University of Sofia
IRC–CoSiM Project

Performance evaluation and optimisation of scientific codes

Hristo Iliev
Monte Carlo Research Group
hristo<at>phys.uni-sofia.bg

Second IRC–CoSiM Workshop, Gyuletchitsa 15–18 October 2009

1. Overview
2. Performance estimation...
3. ... and optimisation
4. Simple optimisations
5. Advanced techniques
6. Case study

Science vs. desktop

- Desktop apps are about **productivity**
  - Funny interactive GUIs
  - Document processing, WEB surfing
  - Complex data structures & complex algorithms but not so many data (~ MiBs)
  - Better ways to manage and represent data
- Scientific apps are about **performance**
  - Scary configuration files
  - Batch execution
  - Simple data structures & simple algorithms* but HUGE amount of data (~ TiBs)
  - Better ways to process the data

* but not necessarily simple in implementation, esp. numerical algorithms

Performance

- Work per unit time
  - Measured in **<u>fl</u>oating point <u>o</u>perations <u>p</u>er <u>s</u>econd** (flops), not in Watts
  - Other units for specific applications:
    - triangles/vertices per second (GPUs)
    - frames per second (video processing)
    - MiB/GiB per second (data processing)
    - simulations per day
    - etc.
- Benchmarks
  - Synthetic tests that measure specific (sub-)system's performance in a comparative way
  - *"Mine FPU is better than yours"*

1. Overview
2. Performance estimation...
3. ... and optimisation
4. Simple optimisations
5. Advanced techniques
6. Case study

LINPACK

- Standard linear algebra benchmark
  - Solves dense $\mathbf{A} \cdot \boldsymbol{x} = \boldsymbol{b}$ in single or double precision floating point numbers
  - Matrix diagonalisation and matrix-vector multiplication
  - $\frac{2}{3} \cdot N^3 + 2 \cdot N^2$ operations where $N = \dim(\mathbf{A})$
  - R_{peak} – peak (theoretical) performance
    - Intel Xeon E5420: Rpeak = 4 cores · 4 flops/cycle · 2.5 Gcycles/sec = 40 Gflops
  - R_{max} – sustained performance
  - N_{max} – $\dim(\mathbf{A})$ to achieve R_{max}
- HPL
  - Parallel implementation of LINPACK
  - Top500.org

