Application behaviour/Profiling/tuning

Ole W. Saastad, Dr.Scient
USIT / UAV / FI / FT
April 18th 2016
NORBIS - HPC in Bioinformatics

Introduction

- Learn about how your application is using the resources
- Memory, CPU, Operating system, I/O and more
- Learn about how to understand what it is doing

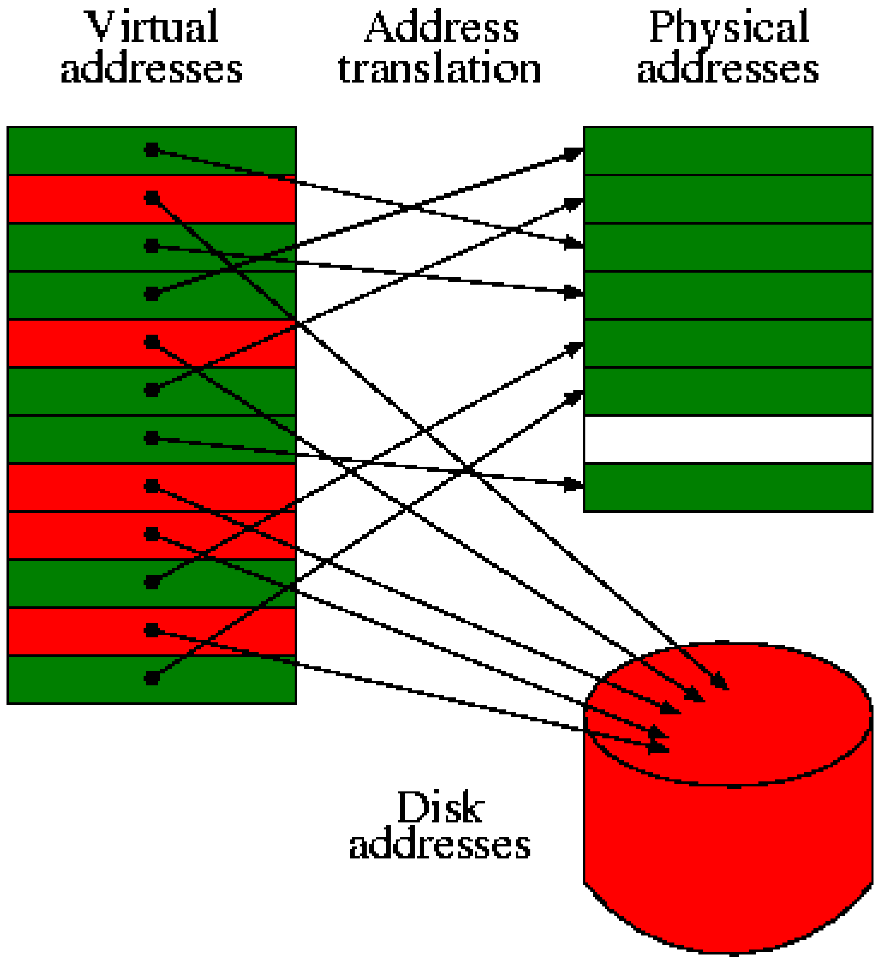
Memory

- All application uses memory
- How much and how does it use memory ?
- How much memory do I need to request ?

Memory

- Linux-based operating systems use a virtual memory system.
- Any address referenced by a user-space application must be translated into a physical address. This is achieved through a combination of page tables and address translation hardware in the underlying computer system.

Virtual memory



Time spent and memory

- Timing is a key in performance tuning
- /usr/bin/time
- Man time

Time spent and memory

- Default format gives:
  %Uuser
  %Ssystem
  %Eelapsed
  %PCPU (%Xtext+%Ddata %Mmax)k
  %Iinputs+%Ooutputs
  (%Fmajor+%Rminor)pagefaults
  %Wswaps

Time spent and memory

- %Uuser
  - Total number of CPU-seconds that the process spent in user mode.

  %Ssystem
  - Total number of CPU-seconds that the process spent in kernel mode.

  %Eelapsed
  - Elapsed real time

    (in [hours:]minutes:seconds).

Time spent and memory

- %PCPU (%Xtext+%Ddata %Mmax)k
  - Percentage of the CPU that this job got, computed as (%U + %S) / %E.

  %Iinputs+%Ooutputs
  - Number of file system inputs by the process.

  (%Fmajor+%Rminor)pagefaults

  %Wswaps

Time spent and memory

- %Fmajor pagefaults
  - Number of major page faults that occurred while the process was running. These are faults where the page has to be read in from disk.

Time spent and memory

- %Rminor pagefaults
  - Number of minor, or recoverable, page faults. These are faults for pages that are not valid but which have not yet been claimed by other virtual pages. Thus the data in the page is still valid but the system tables must be updated.

Time spent and memory

- %Wswaps
  - Number of times the process was swapped out of main memory.

Time spent and memory

- A Gaussian 09 job :

```
2597.79user 42.46system 44:06.57elapsed 99%CPU
(0avgtext+0avgdata 0maxresident)k
0inputs+0outputs (0major+2957369minor)pagefaults 0swaps
```

  - 2598 s user time
  - 43 s system time
  - 2647 s wall clock time
  - 2957369 minor page faults
  - Bug in older kernels, 0 for memory

Time spent and memory

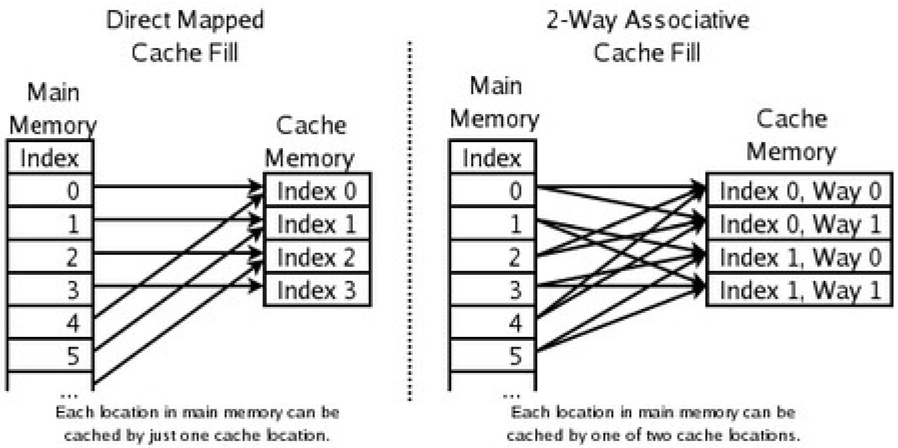
- An evolution run :

```
6.60user 0.60system 1:24.57elapsed 8%CPU
(0avgtext+0avgdata 183984maxresident)k
85312inputs+328outputs (375major+13602minor)pagefaults 0swaps
```

