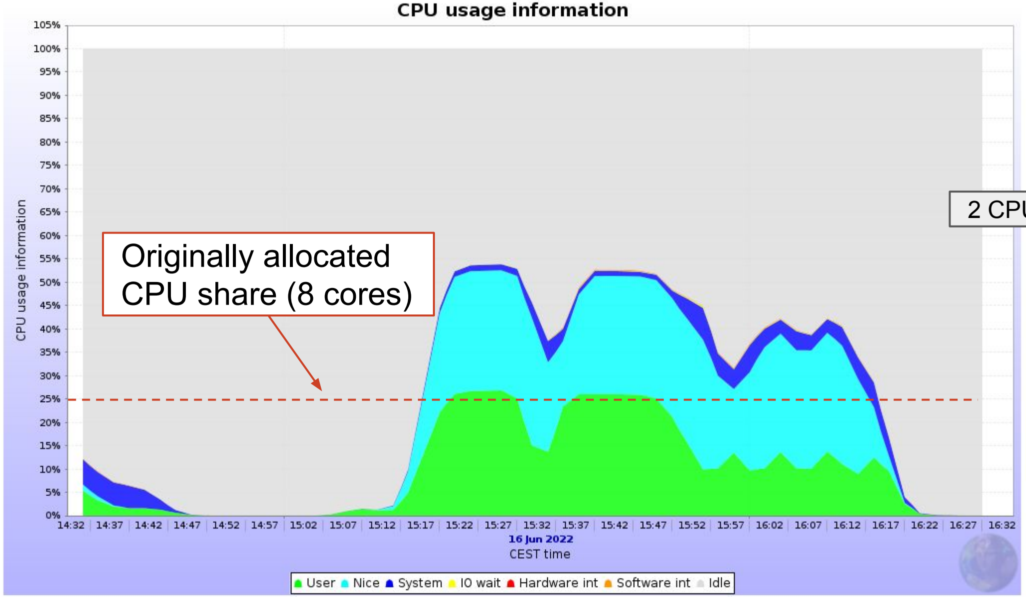
Payload behavioural studies, scheduling, optimization and system tools for payload control

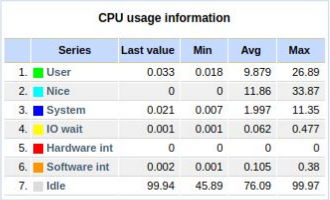
Marta Bertran
26/09/2022
ALICE Tier-1 Tier-2 Workshop

Payload profiling – Understanding its behavior

Observed issues on new multicore workflows



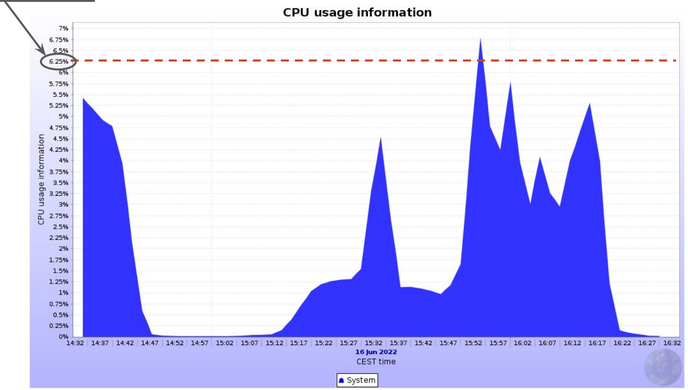
Originally allocated
CPU share (8 cores)

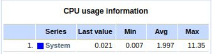

CPU usage information

	Series	Last value	Min	Avg	Max
1.	🟩 User	0.033	0.018	9.879	26.89
2.	🟦 Nice	0	0	11.86	33.87
3.	🟦 System	0.021	0.007	1.997	11.35
4.	🟨 IO wait	0.001	0.001	0.062	0.477
5.	🟥 Hardware int	0	0	0	0
6.	🟧 Software int	0.002	0.001	0.105	0.38
7.	⬜ Idle	99.94	45.89	76.09	99.97

Elevated *System* CPU component

2 CPU cores




CPU usage information

	Series	Last value	Min	Avg	Max
1.	🟦 System	0.021	0.007	1.997	11.35

CPU efficiency →**160.25%**

```
Jun 16 16:21:19 [proc ]: 08:37:20 31040 160.25 0.00 49740.19 85.02 85.02 32 6 1200 59688.23 22970.03 22970.03 1000
```

Observed issues on new multicore workflows

Run 3 jobs are multicore and run on a completely new software stack.

Execution of **high amount of short-lived processes** that requires changes in the monitoring framework

- CPU reported by executing machine did not correspond with the one reported by our job monitoring.

High overhead / **large *System* CPU usage** due to process creation.

- Large amount of folders in `LD_LIBRARY_PATH` => overhead from loading dependent libraries.
- High amount of process initializations.

Deep analysis into **job internal behaviour**.

- Origin of system calls might not be observable at first glance.
- Clear picture of how the resources are being used.

Profiled execution analysis

Job execution wrapped with **`strace`** command:

```
strace -e trace=process -ttt -f -s 10000 -o /tmp/jobId-execution.strace
```

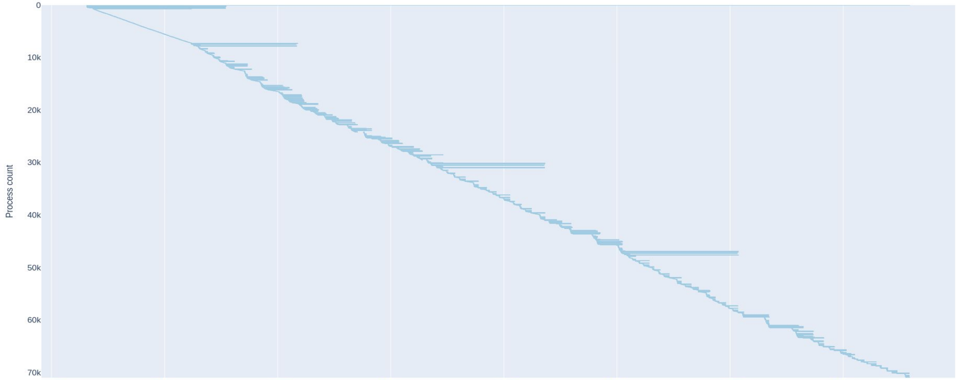
Exhaustive study of the deployed processes and threads, their amount, the execution frequency, the time distribution and the resource usage.

Observations from the reported metrics revealed some **potential areas for improvement**.