LINPACK drawbacks

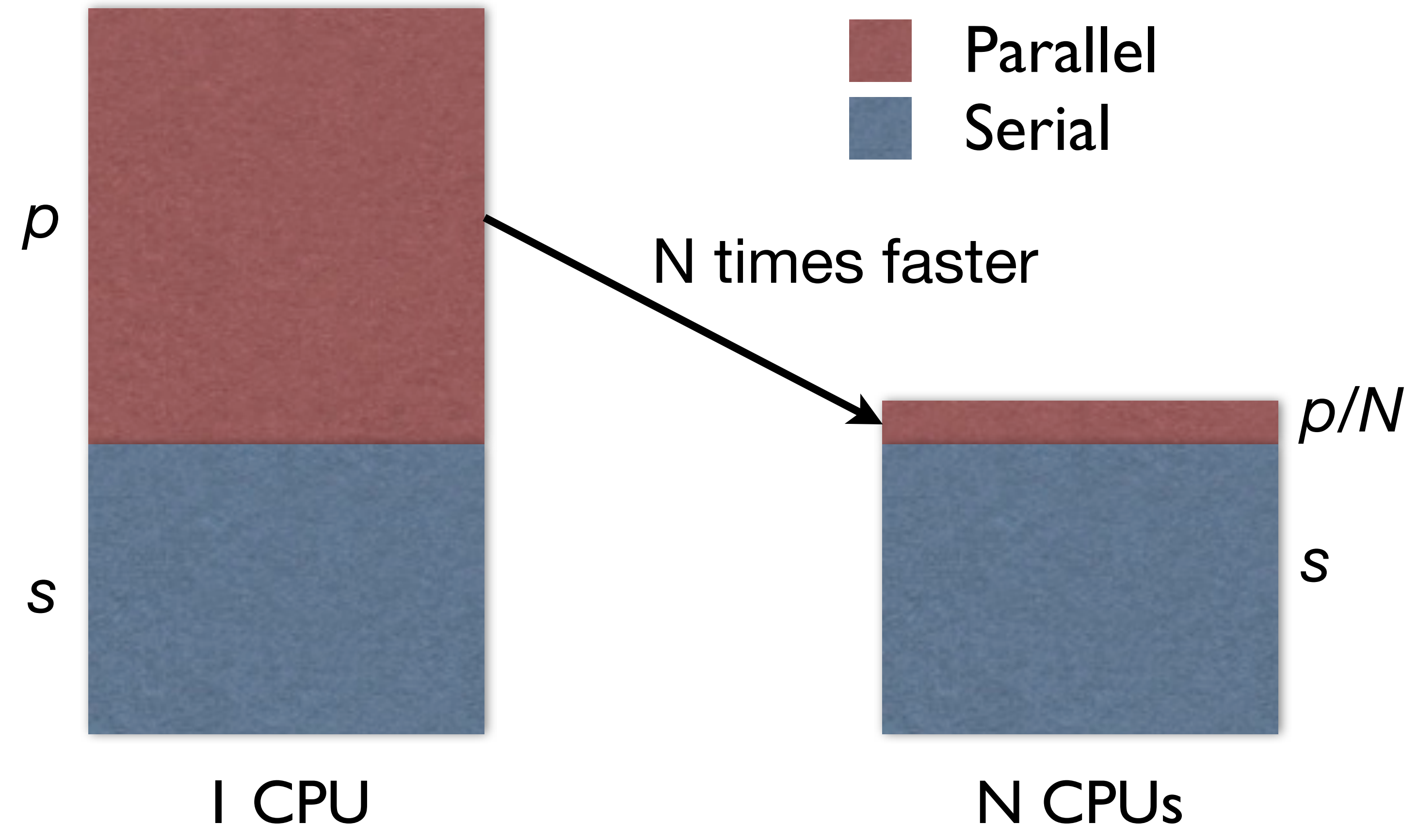
- Only simple **vector math** operations
- Results highly dependent on dim(**A**)
- No transcendental operations used
- Beware!
  - High LINPACK score doesn't always mean high overall computing speed
  - Computer vendors often abuse and/or tweak benchmark results
  - Example: nVidia Tesla C1060 GPGPU
    - 933 Gflops (peak) for IEEE 754 **single precision** numbers
    - **78 Gflops** (peak) for IEEE 754 **double precision** numbers (highly understated in press releases)
    - Thank goodness many scientific codes can run in single precision

Program scaling

- Performance vs. problem size
  - Highly **architecture dependent**
  - Small problems fit in CPU cache (L2 or L3)
  - Memory is the **bottleneck** at large problem sizes
- Performance vs. CPU count
  - Amdahl’s law
- Good to know your program’s scaling
  - Test runs with varying problem size
  - Vary the CPU count (for parallel apps only)
  - Plot it to get the picture!
  - Choose wisely!

Amdahl's law

- Limits the **parallel speed-up**



- Speed-up = $1/(s+p/N) = N/[1+(N-1)s]$
- Maximum speedup = $1/s$
  - More CPUs adds to *s* when global synchronisation is involved

The economist view

- Price for running on N CPUs
  - Price = $T_{\mathrm{CPU}} \cdot \$/\mathrm{hr}$
  - $T_{\mathrm{CPU}} = N \cdot T_{\mathrm{run}}$
  - $T_{\mathrm{run}} = T_1 \cdot [1+(N-1)s]/N$
  - Price = $\mathrm{Price}_1 \cdot [1+(N-1)s] \geq \mathrm{Price}_1$
- Best scenario
  - $s = 0$
  - Price stays the same, but runtime is N times shorter
- Worst scenario
  - $s = 1$
  - Price is N times higher for no gain in runtime
- Usually we are somewhere in between