  - Maximum resident memory : 180 MB
  - Kernel : 2.6.32-27-server (Ubuntu)
  - Still a bug, zero for text and data

Memory hierarchy / cache

- cache and cache misses
  - Cache is not ideal
  - Not fully associative

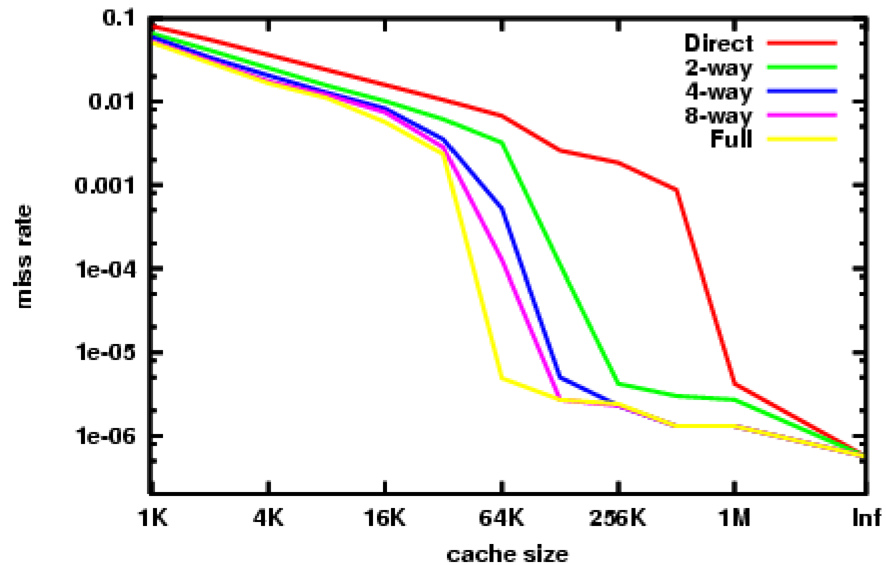


Memory, cache and cache misses

- With 100% cache hit memory access would be 10-20 ns, vs. 0% hit rate yield 100-300 ns.

- Cache misses are costly / painful

- Not easy to spot
  - Need tools
  - Intel Amplifier



Memory and page faults

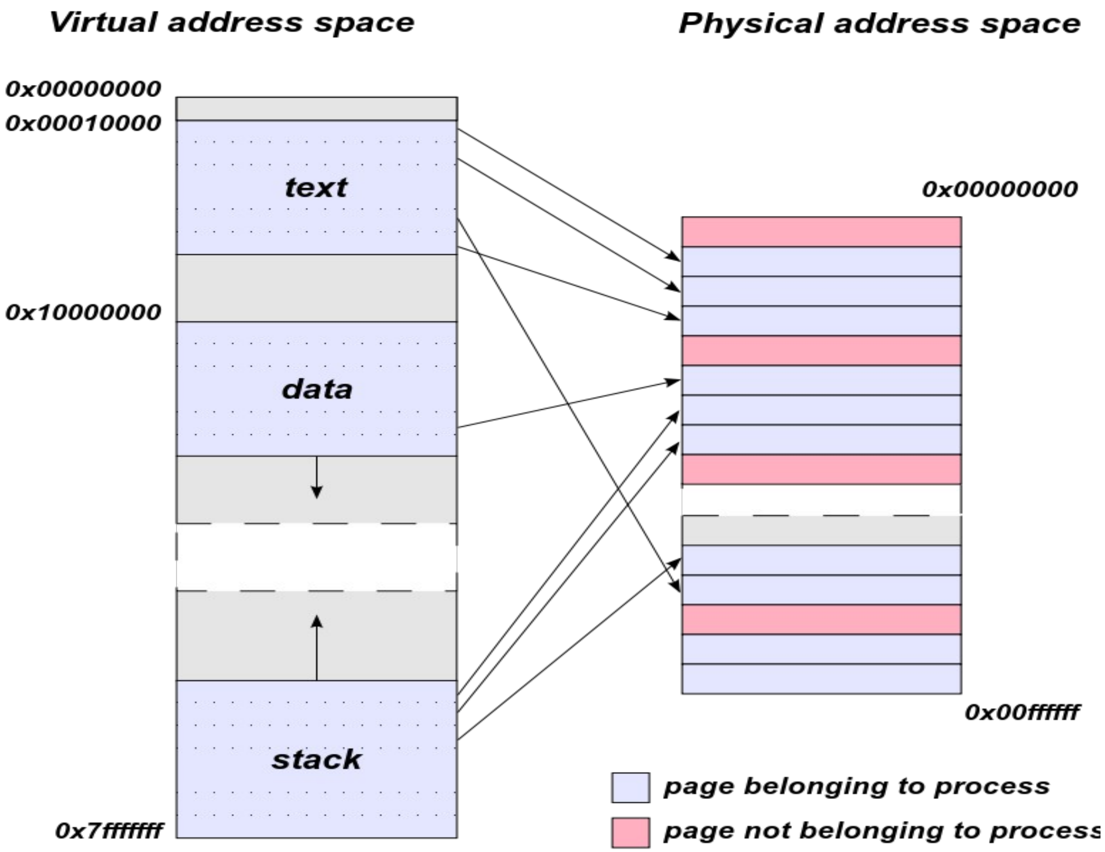
- Major and minor page faults
  - Remember output from /usr/bin/time ?

```
2597.79user 42.46system 44:06.57elapsed 99%CPU
(0avgtext+0avgdata 0maxresident)k
0inputs+0outputs (0major+2957369minor)pagefaults 0swaps
```

Minor page fault

- Minor page faults are unavoidable
- Page is in memory, just a memory admin by OS
- Little cost
- A huge number of these is often a sign of ineffective programming

Virtual to physical mapping



Virtual to Physical

- Page tables
  - TLB is a cache of the page table
  - TLB miss / minor page fault
    - TLB hit time ½ – 1 clock cycle
    - TLB miss time 10 – 100 clock cycles

- How do we avoid TLB misses ?
  - Read about Goto's BLAS libs.