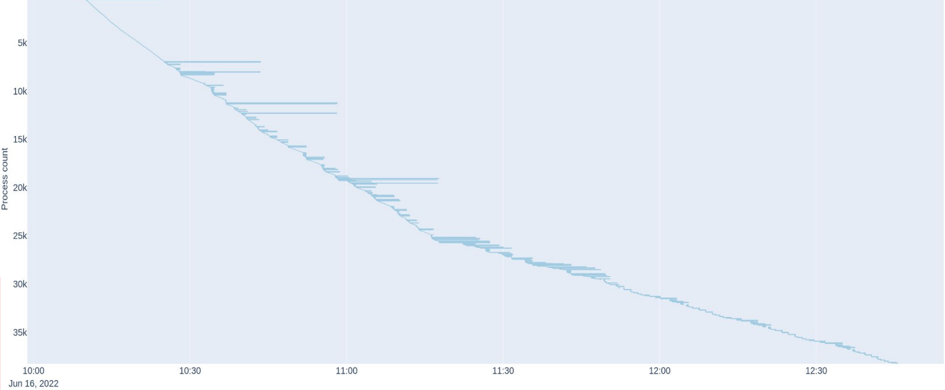
Profiled execution analysis and improvements

Process durations Gantt plot

Originally:
Total process+thread count - **72.5K**

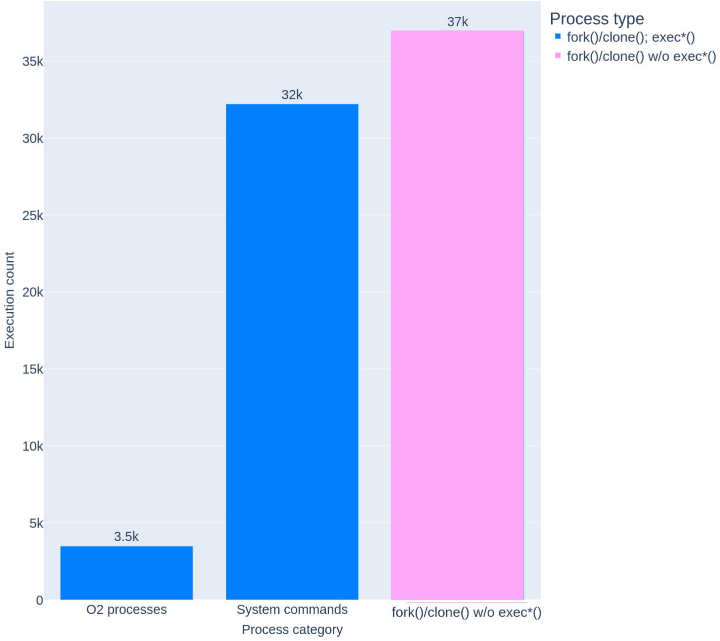


After improvements:
Total process+thread count - **38.3K**

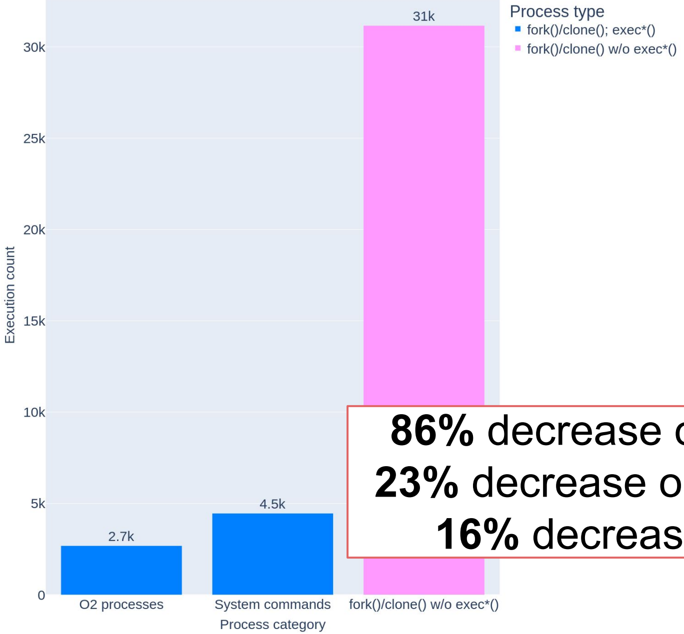


Total count decreased by **47.17%**

Profiled execution analysis and improvements



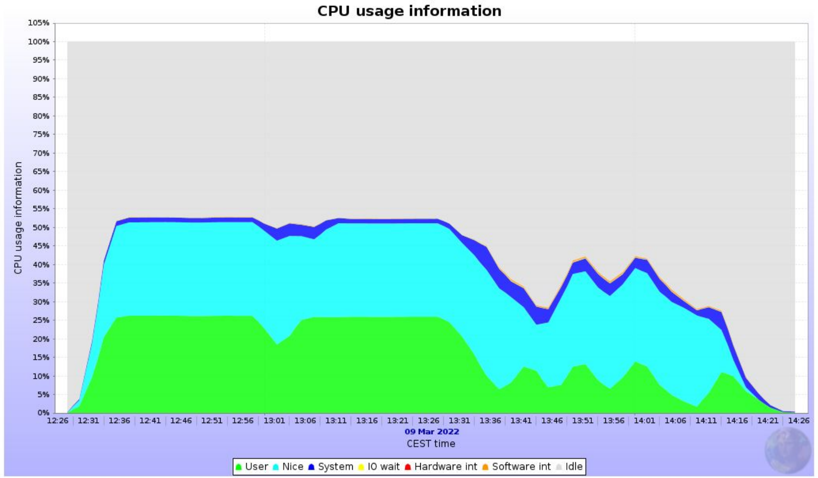
32k system calls
3.5k O2 processes
37k threads



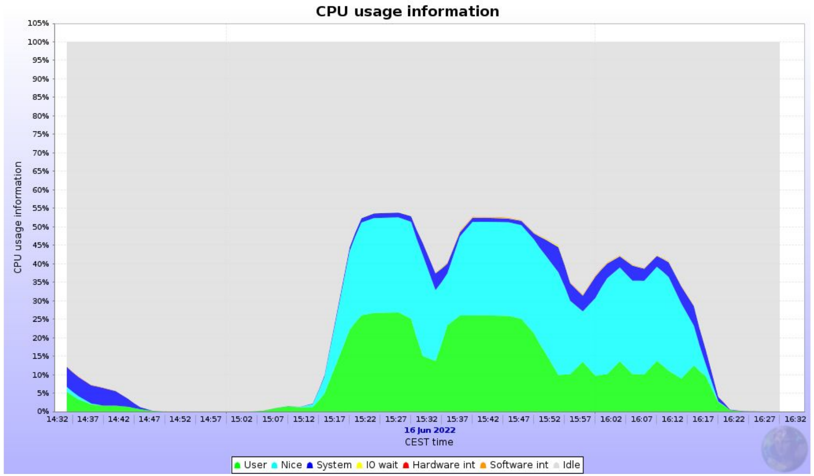
4.5k system calls
2.7k O2 processes
31k threads

