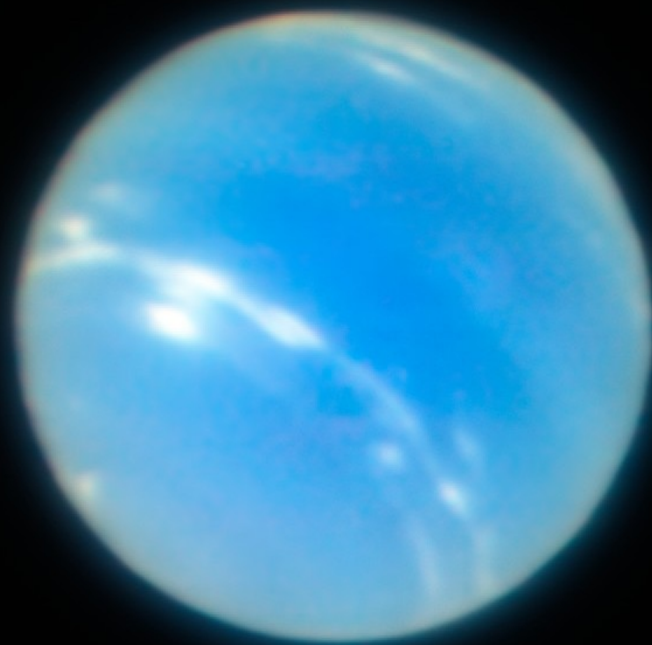
HRTCp:

ELT-sized Hard Real Time Core on common-off-the-shelf hardware

Poul-Henning Kamp
FreeBSD.org

Niels Hald Pedersen
FORCE Technology



HRTCp in terms of 17 computers and 10 VLANs

FORCE TECHNOLOGY

1
From ESO WFS
2
3
4
5
6

NET-WFS1
NET-WFS2
NET-WFS3
NET-WFS4
NET-WFS5
NET-WFS6

FRONTEND1_A
FRONTEND1_B
FRONTEND2 sim
FRONTEND3 sim
FRONTEND4 sim
FRONTEND5 sim
FRONTEND6 sim

NET-TF

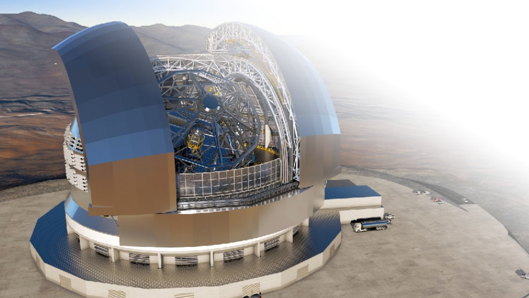
BACKEND
ESO HRTCp
PACER
CONTROL
TEST 0 (wfs sim)
TEST 1 (mirror sim)

NET-OUT

To ESO DMs (and back)

NET-CFG
NET-CMD

ESO Command
ESO Telemetry



HRTCp: Timing diagram

FORCE TECHNOLOGY

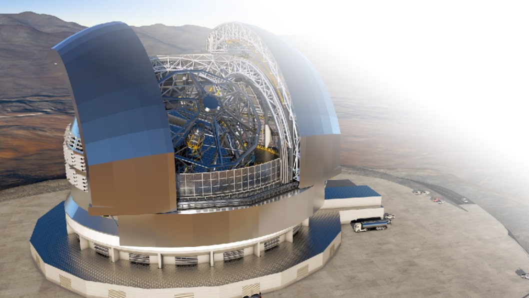
HRTC timing diagram (somewhat simplified)

500 Hz / 2000 microseconds cycle time
Actual timing with 1 real (shown) and 5 simulated frontends (not shown)

0 1000 2000 3000 4000 5000 6000

FRONTEND_A1	
incoming WFS pixels	0000 / 1400
calibration, centroiding, distr.	0100 / 1330
tomo. rec. MVM	0200 / 1250
prepare xmit, etc	1450 / 0030
xmit W to Temp.Filter	1480 / 0040
receive Ck-2 from WFS B	2950 / 0050
Ck-2 processing and storage	3000 / 0050
rec. next cycle slope dist.	0235 / 0025
BACKEND / Temporal Filter	
receive W from 6 WFS A	1480 / 0125
all processing	1605 / 0115
pres. ASAP: xmit U to DM	1720 / 0035
receive cmd echoes from DM	2195 / 0035
echo processing	2230 / 0060
xmit Ck to all WFS	2290 / 0035
rec. next cycle mirror dist.	0215 / 0210
FRONTEND_B1	
receive Ck from TF	2290 / 0045
computations and distr.	2335 / 0020
back projection MVM	2355 / 0585
xmit Sk to WFS A	2940 / 0040
PACER	
determ. Tref (wfspll) from WFS	0000 / 0010
TICK/TOCK exchange	1400 / 0020
disturbances for next cycle	0200 / 0200
dynamic params (future)	0400 / 0600