Memory and page faults

- Major page faults means to little memory for your working set!

- Showstopper !

- Avoid at all cost, slows down your system to halt

- Submit job on a node/system with more memory!

Memory footprint

- "top" is your friend, */usr/bin/time is currently failing*
- Man top will give help
- top is the mostly used tool for memory monitoring

Memory footprint

```
top - 14:25:07 up 5 days, 9 min,  3 users,  load average: 32.86, 24.14, 11.82
Tasks: 427 total,   3 running, 424 sleeping,   0 stopped,   0 zombie
Cpu(s):100.0%us,  0.0%sy,  0.0%ni,  0.0%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
Mem:  264353776k total, 260044844k used,  4308932k free,   186688k buffers
Swap: 25165812k total,        0k used, 25165812k free,   624240k cached

PID USER    PR  NI  VIRT  RES  SHR S %CPU  %MEM   TIME+    COMMAND
13857 olews 25   0  246g 245g  768 R 3197  97.5  198:10.13 stream.large.x
 8573 olews 15   0 12848 1336  800 R    1   0.0   0:10.22  top
```

Memory footprint

```
Mem:  264353776k total, 260044844k used,  4308932k free,   186688k buffers
Swap: 25165812k total,         0k used, 25165812k free,   624240k cached
```

Mem: Total amount of installed physical memory on System in this case 252 GigaBytes

Of which 248 GigaByes are used by user processes 180 MB is used by buffers and 4.11 GB are free

Swap: None of the swap memory is used

Memory footprint

```
  PID USER     PR  NI  VIRT  RES  SHR S %CPU  %MEM    TIME+    COMMAND
13857 olews 25   0  246g 245g  768 R 3197  97.5  198:10.13 stream.large.x
 8573 olews 15   0 12848 1336  800 R    1   0.0    0:10.22  top
```

My application is using :

Virtual memory 246 GigaBytes
Resident memory : 245 GigaBytes

Memory footprint

- VIRT - Virtual memory

  The total amount of virtual memory used by the task. It includes all code, data and shared libraries plus pages that have been swapped out.

  This is the total footprint of everything, and can be regarded as maximum possible size.

Memory footprint

- RSS - Resident memory

  The non-swapped physical memory a task has used.
  RES = CODE + DATA

  When a page table entry has been assigned a physical address, it is referred to as the resident working set.

  This the important number, this is what is actually used. Large unused data segment might be swapped out and not reside in physical memory.

Memory footprint

- CODE
  – The amount of physical memory devoted to executable code

- DATA
  – The amount of physical memory devoted to other than executable code

These two are not standard, read man page

Memory footprint

- SHR - Shared memory

  The amount of shared memory used by a task. It simply reflects memory that could be potentially shared with other processes.

  Normally quite small and little to worry about

More on 'top'

- PID
  - User Process Identification Number
- USER
  - User name
- PR
  - Priority
- NI
  - Nice value, A negative nice value means higher priority, a positive value means lower priority.

More on 'top'

- S – Status which can be one of :
  - 'D' = uninterruptible sleep
  - 'R' = running
  - 'S' = sleeping
  - 'T' = traced or stopped
  - 'Z' = zombie

D is not good at all, it means disk wait
Z is very bad, they cannot be killed, even by kill -9

More on 'top'

- % CPU
  - The task's share of the elapsed CPU time since the last screen update, expressed as a percentage of total CPU time.
- % MEM
  - A task's currently used share of available physical memory
- TIME
  -
- COMMAND

More on 'top'

TIME

– Total CPU time the task has used since it started.

- COMMAND
  – Display the command line used to start a task or the name of the associated program.

Advanced 'top'

- nFLT
  - The number of major page faults that have occurred for a task.
- nDRT
  - The number of pages that have been modified since they were last written to disk.

To see these fields use the 'f' and 'o' interactive commands, read top man page

Open files

- Which files have my application opened ?
  - /usr/sbin/lsof is your friend
  - lsof lists all open files
  - lsof
    - Options for selecting output
    - Type of files
    - User
    - Many many more , read man page

Open files - lsof examples

- lsof -u olews

```
COMMAND    PID  USER   FD   TYPE    DEVICE  SIZE      NODE NAME
xterm     9118 olews  cwd       DIR       0,28    32768   37087057
/xanadu/home/olews/benchmark/Stream
xterm     9118 olews  rtd       DIR        8,1     4096          2 /
xterm     9118 olews  txt       REG        8,1   351464    2850820 /usr/bin/xterm
xterm     9118 olews  mem       REG        8,1   134400    1995244 /lib64/ld-2.5.so
xterm     9118 olews  mem       REG        8,1  1699880    1995245 /lib64/libc-2.5.so
xterm     9118 olews  mem       REG        8,1    23360    1995247 /lib64/libdl-2.5.so
```

This is only a selection of the quite long output. Tools like grep etc can be handy, but lsof has many options for selectiong output.

Open files - lsof examples

- lsof -c more -a -u olews -d 0-1000

```
COMMAND    PID  USER    FD    TYPE    DEVICE   SIZE       NODE NAME
more     12316 olews    0u    CHR    136,76                 78 /dev/pts/76
more     12316 olews    1u    CHR    136,76                 78 /dev/pts/76
more     12316 olews    2u    CHR    136,76                 78 /dev/pts/76
more     12316 olews    3r    REG      0,28 15088     374577 /xanadu/home/
                                           olews/benchmark/Stream/stream.large.f
```

This is a selection of the output, just for the utility 'more', only regular files are reported. We want to know what happen to the open fortran source code file.

File IO – what's going on ?

- Open files
  - Found using lsof
- IO Operations on open files
  - OS calls
- How to look into operations on the open files ?
  - Trace OS calls

Trace OS calls

- /usr/bin/strace
  - strace is a useful diagnostic, instructional, and debugging tool.
  - strace logs and print out OS calls
  - Many options, read man page

What can strace do ?