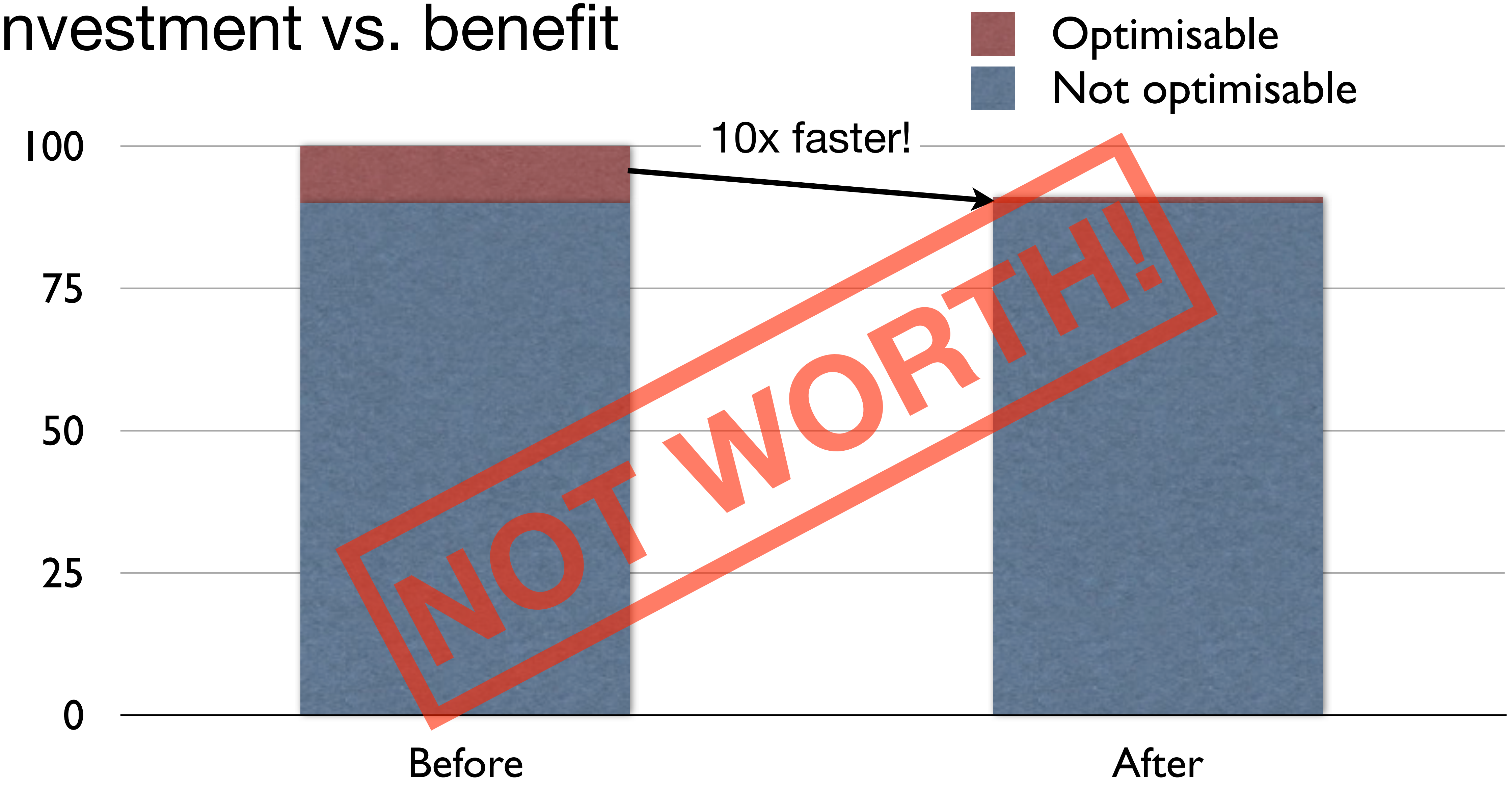
1. Overview
2. Performance estimation...
3. ... and optimisation
4. Simple optimisations
5. Advanced techniques
6. Case study

Optimisation

- Improving program’s performance on the **same hardware**
- No programming involved
  - Better compiler
  - Better libraries
  - Reduce problem size (better/simpler models)
- Programming involved
  - Better algorithms
  - Different data representation
  - Different data alignment
  - Remove redundant code

The big question

- Is it worth?
  - Faster programs vs. longer life when you're on your own
  - Investment vs. benefit



Overall speed-up: 1.1x

1. Overview
2. Performance estimation...
3. ... and optimisation
4. Simple optimisations
5. Advanced techniques
6. Case study

The simplest one

If it works, don’t mess with it!

