools) to collect informations which are related to the "behaviour" of our application during its execution.

Gprof "Sampling"



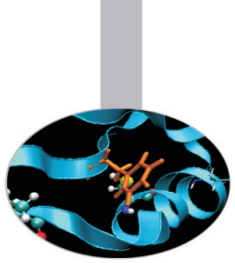
- The "Sampling" technique is used by Gprof (and in general by Profiling tools) to collect informations which are related to the "behaviour" of our application during its execution.
- Gprof is a Time Based Sampling profiler, that is it derives the information it provides by recording the address in the program counter at regular intervals over the course of a run.

Gprof "Sampling"



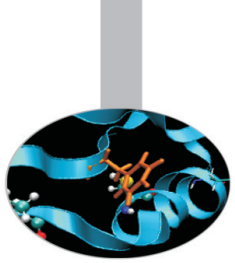
- The "Sampling" technique is used by Gprof (and in general by Profiling tools) to collect informations which are related to the "behaviour" of our application during its execution.
- Gprof is a Time Based Sampling profiler, that is it derives the information it provides by recording the address in the program counter at regular intervals over the course of a run.
- The "program counter" is recording at a fixed rate (e.g. we can fix to 100 for sake of clarity) per second of "run-time". This "number" may vary from machine to another machine.

Gprof "Sampling"



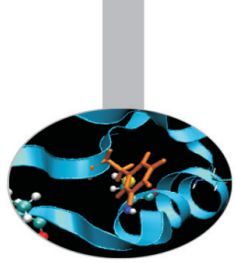
- The "Sampling" technique is used by Gprof (and in general by Profiling tools) to collect informations which are related to the "behaviour" of our application during its execution.
- Gprof is a Time Based Sampling profiler, that is it derives the information it provides by recording the address in the program counter at regular intervals over the course of a run.
- The "program counter" is recording at a fixed rate (e.g. we can fix to 100 for sake of clarity) per second of "run-time". This "number" may vary from machine to another machine.
- The "Sampling" technique is intrinsically statistical, so its effectiveness is strictly depending on this "sampling period".

Gprof "Sampling"

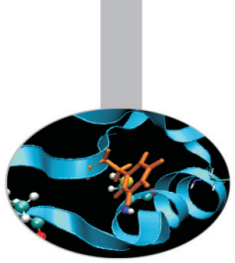


- The "Sampling" technique is used by Gprof (and in general by Profiling tools) to collect informations which are related to the "behaviour" of our application during its execution.
- Gprof is a Time Based Sampling profiler, that is it derives the information it provides by recording the address in the program counter at regular intervals over the course of a run.
- The "program counter" is recording at a fixed rate (e.g. we can fix to 100 for sake of clarity) per second of "run-time". This "number" may vary from machine to another machine.
- The "Sampling" technique is intrinsically statistical, so its effectiveness is strictly depending on this "sampling period".
- Because its little intrusivity we should be granted about the correctness of the profiled execution.

Gprof "Instrumentation"

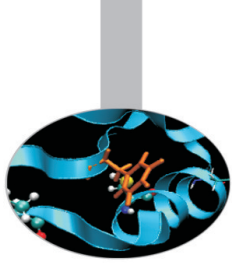


Gprof "Instrumentation"



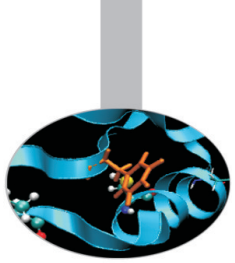
- Using Gprof, instrumentation code is automatically inserted into the program code during compilation to gather "call-graph" data.

Gprof "Instrumentation"



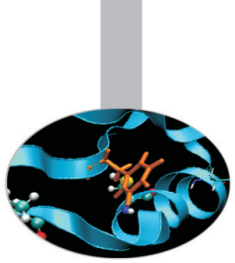
- Using Gprof, instrumentation code is automatically inserted into the program code during compilation to gather "call-graph" data.
- A call to the monitor function mcount is inserted before each function call.

Gprof "Instrumentation"



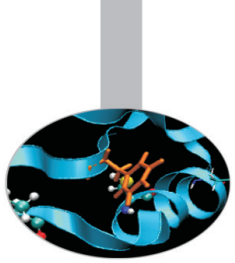
- Using Gprof, instrumentation code is automatically inserted into the program code during compilation to gather "call-graph" data.
- A call to the monitor function mcount is inserted before each function call.
- This techique can generate a limited overhead (depending on the application). Nevertheless, the compiler is (often) able to drive this process in an efficient way.

Gprof:how-to



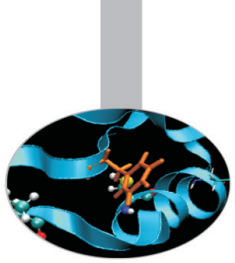
- Using Gprof is a three step process. First, the source code (written in Fortran, C, ...) is compiled and linked with gcc (or your favorite compiler) using options that signal the runtime to collect statistical information (-pg option)
- Use:
  <compiler> -pg programma.f -o nome_eseguibile
  ./nome_eseguibile
  gprof nome_eseguibile

Gprof:how-to



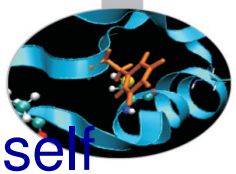
- Using Gprof is a three step process. First, the source code (written in Fortran, C, ...) is compiled and linked with gcc (or your favorite compiler) using options that signal the runtime to collect statistical information (-pg option)
- Use:
  <compiler> -pg programma.f -o nome_eseguibile
  ./nome_eseguibile
  gprof nome_eseguibile
- Then, the program is run one or more times, each time creating a gmon.out file with run-time informations (after successful runs!)

Gprof:how-to



- Using Gprof is a three step process. First, the source code (written in Fortran, C, ...) is compiled and linked with gcc (or your favorite compiler) using options that signal the runtime to collect statistical information (-pg option)
- Use:
  <compiler> -pg programma.f -o nome_eseguibile
  ./nome_eseguibile
  gprof nome_eseguibile
- Then, the program is run one or more times, each time creating a gmon.out file with run-time informations (after successful runs!)
- Gprof is then run against the gmon.out file producing one or more reports of the runtime behavior. Caution, old gmon.out files are oversubscribed.

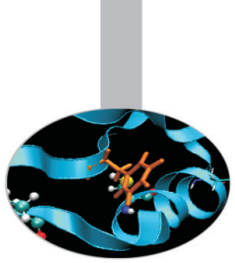
gprof: Flat profile



- Flat profile: This generates flat-profile analytics, where time calls are given for comparison.In particular, cumulative and self calls are given for functions (subroutines) to be profiled. The function ordering is from the higher to the lower in terms of CPU time. Let's see with a simple C example:

```
#include <stdio.h>
int a(void) {
  int i=0,g=0;
  while(i++<100000)
  {
     g+=i;
  }
  return g;
}
int b(void) {
  int i=0,g=0;
  while(i++<400000)
  {
    g+=i;
  }
  return g;
}
int main(int argc, char** argv)
{
   int iterations;

   if(argc != 2)
   {
```

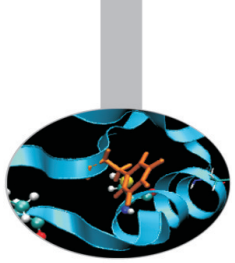
gprof: Flat profile



- Flat profile: to be continue....

```
printf("Usage %s <No of Iterations>\n", argv[0]);
      exit(-1);
   }
   else
      iterations = atoi(argv[1]);
 printf("No of iterations = %d\n", iterations);

   while(iterations--)
   {
      a();
      b();
   }
}
```

gprof: Flat profile

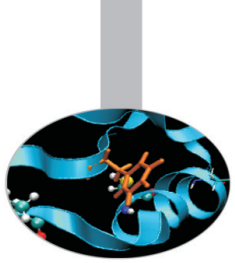


- Flat profile: to be continue....

```
printf("Usage %s <No of Iterations>\n", argv[0]);
      exit(-1);
   }
   else
      iterations = atoi(argv[1]);
 printf("No of iterations = %d\n", iterations);

   while(iterations--)
   {
      a();
      b();
   }
}
```

- The last step above produces an analysis file which is in human readable format.

gprof: Flat profile

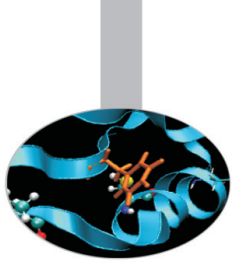


- Flat profile: to be continue....

```
printf("Usage %s <No of Iterations>\n", argv[0]);
      exit(-1);
   }
   else
      iterations = atoi(argv[1]);
 printf("No of iterations = %d\n", iterations);

   while(iterations--)
   {
      a();
      b();
   }
}
```

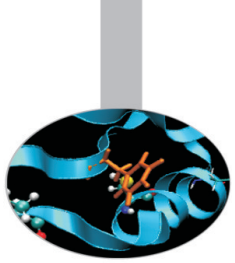
- The last step above produces an analysis file which is in human readable format.
- The file shows that most of the time is spent in the routine b() and that its workload is approximately 4 times the workload of routine a():

gprof: flat profile



```
/usr/bin/time ./Main_example.exe 10000
No of iterations = 10000
3.22user 0.00system 0:03.23elapsed 99%CPU (0avgtext+0avgdata 1760maxresident)k
0inputs+0outputs (0major+131minor)pagefaults 0swaps

 gcc -O Main_example.c -o Main_example_gprof.exe -pg
No of iterations = 10000
12.31user 0.00system 0:12.30elapsed 100%CPU (0avgtext+0avgdata 548maxresident)k
0inputs+0outputs (0major+212minor)pagefaults 0swaps

gprof ./Main_example_gprof.exe > Main_example.gprof

Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 82.19      10.11     10.11   10000   1.01   1.01  b
 19.60      12.144     2.33    10000    0.23    0.23  a
```

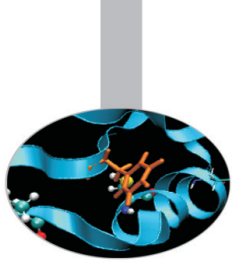
Flat profile:description



1. The percentage of time (with respect to the total time) spent in the routine.
2. The cumulative time spent in the routine and above.
3. The time (in seconds) spent in the routine.
4. The number of times this routine is called.
5. The mean time (in milliseconds) spent in this routine per single call.
6. The total mean time spent in this routine per call (including also its descendents).
7. The name of the routine.

In this example there are no "descendents", so "self and "total" time are practically the same.

gprof: flat profile



Try to vary the workload within source, introducing a simple function:

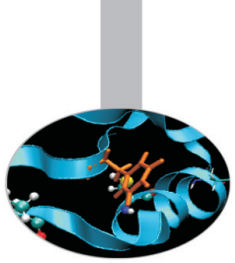
```
int cinsideb(int d) {
  {
  }
  return d;
}
```

and we pose this function within b() in place of g computation:

```
int b(void) {
  int i=0,g=0;
  while(i++<400000)
  {
    g+=cinsideb(i);
  }
  return g;
}
```

Let's see what happens, with this new program:

gprof: flat profile

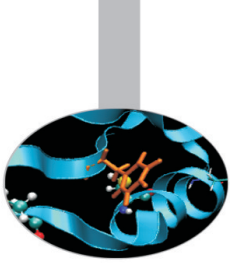


The new Flaf profile:

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self                 self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 45.54      4.39     4.39    10000   438.57   656.84  b
25.50       6.84     2.46    10000   245.56   245.56  a
 22.67      9.02     0.69  4000000000     0.00     0.00  cinsideb
```

Comments:

gprof: flat profile



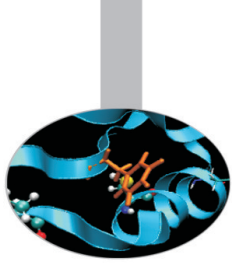
The new Flaf profile:

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self                 self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 45.54      4.39     4.39    10000   438.57   656.84  b
25.50       6.84     2.46    10000   245.56   245.56  a
 22.67      9.02     0.69  4000000000     0.00     0.00  cinsideb
```

Comments:

- Combining the routines b() and cinsideb() together, a global percentage of 80 is reached.

gprof: flat profile



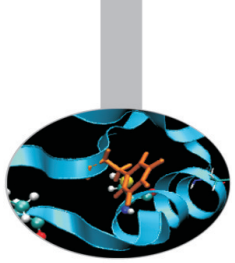
The new Flaf profile:

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self                 self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 45.54      4.39     4.39    10000   438.57   656.84  b
25.50       6.84     2.46    10000   245.56   245.56  a
 22.67      9.02     0.69  4000000000     0.00     0.00  cinsideb
```

Comments:

- Combining the routines b() and cinsideb() together, a global percentage of 80 is reached.
- The "child" function cinsideb() of b() is visible in the Flat-profile and its contribution is responsible for the increasing of the "total" time of function b()

gprof: flat profile



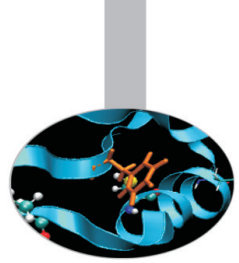
The new Flaf profile:

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self                 self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 45.54      4.39     4.39    10000   438.57   656.84  b
25.50       6.84     2.46    10000   245.56   245.56  a
 22.67      9.02     0.69  4000000000     0.00     0.00  cinsideb
```

Comments:

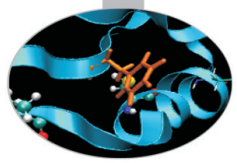
- Combining the routines b() and cinsideb() together, a global percentage of 80 is reached.
- The "child" function cinsideb() of b() is visible in the Flat-profile and its contribution is responsible for the increasing of the "total" time of function b()
- For this example, an increasing overhead is due to the enormous number of calls of function cinsideb()

gprof: Call tree profile



The "Call-tree" profile shows the percentage of time and "self"/"children" calls timings for each routine.