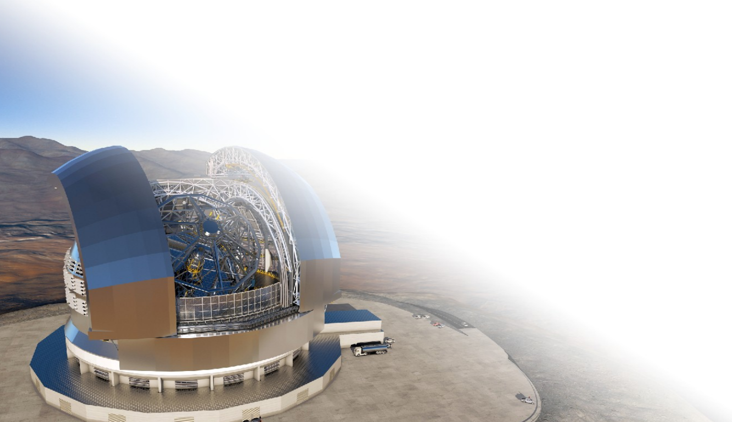
diagram version 0.015 of 2019-01-30



Bleeding (Power)Edge 2018: Dell R7425, dual AMD Epyc 7501:

FORCE TECHNOLOGY

- **64 Zen-1 2.2 GHz cores**,
  cache: 128M L3, 64 x (512K L2, 64K L1i, 32K L1d) on 8 NUMA nodes,
- **16 DDR4 2667 MHz busses** (2 each NUMA node)
  => 341.3 GB/s teoretical BW
- **128 GB RAM** (as 16 x 8)
- **4 x 10 GB Ethernet**



Software & Systems design for prototyping

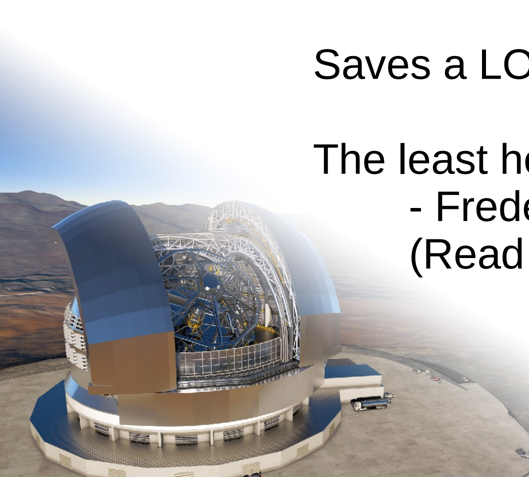
The goal of a prototype is to measure the known unknowns and find the unknown unknows

Make measurement and “what if?” experiments possible, and preferably quick and easy

- Modularity – To know what you are measuring and for “what if?” experiments
- Observability – access to internal state/ad-hoc measurement without side-effects
- Tweakability - everything is a parameter, real-time tweaking, robustness

Write production quality code from the start

- Saves a LOT of debugging time
- The least heeded wisdom in Software Engineering: <u>Always throw the prototype out</u>
  - Frederick P. Brooks: The Mythical Man-Month
  (Read it!)



The important bits for timing

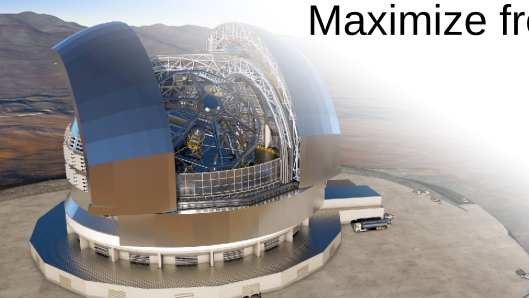
Amazing hardware (CPU, vector-math, network, RAM)

Preparing the “platform”

- As little software as possible – always a good & general principle
- Eliminate as many (potential) sources of jitter as possible – unknown unknows
- Time-synchronization – to be able to measure what we want to know
- Statically linked binaries – reduce randomness in cache-line footprint.

Exploiting the “platform”

- Nailing things firmly down – mostly threads to CPU cores
- Tightly choreographed network traffic – avoiding contention & collisions
- -//- in RAM & between CPUs – cache effects, CPU interconnects
- Code generation from high-level model of the system
- Maximize freedom to experiment – It’s a prototype



Linux as a “hardware abstraction layer”

OS services required:

- Start programs and threads
- Allocate memory
- Thread synchronization primitives
- Take timestamp
- Send packet
- Receive packet (with RX-timestamp)

Nice to have:

- Console connection (for running ps, top etc.)
- TCP connections for OaM

Necessary to maximize HW performance

- Lock thread/memory to specific CPU
- Steer network interrupts to specific CPU

Not wanted:

Pluggable devices
Graphics
Online help
Browser
Cryptography
TimeZones
UniCode
Containers
HTML
JavaScript
Screen Savers
Kubernetes
Internationalization
WLAN
FireWall
Accounting
[…]

SystemDectomy

Linux startup:

Bootloader finds two files: “kernel” and “initramfs”

Loads them into RAM

Jumps to kernel, tells it where to find initramfs

Kernel initializes

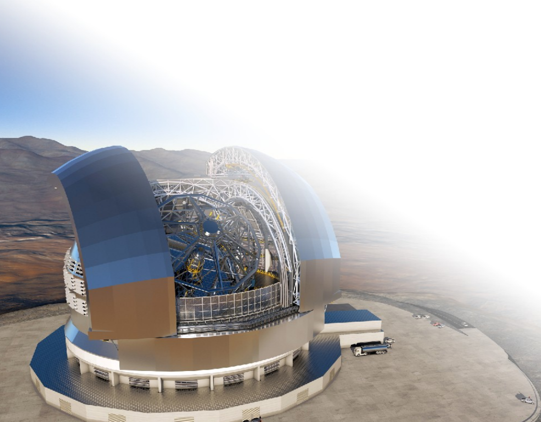
Kernel mounts initramfs as root filesystem