- Remember open files and 'more ....' ?

```
COMMAND    PID   USER    FD    TYPE    DEVICE   SIZE       NODE NAME
more     12316 olews     3r    REG       0,28 15088    374577 /xanadu/home/
                                                 olews/benchmark/Stream/stream.large.f
```

- We can trace the PID 12316 and follow it's OS calls
- strace -p 12316

What can strace do ?

- strace -p 12316

```
Process 12316 attached - interrupt to quit
read(2, " ", 1)                        = 1
write(1, "\r            avgtime(j) = avgt"..., 52) = 52
write(1, "           mintime(j) = min(m"..., 54) = 54
write(1, "           maxtime(j) = max(m"..., 54) = 54

write(1, "...back 1 page\n", 15)        = 15
lseek(3, 15088, SEEK_SET)              = 15088
write(1, "\r           t = mysecond()\n", 26) = 26
```

We see that filedescriptor 3 is our source code file
This is the open file we want to investigate

Using strace

- Monitor read, write, lseek,
- Providing information about record size written or read
  - "`read(2, " ", 1)                                    = 1`"
  - One byte/character (a space) is read from unit 2

Using strace

- Providing information about sequential read or random read.
  - A lot of seek is usually a sign of random disk IO.
  - Read record size is of great importance

“`read(2, " ", 1)                                       = 1`”

Here we have a record size of 1, anything less of a megabyte or so might be random read.

Random read is your enemy !

- With small blocks even more so !

Random read is your enemy !

- A disk can do about 150 IO operations per sec

- At record size 1 M this is 150 MB/s – more than the disk can do anyway => random == sequential

- At record size 4k this is 600 kB/s => grind to a standstill ! This is just a Showstopper!

How to spot Random read ?

- strace yield a lot of seek and read with small record sizes

```
/usr/sbin/lsof -c l906.exe  -a -u olews -d 0-1000
l906.exe 17888 olews    4u   REG   8,17 9396224    49159 /work/Gau-17888.rwf

strace -p 17888
lseek(4, 1571680, SEEK_SET)            = 1571680
read(4, "\2\0\0\0\0\0\0\0\2\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0"..., 46240) = 46240
lseek(4, 348160, SEEK_SET)             = 348160
read(4, "\32\22X\252\372\376\377?\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0"..., 440) = 440
lseek(4, 1675264, SEEK_SET)            = 1675264
read(4, "\1\0\0\0\0\0\0\0\2\0\0\0\0\0\0\0\1\0\0\0\0\0\0\0\1\0\0\0\0\0\0\0"..., 240) = 240
lseek(4, 1662976, SEEK_SET)            = 1662976
read(4, "\2\0\0\0\0\0\0\0\2\0\0\0\0\0\0\0000\0\0\0\0\0\0\0000\0\0\0\0\0\0\0"..., 416) = 416
lseek(4, 1646592, SEEK_SET)            = 1646592
read(4, "\1\0\0\0\0\0\0\0\2\0\0\0\0\0\0\0\3\0\0\0\0\0\0\0\4\0\0\0\0\0\0\0"..., 1536) = 1536
lseek(4, 1671168, SEEK_SET)            = 1671168
read(4, "\1\0\0\0\0\0\0\0\2\0\0\0\0\0\0\0\3\0\0\0\0\0\0\0\4\0\0\0\0\0\0\0"..., 480) = 480
```

iostat – disk monitoring

- /usr/bin/iostat
- Monitor disk usage
- Good for spotting disk bottlenecks
- Part of the sysstat package
- Can be quite complicated !

iostat – disk monitoring

```
avg-cpu:  %user   %nice %system %iowait  %steal   %idle
           2.90    0.00    0.29    0.47    0.00   96.34

Device:         rrqm/s   wrqm/s   r/s   w/s    rMB/s    wMB/s avgrq-sz avgqu-sz   await  svctm  %util
sdb               0.00  6250.20  0.00 54.30     0.00    24.63   928.85    11.47  211.28   2.34  12.70
```

iostat – disk monitoring

- Rrqm, wrqm
  - The number of read or write requests merged per second that were queued to the device.

- r/s, w/s
  - The number of r or w requests that were issued to the device per second.

iostat – disk monitoring

- rMB/s, wMB/s
  - The number of megabytes read or written from or from the device per second.

- avgrq-sz
  - The average size (in sectors) of the requests that were issued to the device.

iostat – disk monitoring

- avgqu-sz
  - The average queue length of the requests that were issued to the device.
- await
  - The average time (in milliseconds) for I/O requests issued to the device to be served. This includes the time spent by the requests in queue and the time spent servicing them.

iostat – disk monitoring

- svctm
  - The average service time (in milliseconds) for I/O requests that were issued to the device.
- %util
  - Percentage of CPU time during which I/O requests were issued to the device (bandwidth utilization for the device). Device saturation occurs when this value is close to 100%.

iostat – what to look for ?

avg time that each request spent in queue (qtime) = await – svctime

In our example : 211.28 ms – 2.32 ms = 209 ms

avg time that each request spent being serviced = 2.32 ms

so averagely each IO request spent 211.28 ms to be processed of which 209 ms were spent just waiting in queue

iostat – what to look for ?

(await-svctim)/await*100: The percentage of time that IO operations spent waiting in queue in comparison to actually being serviced. If this figure goes above 50% then each IO request is spending more time waiting in queue than being processed.