86% decrease on system calls
23% decrease on O2 processes
16% decrease on threads

Profiled execution analysis and improvements



1:46h



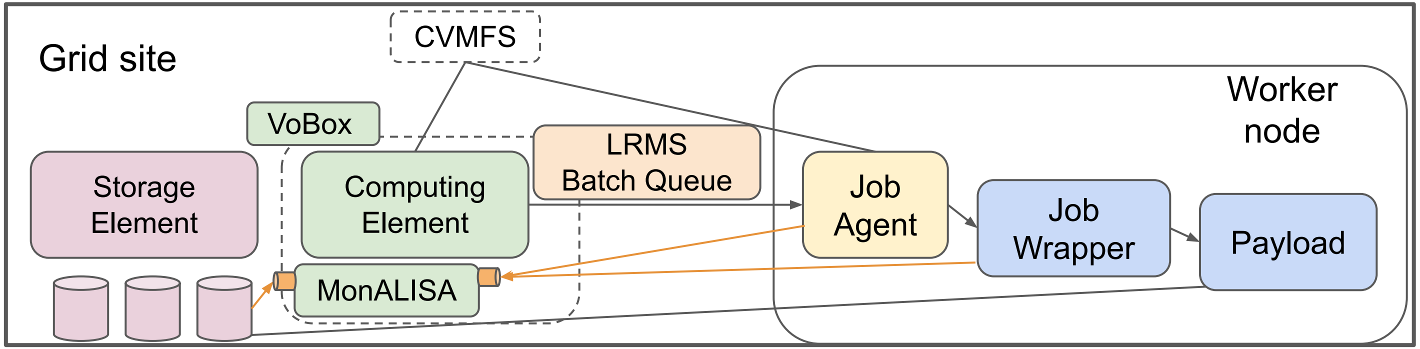
1:09h

Execution time decreased by **35%**

Framework and payload developments

Correct loading of dependent libraries

O2 process merging and decrease of initialisations



Improved monitoring

New efficiency accounting methodology

Per-process fine-grained monitoring

Extended monitoring features

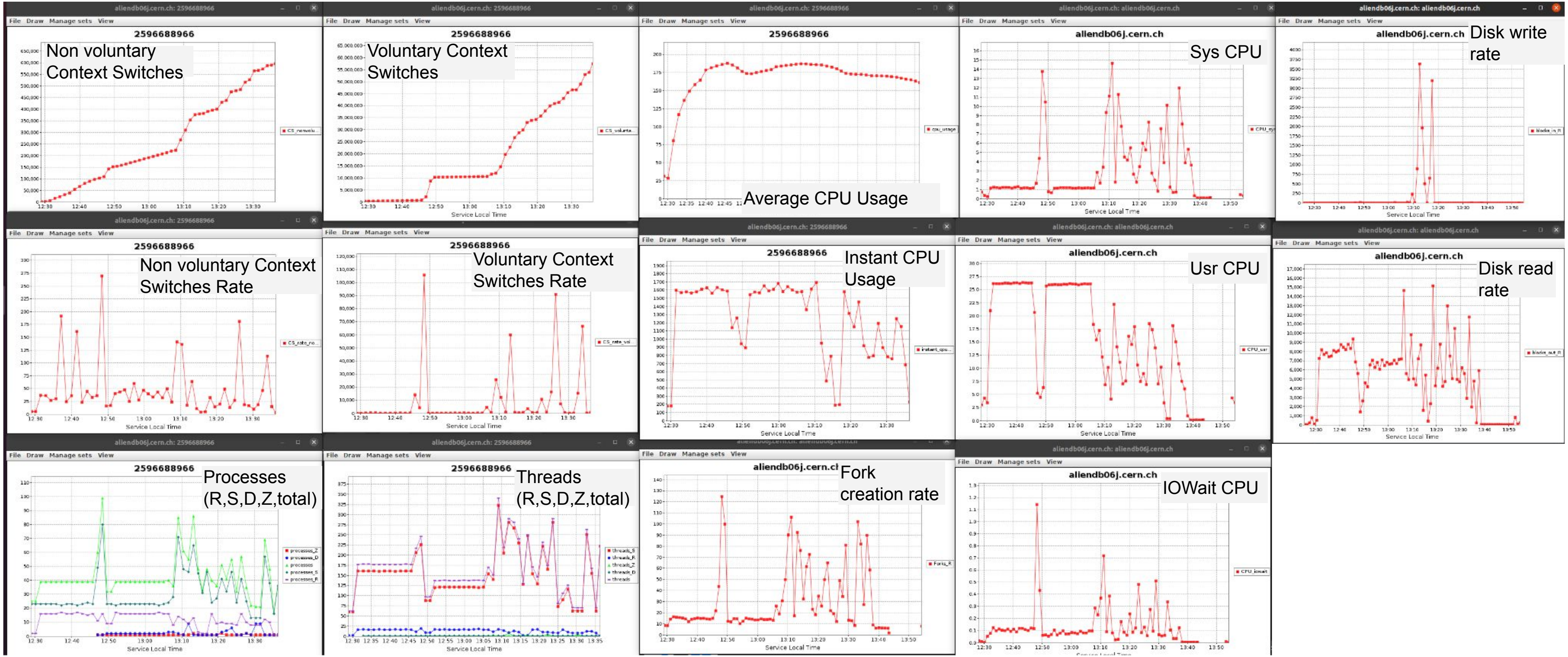
New methodology for **efficiency monitoring** to properly account for child processes and threads. Two values being reported:

- **Instantaneous** CPU efficiency.
  - Reported as absolute percentage (100% = a fully used core).
- **Average** CPU efficiency.
  - Reported as a percentage over the allocated cores.