Kernel executes /sbin/init from initramfs

~~/sbin/init is SystemD~~

~~(abandon all hope ye who enters here)~~

```
set -x

export PATH=$PATH:/usr/sbin

mknod -m 644 /dev/random c 1 8
mknod -m 644 /dev/urandom c 1 9
mount -t proc /proc /proc
mount -t sysfs none /sys

# This list depends on the precise server hardware
/usr/sbin/modprobe -v tg3
/usr/sbin/modprobe -v bnx2
/usr/sbin/modprobe -v ixgbe
/usr/sbin/modprobe -v i40e

# Give drivers a moment
sleep 2

if [ -f /usr/bin/python ] ; then
    # Centos 7.4-6
    export PYTHON=/usr/bin/python
else
    # Centos 8.0
    export PYTHON=/usr/libexec/platform-python
fi

${PYTHON} \
    /network_config.py configure > /_netconf && bash -x /_netconf

/domus &
exec /bin/bash --login
```



System startup & CONTROL

CONTROL is the only “normal” Linux computer in the cluster

- Boots from local disk, ssh access, compilers, source code, email, cron, …
- Does not participate in the choreographed traffic

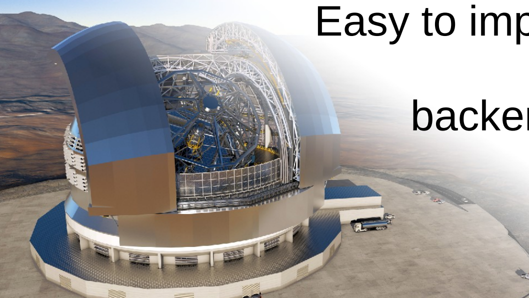
Acts as network boot-server for the rest of the cluster

Runs the “CONTROL” program (377 lines of code)

- Launches, configures, controls & monitors programs on real-time nodes
- Text-based Command Line Protocol (“CLI”) used throughout.

All RT programs have CLI interface reachable through CONTROL’s CLI

- Easy to implement and execute ad-hoc experiments:
  - backend cmdpll dejitter 40

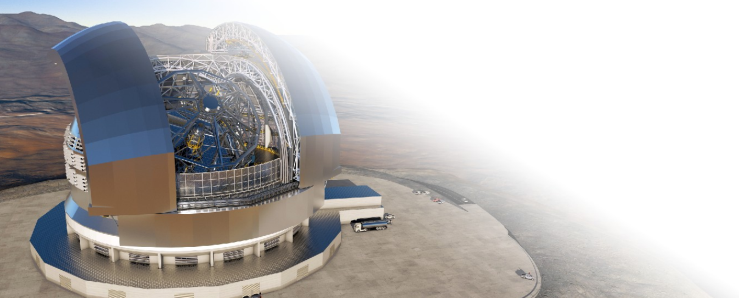


Real time nodes

Boots via network (UEFI standard supported by BIOS)

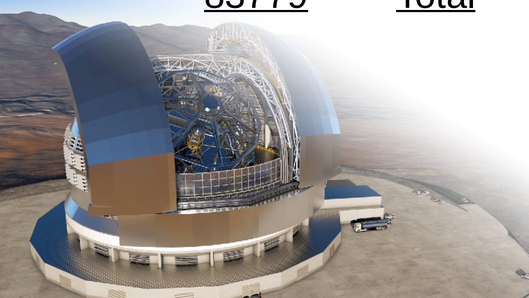
/sbin/init script starts “DOMUS” which:

- Listens for TCP connection from CONTROL
- Executes CLI commands from CONTROL
  - Receive file
  - Transmit file
  - Launch (a single) RT-program
- Pass CLI traffic between CONTROL and RT-program



Lines of code expended:

2223	PACER – Timekeeping, choreography, reporting
4445	FRONTEND[N]a – Big “forward” MVM math
1854	FRONTEND[N]b – Big “backwards” MVM math
3059	BACKEND – “minor” math, filters, mirror split etc.
324	WFS – WFS sensor simulator
181	MIRROR – MIRROR simulator
632	RECORD – Can capture any traffic in system to FITS file(s)
5458	Include files
12315	Library functions
7671	Python data model of all “NetVar” in cluster
<u>38152</u>	<u>Subtotal</u>
45627	Python generated NetVar Alloc+Tx+Rx+RTMS+MUDPI+Timestamping code
<u>83779</u>	<u>Total</u> (Comparison: /bin/bash is 168330 LOC)



Work cycle starts with first WFS[0] Packet (= rate controlled by WFS)

Every WS:

PACER sends TICKTOCK
- UTC timestamp
- Parameter orders

RT-nodes send TOCKTICK
- Parameter confirmation
- Stats, states, timing

All NetVar’s have assigned Tx time(-slot)

PACER xmits parameter updates in defined holes between science data xmissions.