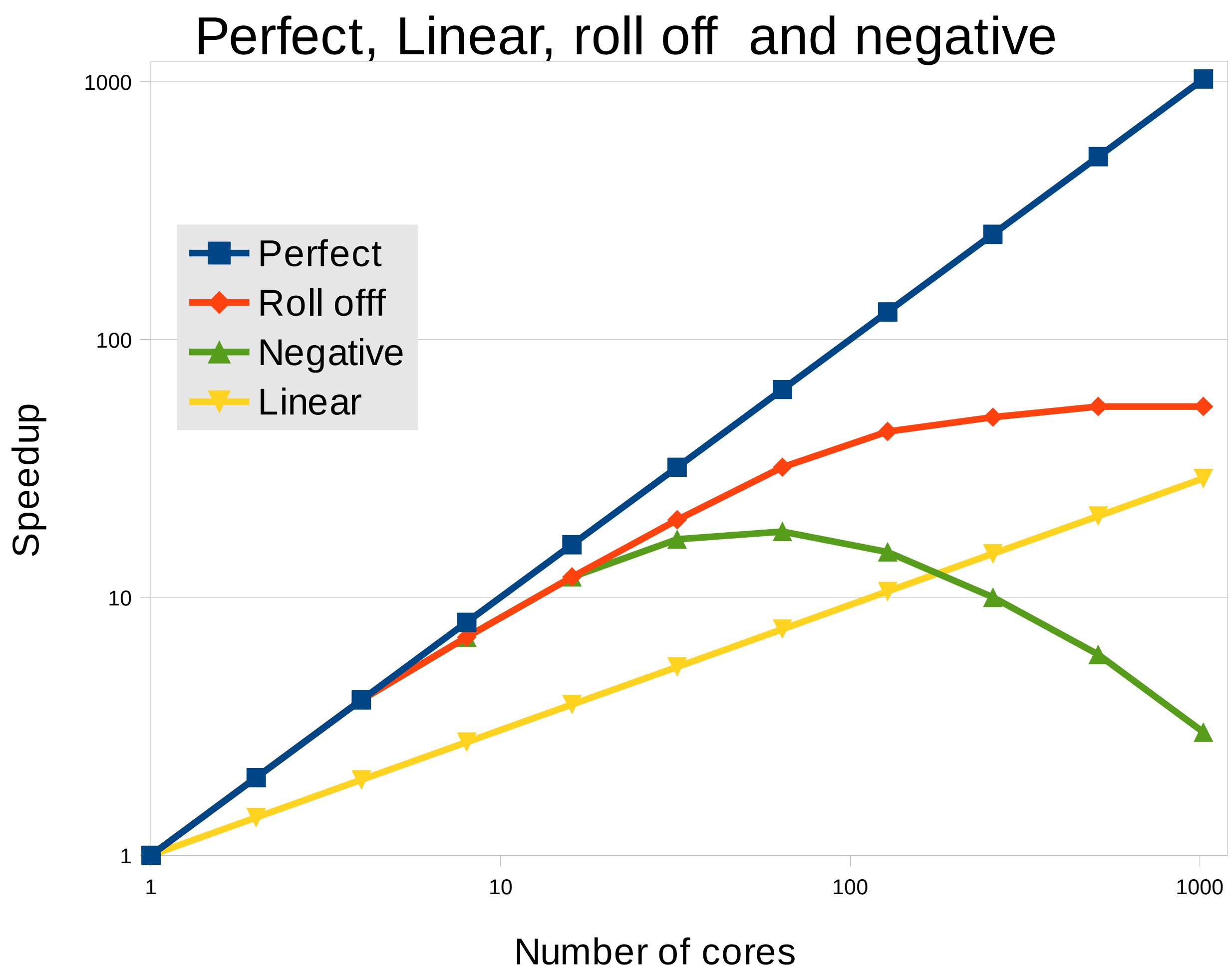
The figure in the await column should be as close to that in the svctim column as possible. If await goes much above svctim, watch out! The IO device is probably overloaded.

We have : await = 211 ms and svctim 2.32 ms
Not very good !

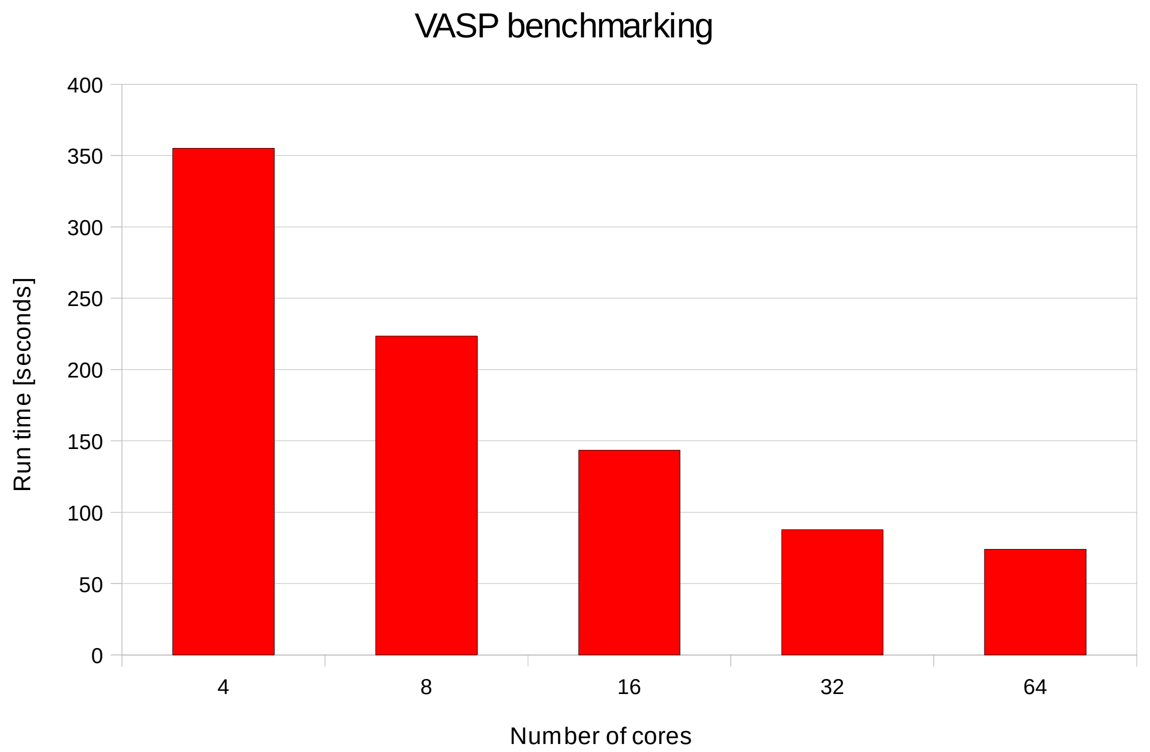
Scaling - Speedup

- Run time and core count
- Wall time and cpu time
- Always use wall time for timing
- Record wall time and core count

