An Introduction to the POP Methodology

Jonathan Boyle, Numerical Algorithms Group

EU H2020 Centre of Excellence (CoE)

1 December 2018 – 30 November 2021

Grant Agreement No 824080

Introduction to the POP CoE

The POP Centre of Excellence

- POP is a CoE in **Performance Optimisation and Productivity**
  - Promotes **best practices in parallel programming**
- POP provides **FREE** services
  - for (EU) industrial and academic, (parallel) codes and users
  - across all application areas, platforms, scales
- giving users
  - a precise understanding of application and system behaviour
  - suggestions & support on how to refactor code in the most productive way
- often leading to
  - faster code / bigger jobs / better science
  - a significant Return on Investment
  - an edge against the competition

Return on Investment Examples

Application Savings after POP Proof-of-Concept

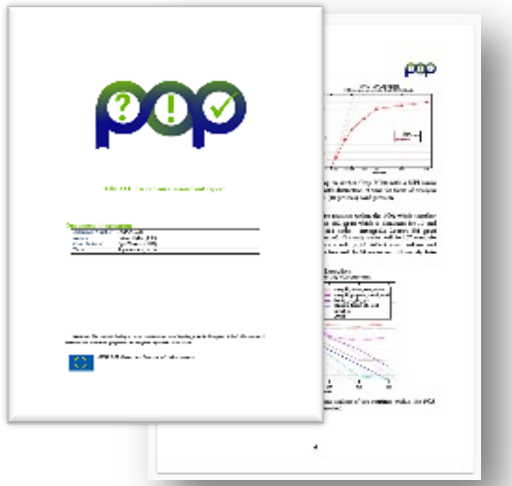
- POP PoC resulted in 72% faster-time-to-solution
- Production runs on ARCHER (UK national academic supercomputer)
- Improved code saves €15.58 per run
- Yearly savings of around €56,000 (from monthly usage data)

Application Savings after POP Performance Assessment

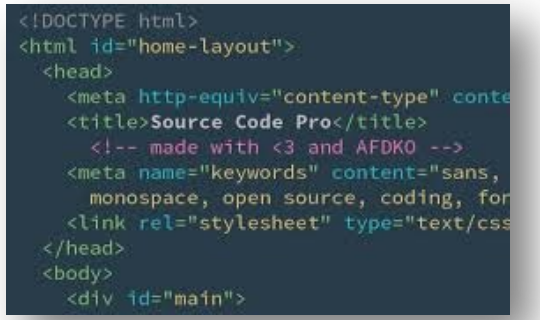
- Cost for customer implementing POP recommendations: €2,000
- Achieved improvement of 62%
- €20,000 yearly operating cost
- Resulted in yearly saving of €12,400 in compute costs ⇨ ROI of 620%

FREE Services provided by the CoE

- **Parallel Application Performance Assessment**
  - Primary service
  - Identifies performance issues of customer code
  - If needed, identifies the root causes of the issues found and qualifies and quantifies approaches to address them (recommendations)
  - 1-3 months effort



- **Proof-of-Concept**
  - Follow-up service
  - Experiments and mock-up tests for customer codes
  - Kernel extraction, parallelisation, mini-app experiments to show effect of proposed optimisations
  - 3-6 months effort



Note: Effort shared between our experts and customers!

Module 1

Introduction to Parallel Performance Analysis

What you'll learn

Why it's difficult to understand poor parallel performance

- The limitations of traditional speedup and efficiency plots
- What trace data is
- Challenges of interpreting trace data
- The philosophy behind the POP metrics

A brief introduction to profiling and tracing

The importance of performance

Q: Are we making good use of parallel hardware?

- **To speed up computation we run on multiple cores**
  - Modern processors are multicore e.g. desktops, mobile devices
  - Computers may contain multiple processors e.g. supercomputers
- **Huge speedup is possible on large HPC machines**
  - Single processor speedup is often important too

Q: Is our speedup close to the maximum possible?

- Ideal speedup = number of CPU cores used
  - Relative to 1 core
- Anything less is a waste of resources
  - e.g. hardware, electricity, money

Q: How do we improve low speedup?

