Concurrency in ROOT

S. Hageboeck (CERN, EP-SFT) for the ROOT team

1. IMT and Shared Resources

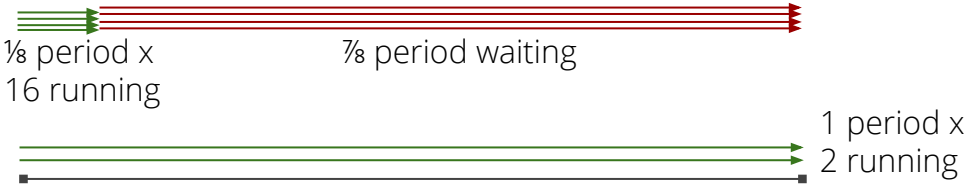
- All our tutorials use `ROOT::EnableImplicitMT()`
- This uses the hardware concurrency (#logical)
  - Special case: CPU pinning → Number of pinned cores (tbb default)
- Bad idea on shared resources
  - 1. "Classic" batch cluster (no containers):
    - Put 64 jobs on machine with 128 (logical) cores
    - IMT spawns 128 threads each → 8192 threads
    - Node goes down (Happened last week in Lausanne)
  - 2. SWAN (=docker): Limit CPU usage with linux control groups

SWAN Example

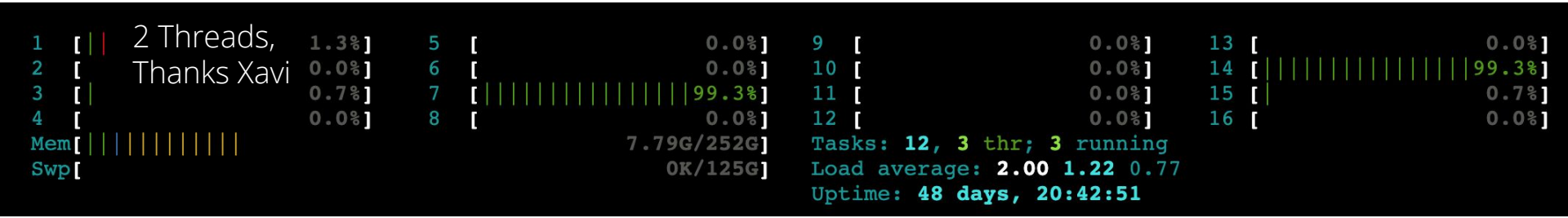
- SWAN (run inside linux control group)
  - SWAN uses control groups, but **does not pin processes**
    (→ tbb infers the wrong number of threads)
  - Test: request 2 CPUs, happen to run on 16 core machine
    - Simple RDF analysis (few filters + invariant mass of muons)
    - Define, run & destroy comp. graph 10x, take best of 3 runs

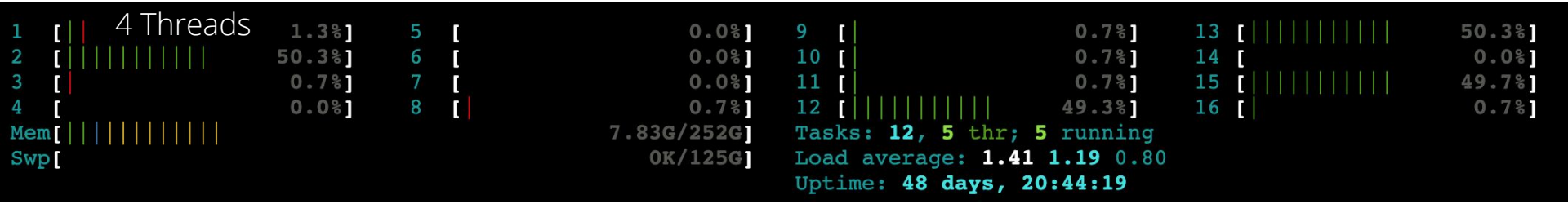
No IMT	IMT(2)	**IMT()** = 16 threads	IMT(4)
1:02 min	0:37 min	1:05 min	0:46 min