Added per process type grained monitoring

- Enabled with the `Monitoring = "payload";` option in the JDL.
- Reported metrics (per-process and total accumulated):
  - CPU usage
  - Voluntary and non-voluntary context switches

System monitored parameters



Non voluntary Context Switches

Non voluntary Context Switches Rate

Processes (R,S,D,Z,total)

Voluntary Context Switches

Voluntary Context Switches Rate

Threads (R,S,D,Z,total)

Average CPU Usage

Instant CPU Usage

Fork creation rate

Sys CPU

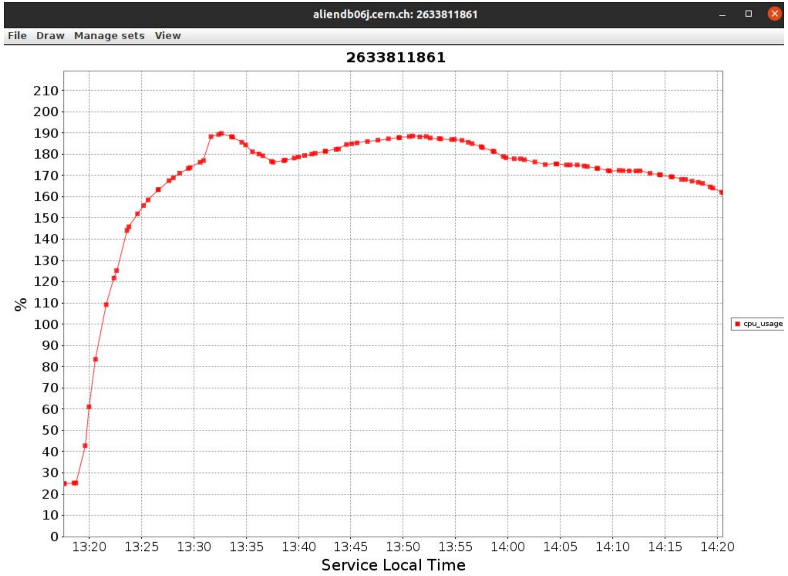
Usr CPU

IOWait CPU

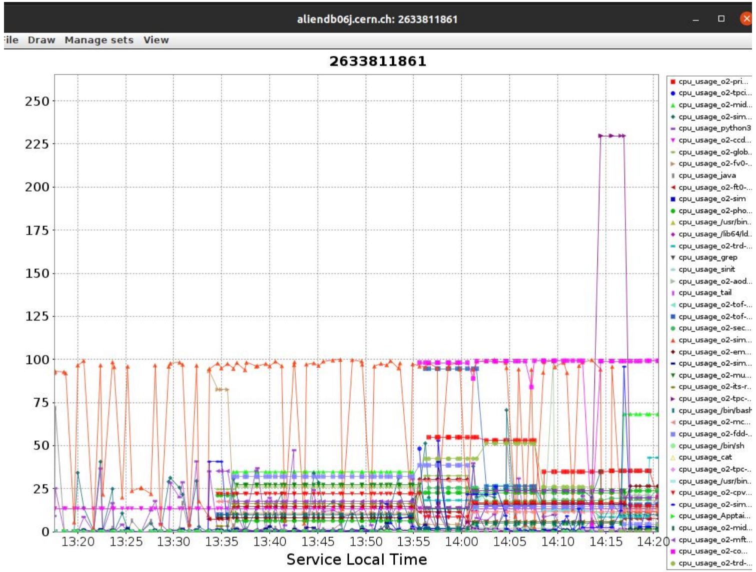
Disk write rate

Disk read rate

CPU usage reporting

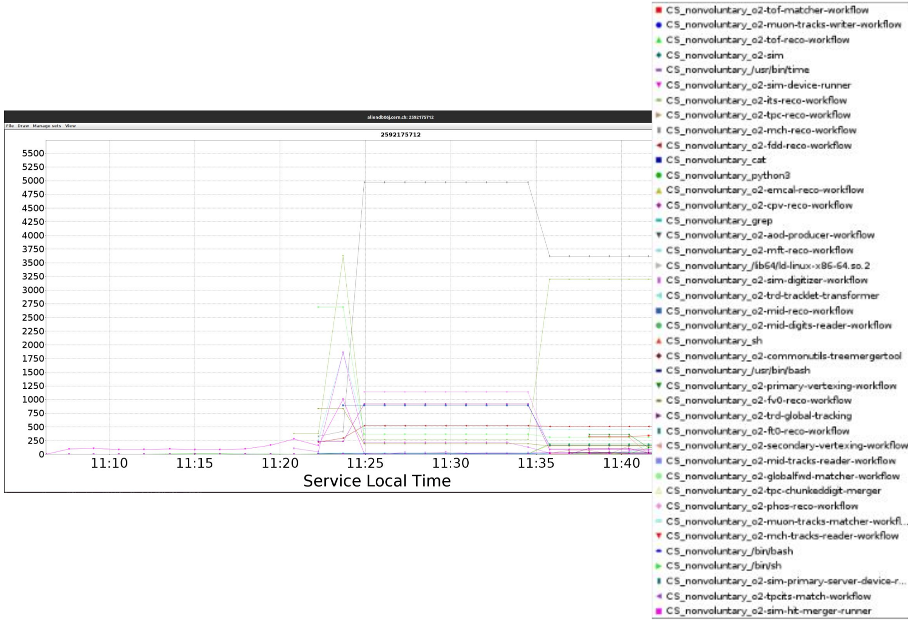


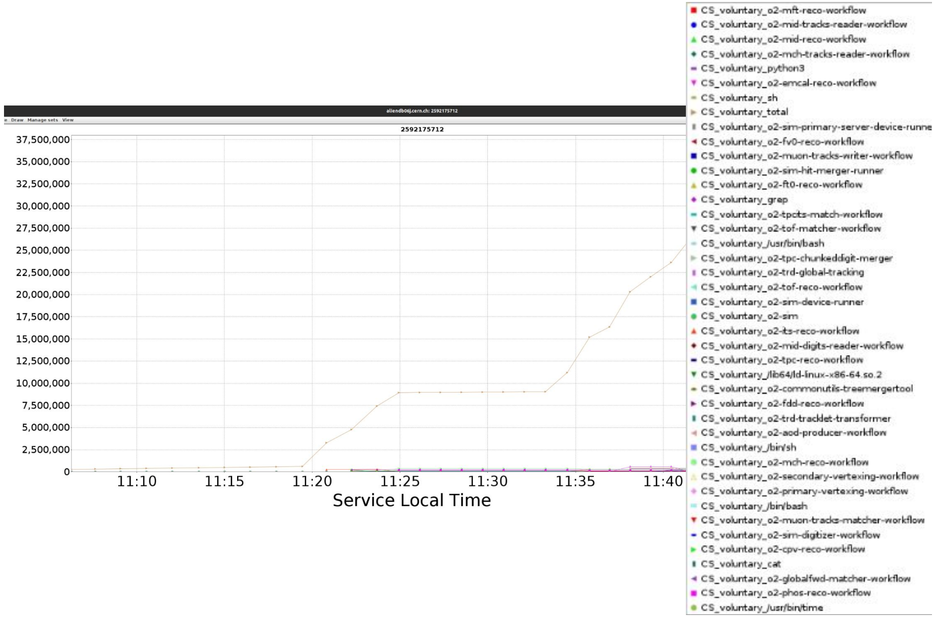
Average CPU usage of job



Instantaneous CPU usage per process type

Per-process context switches reporting





Non-voluntary context switches per process type

Voluntary context switches per process type

Improving efficiency in monitoring

Accounting for the **Proportional Set Size (PSS)** of processes.