Traditional parallel performance

- Plot **speedup** or **efficiency** to measure **relative** performance, i.e.

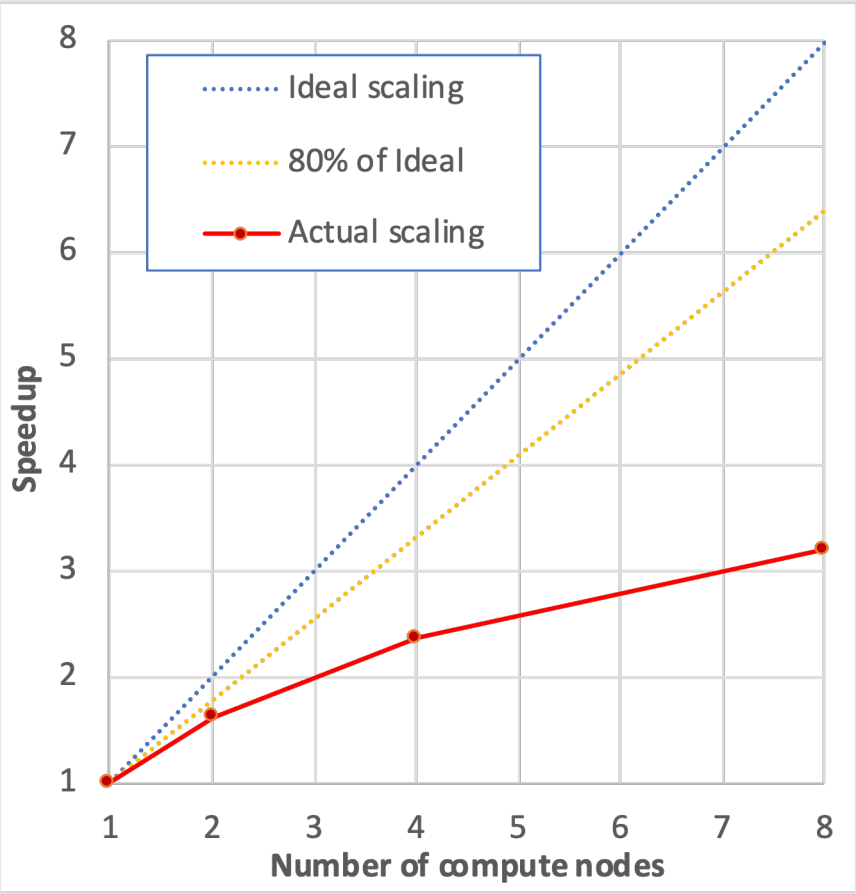
- **Step 1**: measure run time $\mathbf{T_N}$ for some range of **N**
  - **N** is usually **number of compute nodes** on HPC hardware
    - Or perhaps number of CPU cores for thread based parallelism