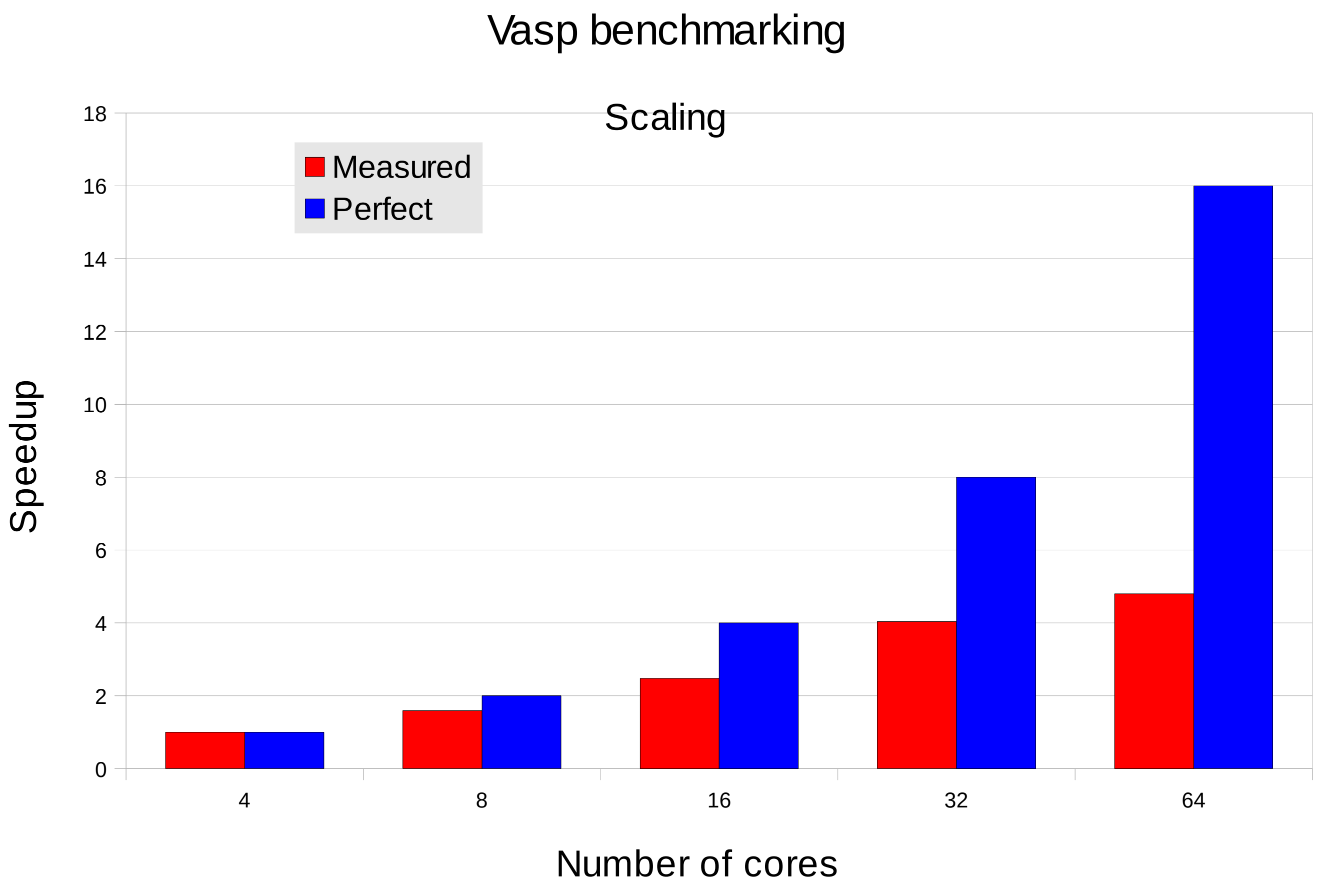
Scaling – types of



Scaling – wall / run time



Scaling – speedup



Scaling

- Perfect scaling
  - Scale to a tuning point
  - Where does it roll off
  - Can it scale to very large core count ?
  - 1 million cores ?

Scaling

- Linear scaling
  - What is the angle of the scaling line ?
  - Scale to a tuning point
  - Where does it roll off
  - Can it scale to very large core count ?
  - 1 million cores ?

Scaling - Poor scaling

- Roll off or negative
  - Can a better interconnect help ?
  - Maybe you can rewrite application or change algorithm ?
  - Another application ?
  - Get help !

iostat – what to look for ?

We can change the IO scheduler, the default (cfg) one is optimzed for a single spinning disk.
If we have a disk RAID and controller the cfg is not optimal.

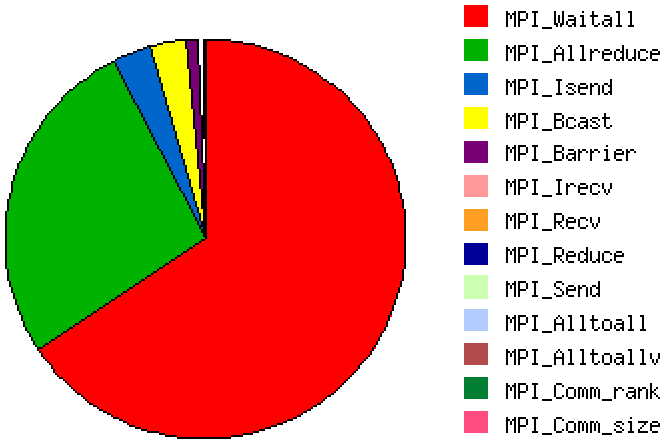
For RAID no scheduler is needed, this is the controllers job!