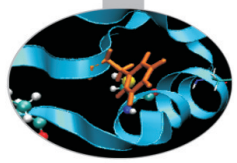
gprof: Call tree profile



The "Call-tree" profile shows the percentage of time and "self"/"children" calls timings for each routine.

The ordering is based on the percentage of time spent in each single routine (and descendents) in decreasing order.

gprof: Call tree profile



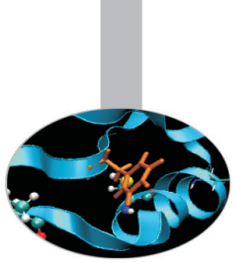
The "Call-tree" profile shows the percentage of time and "self"/"children" calls timings for each routine.

The ordering is based on the percentage of time spent in each single routine (and descendents) in decreasing order.

Let's see the Call tree profile of the last modified version of code:

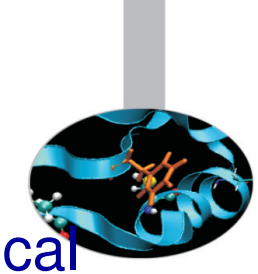
```
                     Call graph (explanation follows)

granularity: each sample hit covers 2 byte(s) for 0.11% of 9.13 seconds

index % time    self  children       called                          name
                                                  <spontaneous>
[1]    100.0    0.57    8.55                                        main [1]
                4.33    1.96   10000/10000                        b [2]
                2.27    0.00   10000/10000                       a [3]
-----------------------------------------------
                4.33    1.96   10000/10000                        main [1]
[2]     68.8    4.33    1.96   10000                               b [2]
                1.96    0.00 4000000000/4000000000     cinsideb [4]
-----------------------------------------------
                2.27    0.00   10000/10000                        main [1]
[3]     24.9    2.27    0.00   10000                              a [3]
-----------------------------------------------
                1.96    0.00 4000000000/4000000000     b [2]
[4]     21.4    1.96    0.00 4000000000                         cinsideb [4]
-----------------------------------------------

...
```

Call tree profile:description



1. An index defining "main program" and different routines within Flat-profile.
2. The percentage of time spent in each single routine and "childs" with respect to the total.
3. The total time spent in the routine.
4. The total time spent in its "childs".
5. The number of times the routine is called as "parent" and "child" with respect to the total number of calls in the entire program.
6. The name of the routine.

gprof: limitations



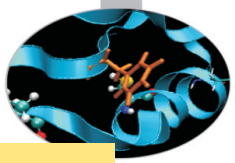
Let's see what happens with a simple program doing classical "matrix-matrix" product.Two approaches:

1. a program linking the MKL system libraries (which are fully optimized for the given hardware)
2. or using oppure a "standalone" library built compiling and linking BLAS sources on target machine

Results, in terms of "Flat Profile", profiling the two different versions:
MKL usage profiling:

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 71.43      0.10     0.10                             for_simd_random_number
 14.29      0.12     0.02        1    20.00    20.00  MAIN__
 14.29      0.14     0.02                             __intel_memset
  0.00      0.14     0.00        4     0.00     0.00  timing_module_mp_timing_
```

gprof: limitations



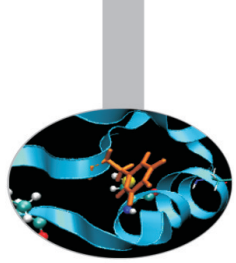
BLAS usage profiling:

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 97.76      6.10     6.10                             dgemm_
  1.60      6.20     0.10                             for_simd_random_number
  0.32      6.22     0.02        1    20.00    20.00  MAIN__
  0.32      6.24     0.02                             __intel_memset
  0.00      6.24     0.00        4     0.00     0.00  timing_module_mp_timing_
```

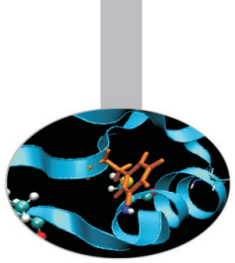
Comments:

- Gprof cannot measure time spent in kernel mode (syscalls, waiting for CPU or I/O waiting), and only user-space code is profiled. So, for the MKL version, there is no useful informations apart some auxiliary library call with some (meaningless) time report.
- The profiling of the BLAS version is correctly reporting the *dgemm_* call, which is responsible for the virtually all the total time.

gprof: other limitations

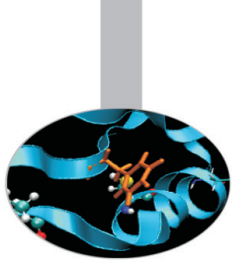


gprof: other limitations



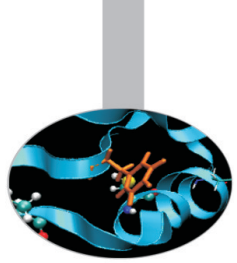
- Gprof "granularity" (which is strictly related to its "sampling rate") is quite high. Therefore, is not so easy to estimate complex real-world application performances and fine tuning is quite difficult (also, knowing in advance which is the function/routine which is responsible for the most of the cycles of a given applications).

gprof: other limitations



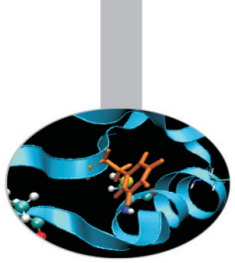
- Gprof "granularity" (which is strictly related to its "sampling rate") is quite high. Therefore, is not so easy to estimate complex real-world application performances and fine tuning is quite difficult (also, knowing in advance which is the function/routine which is responsible for the most of the cycles of a given applications).
- Gprof intrusivity may be quite huge. It is a good practice to compare the "naive" execution with the "gprof" execution and verify the impact of Gprof with the elapsed time of our application.

gprof: other limitations



- Gprof "granularity" (which is strictly related to its "sampling rate") is quite high. Therefore, is not so easy to estimate complex real-world application performances and fine tuning is quite difficult (also, knowing in advance which is the function/routine which is responsible for the most of the cycles of a given applications).
- Gprof intrusivity may be quite huge. It is a good practice to compare the "naive" execution with the "gprof" execution and verify the impact of Gprof with the elapsed time of our application.
- The sampling period (that is printed at the beginning of the flat profile) says how often samples are taken. The rule of thumb is that a run-time execution will be accurate if it is considerably bigger than the sampling period (this give us a brute-force estimate of the expected error (in seconds) of the Gprof analysis.

Gprof: Hands-on

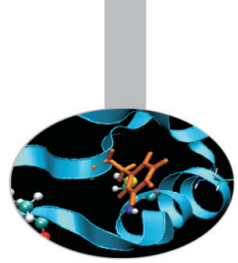


- Profiling of a serial code solving a Partial Differential Equation problem.
  - Use Gprof tool to profile the applications. Compare the results of Gprof analysis changing the size of the problem.
  - Source files under hpc-forge.cineca.it:

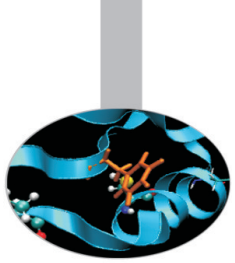
```
tar xvfz Gprof_Profiling_exercise.tar.gz
cd GPROF/JACOBI
```

  - read the content of README file
  - execute benchmarks
  - try to explain the profiling results in terms of the expected performances

Time a Function

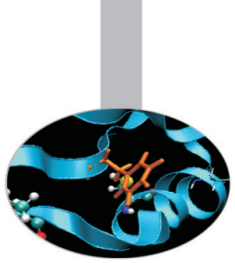


Time a Function



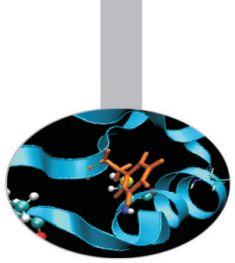
- Gprof granularity is well suited for finding bottlenecks in a program at a "function level" (e.g. to find which is the function responsible for the most of the time).

Time a Function



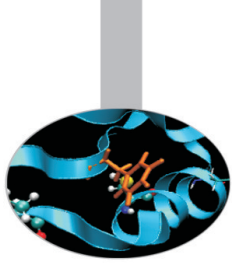
- Gprof granularity is well suited for finding bottlenecks in a program at a "function level" (e.g. to find which is the function responsible for the most of the time).
- After completing the gprof's analysis, we can manually "instrument" this routine with time measurement functions to finalize at a deeper level our analysis.

Time a Function



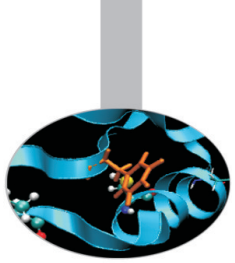
- Gprof granularity is well suited for finding bottlenecks in a program at a "function level" (e.g. to find which is the function responsible for the most of the time).
- After completing the gprof's analysis, we can manually "instrument" this routine with time measurement functions to finalize at a deeper level our analysis.
- This effective technique is used to avoid the large overhead of Gprof "line-by-line" analysis.

Time a Function



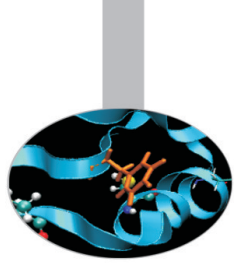
- Gprof granularity is well suited for finding bottlenecks in a program at a "function level" (e.g. to find which is the function responsible for the most of the time).
- After completing the gprof′s analysis, we can manually "instrument" this routine with time measurement functions to finalize at a deeper level our analysis.
- This effective technique is used to avoid the large overhead of Gprof "line-by-line" analysis.
- the drawbacks of this kind of technique is the lack of "portability" and " the level of coding intrusivity, that is particularly true for "third-party" complex applications. Some example:

Time a Function



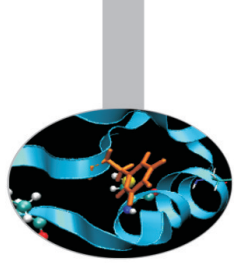
- Gprof granularity is well suited for finding bottlenecks in a program at a "function level" (e.g. to find which is the function responsible for the most of the time).
- After completing the gprof's analysis, we can manually "instrument" this routine with time measurement functions to finalize at a deeper level our analysis.
- This effective technique is used to avoid the large overhead of Gprof "line-by-line" analysis.
- the drawbacks of this kind of technique is the lack of "portability" and " the level of coding intrusivity, that is particularly true for "third-party" complex applications. Some example:
  - etime(),dtime() (Fortran 77)

Time a Function



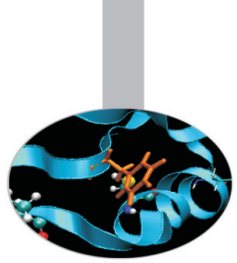
- Gprof granularity is well suited for finding bottlenecks in a program at a "function level" (e.g. to find which is the function responsible for the most of the time).
- After completing the gprof's analysis, we can manually "instrument" this routine with time measurement functions to finalize at a deeper level our analysis.
- This effective technique is used to avoid the large overhead of Gprof "line-by-line" analysis.
- the drawbacks of this kind of technique is the lack of "portability" and " the level of coding intrusivity, that is particularly true for "third-party" complex applications. Some example:
  - etime(),dtime() (Fortran 77)
  - cputime(),system_clock(), date_and_time() (Fortran 90)

Time a Function



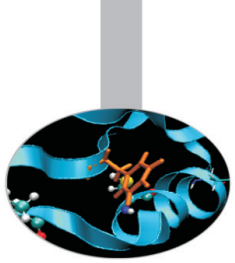
- Gprof granularity is well suited for finding bottlenecks in a program at a "function level" (e.g. to find which is the function responsible for the most of the time).
- After completing the gprof's analysis, we can manually "instrument" this routine with time measurement functions to finalize at a deeper level our analysis.
- This effective technique is used to avoid the large overhead of Gprof "line-by-line" analysis.
- the drawbacks of this kind of technique is the lack of "portability" and " the level of coding intrusivity, that is particularly true for "third-party" complex applications. Some example:
  - etime(),dtime() (Fortran 77)
  - cputime(),system_clock(), date_and_time() (Fortran 90)
  - clock() (C/C++)
  - ...

C/C++ example



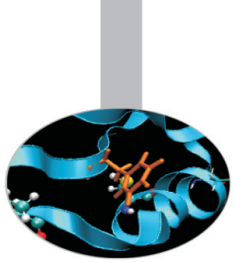
```
#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include <time.h>
clock_t time1, time2;
double dub_time;
int main(){
int i,j,k,nn=1000;
double c[nn][nn],a[nn][nn],b[nn][nn];
...
time1 = clock();
for (i = 0; i < nn; i++)
for (k = 0; k < nn; k++)
for (j = 0; j < nn; j ++)
c[i][j] = c[i][j] + a[i][k]*b[k][j];
time2 = clock();
dub_time = (time2 - time1)/(double) CLOCKS_PER_SEC;
printf("Time ----------------> %lf \n", dub_time);
...
return 0;
}
```

Fortran example

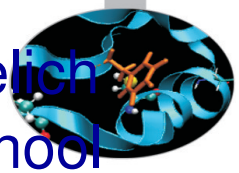


```
real(8)::a(1000,1000),b(1000,1000),c(1000,1000)
real(8) ::t1,t2
integer :: time_array(8)
a=0;b=0;c=0;n=1000
...
...
call date_and_time(values=time_array)
t1 = 3600.*time_array(5)+60.*time_array(6)+time_array(7)+time_array(8)/1000.
do j = 1,n
do k = 1,n
do i = 1,n
c(i,j) = c(i,j) + a(i,k)*b(k,j)
enddo
enddo
enddo
call date_and_time(values=time_array)
t2 = 3600.*time_array(5)+60.*time_array(6)+time_array(7)+time_array(8)/1000.
write(6,*) t2-t1
...
...
end
```

Outline

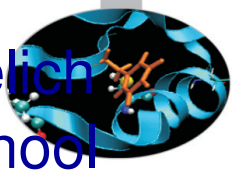


Scalasca: overview



- Tool developed by Felix Wolf in a collaboration between Juelich Supercomputing Centre and and the German Research School for Simulation Sciences.