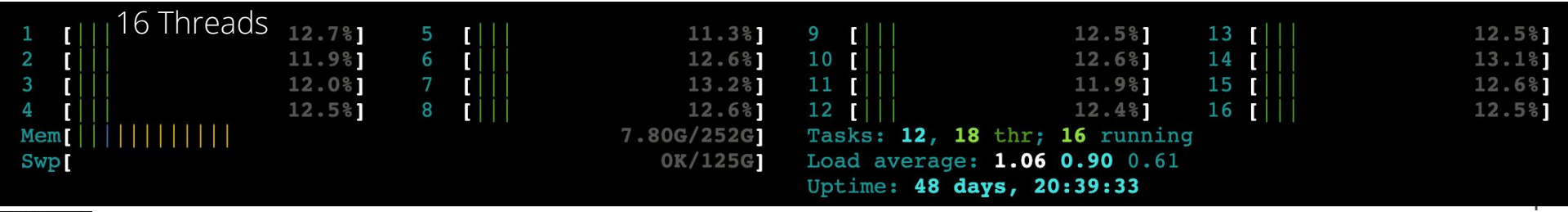
- Why? Linux scheduler starts & stops threads when over-allocating



SWAN, 2 CPUs







Remedies

- In control groups (docker == SWAN), standard in linux since >5 years:

```
bash-4.2$ more /sys/fs/cgroup/cpu/cpu.cfs_quota_us
200000
bash-4.2$ more /sys/fs/cgroup/cpu/cpu.cfs_period_us
100000
```

- Batch systems
  - PBS/torque: ${PBS_NUM_PPN}
  - condor (requested 2 CPUs):

```
$ echo ${PBS_NUM_PPN}
2
```

```
$ grep "Cpus.*=" ${_CONDOR_JOB_AD}
MachineAttrDetectedCpus0 = 10
Cpus = "2"
MachineAttrCpus0 = 2
RequestCpus = 2
```

- I think that ROOT should support these. In use at CERN (+ grid?)
  https://sft.its.cern.ch/jira/browse/ROOT-10479