All transmissions

Tx0
Rx0
RxN

Number of packets

1e+08
1e+07
1e+06
100000
10000
1000
100
10
1

0 500 1000 1500 2000 2500 3000 3500

Time in work-cycle [μs]

TICKTOCK/TOCKTICK

FORCE TECHNOLOGY

Reports TX_0 timestamp for all NetVar transmissions, RX_0, RX_N for all receptions

Also multicast, so other programs can listen in with zero impact.

Plot from development, showing parameter update transmission interfering with science data timeslot

_fold_MIRROR.INSDM1.ACTUATOR_COMMAND RX0 + RXN

Microseconds after Tref

Workcycle# modulus 30000

Reflections on trusting Intel, AMD and Linus but not Lennart

Safe: Huge MVM’s/fast math have large overlap with traditional supercomputing

Safe: Microsecond timing has overlap with very profitable “fast trading” market segment

Safe: What we want from the linux kernel is minimal and totally uncontroversial

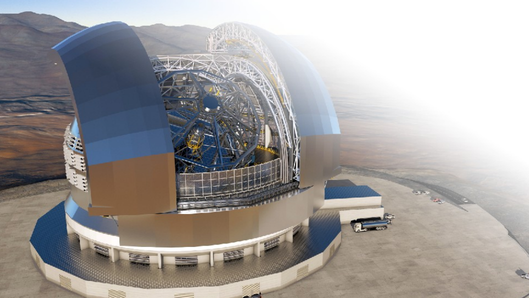
Mitigated Risk: Linux userland, and SystemD goals and direction is totally different

Available fallbacks:

- ARM CPUs – Harder to get, except Apple, which have astonishing performance
- FreeBSD OS – Does (almost) everything Linux does, but boringly.

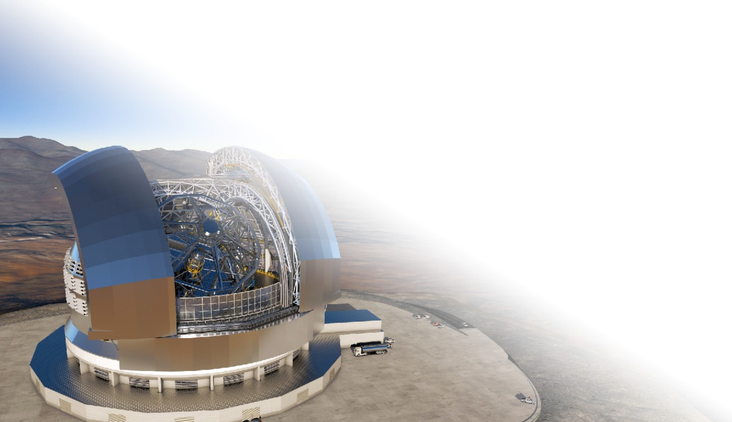
Interesting question:

- MTBF of 3nm semiconductors in cosmic rays at 3km altitude



A highly NUMA platform: Dell R7425, dual AMD Epyc 7501

- **64 Zen-1 2.2 GHz cores**,
  cache: 128M L3, 64 x (512K L2, 64K L1i, 32K L1d) on 8 NUMA nodes,
- **16 DDR4 2667 MHz busses** (2 each NUMA node)
  => 341.3 GB/s teoretical BW
- **128 GB RAM** (as 16 x 8)
- **4 x 10 GB Ethernet**