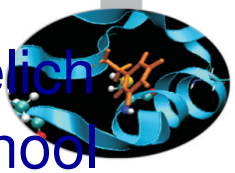
Scalasca: overview



- Tool developed by Felix Wolf in a collaboration between Juelich Supercomputing Centre and and the German Research School for Simulation Sciences.
- Scalasca was borned as a "successor" to another well-known tool (KOJAK)

- It is the reference toolset for the "scalable" "performance analysis" of large-scale parallel applications (MPI & OpenMP).

Scalasca: overview

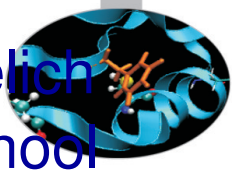


- Tool developed by Felix Wolf in a collaboration between Juelich Supercomputing Centre and and the German Research School for Simulation Sciences.
- Scalasca was borned as a "successor" to another well-known tool (KOJAK)

- It is the reference toolset for the "scalable" "performance analysis" of large-scale parallel applications (MPI & OpenMP).

- Can be installed and used on practically all the modern High Performance Computing (HPC) machines with dozens of thousand of "cores"....

Scalasca: overview



- Tool developed by Felix Wolf in a collaboration between Juelich Supercomputing Centre and and the German Research School for Simulation Sciences.
- Scalasca was borned as a "successor" to another well-known tool (KOJAK)

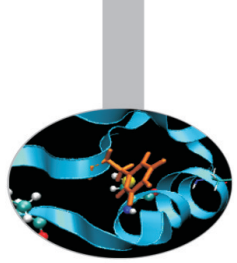
- It is the reference toolset for the "scalable" "performance analysis" of large-scale parallel applications (MPI & OpenMP).

- Can be installed and used on practically all the modern High Performance Computing (HPC) machines with dozens of thousand of "cores"....
- ...but also on "medium-size" parallel architectures.

- It's an "open-source" (but licensed) tool, continuously updated and mantained from Juelich.

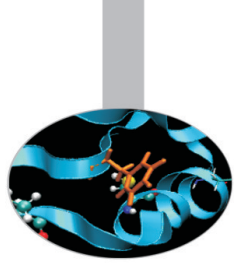
- See: www.scalasca.org

Scalasca: overview



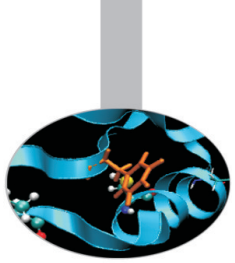
- Stuitable for Fortran, C e C++ applications.

Scalasca: overview



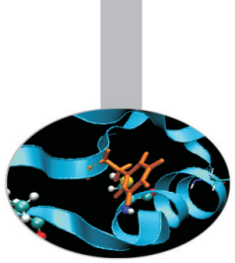
- Stuitable for Fortran, C e C++ applications.
- Scalasca analysis can be done using two different workflows:

Scalasca: overview



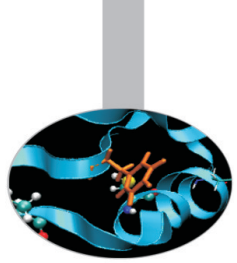
- Stuitable for Fortran, C e C++ applications.
- Scalasca analysis can be done using two different workflows:
  - "summary" workflow, suitable to obtain aggregated informations for our application (but detailed at the single instruction level) and

Scalasca: overview



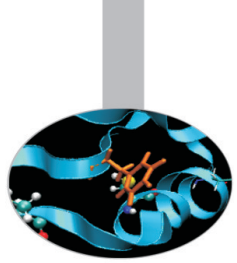
- Stuitable for Fortran, C e C++ applications.
- Scalasca analysis can be done using two different workflows:
  - "summary" workflow, suitable to obtain aggregated informations for our application (but detailed at the single instruction level) and
  - a "tracing" workflow, "process–local" and suitable to aggregate a huge variety of informations (qualitatively enriched). This report can be particolarly demanging in terms of computing, memory and storage resources.

Scalasca: overview



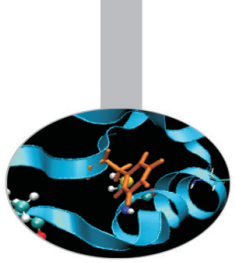
- Stuitable for Fortran, C e C++ applications.
- Scalasca analysis can be done using two different workflows:
  - "summary" workflow, suitable to obtain aggregated informations for our application (but detailed at the single instruction level) and
  - a "tracing" workflow, "process–local" and suitable to aggregate a huge variety of informations (qualitatively enriched). This report can be particolarly demanging in terms of computing, memory and storage resources.
- After running the instrumented code on the parallel machine, Scalasca is able to load "tracing" files in memory and analize them in parallel using the same number of cores of the original application.

Scalasca: comprehensive guide



The entire analysis process can be divided into three steps:

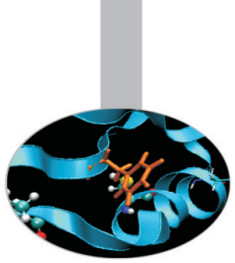
Scalasca: comprehensive guide



The entire analysis process can be divided into three steps:

- Compilation (source code is "instrumented"):

Scalasca: comprehensive guide



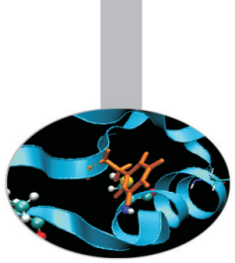
The entire analysis process can be divided into three steps:

- Compilation (source code is "instrumented"):

  ifort -openmp [other_options] <codice_sorgente>

  is created

Scalasca: comprehensive guide

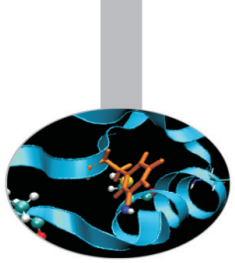


The entire analysis process can be divided into three steps:

- Compilation (source code is "instrumented"):
  scalasca -instrument [options_scalasca] ifort -openmp [other_options] <codice_sorgente>

is created

Scalasca: comprehensive guide



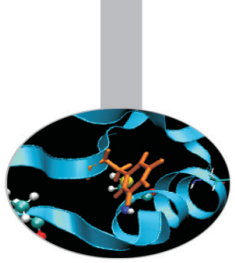
The entire analysis process can be divided into three steps:

- Compilation (source code is "instrumented"):
  scalasca -instrument [options_scalasca] ifort -openmp [other_options] <codice_sorgente>

  mpif90 [options] <source_code>

is created

Scalasca: comprehensive guide

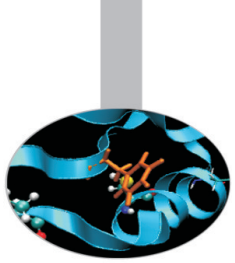


The entire analysis process can be divided into three steps:

- Compilation (source code is "instrumented"):
  scalasca -instrument [options_scalasca] ifort -openmp [other_options] <codice_sorgente>
  scalasca -instrument [options_scalasca] mpif90 [options] <source_code>

is created

Scalasca: comprehensive guide

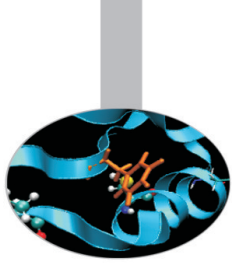


The entire analysis process can be divided into three steps:

- Compilation (source code is "instrumented"):
  scalasca -instrument [options_scalasca] ifort -openmp [other_options] <codice_sorgente>
  scalasca -instrument [options_scalasca] mpif90 [options] <source_code>
- Execution:

is created

Scalasca: comprehensive guide



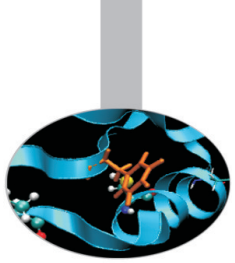
The entire analysis process can be divided into three steps:

- Compilation (source code is "instrumented"):
  scalasca -instrument [options_scalasca] ifort -openmp [other_options] <codice_sorgente>
  scalasca -instrument [options_scalasca] mpif90 [options] <source_code>
- Execution:

<executable_code>

is created

Scalasca: comprehensive guide

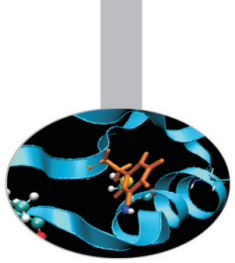


The entire analysis process can be divided into three steps:

- Compilation (source code is "instrumented"):
  scalasca -instrument [options_scalasca] ifort -openmp [other_options] <codice_sorgente>
  scalasca -instrument [options_scalasca] mpif90 [options] <source_code>
- Execution:
  scalasca -analyze [options_scalasca] <executable_code>

is created

Scalasca: comprehensive guide

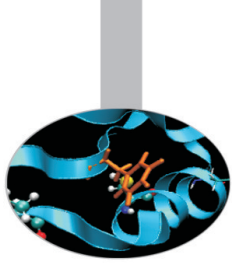


The entire analysis process can be divided into three steps:

- Compilation (source code is "instrumented"):
  scalasca -instrument [options_scalasca] ifort -openmp [other_options] <codice_sorgente>
  scalasca -instrument [options_scalasca] mpif90 [options] <source_code>
- Execution:
  scalasca -analyze [options_scalasca] <executable_code>
  mpirun [options] <executable_code>

is created

Scalasca: comprehensive guide

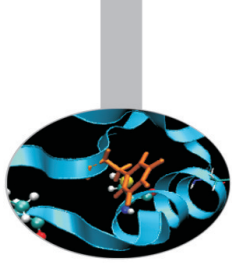


The entire analysis process can be divided into three steps:

- Compilation (source code is "instrumented"):
  scalasca -instrument [options_scalasca] ifort -openmp [other_options] <codice_sorgente>
  scalasca -instrument [options_scalasca] mpif90 [options] <source_code>
- Execution:
  scalasca -analyze [options_scalasca] <executable_code>
  scalasca -analyze [options_scalasca] mpirun [options] <executable_code>

is created

Scalasca: comprehensive guide

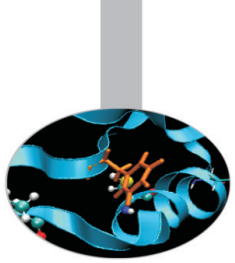


The entire analysis process can be divided into three steps:

- Compilation (source code is "instrumented"):
  scalasca -instrument [options_scalasca] ifort -openmp [other_options] <codice_sorgente>
  scalasca -instrument [options_scalasca] mpif90 [options] <source_code>
- Execution:
  scalasca -analyze [options_scalasca] <executable_code>
  scalasca -analyze [options_scalasca] mpirun [options] <executable_code>

  A directory epik_[characteristics] is created

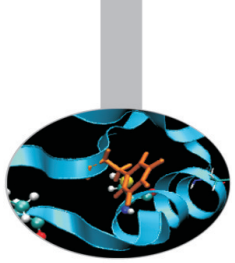
Scalasca: comprehensive guide



The entire analysis process can be divided into three steps:

- Compilation (source code is "instrumented"):
  scalasca -instrument [options_scalasca] ifort -openmp [other_options] <codice_sorgente>
  scalasca -instrument [options_scalasca] mpif90 [options] <source_code>
- Execution:
  scalasca -analyze [options_scalasca] <executable_code>
  scalasca -analyze [options_scalasca] mpirun [options] <executable_code>
  A directory epik_[characteristics] is created
- Analysis results :

Scalasca: comprehensive guide

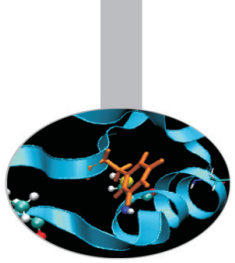


The entire analysis process can be divided into three steps:

- Compilation (source code is "instrumented"):
  scalasca -instrument [options_scalasca] ifort -openmp [other_options] <codice_sorgente>
  scalasca -instrument [options_scalasca] mpif90 [options] <source_code>
- Execution:
  scalasca -analyze [options_scalasca] <executable_code>
  scalasca -analyze [options_scalasca] mpirun [options] <executable_code>
  A directory epik_[characteristics] is created
- Analysis results :

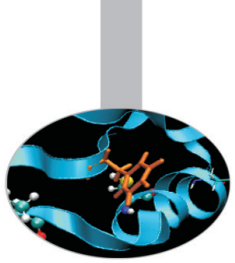
  scalasca -examine [options_scalasca] epik_[characteristics]

Scalasca: score-P integration



Starting from Scalasca 2.x release, users are strongly encouraged to use the score-P instrumenter directly:

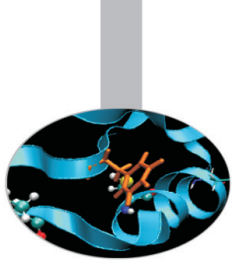
Scalasca: score-P integration



Starting from Scalasca 2.x release, users are strongly encouraged to use the score-P instrumenter directly:

- Compilation (source code is "instrumented" using scorep command):

Scalasca: score-P integration



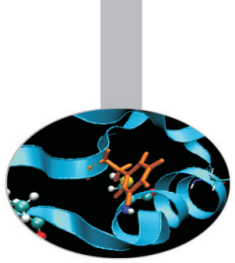
Starting from Scalasca 2.x release, users are strongly encouraged to use the score-P instrumenter directly:

- Compilation (source code is "instrumented" using scorep command):

ifort -openmp [other_options] <codice_sorgente>

is created

Scalasca: score-P integration

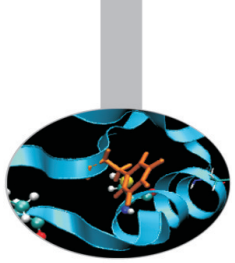


Starting from Scalasca 2.x release, users are strongly encouraged to use the score-P instrumenter directly:

- Compilation (source code is "instrumented" using scorep command):
  scorep [options_scorep] ifort -openmp [other_options] <codice_sorgente>

is created

Scalasca: score-P integration



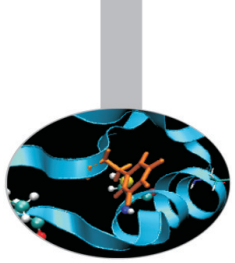
Starting from Scalasca 2.x release, users are strongly encouraged to use the score-P instrumenter directly:

- Compilation (source code is "instrumented" using scorep command):

scorep [options_scorep] ifort -openmp [other_options] <codice_sorgente>
mpif90 [options] <source_code>

is created

Scalasca: score-P integration



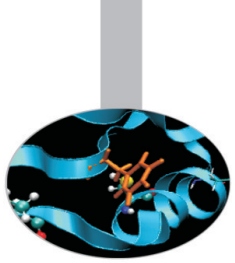
Starting from Scalasca 2.x release, users are strongly encouraged to use the score-P instrumenter directly:

- Compilation (source code is "instrumented" using scorep command):

scorep [options_scorep] ifort -openmp [other_options] <codice_sorgente>
scorep [options_scorep] mpif90 [options] <source_code>

is created

Scalasca: score-P integration

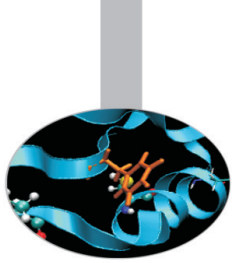


Starting from Scalasca 2.x release, users are strongly encouraged to use the score-P instrumenter directly:

- Compilation (source code is "instrumented" using scorep command):
  scorep [options_scorep] ifort -openmp [other_options] <codice_sorgente>
  scorep [options_scorep] mpif90 [options] <source_code>
- Execution:

is created

Scalasca: score-P integration



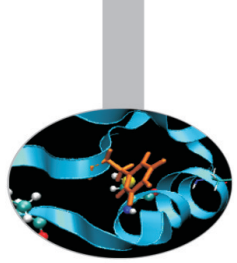
Starting from Scalasca 2.x release, users are strongly encouraged to use the score-P instrumenter directly:

- Compilation (source code is "instrumented" using scorep command):
  scorep [options_scorep] ifort -openmp [other_options] <codice_sorgente>
  scorep [options_scorep] mpif90 [options] <source_code>
- Execution:

<executable_code>

is created

Scalasca: score-P integration

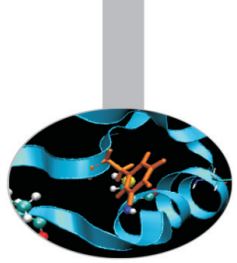


Starting from Scalasca 2.x release, users are strongly encouraged to use the score-P instrumenter directly:

- Compilation (source code is "instrumented" using scorep command):
  scorep [options_scorep] ifort -openmp [other_options] <codice_sorgente>
  scorep [options_scorep] mpif90 [options] <source_code>
- Execution:
  scalasca -analyze [options_scalasca] <executable_code>

is created

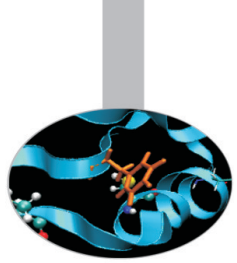
Scalasca: score-P integration



Starting from Scalasca 2.x release, users are strongly encouraged to use the score-P instrumenter directly:

- Compilation (source code is "instrumented" using scorep command):
  scorep [options_scorep] ifort -openmp [other_options] <codice_sorgente>
  scorep [options_scorep] mpif90 [options] <source_code>
- Execution:
  scalasca -analyze [options_scalasca] <executable_code>
  mpirun [options] <executable_code>
  is created

Scalasca: score-P integration

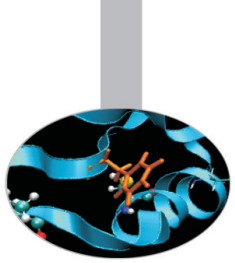


Starting from Scalasca 2.x release, users are strongly encouraged to use the score-P instrumenter directly:

- Compilation (source code is "instrumented" using scorep command):
  scorep [options_scorep] ifort -openmp [other_options] <codice_sorgente>
  scorep [options_scorep] mpif90 [options] <source_code>
- Execution:
  scalasca -analyze [options_scalasca] <executable_code>
  scalasca -analyze [options_scalasca] mpirun [options] <executable_code>

is created

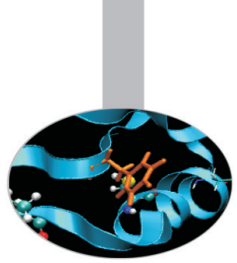
Scalasca: score-P integration



Starting from Scalasca 2.x release, users are strongly encouraged to use the score-P instrumenter directly:

- Compilation (source code is "instrumented" using scorep command):
  scorep [options_scorep] ifort -openmp [other_options] <codice_sorgente>
  scorep [options_scorep] mpif90 [options] <source_code>
- Execution:
  scalasca -analyze [options_scalasca] <executable_code>
  scalasca -analyze [options_scalasca] mpirun [options] <executable_code>
  A directory scorep_[characteristics] is created

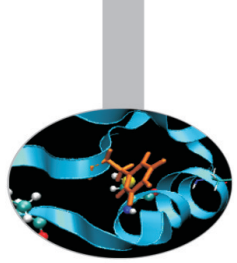
Scalasca: score-P integration



Starting from Scalasca 2.x release, users are strongly encouraged to use the score-P instrumenter directly:

- Compilation (source code is "instrumented" using scorep command):
  scorep [options_scorep] ifort -openmp [other_options] <codice_sorgente>
  scorep [options_scorep] mpif90 [options] <source_code>
- Execution:
  scalasca -analyze [options_scalasca] <executable_code>
  scalasca -analyze [options_scalasca] mpirun [options] <executable_code>
  A directory scorep_[characteristics] is created
- Analysis results :

Scalasca: score-P integration

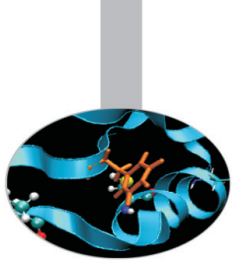


Starting from Scalasca 2.x release, users are strongly encouraged to use the score-P instrumenter directly:

- Compilation (source code is "instrumented" using scorep command):
  scorep [options_scorep] ifort -openmp [other_options] <codice_sorgente>
  scorep [options_scorep] mpif90 [options] <source_code>
- Execution:
  scalasca -analyze [options_scalasca] <executable_code>
  scalasca -analyze [options_scalasca] mpirun [options] <executable_code>
  A directory scorep_[characteristics] is created
- Analysis results :

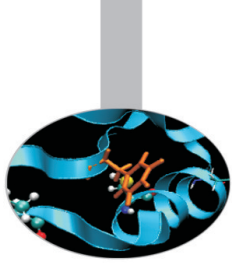
  scalasca -examine [options_scalasca] scorep_[characteristics]

Scalasca: a simple example



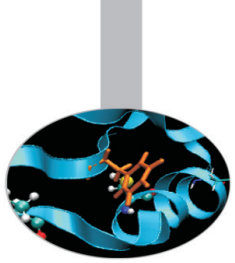
- The benchmark is the well known Himeno Benchmark (simplified Poisson solver):

Scalasca: a simple example



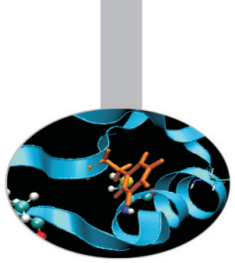
- The benchmark is the well known Himeno Benchmark (simplified Poisson solver):
- Parallelized using OpenMP.

Scalasca: a simple example



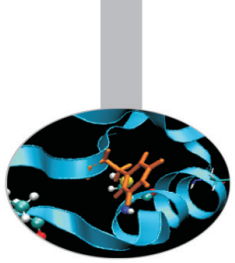
- The benchmark is the well known Himeno Benchmark (simplified Poisson solver):
- Parallelized using OpenMP.
- Run up to 16 cores ("moderate parallelism") on a single Galileo node

Scalasca: a simple example



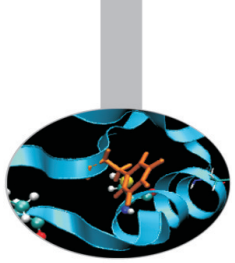
- The benchmark is the well known Himeno Benchmark (simplified Poisson solver):
- Parallelized using OpenMP.
- Run up to 16 cores ("moderate parallelism") on a single Galileo node
- Intel compiler(ifort -O3 -openmp...command)

Scalasca: a simple example



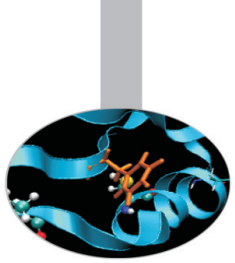
- The benchmark is the well known Himeno Benchmark (simplified Poisson solver):
- Parallelized using OpenMP.
- Run up to 16 cores ("moderate parallelism") on a single Galileo node
- Intel compiler(ifort -O3 -openmp...command)
- Some useful numbers:

Scalasca: a simple example



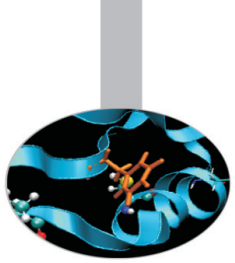
- The benchmark is the well known Himeno Benchmark (simplified Poisson solver):
- Parallelized using OpenMP.
- Run up to 16 cores ("moderate parallelism") on a single Galileo node
- Intel compiler(ifort -O3 -openmp...command)
- Some useful numbers:
  - Number of (grid) points 4276737.

Scalasca: a simple example



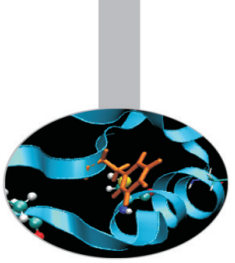
- The benchmark is the well known Himeno Benchmark (simplified Poisson solver):
- Parallelized using OpenMP.
- Run up to 16 cores ("moderate parallelism") on a single Galileo node
- Intel compiler(ifort -O3 -openmp...command)
- Some useful numbers:
  - Number of (grid) points 4276737.
  - Number of iterations 100.

Scalasca: compilation ed execution



```
scorep ifort -O3 -openmp himenoBMTxp_omp.f90
```

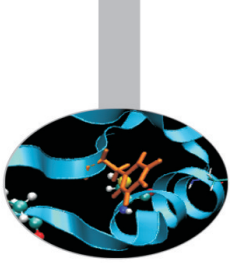
Scalasca: compilation ed execution



```
scorep ifort -O3 -openmp himenoBMTxp_omp.f90
```

```
export OMP_NUM_THREADS=2;scan ./a.out
```

Scalasca: compilation ed execution

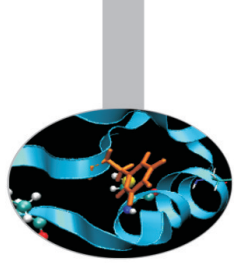


```
scorep ifort -O3 -openmp himenoBMTxp_omp.f90
```

```
export OMP_NUM_THREADS=2;scan ./a.out
```

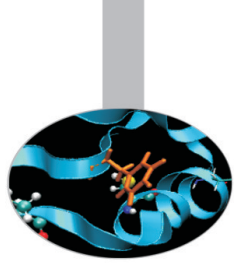
```
S=C=A=N: Scalasca 2.2 runtime summarization
S=C=A=N: Abort: measurement blocked by existing archive ./scorep_a_0x2_sum
rm -rf scorep_a_0x2_sum/
export OMP_NUM_THREADS=2;scan ./a.out
S=C=A=N: Wed Apr 13 11:25:10 2016: Collect done (status=0) 5s
S=C=A=N: ./scorep_a_0x2_sum complete.
```

Scalasca: analysis of results



```
square -s scorep_a_0x2_sum/
INFO: Post-processing runtime summarization report...
/cineca/prod/tools/scalasca/2.2/intelmpi--5.0.2--binary/bin/scorep-score -r ./scorep_a_0x2_su
INFO: Score report written to ./scorep_a_0x2_sum/scorep.score
```

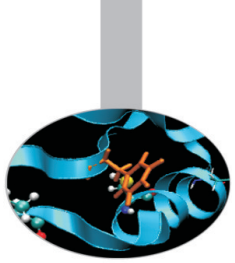
Scalasca: analysis of results



```
square -s scorep_a_0x2_sum/
INFO: Post-processing runtime summarization report...
/cineca/prod/tools/scalasca/2.2/intelmpi--5.0.2--binary/bin/scorep-score -r ./scorep_a_0x2_su
INFO: Score report written to ./scorep_a_0x2_sum/scorep.score
```

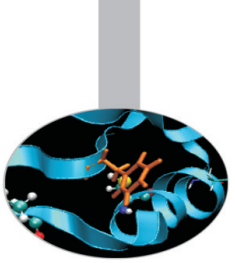
```
Estimated aggregate size of event trace:                   28kB
Estimated requirements for largest trace buffer (max_buf): 28kB
Estimated memory requirements (SCOREP_TOTAL_MEMORY):       7MB
(hint: When tracing set SCOREP_TOTAL_MEMORY=7MB to avoid intermediate flushes
 or reduce requirements using USR regions filters.)
...
flt     type max_buf[B] visits     time[s]     time[%] time/visit[us]   region
         ALL     27,918     1,157     6.97       100.0        6025.80         ALL
         OMP    27,654     1,146     3.46        49.6        3016.49         OMP
         USR    192            8     3.51        50.4      438994.57         USR
         COM    72             3     0.00         0.0        1001.07         COM
...
```

Scalasca output: short description



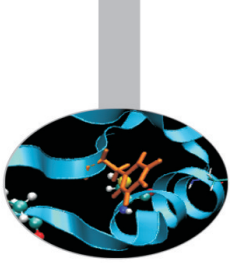
1. Output is divided into different categories, determined for each region according to its type of call path.