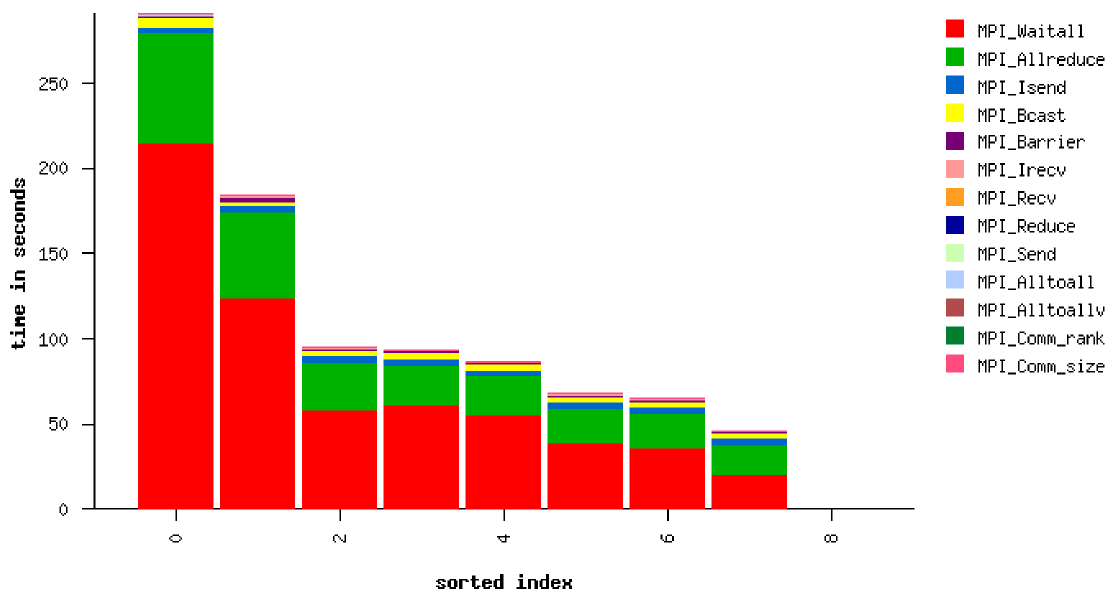
```
echo noop > /sys/block/sdb/queue/scheduler
```

Integrated Performance Monitoring

- MPI
  - communication topology and statistics for each MPI call and buffer size
- Memory
  - wallclock, user and system timings

IPM – Vasp using 8 cpurs





PAPI - instruction level profiling

- Detailed CPU profiling of application
- CPU counters for events like Instructions, cache misses, Stalls etc.
- A very detailed application profile can be obtained

- We focus on the easy ones !

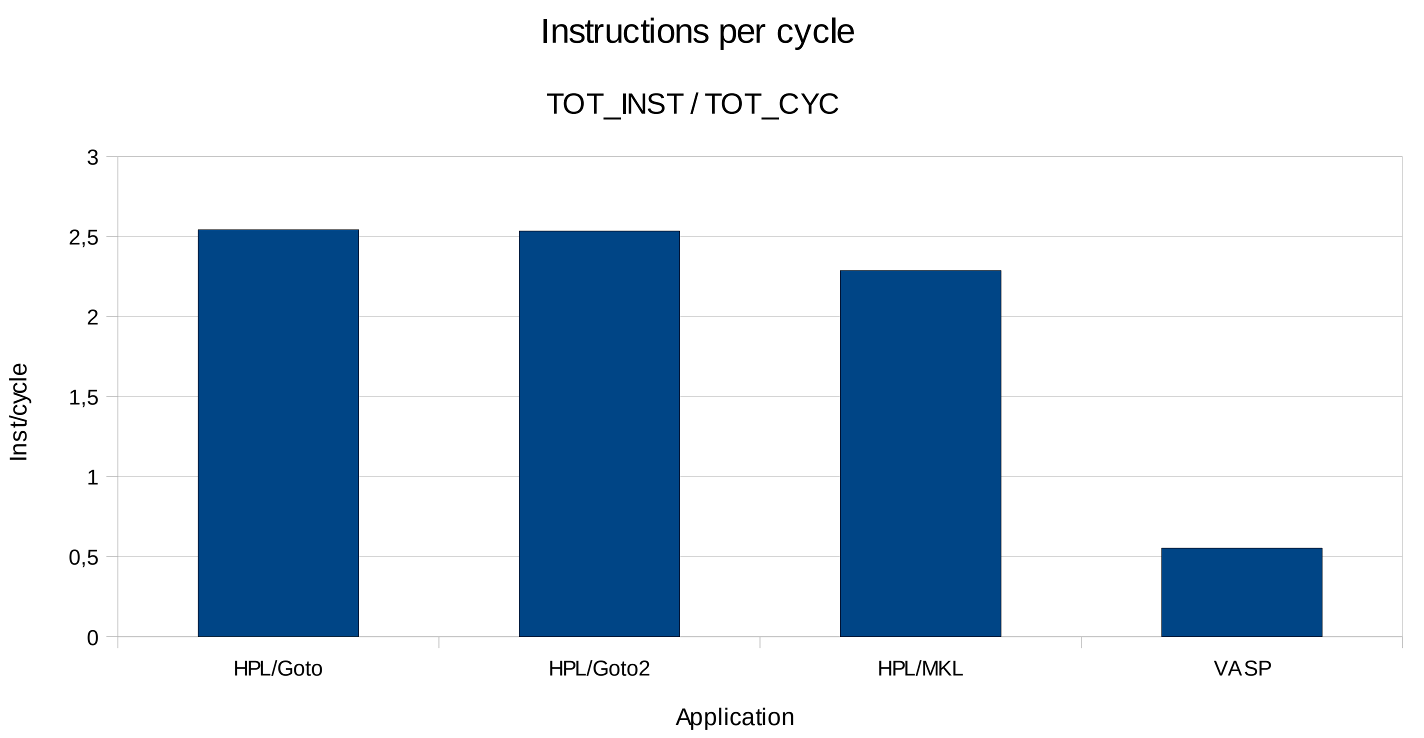
PAPI - instruction level profiling

- Total cycles - PAPI_TOT_CYC
- Total Instructions – PAPI_TOT_INS
- Resources Stalled Instructions - PAPI_RES_STL