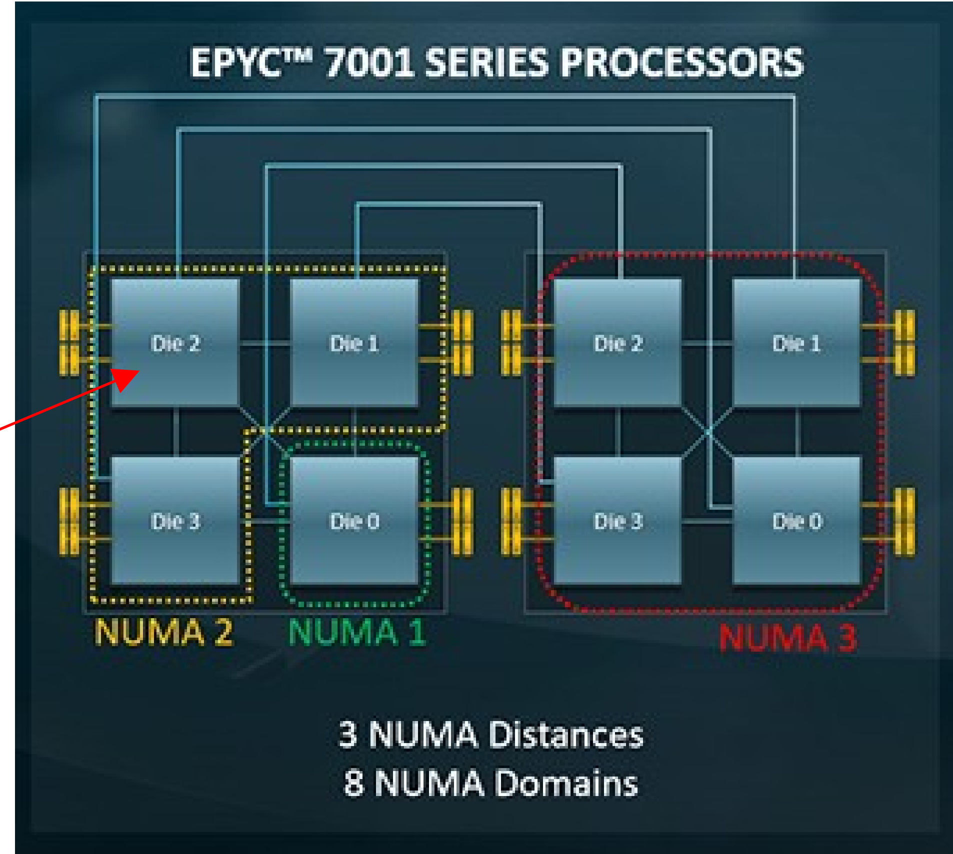


A highly NUMA platform: R7425 The NUMA topology

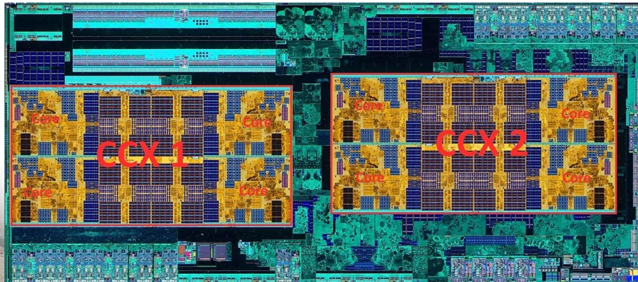
Taking the liberty of interpreting “von Neumann machine” as something with a single memory space, but possibly more than one hardware thread of execution, we may state:

This is not exactly a von Neumann machine.

It is more like 8 (16?) von Neumann machines, connected with 40GB-ish network segments (16 hereof), simulating being a single von Neuman machine.



Each Zen-1 die (“Zeppelin”) holds two core complexes each with 4 cores and 8 MB L3 cache



Latency ping-pong (practical latency mapping)

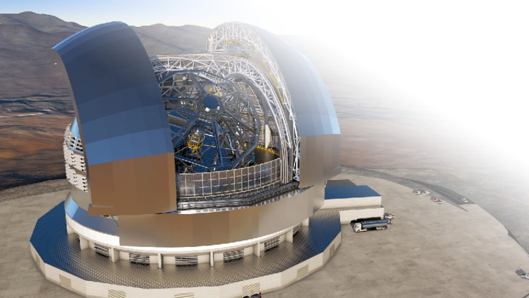
Dell PowerEdge R7425

2 x AMD Epyc 7501: 64 Zen gen.1 cores, 128 MB L3
8 NUMA nodes

Each side of the 64 x 64 pix graphic represents cores 0-63.
Each of the 4032 non-green pixels thus represents a pair of two cores and the colour code the middle latency, when these two play a game of **latency-pingpong:**

Each core hotspins on a variable local to its own NUMA domain.
When it sees the value having been incremented by its opponent core, it will increment this cores local variable and note the time.
Repeat 10000 times, compute average roundtrip latency.

White	100	ns,	same CCX (same L3)
Light blue	340	ns,	same Zeppelin, other CCX
Just blue	560	ns,	same socket, other Zeppelin
Dark blue	840	ns,	other socket, shortest InFab path
Blackish blue	950+	ns,	other socket, longer path



NUMA/cache/BW-driven Ground Rules

A real time node is dedicated to a single task. You should own it.

Prepare the platform for real time use:
ISOLCPU 1-63 and friends, interrupt routing, no SMP, move kernel tasks...

do {

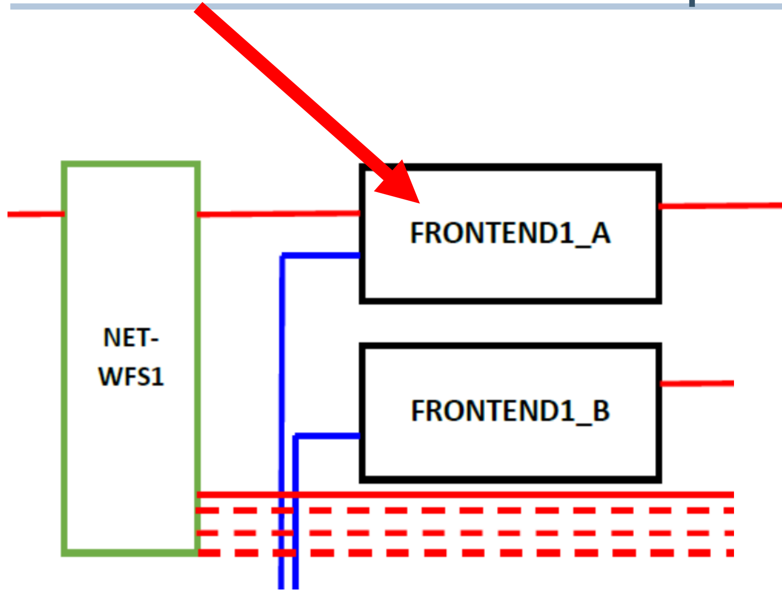
Analyze the application:
Data storage, reads and writes.
Computational requirements. Cache hits necessary?