Scalasca output: short description



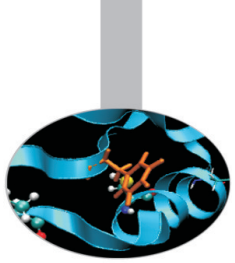
1. Output is divided into different categories, determined for each region according to its type of call path.
   - ALL: aggregated results containing all the regions or "function calls" within program source(s)

Scalasca output: short description



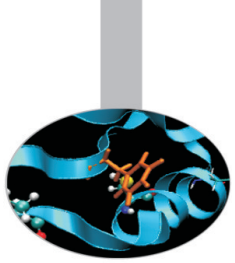
1. Output is divided into different categories, determined for each region according to its type of call path.
   - ALL: aggregated results containing all the regions or "function calls" within program source(s)
   - OMP: regions containing parallelization constructs (OpenMP or MPI or both).

Scalasca output: short description



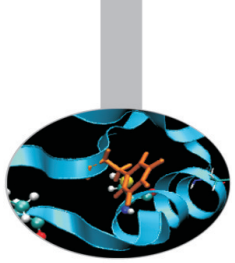
1. Output is divided into different categories, determined for each region according to its type of call path.
   - ALL: aggregated results containing all the regions or "function calls" within program source(s)
   - OMP: regions containing parallelization constructs (OpenMP or MPI or both).
   - USR: regions which are involved in purely local operations within process.

Scalasca output: short description



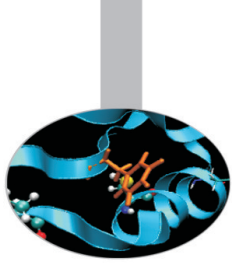
1. Output is divided into different categories, determined for each region according to its type of call path.
   - ALL: aggregated results containing all the regions or "function calls" within program source(s)
   - OMP: regions containing parallelization constructs (OpenMP or MPI or both).
   - USR: regions which are involved in purely local operations within process.
   - COM:the rest of not (USR and OMP (or MPI)).

Scalasca output: short description



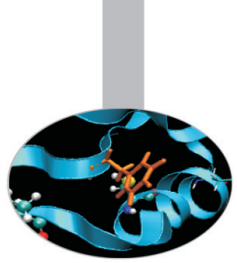
1. Output is divided into different categories, determined for each region according to its type of call path.
   - ALL: aggregated results containing all the regions or "function calls" within program source(s)
   - OMP: regions containing parallelization constructs (OpenMP or MPI or both).
   - USR: regions which are involved in purely local operations within process.
   - COM:the rest of not (USR and OMP (or MPI)).
2. The maximum estimated "trace-buffer" capacity (in bytes) and other parameters.

Scalasca output: a greater detail



```
cat scorep_a_0x2_sum/scorep.score
```

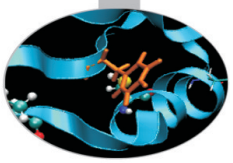
Scalasca output: a greater detail

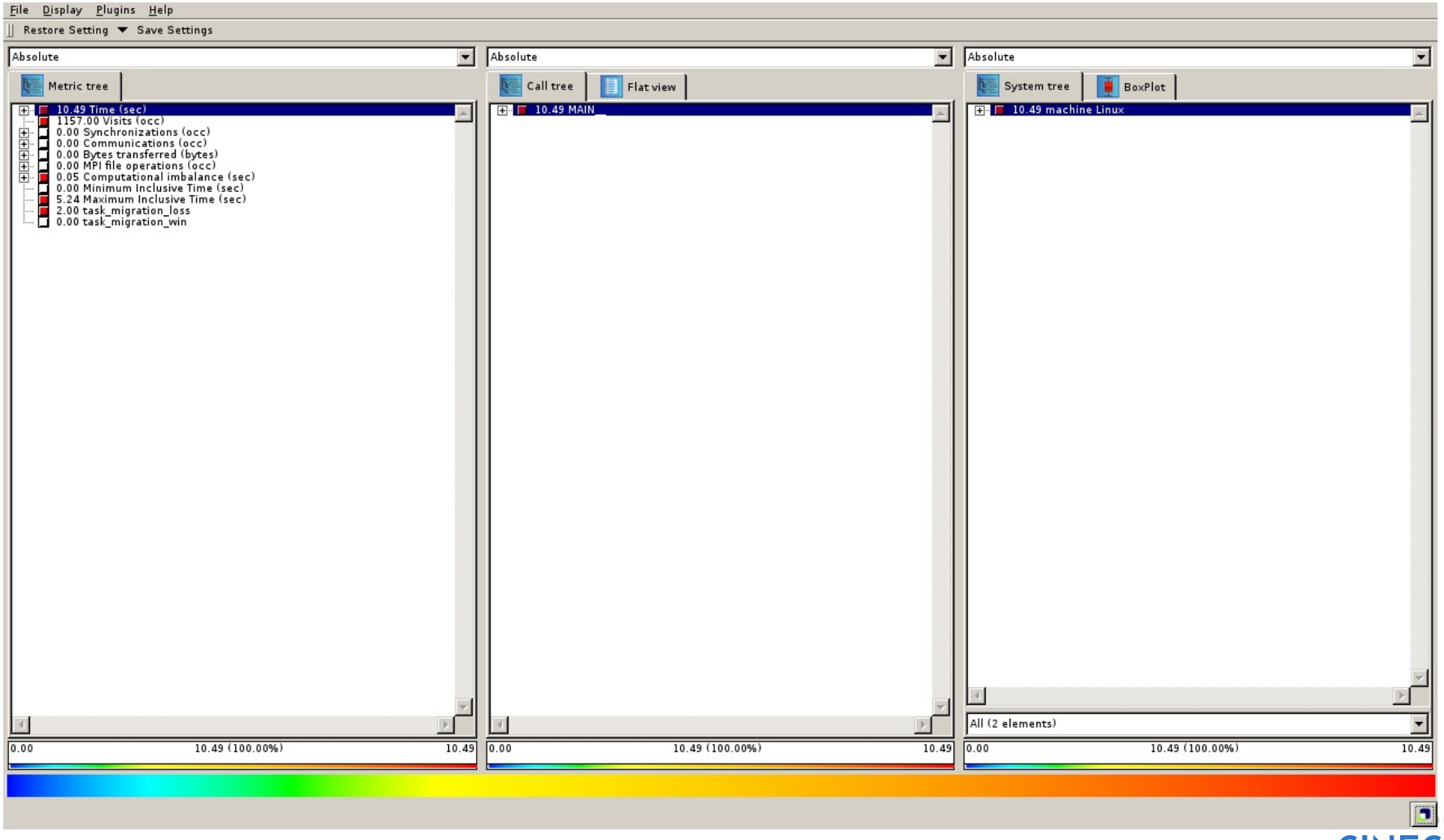


```
cat scorep_a_0x2_sum/scorep.score
```

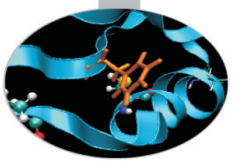
```
...
type max_buf[B] visits time[s] time[%] time/visit[us]  region
OMP       4,944    206    0.00     0.0           0.64  !$omp atomic @himenoBMTxp_omp.f90:335
OMP       4,944    206    1.26    18.1        6131.05  !$omp do @himenoBMTxp_omp.f90:329
OMP       4,944    206    0.04     0.6         192.49  !$omp barrier @himenoBMTxp_omp.f90:328
OMP       4,944    206    2.12    30.4       10297.42  !$omp do @himenoBMTxp_omp.f90:305
OMP       4,944    206    0.03     0.5         152.34  !$omp barrier @himenoBMTxp_omp.f90:300
OMP       2,472    103    0.00     0.0           0.35  !$omp master @himenoBMTxp_omp.f90:301
OMP         196      4    0.00     0.0         174.94  !$omp parallel @himenoBMTxp_omp.f90:297
OMP          98      2    0.00     0.0           5.88  !$omp parallel @himenoBMTxp_omp.f90:97
USR          96      4    0.00     0.0           0.94  second_
OMP          96      4    0.00     0.0         161.78  !$omp implicit barrier @himenoBMTxp_omp
USR          24      1    0.07     1.0       66482.94  initmt_
USR          24      1    0.00     0.0        2589.32  initmem_
USR          24      1    0.00     0.0          16.33  grid_set_
OMP          24      1    0.00     0.0          11.00  !$omp master @himenoBMTxp_omp.f90:98
USR          24      1    3.44    49.4     3442864.26  readparam_
COM          24      1    0.00     0.0        2968.85  MAIN__
      ....
```