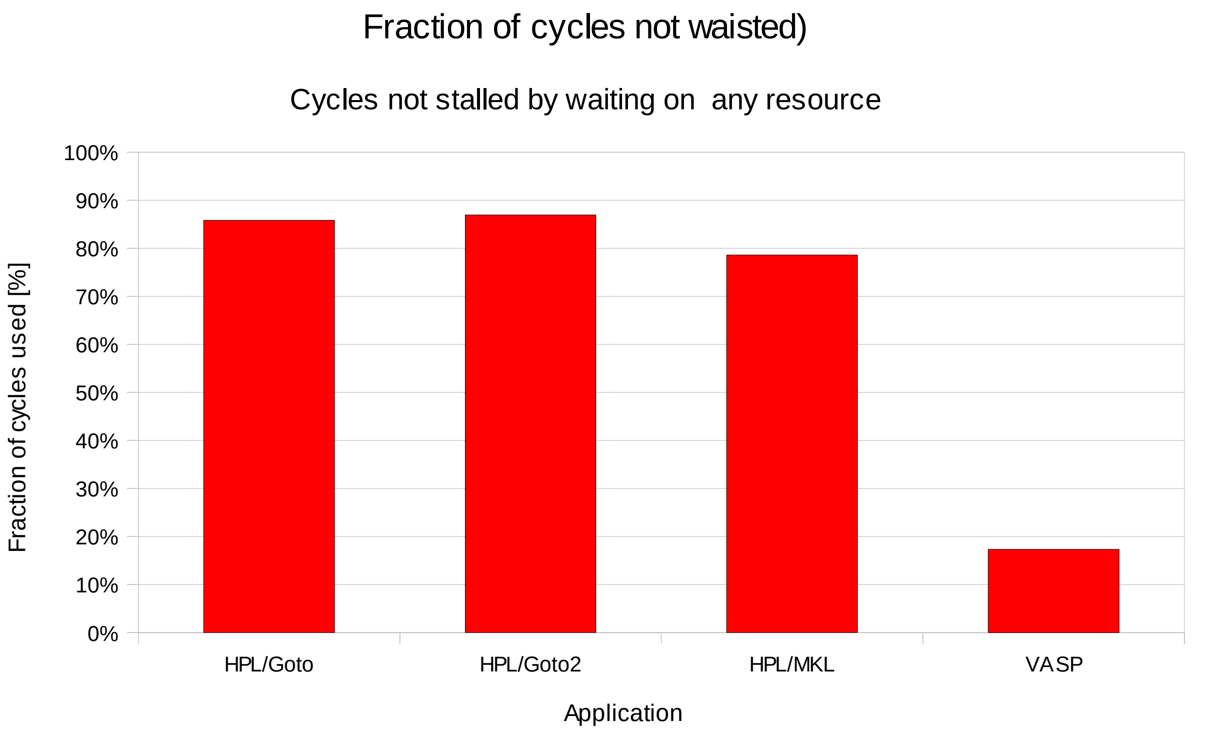
- The ratio between total cycles and total instructions is a measure of efficiency.

- Instructions stalled waiting for resources is waisted instructions.

PAPI results

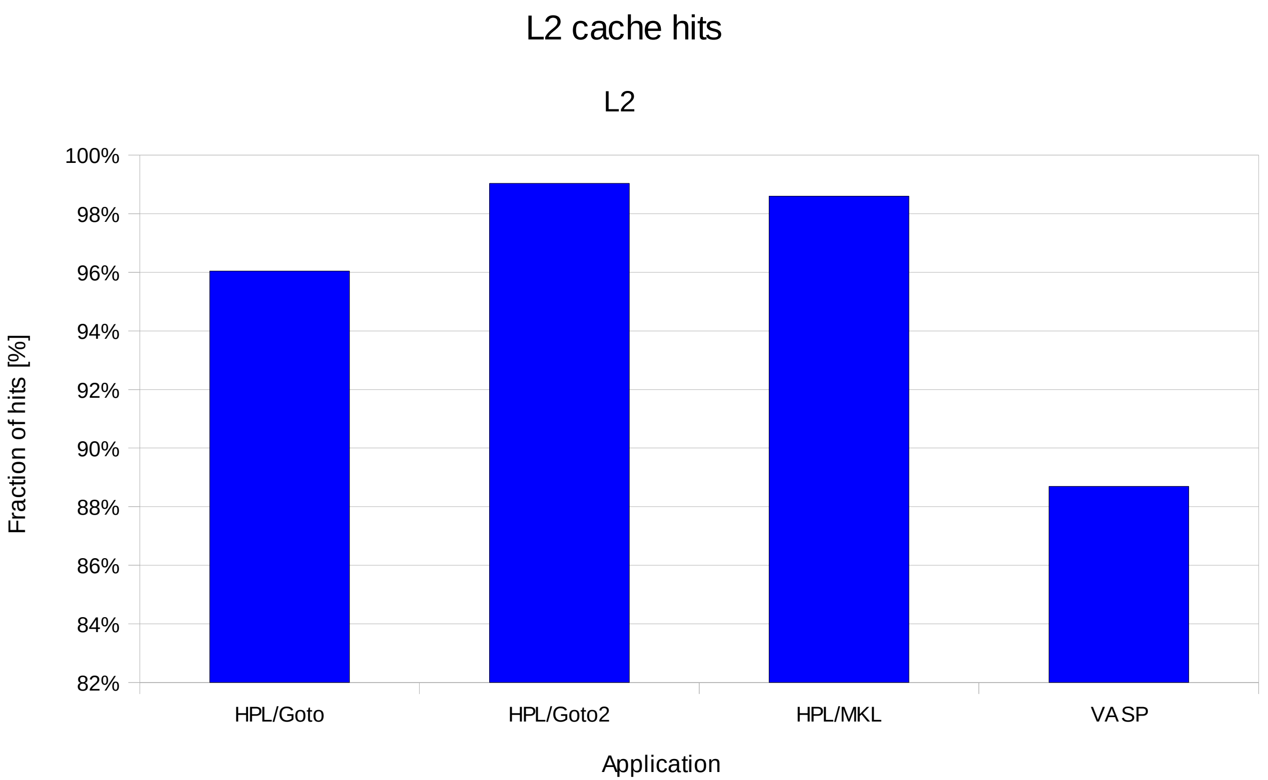


PAPI results



UNIVERSITETET I OSLO

PAPI results



Application behaviour/Profiling/tuning

Thank you for attending

Follow us on twitter : titancluster
Email list titan-operations

Ole W. Saastad, Dr.Scient
NOTUR training
USIT / SUF / VD