Change the compiler

- Different vendors
  (a.k.a. *“anything but GCC”*)

gfortran 4.2.4 (-O3 -msse3 -mfpmath=sse -march=nocona -static)	1.00x
ifort 11.1 (-O3 -xSSE4.1 -static)	1.67x
sunf90 8.4 (-xO3 -xarch=sse4_1 -xcache=32/64/8:6144/64/24 \ -xchip=penryn -dalign -fsimple=2 -fns=yes -ftrap=common -xlibmil \ -xlibmopt -nofstore -xregs=frameptr -xvector=simd -Bstatic)	1.43x

- Newer versions (sometimes) perform better

ifort 10.0 (-O3 -xT -ipo -static)	1.00x
ifort 11.0 (-O3 -xSSE4.1 -ipo -static)	1.13x
ifort 11.1 (-O3 -xSSE4.1 -ipo -static)	1.13x

Know your options

- Many compiler options affect performance, but most require programmer's knowledge
  - Use register arguments (breaks profiling)
  - Use function inlining (ditto)
  - Static linkage gives a few % faster code
  - Specify correct cache properties (e.g. to Sun Studio)
  - Enable omission of frame pointers (breaks profiling)
  - Enable extended processor instructions
  - Enable vectorisation (3DNow!, SSE2, SSE3, etc.)
- **Beware: Optimisations can break unstable (numerical) codes!!!**

Link with better libs

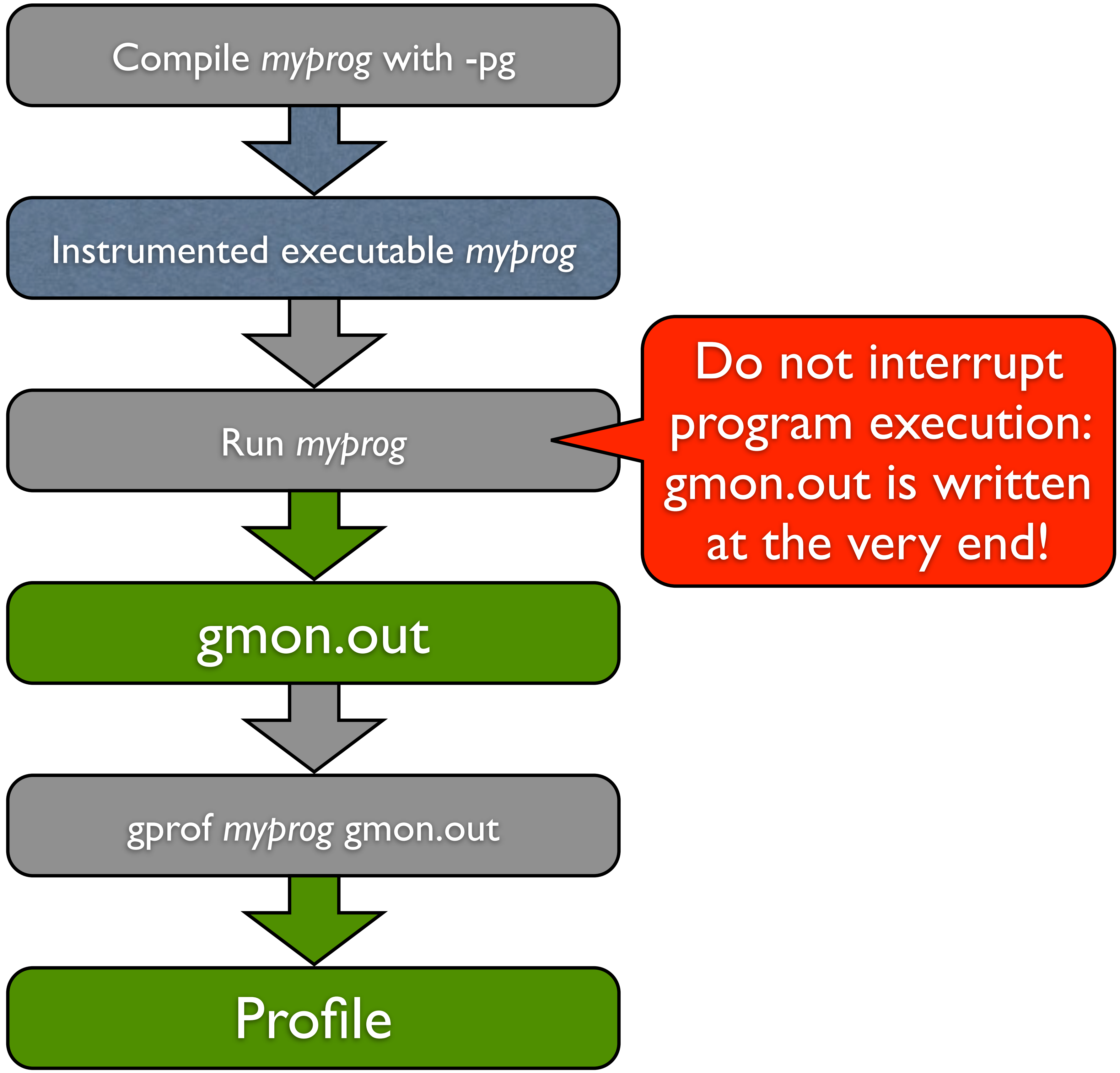
- Vendor libraries are usually better than generic versions
- Intel
  - Intel Performance Primitives (vector ops)
  - Intel Math Kernel Library (BLAS, LAPACK, FFT)
- Sun
  - Sun Performance Library
- Generic (but still fast)
  - ATLAS
  - FFTW
- Most software automatically recognises and uses vendor libraries