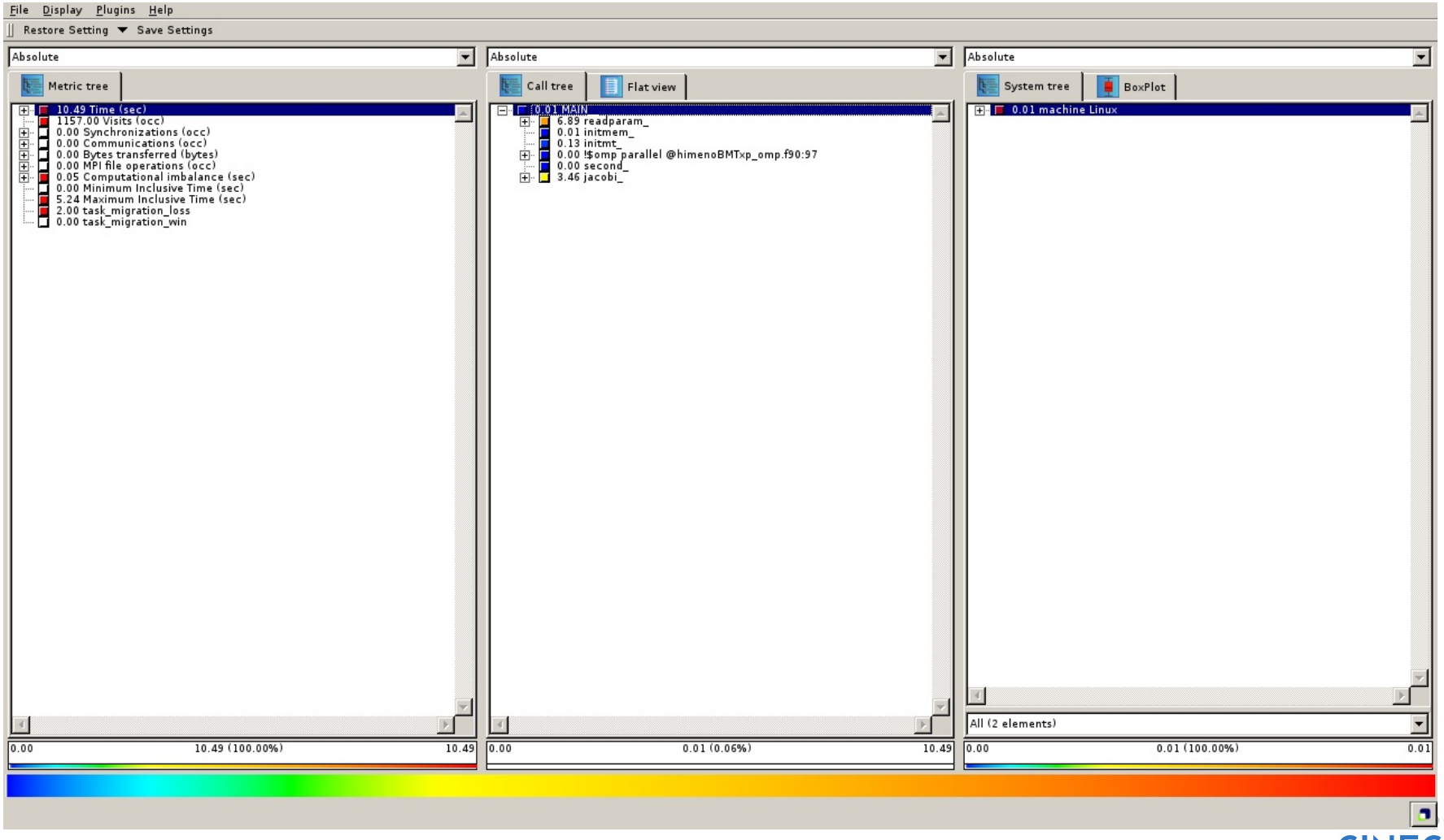
Scalasca: graphic output



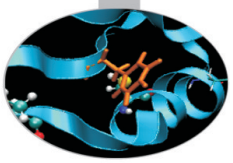


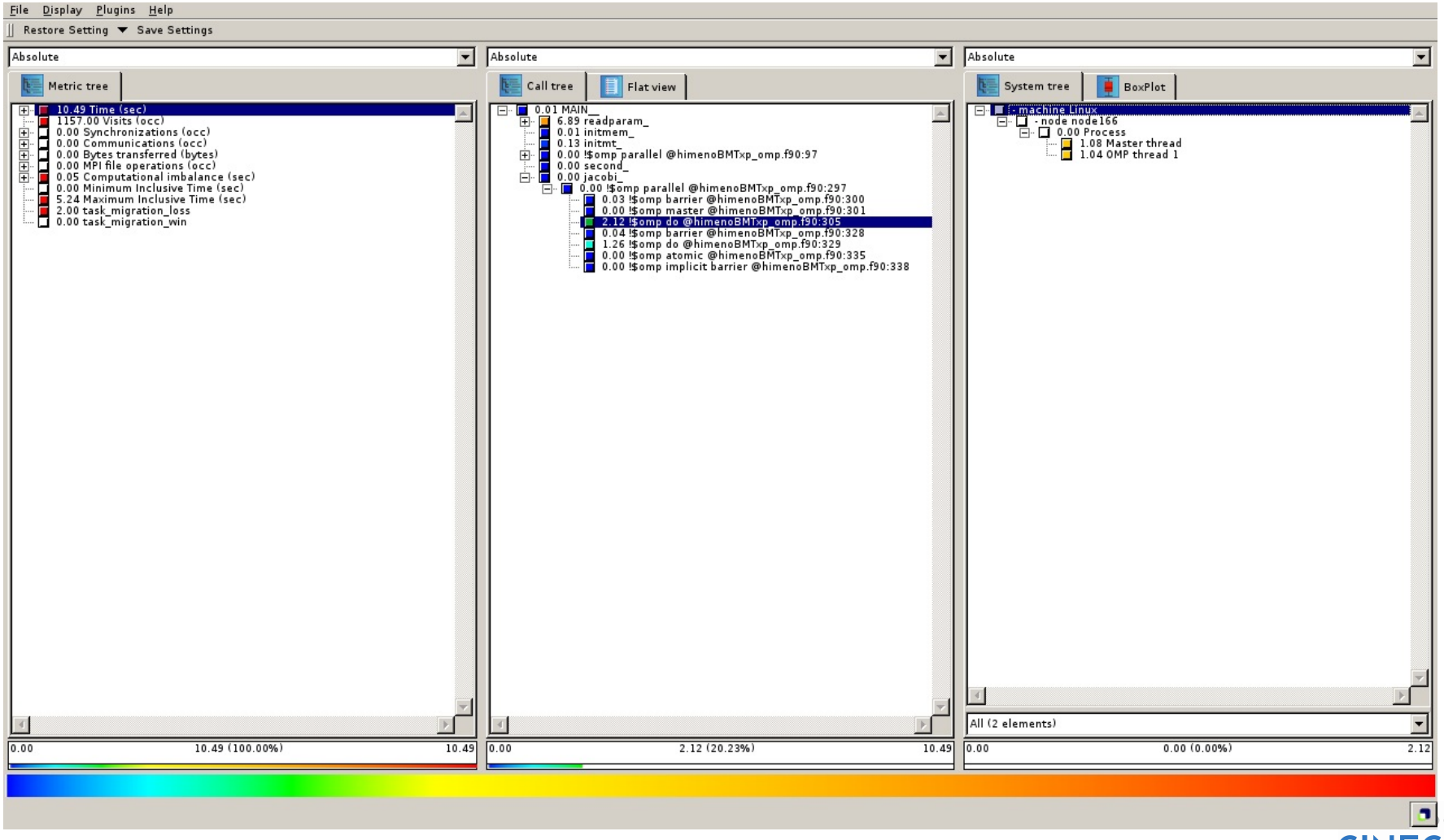
Scalasca: graphic output



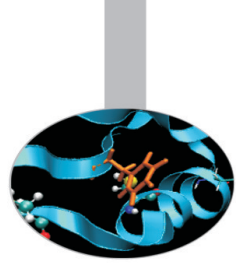


Scalasca: graphic output

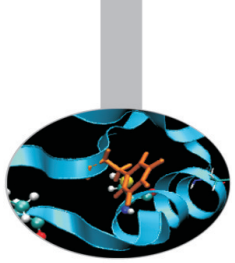




Scalasca: summary vs tracing

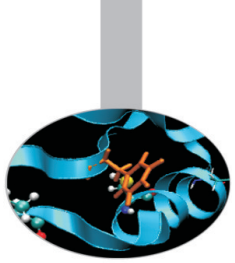


Scalasca: summary vs tracing



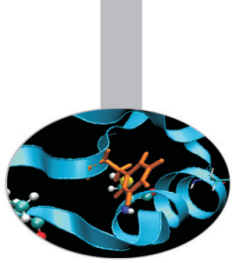
- Both analysis workflows are useful.

Scalasca: summary vs tracing



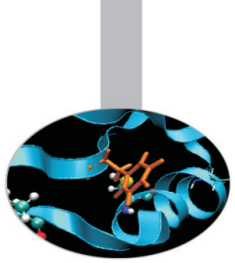
- Both analysis workflows are useful.
  - "summary" it generates aggregated informations (but detailed at a single instruction level)

Scalasca: summary vs tracing



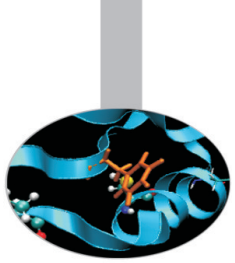
- Both analysis workflows are useful.
  - "summary" it generates aggregated informations (but detailed at a single instruction level)
  - a "tracing", "process–local" analysis (much more informations) that can be require huge amount of "resources"

Scalasca: summary vs tracing



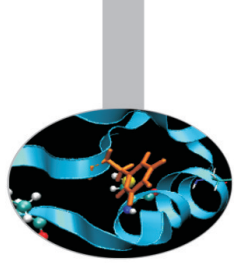
- Both analysis workflows are useful.
  - "summary" it generates aggregated informations (but detailed at a single instruction level)
  - a "tracing", "process–local" analysis (much more informations) that can be require huge amount of "resources"
- the "tracing" analysis report includes metrics that are not available in the "summary" report.

Scalasca: tracing



```
[planucar@node166 SCALASCA]$ export OMP_NUM_THREADS=2;scan -t ./a.out
```

Scalasca: tracing

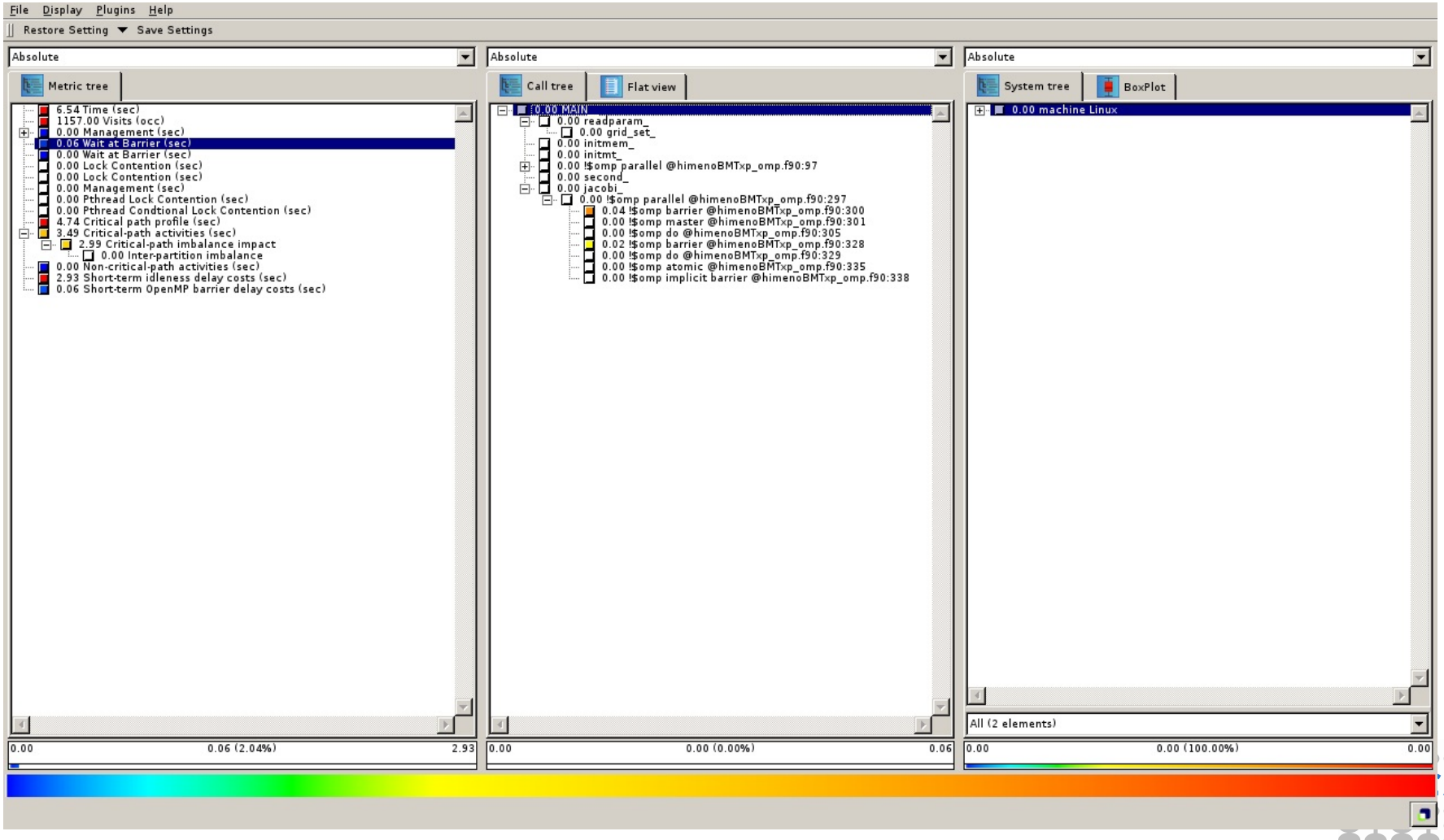


```
[planucar@node166 SCALASCA]$ export OMP_NUM_THREADS=2;scan -t ./a.out
```

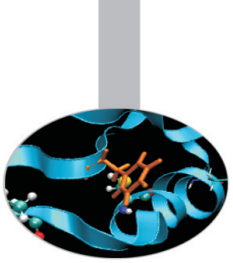
```
S=C=A=N: Scalasca 2.2 trace collection and analysis
S=C=A=N: ./scorep_a_Ox2_trace experiment archive
S=C=A=N: Wed Apr 13 14:47:05 2016: Collect start
./a.out
...
S=C=A=N: Wed Apr 13 14:47:11 2016: Collect done (status=0) 6s
S=C=A=N: Wed Apr 13 14:47:11 2016: Analyze start
/cineca/prod/tools/scalasca/2.2/intelmpi--5.0.2--binary/bin/scout.omp ./scorep_a_Ox2_trace/tra
...
Analyzing experiment archive ./scorep_a_Ox2_trace/traces.otf2
Writing analysis report    ... done (0.034s).
Max. memory usage          : 15.383MB
Total processing time      : 0.070s
S=C=A=N: Wed Apr 13 14:47:11 2016: Analyze done (status=0) 0s
Warning: analyzed trace data retained in ./scorep_a_Ox2_trace/traces!
S=C=A=N: ./scorep_a_Ox2_trace complete.
```