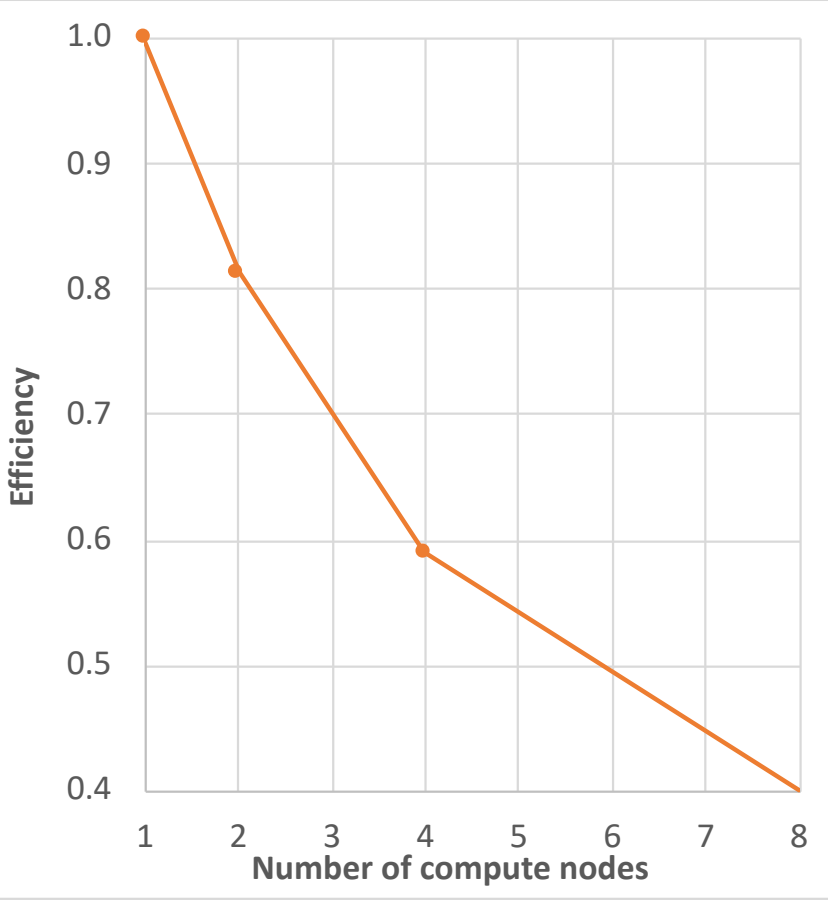
- **Step 2**: plot scaling or efficiency

$$\text{Speedup} = \frac{T_{reference}}{T_N}$$

$$\text{Efficiency} = \frac{T_{reference}}{(N \times T_N)}$$

Traditional scaling & efficiency plots





- Reference case is **a 68 core** compute node
  - Note: speedup and efficiency is always 1 for the reference case

The problem with these metrics

- **They tell us nothing about the causes of poor performance**
  - e.g. load imbalance, idle cores, parallelism overheads, etc

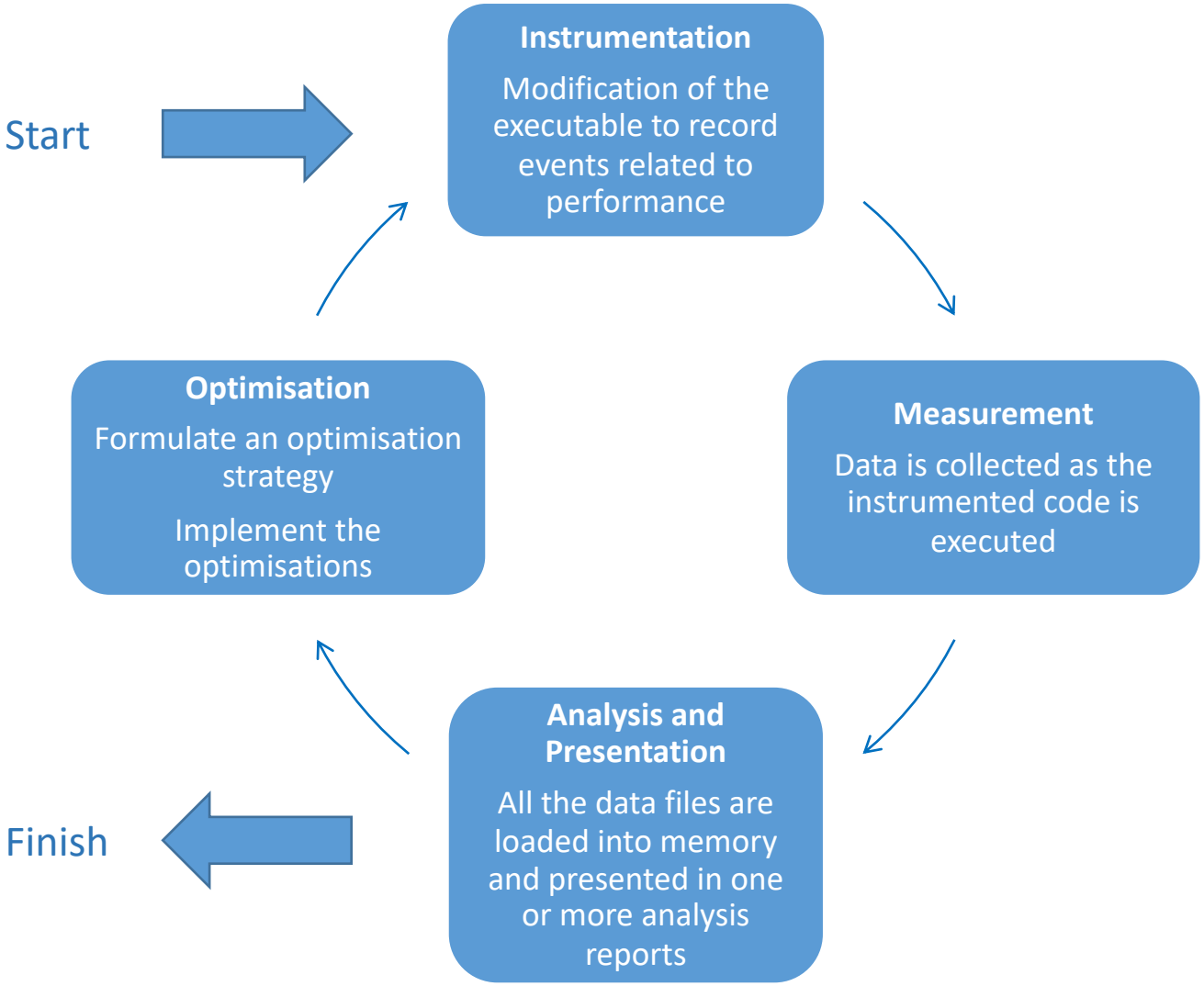
- For a parallel reference case they tell us nothing about absolute performance
  - Scaling and efficiency for the reference case is always 1
  - And running a single core case may be impractical