- Iteration over `/proc/`*`PID`*`/smaps` file of every deployed process.
- Parsing of these files is a **costly operation in CentOS7** format.
- CentOS8 introduces `/proc/`*`PID`*`/smaps_rollup` with **pre-summed memory information** for a process.

Upgrading to CentOS8 would allow for a **higher monitoring rate** with less overhead and increased monitoring precision

- Also quicker to react on jobs running over the memory limits

Payload control methodology

Multicore jobs running on the Grid

Multicore jobs can run in two configurations:

- **Whole node**: JAliEn manages resources and splits them into job execution slots.
- **Slots of *N* cores**: Jobs are assigned a set of resources.
  - This might be enforced by some resource constraining mechanism.
  - These slots can be further partitioned to run finer-grained jobs.

We are particularly interested on being granted a **whole node** for increased flexibility and optimal resource management.

ALICE Run 3 payloads

New workflows make use of multicore slots (8-core slots).

- All kind of payloads are using multicore: Asynchronous reconstruction, Monte Carlo, Analysis…
- Grid still running some legacy workflows with 1 core.

Some of the processes **consume more than the given 8-core share** if they can freely use all the resources of the working node.

For **resource fairness**, we need to **enforce constraining** of these jobs to a set of resources.

Constraining jobs to a set of resources

`Cgroups v2` is available in CentOS8 to constrain the job to a set of cores.

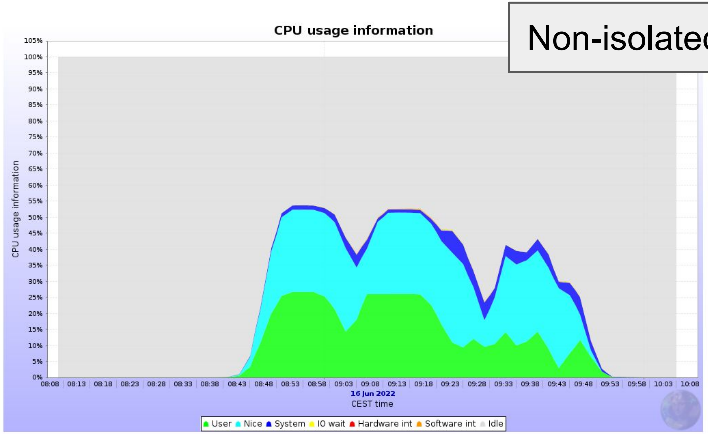
- However, it is **not available in CentOS7**.
- This would be the best alternative.

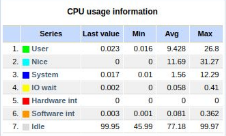
We already have an implementation of **CPU pinning** to constrain jobs to run on a set of specific CPU cores that works for CentOS7.

- We explore what cores are pinned to pin those that are free.
- To pin specific CPU cores to jobs we use the `taskset` **Linux tool**.

Constraining jobs to a set of resources

Non-isolated job



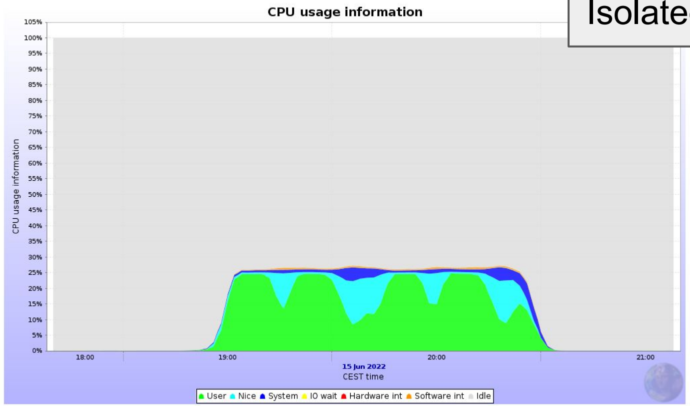


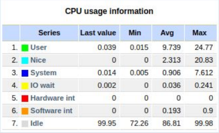
CPU usage information

	Series	Last value	Min	Avg	Max
1.	🟩 User	0.023	0.016	9.428	26.8
2.	🟦 Nice	0	0	11.69	31.27
3.	🟦 System	0.017	0.01	1.56	12.29
4.	🟨 IO wait	0.002	0	0.058	0.41
5.	🟥 Hardware int	0	0	0	0
6.	🟧 Software int	0.003	0.001	0.081	0.362
7.	⬜ Idle	99.95	45.99	77.18	99.97

Idle machine just running our job
Total CPU usage - **Goes over 50%**, i.e. 16 cores of 32 available

Isolated job





CPU usage information

	Series	Last value	Min	Avg	Max
1.	🟩 User	0.039	0.015	9.739	24.77
2.	🟦 Nice	0	0	2.313	20.83
3.	🟦 System	0.014	0.005	0.906	7.612
4.	🟨 IO wait	0.002	0	0.036	0.241
5.	🟥 Hardware int	0	0	0	0
6.	🟧 Software int	0	0	0.193	0.9
7.	⬜ Idle	99.95	72.26	86.81	99.98

Idle machine just running our job
CPU usage is limited with `taskset`
Total CPU usage - 100%, i.e. 8 cores of 8 cores requested

Constraining jobs to a set of resources

Depending on the node configuration:

- **Jobs running inside a container** in the worker node: Containers should already have *cpuset* assigned to limit resource consumption.
  - We can not see configuration of other processes for optimal core selection.
  - We might get assigned a resource slice that can be further split between the running payloads.

- **Whole node without containers**: JAliEn takes care of the CPU management assigning the cores to the running workflows.

Constraining jobs to a set of resources

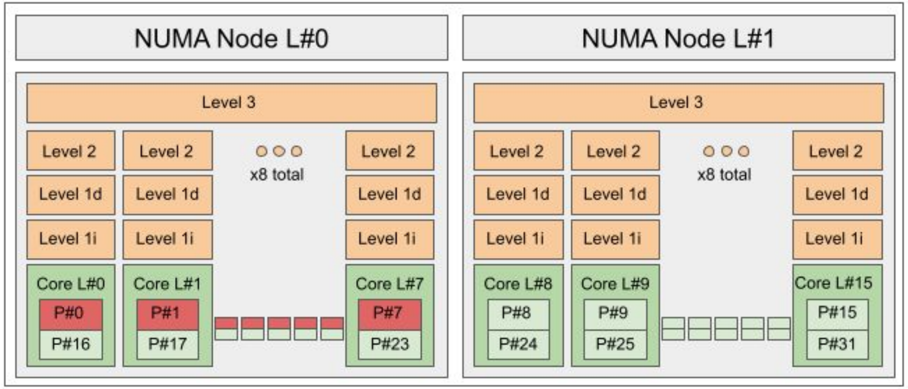
Hybrid strategy implemented:

- Jobs can expand when there is enough resources but once overconsumption detected (and after a grace period) they are constrained to a set of cores (originally requested amount).
- This prevents that jobs make use of resources granted to other jobs.
- Performed study on pinning configurations with a focus on exploiting **data locality of NUMA nodes**.
- Still in testing phase but planned to be added in production environment.