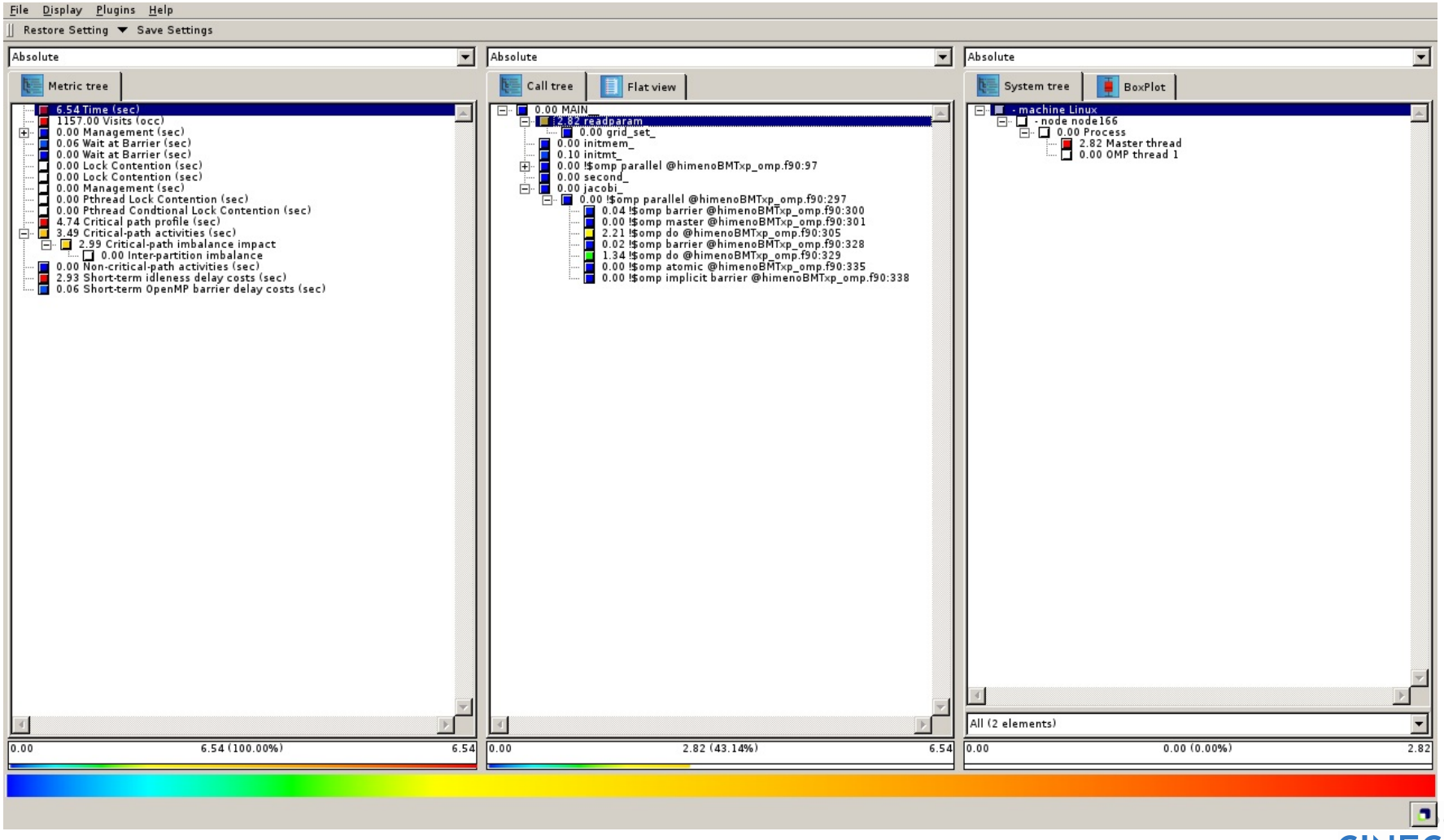
Scalasca: "tracing" analysis

```
cube scorep_a_0x2_trace/
```

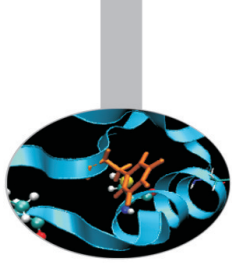


Scalasca: "tracing" analysis

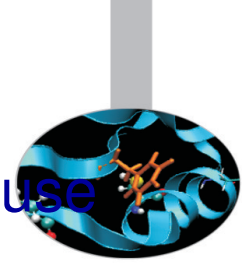




Outline

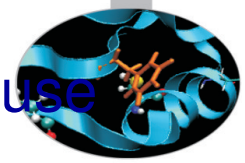


Papi



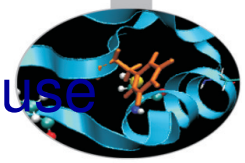
- Gprof and other tools strengths are clearly the simplicity of use and the poor level of intrusivity.

Papi



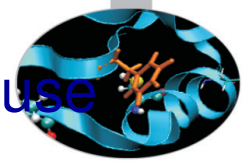
- Gprof and other tools strengths are clearly the simplicity of use and the poor level of intrusivity.
- Often, this is good enough to "win". Sometimes not. This is exactly the case when we want to increase the level of accuracy of the profiling, for example trying to explore techniques which are related to the underlying hardware. This is difficult with standard tools. We need something more accurate.

Papi



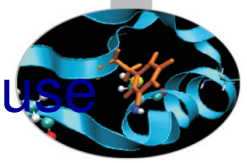
- Gprof and other tools strengths are clearly the simplicity of use and the poor level of intrusivity.
- Often, this is good enough to "win". Sometimes not. This is exactly the case when we want to increase the level of accuracy of the profiling, for example trying to explore techniques which are related to the underlying hardware. This is difficult with standard tools. We need something more accurate.
- PAPI (Performance Application Programming Interface) can be used having in mind the relation between software performance and processor events. Main topics:

Papi



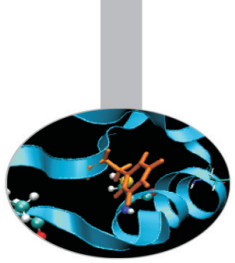
- Gprof and other tools strengths are clearly the simplicity of use and the poor level of intrusivity.
- Often, this is good enough to "win". Sometimes not. This is exactly the case when we want to increase the level of accuracy of the profiling, for example trying to explore techniques which are related to the underlying hardware. This is difficult with standard tools. We need something more accurate.
- PAPI (Performance Application Programming Interface) can be used having in mind the relation between software performance and processor events. Main topics:
  - portability on a huge variety of Linux, Windows,...machines (including recent hardware like GPUs, "accelerators" (Intel MIC), ...)

Papi



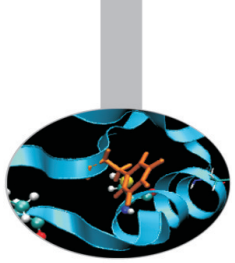
- Gprof and other tools strengths are clearly the simplicity of use and the poor level of intrusivity.
- Often, this is good enough to "win". Sometimes not. This is exactly the case when we want to increase the level of accuracy of the profiling, for example trying to explore techniques which are related to the underlying hardware. This is difficult with standard tools. We need something more accurate.
- PAPI (Performance Application Programming Interface) can be used having in mind the relation between software performance and processor events. Main topics:
  - portability on a huge variety of Linux, Windows,...machines (including recent hardware like GPUs, "accelerators" (Intel MIC), ...)
  - PAPI is based on the use of so-called *Hardware Counters*: "special-purpose" registers built into processor and able to measure a set of "events" occurring during the execution of our program.

Papi-cont.



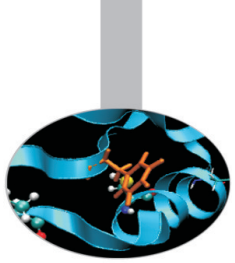
- PAPI is essentially an interface to *Hardware Counters*. We can distinguish two different interfaces:

Papi-cont.



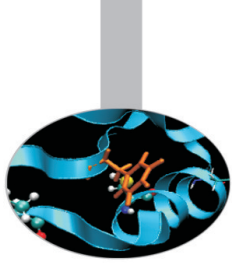
- PAPI is essentially an interface to *Hardware Counters*. We can distinguish two different interfaces:
  - *High level interface*, a set of (high-level) routines able to collect informations from a (pre-defined) list of events (*PAPI Preset Events*)

Papi-cont.



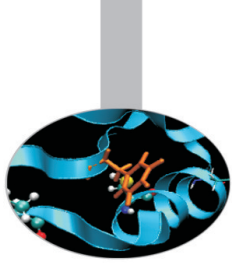
- PAPI is essentially an interface to *Hardware Counters*. We can distinguish two different interfaces:
  - *High level interface*, a set of (high-level) routines able to collect informations from a (pre-defined) list of events (*PAPI Preset Events*)
  - *Low level interface*, which can be used to manage specific hardware events. It is meant for experienced application programmers and tool developers wanting fine-grained measurement and control of the PAPI interface.

Papi-cont.



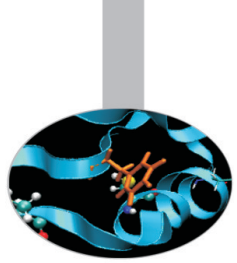
- PAPI is essentially an interface to *Hardware Counters*. We can distinguish two different interfaces:
  - *High level interface*, a set of (high-level) routines able to collect informations from a (pre-defined) list of events (*PAPI Preset Events*)
  - *Low level interface*, which can be used to manage specific hardware events. It is meant for experienced application programmers and tool developers wanting fine-grained measurement and control of the PAPI interface.
- Please, pay attention to the number of *Hardware Counters* available on your machine. This number will return the maximum number of "events" that can be tracked at the same time on this machine.

PAPI Preset Events



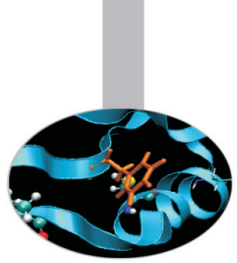
- This set is a collection of events typically found in many CPUs that provide performance counters. A PAPI preset event is something we can always define and use when we want to tune the performance of a given application.

PAPI Preset Events



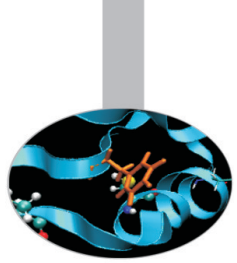
- This set is a collection of events typically found in many CPUs that provide performance counters. A PAPI preset event is something we can always define and use when we want to tune the performance of a given application.
- PAPI define something like "hundreds" of *Preset Events*. For a given platform, a subset of these preset events can be counted. Let's see someone:

PAPI Preset Events



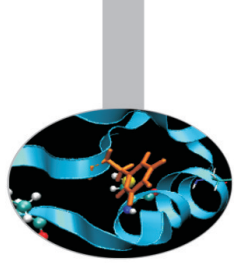
- This set is a collection of events typically found in many CPUs that provide performance counters. A PAPI preset event is something we can always define and use when we want to tune the performance of a given application.
- PAPI define something like "hundreds" of *Preset Events*. For a given platform, a subset of these preset events can be counted. Let's see someone:
  - PAPI_TOT_CYC - number of total cycles

PAPI Preset Events



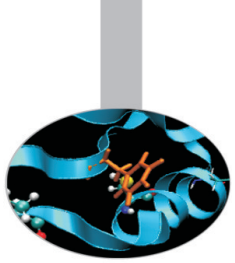
- This set is a collection of events typically found in many CPUs that provide performance counters. A PAPI preset event is something we can always define and use when we want to tune the performance of a given application.
- PAPI define something like "hundreds" of *Preset Events*. For a given platform, a subset of these preset events can be counted. Let's see someone:
  - PAPI_TOT_CYC - number of total cycles
  - PAPI_TOT_INS - number of instructions completed

PAPI Preset Events



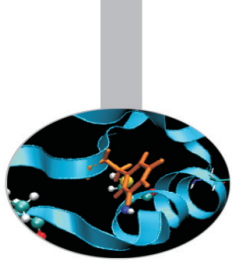
- This set is a collection of events typically found in many CPUs that provide performance counters. A PAPI preset event is something we can always define and use when we want to tune the performance of a given application.
- PAPI define something like "hundreds" of *Preset Events*. For a given platform, a subset of these preset events can be counted. Let's see someone:
  - PAPI_TOT_CYC - number of total cycles
  - PAPI_TOT_INS - number of instructions completed
  - PAPI_FP_INS - floating-point instructions

PAPI Preset Events



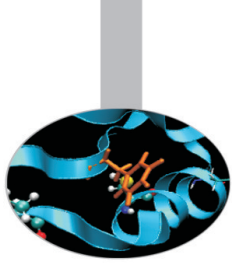
- This set is a collection of events typically found in many CPUs that provide performance counters. A PAPI preset event is something we can always define and use when we want to tune the performance of a given application.
- PAPI define something like "hundreds" of *Preset Events*. For a given platform, a subset of these preset events can be counted. Let's see someone:
  - PAPI_TOT_CYC - number of total cycles
  - PAPI_TOT_INS - number of instructions completed
  - PAPI_FP_INS - floating-point instructions
  - PAPI_L1_DCA - L1 cache accesses

PAPI Preset Events



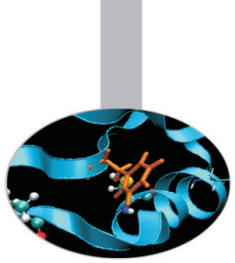
- This set is a collection of events typically found in many CPUs that provide performance counters. A PAPI preset event is something we can always define and use when we want to tune the performance of a given application.
- PAPI define something like "hundreds" of *Preset Events*. For a given platform, a subset of these preset events can be counted. Let's see someone:
  - PAPI_TOT_CYC - number of total cycles
  - PAPI_TOT_INS - number of instructions completed
  - PAPI_FP_INS - floating-point instructions
  - PAPI_L1_DCA - L1 cache accesses
  - PAPI_L1_DCM - L1 cache misses

PAPI Preset Events



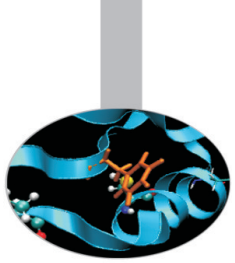
- This set is a collection of events typically found in many CPUs that provide performance counters. A PAPI preset event is something we can always define and use when we want to tune the performance of a given application.
- PAPI define something like "hundreds" of *Preset Events*. For a given platform, a subset of these preset events can be counted. Let's see someone:
  - PAPI_TOT_CYC - number of total cycles
  - PAPI_TOT_INS - number of instructions completed
  - PAPI_FP_INS - floating-point instructions
  - PAPI_L1_DCA - L1 cache accesses
  - PAPI_L1_DCM - L1 cache misses
  - PAPI_SR_INS - store instructions

PAPI Preset Events



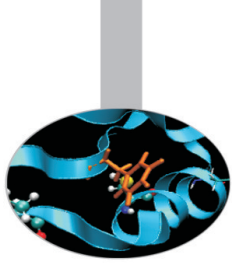
- This set is a collection of events typically found in many CPUs that provide performance counters. A PAPI preset event is something we can always define and use when we want to tune the performance of a given application.
- PAPI define something like "hundreds" of *Preset Events*. For a given platform, a subset of these preset events can be counted. Let's see someone:
  - PAPI_TOT_CYC - number of total cycles
  - PAPI_TOT_INS - number of instructions completed
  - PAPI_FP_INS - floating-point instructions
  - PAPI_L1_DCA - L1 cache accesses
  - PAPI_L1_DCM - L1 cache misses
  - PAPI_SR_INS - store instructions
  - PAPI_TLB_DM - TLB misses