- **So we add:**
  - **Step 3**: generate trace data to profile the performance
  - **Step 4**: interpret the trace data

Performance Profiling

- *Profiling* refers to the monitoring of a code's behaviour as it executes
  - To capture the behaviour of the application under production conditions
  - To understand and quantify the efficiency of resource usage
  - To identify inefficiencies and where improvements can be made
- Profiling helps build an overall picture of the application e.g.
  - Where is the time spent?
  - Is performance consistent?
  - Where are the underperforming regions?
  - Does parallel code make good use of all cores?
- The profiling results can be used to:
  - Guide the code refactoring effort
  - Provide a baseline from which improvements can be measured

The Profiling-Optimisation Cycle



What is trace data?

- Tracing tools record data **at specific points** during program execution
  - i.e. a timestamp plus various information about what's going on at that point
    - Tracing tools will vary in what they record
- For parallel performance we typically record (at least) all parallel events
  - Usually also hardware counter data
    - e.g. number of processor cycles and instructions

- Trace files usually contain a huge amount of data
  - There are often many parallel events
  - They may also collect data for every function call, perhaps for every loop

- Trace visualisation tools display trace data, e.g.
  1. Timelines showing selected events per core - often too detailed to interpret
  2. Metrics - usually an overwhelming amount of data

Understanding trace data is usually a big challenge

Why analysing trace data is hard (1)

There are many possible causes of poor performance e.g.

- Imbalance in the amount of computation per core
- Dependencies between computation on different cores
  - e.g. MPI synchronisation issues leading to idle cores
- Additional work arising from the parallelism e.g.
  - Useful work which can't be parallelised & must be replicated over the cores
  - Parallelism overheads e.g. time in MPI data transfer
- Reduction in processor instruction throughput (IPC) or frequency
  - e.g. Memory issues
    - Such as NUMA (non-uniform memory access), memory cache use, etc.

Qs: Which issues are impacting performance? Which issues to fix first?

Why analysing trace data is hard (2)

- There are typically a huge number of (parallel) events during execution
- **The trace data is often too complex to view as a single timeline**

- Some trace visualisation tools can post-process trace data to calculate a range of metrics for profiling
- **But the number of metrics and amount of data is often overwhelming**

Q: How do we know where to start with the trace data? What are we looking for?