Tested pinning configurations

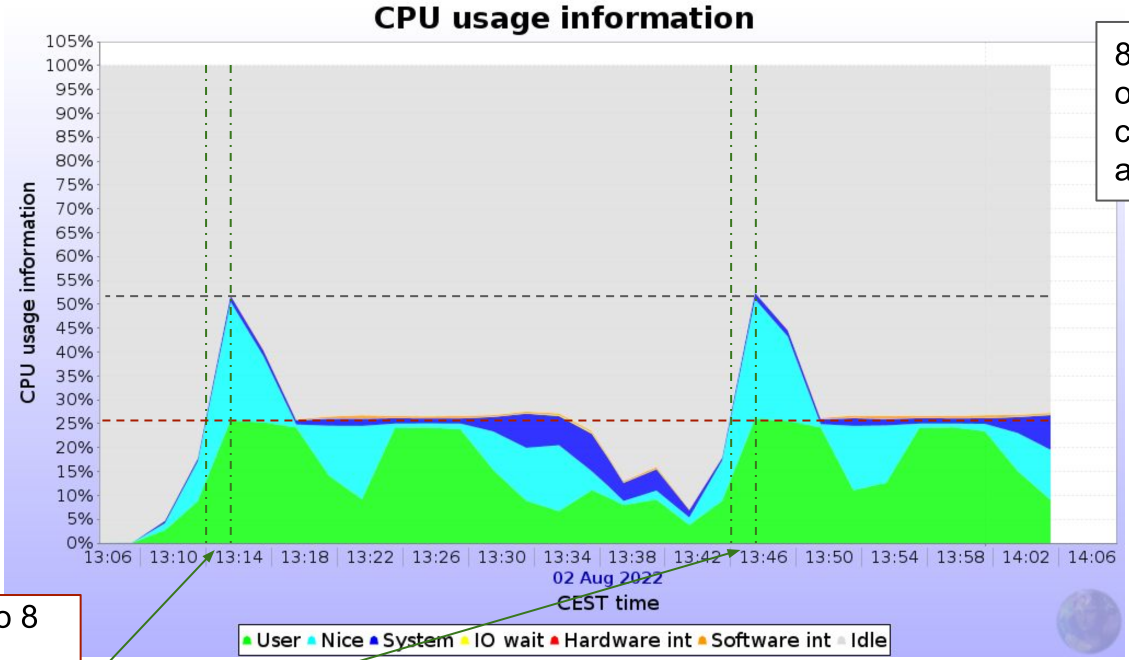
Tested 5 different configurations:

1. **Same NUMA Node and independent L1,L2 cache**
2. Different NUMA Nodes and independent L1,L2 cache
3. Same NUMA Node and sharing L1,L2 cache
4. Random core assignment
5. No pinning - jobs run freely on the machine



Configuration 1 proved best performance.

CPU resource consumption on over consuming job



8-core job using 50% of a 32 cores node (16 cores - 2x requested amount)

Enforce constrainment to 8 cores when detected prolonged overconsumption. Adjustable grace period given.

Exploiting optimal resource usage – Executing in oversubscription regime

Whole node oversubscription

Workflows of different nature running on the Grid use resources differently.

- Analysis jobs are IO intensive and they do not fully use the allocated CPU slot at all times.
- **Idle portions of CPU** can be gathered and used to execute computing intensive jobs (MonteCarlo).
  - No added pressure in IO.

Many of the worker nodes (more than 76%) have **spare memory resources** not used by the running jobs, which are allocated 2GB RAM (allowed up to 8GB/core when considering SWAP).

Whole node oversubscription

In whole node scheduling scenarios we want to use the available resources in the most efficient way.

- Memory is a constraining factor in job execution.
- Compute memory per CPU core ratio and see if they can be oversubscribed.
- Node's *Idle* CPU above a threshold and at least 2GB free memory in the system. → It has room for an extra job.
- Advertised "*extra*" free resources to the CS.
- **Extra job of specific nature** with complementary resource usage patterns.
- **Assign unused memory to new jobs and oversubscribe CPU cores.**
- **Constant monitoring** of machine CPU and memory consumption to preempt extra jobs if limits surpassed.
- Preempted jobs are rescheduled without penalty.

Whole node oversubscription

The amount of CPUs available for oversubscription will depend on the node's memory.

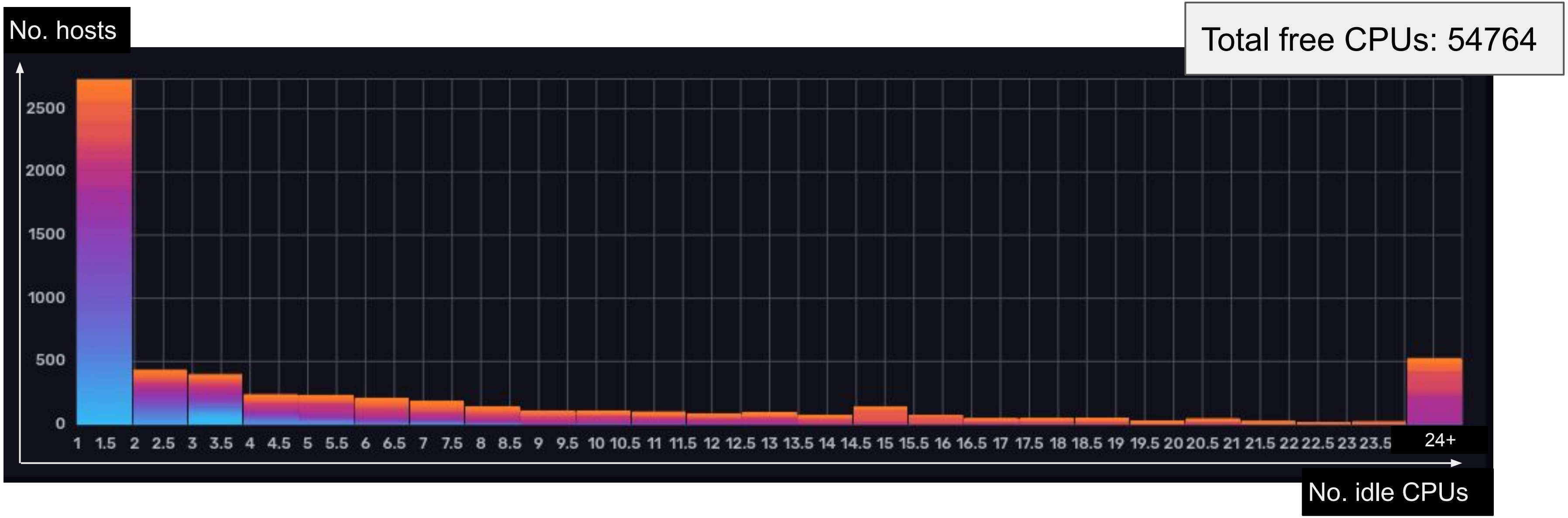
- Minimum between [RAM/2, (RAM+SWAP)/8].