PAPI Preset Events



- This set is a collection of events typically found in many CPUs that provide performance counters. A PAPI preset event is something we can always define and use when we want to tune the performance of a given application.
- PAPI define something like "hundreds" of *Preset Events*. For a given platform, a subset of these preset events can be counted. Let's see someone:
  - PAPI_TOT_CYC - number of total cycles
  - PAPI_TOT_INS - number of instructions completed
  - PAPI_FP_INS - floating-point instructions
  - PAPI_L1_DCA - L1 cache accesses
  - PAPI_L1_DCM - L1 cache misses
  - PAPI_SR_INS - store instructions
  - PAPI_TLB_DM - TLB misses
  - PAPI_BR_MSP - conditional branch mispredicted

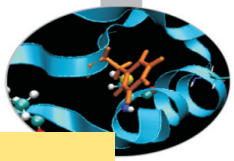
Papi: C and Fortran interface



Calls to the high-level API are sufficiently clear. Furthermore, is always possible to call PAPI APIs from C and Fortran sources (even if PAPI is natively written in C).
Fortran example:

```
#include "fpapi_test.h"
...  ; integer events(2), retval   ;   integer*8 values(2)
...  ;
events(1) = PAPI_FP_INS     ;   events(2) = PAPI_L1_DCM
...
        call PAPIf_start_counters(events, 2, retval)
        call PAPIf_read_counters(values, 2, retval)  ! Clear values
             [sezione di codice da monitorare]
        call PAPIfstop_counters(values, 2, retval)
           print*,'Floating point instructions:  ',values(1)
           print*,' L1 Data Cache Misses:   ',values(2)
...
```

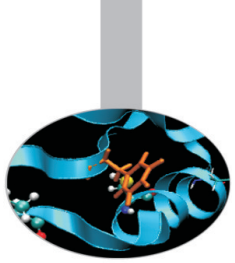
Papi: C and Fortran interface



C example:

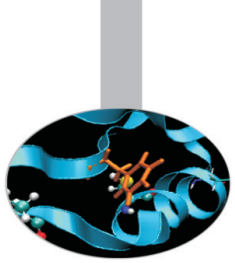
```
#include <stdio.h>
#include <stdlib.h>
#include "papi.h"

#define NUM_EVENTS 2
#define THRESHOLD 10000
#define ERROR_RETURN(retval) { fprintf(stderr, "Error %d %s:line %d: \n",
retval,__FILE__,__LINE__);  exit(retval); }
...
/* stupid codes to be monitored */
void computation_add()
   ....
int main()
{
   int Events[2] = {PAPI_TOT_INS, PAPI_TOT_CYC};
   long long values[NUM_EVENTS];
...
if ( (retval = PAPI_start_counters(Events, NUM_EVENTS)) != PAPI_OK)
       ERROR_RETURN(retval);
   printf("\nCounter Started: \n");
if ( (retval=PAPI_read_counters(values, NUM_EVENTS)) != PAPI_OK)
      ERROR_RETURN(retval);
   printf("Read successfully\n");
   computation_add();
if ( (retval=PAPI_stop_counters(values, NUM_EVENTS)) != PAPI_OK)
      ERROR_RETURN(retval);
   printf("Stop successfully\n");
   printf("The total instructions executed for addition are %lld \n",values[0]);
   printf("The total cycles used are %lld \n", values[1] );
}
```

PAPI: high-level functions



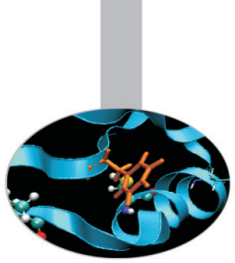
- A small set of routines which can be used to "instrument" a program. The functions are:

PAPI: high-level functions



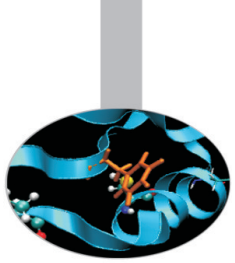
- A small set of routines which can be used to "instrument" a program. The functions are:
  - PAPI_num_counters - ritorna il numero di hw counters disponibili

PAPI: high-level functions



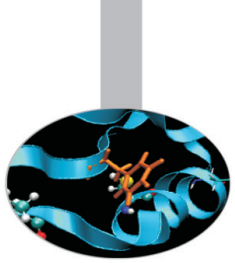
- A small set of routines which can be used to "instrument" a program. The functions are:
  - PAPI_num_counters - ritorna il numero di hw counters disponibili
  - PAPI_flips - floating point instruction rate

PAPI: high-level functions



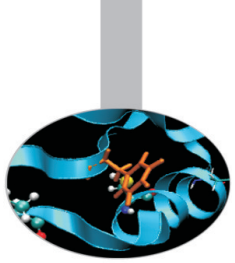
- A small set of routines which can be used to "instrument" a program. The functions are:
  - PAPI_num_counters - ritorna il numero di hw counters disponibili
  - PAPI_flips - floating point instruction rate
  - PAPI_flops - floating point operation rate

PAPI: high-level functions



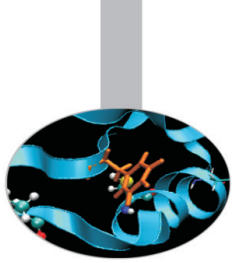
- A small set of routines which can be used to "instrument" a program. The functions are:
  - PAPI_num_counters - ritorna il numero di hw counters disponibili
  - PAPI_flips - floating point instruction rate
  - PAPI_flops - floating point operation rate
  - PAPI_ipc - instructions per cycle and time

PAPI: high-level functions



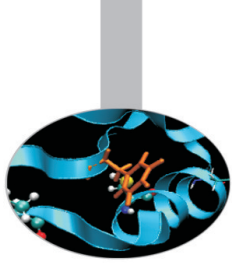
- A small set of routines which can be used to "instrument" a program. The functions are:
  - PAPI_num_counters - ritorna il numero di hw counters disponibili
  - PAPI_flips - floating point instruction rate
  - PAPI_flops - floating point operation rate
  - PAPI_ipc - instructions per cycle and time
  - PAPI_accum_counters

PAPI: high-level functions



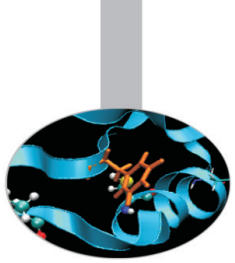
- A small set of routines which can be used to "instrument" a program. The functions are:
  - PAPI_num_counters - ritorna il numero di hw counters disponibili
  - PAPI_flips - floating point instruction rate
  - PAPI_flops - floating point operation rate
  - PAPI_ipc - instructions per cycle and time
  - PAPI_accum_counters
  - PAPI_read_counters - read and reset counters

PAPI: high-level functions



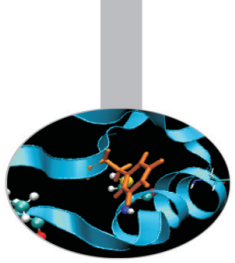
- A small set of routines which can be used to "instrument" a program. The functions are:
  - PAPI_num_counters - ritorna il numero di hw counters disponibili
  - PAPI_flips - floating point instruction rate
  - PAPI_flops - floating point operation rate
  - PAPI_ipc - instructions per cycle and time
  - PAPI_accum_counters
  - PAPI_read_counters - read and reset counters
  - PAPI_start_counters - start counting hw events

PAPI: high-level functions



- A small set of routines which can be used to "instrument" a program. The functions are:
  - PAPI_num_counters - ritorna il numero di hw counters disponibili
  - PAPI_flips - floating point instruction rate
  - PAPI_flops - floating point operation rate
  - PAPI_ipc - instructions per cycle and time
  - PAPI_accum_counters
  - PAPI_read_counters - read and reset counters
  - PAPI_start_counters - start counting hw events
  - PAPI_stop_counters - stop counters return current counts

PAPI: hands-on

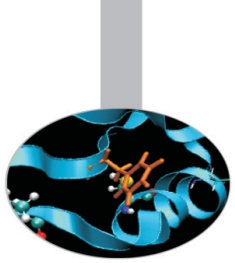


- Profiling using PAPI API of a simple serial code using BLAS library for implementing "linear-algebra" kernels.
  - The analysis follows the next steps:
  - Go-to the hpc-forge.cineca.it link:

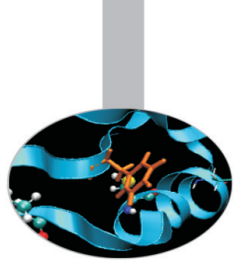
```
tar xvfz PAPI_Profiling_exercise.tar.gz
cd PAPI
```

  - analyse README file
  - run the benchmarks
  - Please, make your comments on the results obtained using PAPI

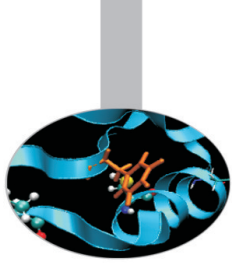
Outline



The "art" of Profiling...

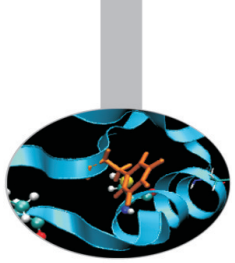


The "art" of Profiling...



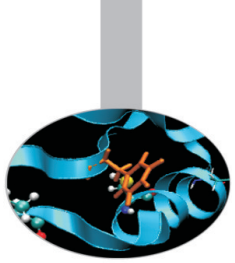
- the lesson does not pretend to be exhaustive or conclusive...

The "art" of Profiling...



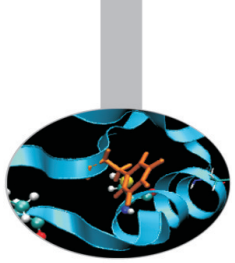
- the lesson does not pretend to be exhaustive or conclusive...
- ...there is a "pletora" of Profiling tools...and much more to say of the presented Profiling tools!

The "art" of Profiling...



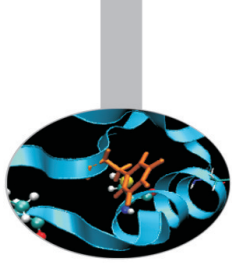
- the lesson does not pretend to be exhaustive or conclusive...
- ...there is a "pletora" of Profiling tools...and much more to say of the presented Profiling tools!
- some "practical" suggestions:

The "art" of Profiling...



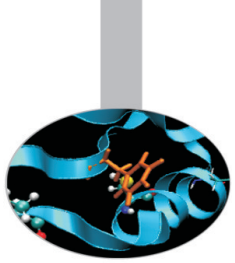
- the lesson does not pretend to be exhaustive or conclusive...
- ...there is a "pletora" of Profiling tools...and much more to say of the presented Profiling tools!
- some "practical" suggestions:
  - use always more than one test case. Try to activate each part of a complex application

The "art" of Profiling...



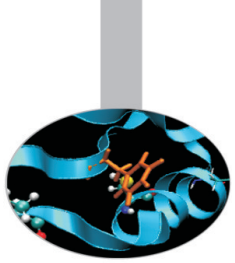
- the lesson does not pretend to be exhaustive or conclusive...
- ...there is a "pletora" of Profiling tools...and much more to say of the presented Profiling tools!
- some "practical" suggestions:
  - use always more than one test case. Try to activate each part of a complex application
  - use always "realistic" test cases to profile your application

The "art" of Profiling...



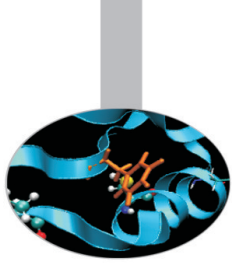
- the lesson does not pretend to be exhaustive or conclusive...
- ...there is a "pletora" of Profiling tools...and much more to say of the presented Profiling tools!
- some "practical" suggestions:
  - use always more than one test case. Try to activate each part of a complex application
  - use always "realistic" test cases to profile your application
  - use always different data sizes for your problem

The "art" of Profiling...



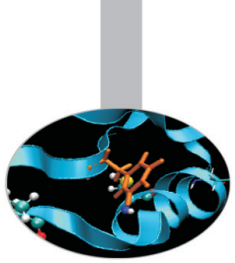
- the lesson does not pretend to be exhaustive or conclusive...
- ...there is a "pletora" of Profiling tools...and much more to say of the presented Profiling tools!
- some "practical" suggestions:
  - use always more than one test case. Try to activate each part of a complex application
  - use always "realistic" test cases to profile your application
  - use always different data sizes for your problem
  - pay attention to input/output

The "art" of Profiling...



- the lesson does not pretend to be exhaustive or conclusive...
- ...there is a "pletora" of Profiling tools...and much more to say of the presented Profiling tools!
- some "practical" suggestions:
  - use always more than one test case. Try to activate each part of a complex application
  - use always "realistic" test cases to profile your application
  - use always different data sizes for your problem
  - pay attention to input/output
  - use more than one profiling tool (trying to refine a previous analysis)

The "art" of Profiling...



- the lesson does not pretend to be exhaustive or conclusive...
- ...there is a "pletora" of Profiling tools...and much more to say of the presented Profiling tools!
- some "practical" suggestions:
  - use always more than one test case. Try to activate each part of a complex application
  - use always "realistic" test cases to profile your application
  - use always different data sizes for your problem
  - pay attention to input/output
  - use more than one profiling tool (trying to refine a previous analysis)
  - use, when possible, different architectures.