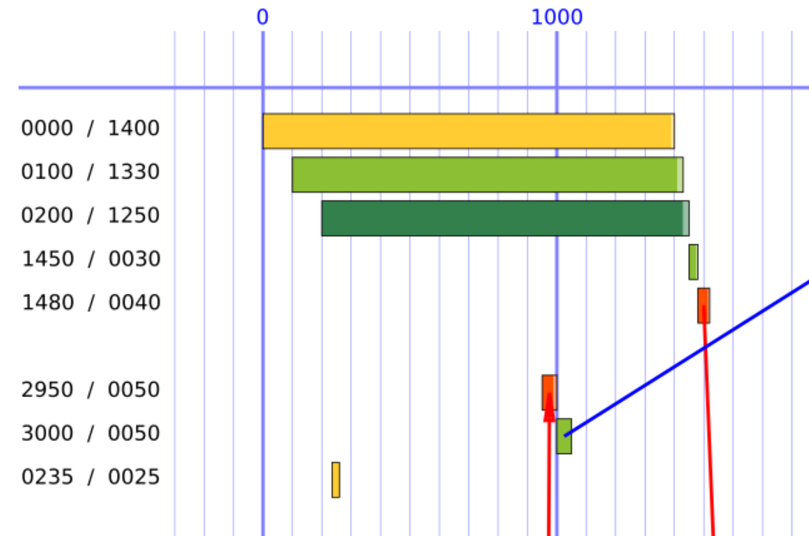
[**Re**]**Design the application** (core-ography):
Pin all threads to a specific core
Evaluate the effect of reads on content of shared cache
Consider memory BW

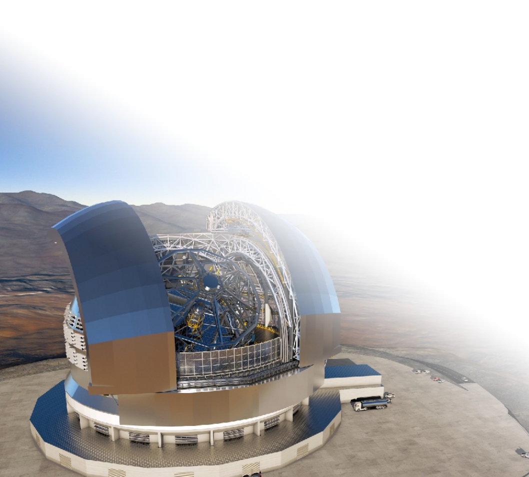
Test achieved timing/performance (RDTSC is your friend)
Consider integrating a nanosec-granular test infrastructure.

} **while (!sufficient);**

Frontend A: Real time programming a NUMA platform







Frontend A: Sensor image processing

FORCE TECHNOLOGY

Run-time
Configuration Data

Dark Map (dynamic)
Flat-field Map (dynamic)
Pixel to Sub-aperture Number Map (static)
Pixel to Sub-aperture Position Map (static)
Centroiding Matrix (dynamic)
Reference Slope Map (dynamic)
Sub-aperture Underillumination Threshold (static)

Disturbance
Injection Interface

I^{dark} I^{flat} c_{ref} f_{thresh} s_k^{d}

WFS Real-Time
Data Interface

I_k^{raw} I_k^{cal} c_k s_k^{l} s_k^{m} s_k

Sub-aperture
Centroiding

$f_k(i) < f_{thresh} \rightarrow s_k(i)=0$

Internal Real-time
Data Interface

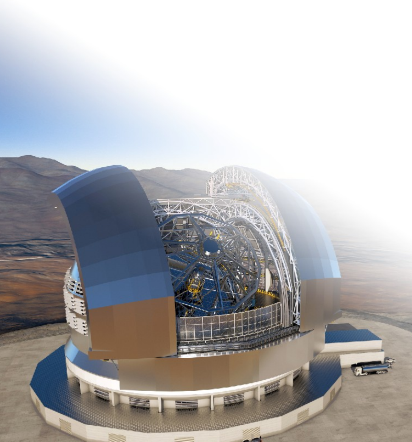
Pixel Calibration

f_k

Reference Subtraction &
Under-illumination Handling

Pixel Telemetry
Data Interface

Loop Telemetry
Data Interface



Cache planning: Calibrate, Centroids, Move

FORCE TECHNOLOGY

Incoming raw sensor data lives here, node 0 memory

In NUMA node 1, core 8, the **calibrator** thread lives, computing the calibrated sensor image.

In node 1 DDR memory, its coefficients (5 MB) are stored and in principle read every cycle. As L3 cache is 8 MB, this lead to a very high cache hit rate.

Likewise, the **centroid** thread keeps its coefficients (3 MB) mostly in the other half of the node 1 L3 cache.

Core-ography, frontend A

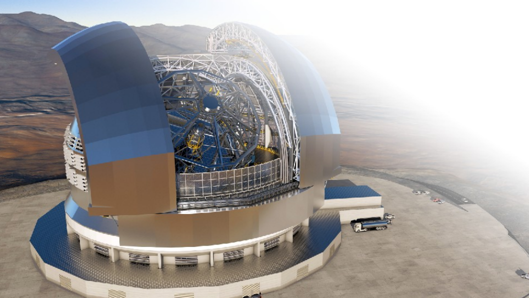
L3 cache 8M					L3 cache 8M			
core 0 OS init pixel tele	core 1 receive WFS	core 2 ticktock	core 3 supervisor	dual DDR3 bus logic	core 4 transmit cmds	core 5 receive c.errors	core 6 receive dynvars	core 7 parameter install.