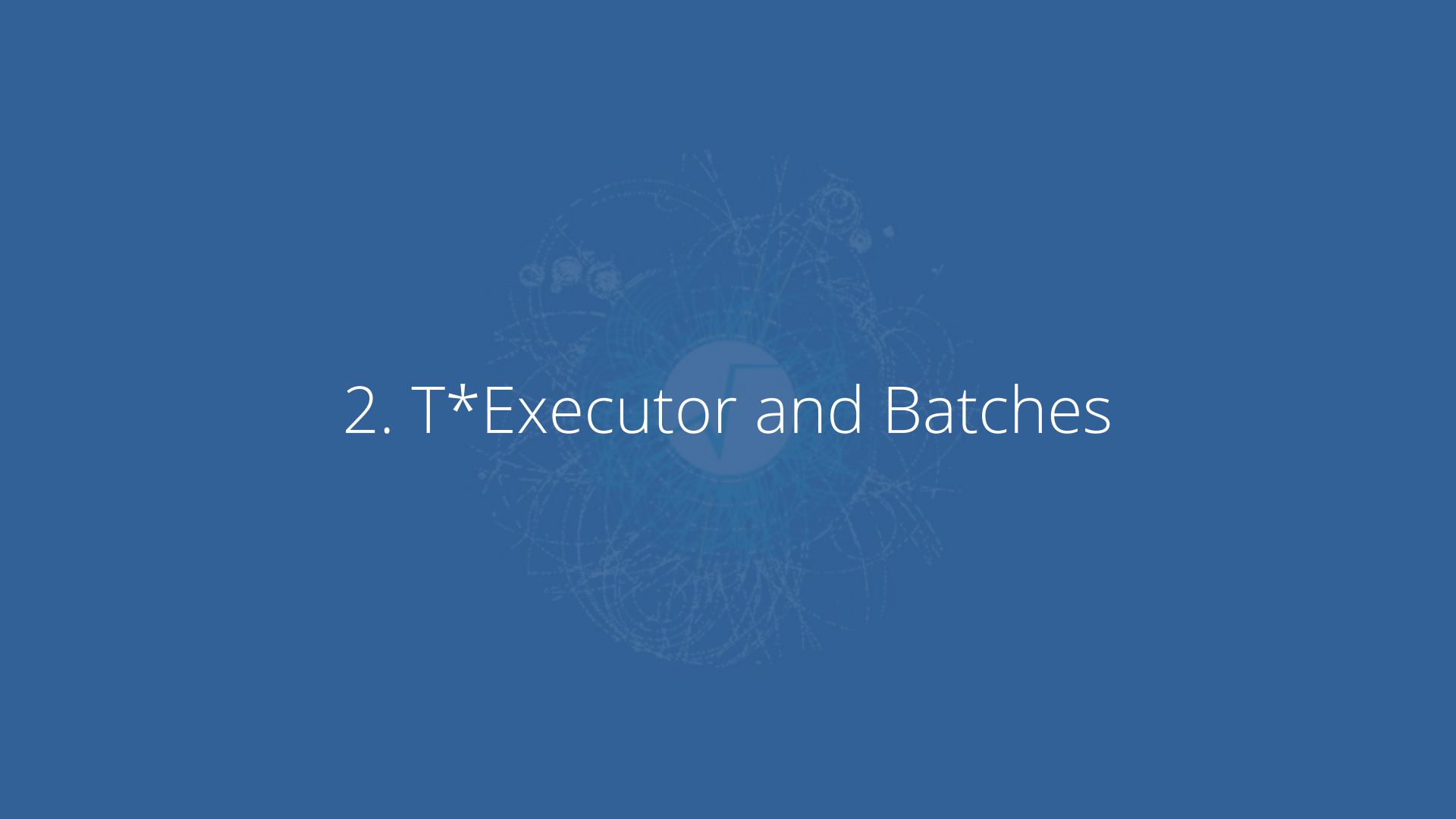
Possible Strategy

Either: User set number of threads manually - we are done

OR:

1. Check if in control group with limited CPU quota
   https://github.com/root-project/root/pull/4684/files
2. Check CPU affinity
3. Check if batch system allocated specific number of CPUs
4. Check for number of logical CPUs

2. T*Executor and Batches



2. T*Executor and Batches

- TExecutor is designed to process one element per call:

```
template<class F >
void Foreach (F func, unsigned nTimes, unsigned nChunks=0)
    Execute func (with no arguments) nTimes in parallel. More...

template<class F , class INTEGER >
void Foreach (F func, ROOT::TSeq< INTEGER > args, unsigned nChunks=0)
    Execute func in parallel, taking an element of a sequence as argument. More...

template<class F , class T >
void Foreach (F func, std::vector< T > &args, unsigned nChunks=0)
    Execute func in parallel, taking an element of an std::vector as argument. More...

template<class F , class T >
void Foreach (F func, const std::vector< T > &args, unsigned nChunks=0)
    Execute func in parallel, taking an element of a std::vector as argument. More...
```

- func can be:

```
T func();
T func(int_t i);
```

2. T*Executor and Batches

- For efficient batch processing (+ vectorisation) propose to extend to `T func(Int_t begin, Int_t end)`
- Or imitate tbb approach (my favourite):

```
func = [&](const tbb::blocked_range<std::size_t>& range, double init) ->
double {
  double f = init;
  for (std::size_t j = range.begin(); j < range.end(); ++j) {
    … //Vectorised batch processing
  }
  return f;
};
```

  - blocked_range automatically partitioned by tbb

Need for a General Executor

- Currently, cannot program against TThreadExecutor
  - ifdef-ed out if IMT disabled
  - TExecutor has a reduced interface (e.g. no ForEach) and takes backend as template parameter
- Proposal:
  - Refactor TExecutor:
    `TExecutor(nThread = 0, ExecutionPolicy = Thread)`
  - Give TExecutor the full interface
    Map(Reduce), ForEach, chunks, Func(void), Func(i), Func(begin, end)
  - IMT on → ExecutionPolicy = Thread
  - IMT off/not available → ExecutionPolicy = Sequential

RDF Interface

3. RDF and Scalar + Vector Fills

- https://root-forum.cern.ch/t/use-of-rdataframe-to-make-profile-plots-from-one-scalar-and-one-vector/37106
- What do we do for this?

```
df.Histo2D<double>(TH2DModel,"scalar", "vector<double>");
```

- NB: This broadcasts automatically

```
df.Histo2D<double>(TH2DModel,
    "vector<double>", "vector<double>", "weight");
```

Broadcast the first double to the size of the vector, and fill once for each entry in the vector. (5 votes) @Stephan Hageboeck, @Xavier Valls Pla, @Oliver Lantwin, @Olivier Couet and @Patrick Bos	Broadcasting is cool, but it needs to be an opt-in feature. Something like `broadcast=true` is required. (1 vote) @Massimiliano Galli
Error out because this doesn't make sense. (2 votes) @Stefan Wunsch and @Jim Pivarski	Not sure what's the right thing in this situation. (1 vote) @Jakob Blomer

3. RDF and Scalar + Vector Fills

- Some users might be surprised by automatic broadcasting
- Current interface could be adapted:

```
Profile1D(const TProfile1DModel & model,
  std::string_view    v1Name = "",
  std::string_view    v2Name = "",
  broadcast = true)
```

- `Profile1DBroadcast( … )`