1. Overview
2. Performance estimation...
3. ... and optimisation
4. Simple optimisations
5. Advanced techniques
6. Case study

Profiling

- Profilers profile your **instrumented code**
  - Can include statistical sampling
  - **Instruction pointer** sampling (what's running now?)
  - **Call stack** recording (who called who?)
  - Much more informative when **debug info** is present (gives familiar function names in the output rather than obscure addresses)
- Requires compiler support
  - GCC: `-p` (prof) or `-pg` (gprof)
  - Sun: `-p` (prof) or `-xpg` (gprof)
  - Intel: `-p` (gprof)
- Beware of the optimisation!

Profiling workflow



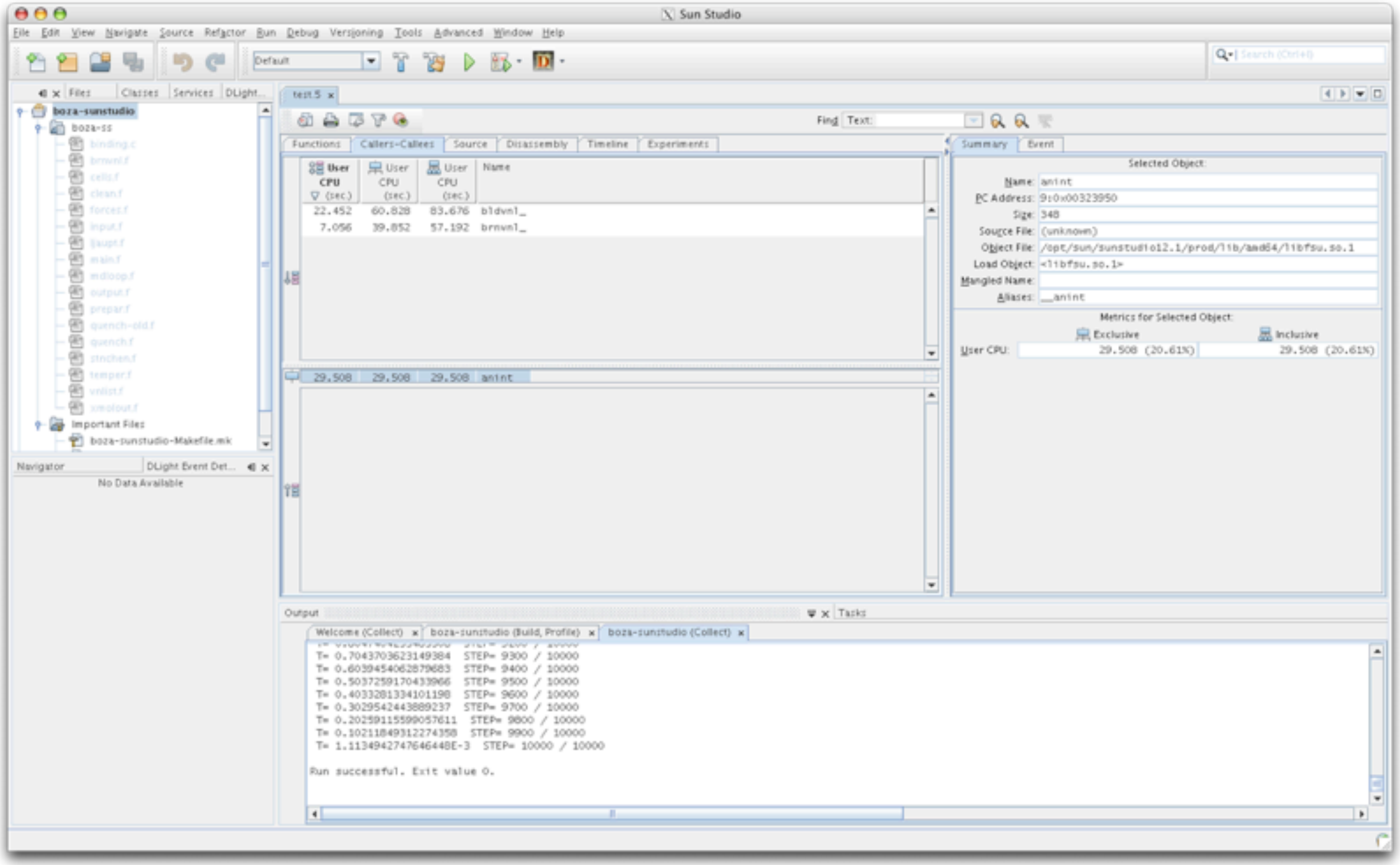
Compile *myprog* with -pg
Instrumented executable *myprog*
Run *myprog*
gmon.out
gprof *myprog* gmon.out
Profile
Do not interrupt program execution: gmon.out is written at the very end!