Oversubscribed jobs are executed with a lower priority.

Whole node scheduling is needed for managing all available resources.

Running additional jobs with **complementary resource usage patterns** on a worker node has a great potential to compensate for inefficiencies.

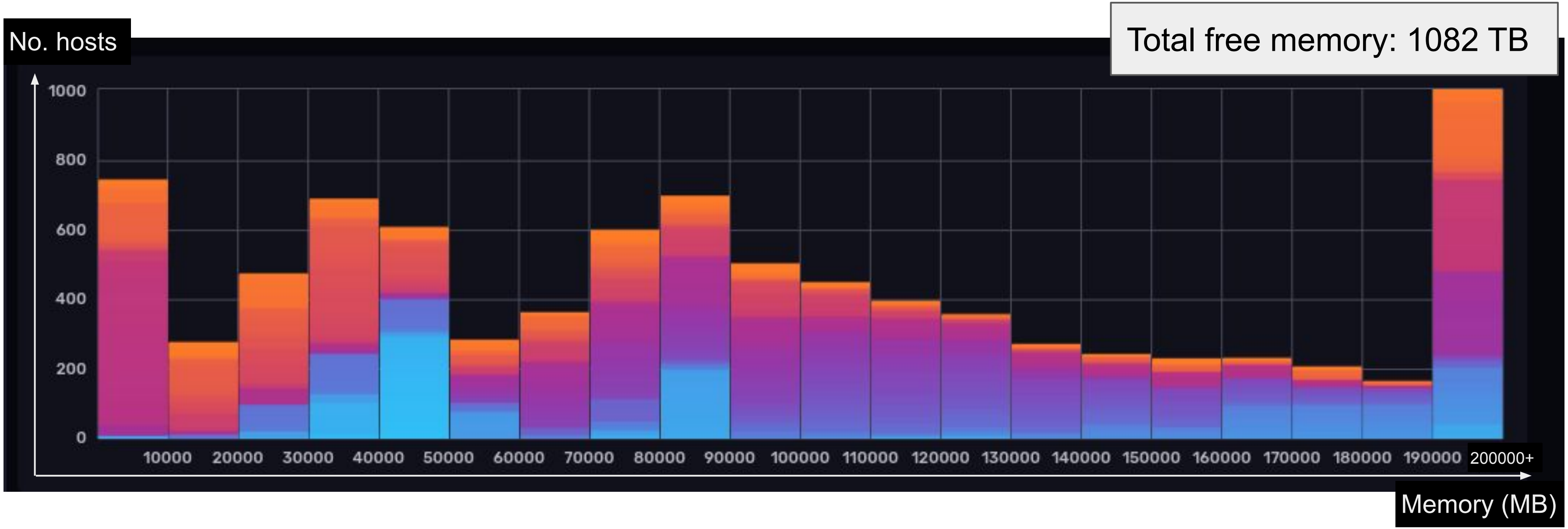
Idle CPU cores from Grid hosts



Total free CPUs: 54764

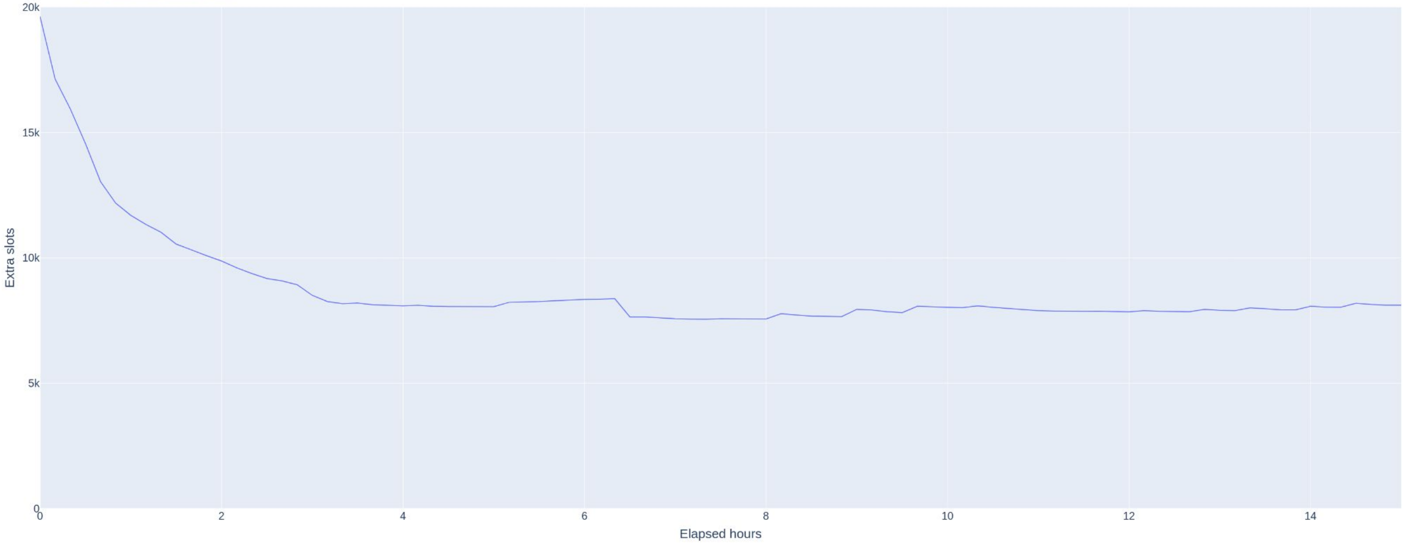
- Minimum amount of idle CPU cores per host during a period of 1 hour.
  - Percentage of idle CPU parsed from `/proc/stat`.
  - Multiplied by the total amount of cores → Idle cores.