L3 cache 8M					L3 cache 8M			
core 8 calibrate pixels	core 9	core 10 pixel tele. --> node 0	core 11	dual DDR3 bus logic	core 12 centroid	core 13 mover	core 14	core 15

L3 cache 8M					L3 cache 8M			
core 16 MVM lane 0	core 17 MVM lane 1	core 18 MVM lane 2	core 19 MVM ...	dual DDR3 bus logic	core 20 MVM	core 21 MVM	core 22 MVM	core 23 MVM

L3 cache 8M					L3 cache 8M			
core 24 MVM lane 8	core 25 MVM	core 26 MVM	core 27 MVM	dual DDR3 bus logic	core 28 MVM	core 29 MVM	core 30 MVM	core 31 MVM

(32 additional MVM worker cores in 4 NUMA nodes following...)



Frontend A: Tomographic reconstruction, and back...

FORCE TECHNOLOGY

x,y: 116.6 MFLOPS/cycle
(these two constitutes 99% of total work)

for this reason, **x** is located in WFSB.

Interaction Matrix (dynamic)
Control Matrix (dynamic)
Run-time Configuration Data

M
R

Internal Real-Time Data Interface

s_k

+ +

s_k^*

x $s_k^* = M \cdot v_k$

v_k

$v_k = \alpha \cdot c_{k-\delta-3} + \beta \cdot c_{k-\delta-2} + (1-\alpha-\beta) \cdot c_{k-\delta-1}$

c_k

Internal Real-Time Data Interface

POL Slope Computation

s_k^{pol}

y $w_k = R \cdot s_k^{pol}$

w_k

Internal Real-Time Data Interface

Tomographic Reconstruction

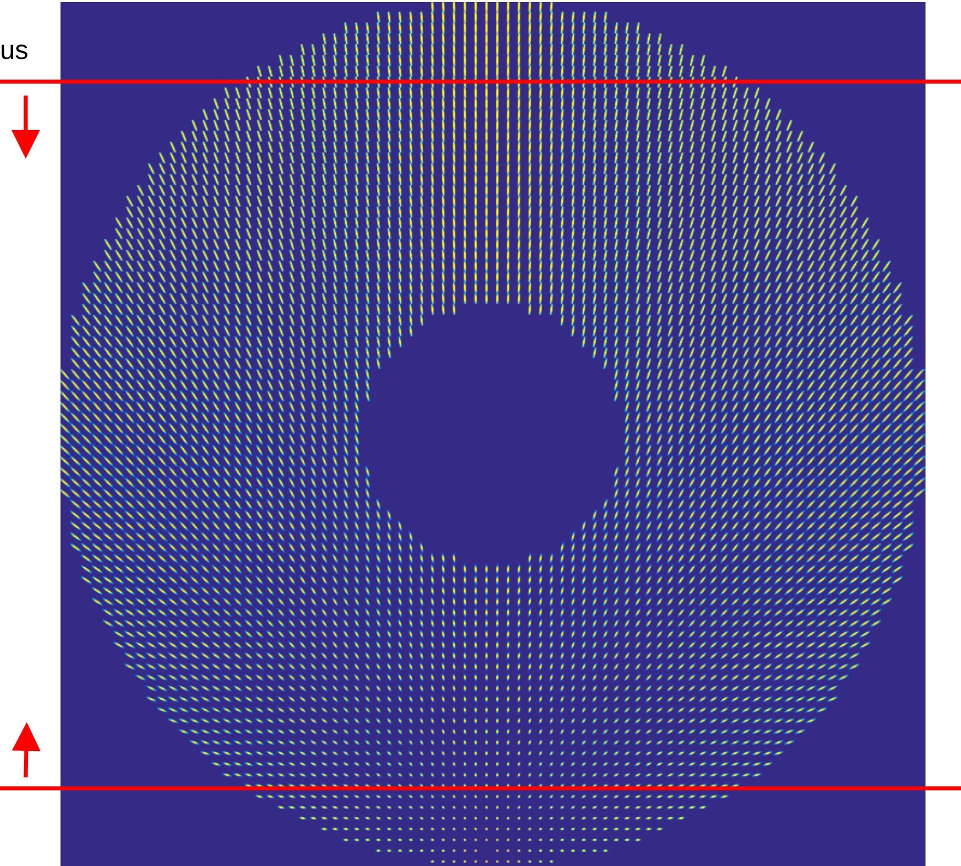
Pipeline order:

Data is read out of the CCD in an nontrivial order given by sensor, and described by suitable maps, available at runtime.

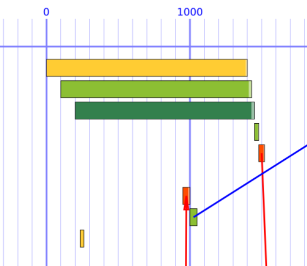
From these maps can be computed the order of which subapertures become complete.

This, the pipeline order, is used in computing the tomographic reconstruction MVM.

This implies that matrix coefficients should be scrambled to pipeline order at installation, and MVM result should be unscrambled to user order afterwards.