gprof output

- Flat profile
  - Total time spent in each function
  - Number of calls to each function
  - Time per call
  - Self times
- Call graph profile
  - Time spent in each child function
  - Number of calls to each child from the current one
  - Total number of calls to each child
- Compiler optimisations and function inlining may result in weird output!

Alternatives to gprof

- Hey, it’s XXI century. We’ve got Windows, and Macs, and Java, and stuff!


Sun Studio
File Edit View Navigate Source Refactor Run Debug Versioning Tools Advanced Window Help
Files Classes Services DLight...
boza-sunstudio
boza-ss
Important Files
boza-sunstudio-Makefile.mk
Navigator
DLight Event Det...
No Data Available
test.5
Find Text:
Functions Callers-Callees Source Disassembly Timeline Experiments
Summary Event
Output
Tasks
Welcome (Collect)
boza-sunstudio (Build, Profile)
boza-sunstudio (Collect)

User CPU (sec.)	User CPU (sec.)	User CPU (sec.)	Name
22.452	60.828	83.676	bldvnl_
7.056	39.852	57.192	brnvnl_
29.508	29.508	29.508	anint

Selected Object:

Name:	anint
PC Address:	9:0x00323950
Size:	348
Source File:	(unknown)
Object File:	/opt/sun/sunstudio12.1/prod/lib/amd64/libfsu.so.1
Load Object:	<libfsu.so.1>
Mangled Name:	
Aliases:	__anint

Metrics for Selected Object:

	Exclusive	Inclusive
User CPU:	29.508 (20.61%)	29.508 (20.61%)

```
 T= 0.7043703623149384  STEP= 9300 / 10000
 T= 0.6039454062879683  STEP= 9400 / 10000
 T= 0.5037259170433966  STEP= 9500 / 10000
 T= 0.4033281334101198  STEP= 9600 / 10000
 T= 0.3029542443889237  STEP= 9700 / 10000
 T= 0.20259115599057611  STEP= 9800 / 10000
 T= 0.10211849312274358  STEP= 9900 / 10000
 T= 1.1134942747646448E-3  STEP= 10000 / 10000

Run successful. Exit value 0.
```