Free memory from Grid hosts



- Minimum amount of memory available per host during a period of 1 hour.
- Free memory computed parsing `/proc/meminfo` → (`MemFree+Buffers+Cached)` .

Potential extra 8-core slots on Grid hosts



- Minimum amount of extra jobs that would not be preempted running for the set amount of hours.
- Extra slots as `min(floor(idle_cores / 8) , floor(free_memory / 16 GB))`

Conclusions and final remarks

Need to adapt Grid framework to new payload software stack.

- Enhanced and extended monitoring.
- Optimized fairness on resource usage. → Job confinement.
- Improved resource utilisation levels. →Whole node oversubscription.

For observable results in overall Grid performance we need:

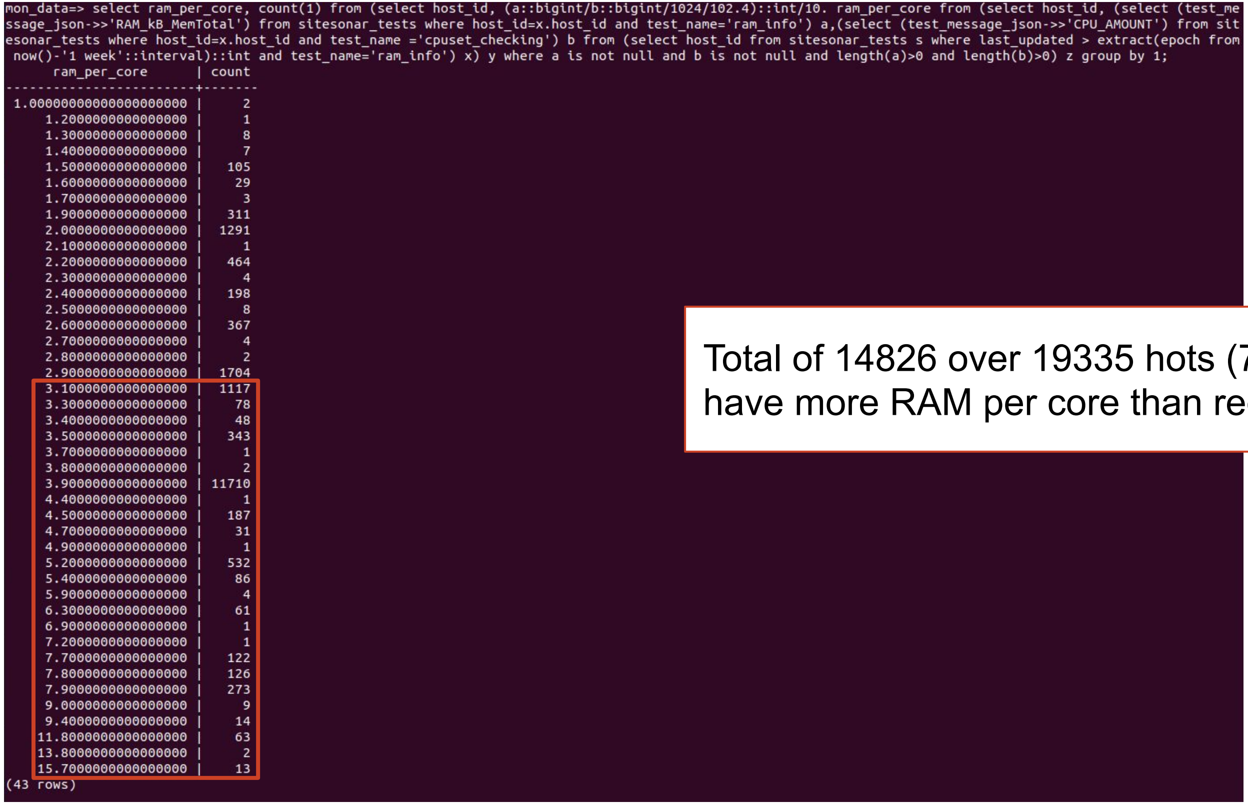
- **Whole-node** allocations.
  - Accurate accounting of node’s activity. →Efficient resource management.
- Migration to **CentOS8**.
  - Support for `cgroups2`
  - More efficient monitoring
  - Growing need for features not available in prior operating systems.

Payload behavioural studies, scheduling, optimization and system tools for payload control

Marta Bertran
26/09/2022
ALICE Tier-1 Tier-2 Workshop

Extra slides

RAM/CPU Core ratio analysis



```
mon_data=> select ram_per_core, count(1) from (select host_id, (a::bigint/b::bigint/1024/102.4)::int/10. ram_per_core from (select host_id, (select (test_me
ssage_json->>'RAM_kB_MemTotal') from sitesonar_tests where host_id=x.host_id and test_name='ram_info') a,(select (test_message_json->>'CPU_AMOUNT') from sit
esonar_tests where host_id=x.host_id and test_name ='cpuset_checking') b from (select host_id from sitesonar_tests s where last_updated > extract(epoch from
 now()-'1 week'::interval)::int and test_name='ram_info') x) y where a is not null and b is not null and length(a)>0 and length(b)>0) z group by 1;
      ram_per_core      | count
------------------------+-------
 1.00000000000000000000 |     2
     1.2000000000000000 |     1
     1.3000000000000000 |     8
     1.4000000000000000 |     7
     1.5000000000000000 |   105
     1.6000000000000000 |    29
     1.7000000000000000 |     3
     1.9000000000000000 |   311
     2.0000000000000000 |  1291
     2.1000000000000000 |     1
     2.2000000000000000 |   464
     2.3000000000000000 |     4
     2.4000000000000000 |   198
     2.5000000000000000 |     8
     2.6000000000000000 |   367
     2.7000000000000000 |     4
     2.8000000000000000 |     2
     2.9000000000000000 |  1704
     3.1000000000000000 |  1117
     3.3000000000000000 |    78
     3.4000000000000000 |    48
     3.5000000000000000 |   343
     3.7000000000000000 |     1
     3.8000000000000000 |     2
     3.9000000000000000 | 11710
     4.4000000000000000 |     1
     4.5000000000000000 |   187
     4.7000000000000000 |    31
     4.9000000000000000 |     1
     5.2000000000000000 |   532
     5.4000000000000000 |    86
     5.9000000000000000 |     4
     6.3000000000000000 |    61
     6.9000000000000000 |     1
     7.2000000000000000 |     1
     7.7000000000000000 |   122
     7.8000000000000000 |   126
     7.9000000000000000 |   273
     9.0000000000000000 |     9
     9.4000000000000000 |    14
    11.8000000000000000 |    63
    13.8000000000000000 |     2
    15.7000000000000000 |    13
(43 rows)
```

Total of 14826 over 19335 hots (76.68%) have more RAM per core than required