FRONTEND_A1	
incoming WFS pixels	0000 / 1400
calibration, centroiding, distr.	0100 / 1330
tomo. rec. MVM	0200 / 1250
prepare xmit, etc	1450 / 0030
xmit W to Temp.Filter	1480 / 0040
receive Ck-2 from WFS B	2950 / 0050
Ck-2 processing and storage	3000 / 0050
rec. next cycle slope dist.	0235 / 0025



Frontend A: The MVM bucket brigade

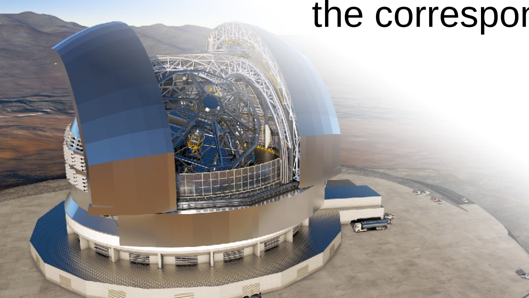
Sensor data packet X arrives with its payload of sensor data. Then:

1) **calibrator** thread looks up which pixels came with particular packet X,
applies dark field/flat field correction to these pixels, converting to **user order** in the process,
and signals to thread centroid that this is done.

2) **centroid** thread looks up which subapertures can now be computed, does this,
converting to **pipeline order** in the process,
and signals to thread mover that this is done.

3) **mover thread** sees whether there are enough new slope values to bother MVM workers with that.
If so, the new values are copied to a node-local copy of the slope vector on nodes 2-7.
After this, a likewise node-local counter stating the validity of the local slope vector is updated.
No one is signalled.

Meanwhile, and asynchronous to the above, each of the 48 MVM workers hotspins on their local copy of the validity variable; when this changes (from above actions), the worker computes the corresponding slice of the MVM.

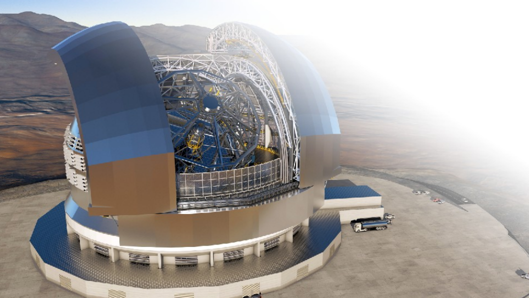


Frontend A: The MVM bucket brigade II

signal →
read →
write →

Core-ography, frontend A

(32 additional MVM worker cores in 4 NUMA nodes following...)



Results: (early) Timing details from Frontend A big MVM

FORCE TECHNOLOGY

These are early stage results, using a configuration with 56 cores / 7 NUMA-nodes allocated to doing the MVM.

Graph show the time interval 1390 to 1470 microseconds from arrival of first WFS packet.

We see the winding up of the bucket brigade after last packet: first **calibrator**, then **centroid**, then **mover** and at last the **MVM workers**.

printout of 19/05/09 08:01:28, page 1

Last packet signalled from pacer thread

calibrator thread processing done

0000 / 1400
0100 / 1330
0200 / 1250
1450 / 0030

Centroid thread processing done

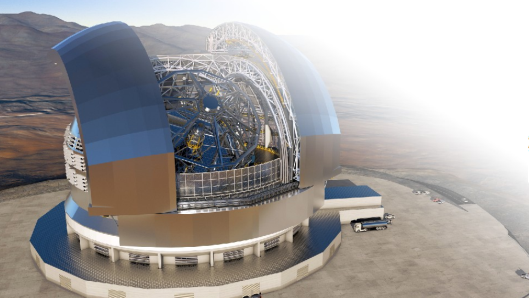
Mover thread processing done

All 56 MVM threads processing done, result on node 0

his.dat
2] vs [1]

x-axis: 1390, 1395, 1400, 1405, 1410, 1415, 1420, 1425, 1430, 1435, 1440, 1445, 1450, 1455, 1460, 1465, 1470

y-axis: 0, 1000, 2000, 3000, 4000, 5000, 6000, 7000, 8000



Results: Latency distribution from performance test

FORCE TECHNOLOGY

12-hour test as per req. 374:

- Disturbances (mirrors, slopes) continuously
- Dark/scale maps, centroid matrices, etc updated every 10 sec (as per req.).
- Big matrices (R,M) updated every 30 sec (NB: 10 times the req).

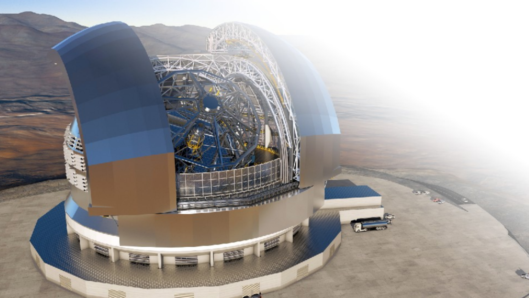
1500000 cycles (50 min)

Note logarithmic y-axis

M4 Command End-to-End Latency: $\Delta t = t_{M4,Trailer} - t_{LVSM,Trailer}$

y-axis: n/1 (10^0 – 10^6); x-axis: Δt/µs (350 – 600)

These 19 outliers, approx. 160 microsecs late, was difficult to reproduce later and has never been explained.



Thank you for listening...