Sun Studio

- Free C/C++/Fortran IDE from Sun
  - Written in Java, of course, thus kind of slow ;)
- Available for Solaris and Linux
- GUI plug-ins that wrap command-line tools
- Project D-Light
  - Interface to the **D-Trace** toolkit
  - Omnipotent **system wide profiling**
  - Scripts written in D
  - Many Solaris components provide D-Trace hooks
- collect/analyzer
  - gprof on steroids
  - Can trace threads, synchronisation and **MPI** calls

Basic performance principles

- Data locality
  - Spatial – group related data structures together in memory, do not scatter them
  - Temporal – use variables as soon as possible after their value is computed
- Streams are good
  - Streaming data is a good candidate for vectorisation
  - Streams play nice with prefetching
- Simple data structures
  - Use pointers only when necessary
  - Pointers confuse code optimisers
- It's all about the **loops**

Data locality

- CPU cache is copied from/to memory in "lines" (64 bytes in modern x86)
  - **Spatial locality** maximises the chance that related data parts are in the same cache line
- Each CPU has a limited number of very fast registers
  - **Temporal locality** maximises the chance that variables stay in CPU registers and not in main memory

Nested loops

- When nesting loops make the one with most iterations the innermost

good:
```
DO i = 1,10
  DO j = 1,100000
  ...
  END DO !j
END DO !i
```

bad:
```
DO i = 1,100000
  DO j = 1,10
  ...
  END DO !j
END DO !i
```

- Mind how multidimensional arrays are laid in memory (spatial locality!)

good:
```
DO j = 1,100
  DO i = 1,100
    a(i,j) = ...
  END DO !i
END DO !j
```

bad:
```
DO i = 1,100
  DO j = 1,100
    a(i,j) = ...
  END DO !j
END DO !i
```

Innermost loops

- Put as much work as possible in innermost loops
- Small innermost loops (2 to 4 steps) can be **unrolled**:

```
DO d = 1,3
  R(d,num) = ...
END DO
```

$\Rightarrow$

```
R(1,num) = ...
R(2,num) = ...
R(3,num) = ...
```

- Most compilers automatically unroll loops in higher optimisation levels
- Small loops with no interdependencies can be vectorised – better don't unroll by hand

IRC CoSiM

Conditionals

- Conditionals are enemy to the performance
- Conditional statements inside tight loops are **performance killers**
- Conditionals on global flags inside loops are insanity

```
DO i = 1,1000
  IF (flag) THEN
    statements 1
  ELSE
    statements 2
  END IF
END DO
```

```
IF (flag) THEN
  DO i = 1,1000
    statements 1
  END DO
ELSE
  DO i = 1,1000
    statements 2
  END DO
END IF
```

Cache trashing

- Simultaneous access to memory blocks that map on the same cache line

```
A(:) = B(:) + C(:)
```

  - Causes continuous cache reloads from main memory
  - Modern CPUs have highly associative L2 and L3 caches to prevent most cache trashes

- Artificial padding can help reduce trashing

```
REAL A(1024)
REAL B(1024)
REAL C(1024)
```

```
REAL A(1024)
REAL PADA(16)
REAL B(1024)
REAL PADB(16)
REAL C(1024)
```

An “obvious” one

- Skip unnecessary data initialisation
  - No need to clear variables that are assigned to later
  - Split loops where variable values are accumulated in each step

```
a = 0.0
...
DO i = 1,1000
  a = a + i**2
END DO
```

```
a = 1**2
DO i = 2,1000
  a = a + i**2
END DO
```

- Initialisation of large arrays on each computation step can be very time consuming!

1. Overview
2. Performance estimation...
3. ... and optimisation
4. Simple optimisations
5. Advanced techniques
6. Case study

BOZA

- Molecular Dynamics code for simulations of carbon and carbon–metal system
- Brenner's potential – naive $O(N^3)$ alogrithm
- Optimisations
  - Linked cells + Verlet neighbour list
  - Removed unnecessary initialisations of large arrays
  - Reduced the number of conditionals
  - Reordered some loops
  - Better compiler options
  - Better compilers
- Net result: ~40x speed-up

Acknowledgements

The author acknowledges the financial support of the National Scientific Research Fund under contract DO-02-136/2008 (IRC-CoSiM project).

Thank you for your attention
and
have a pleasant dinner time!