A solution - The POP metrics

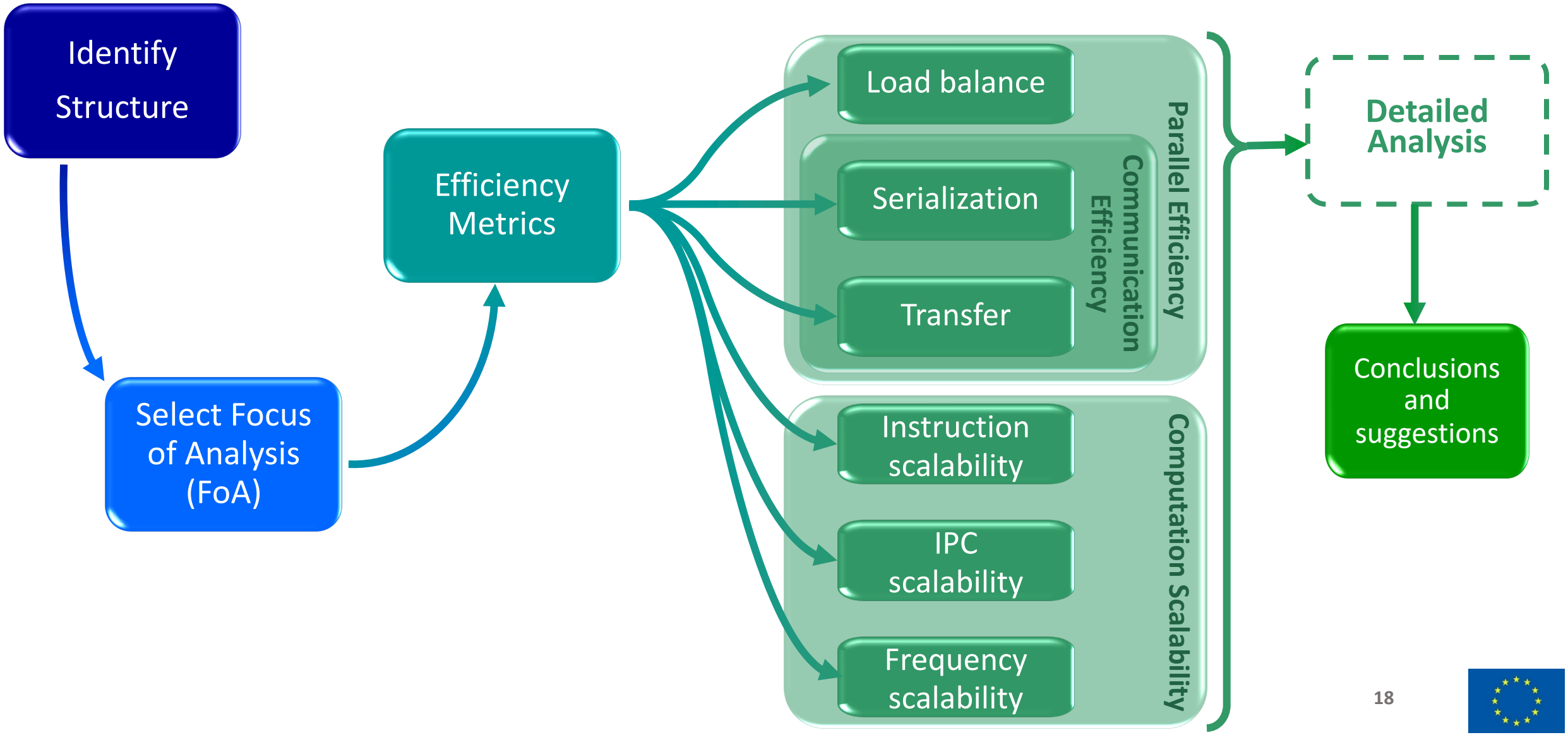
The idea is simple but extremely powerful

- Devise a **simple** set of performance metrics using values easily obtained from the trace data i.e.
  - Absolute efficiency metrics
  - Scaling metrics
- Low values indicate **specific** causes of poor parallel performance

We use these metrics to understand

1. **What are the causes of poor performance**
2. **What to look for in the trace data**

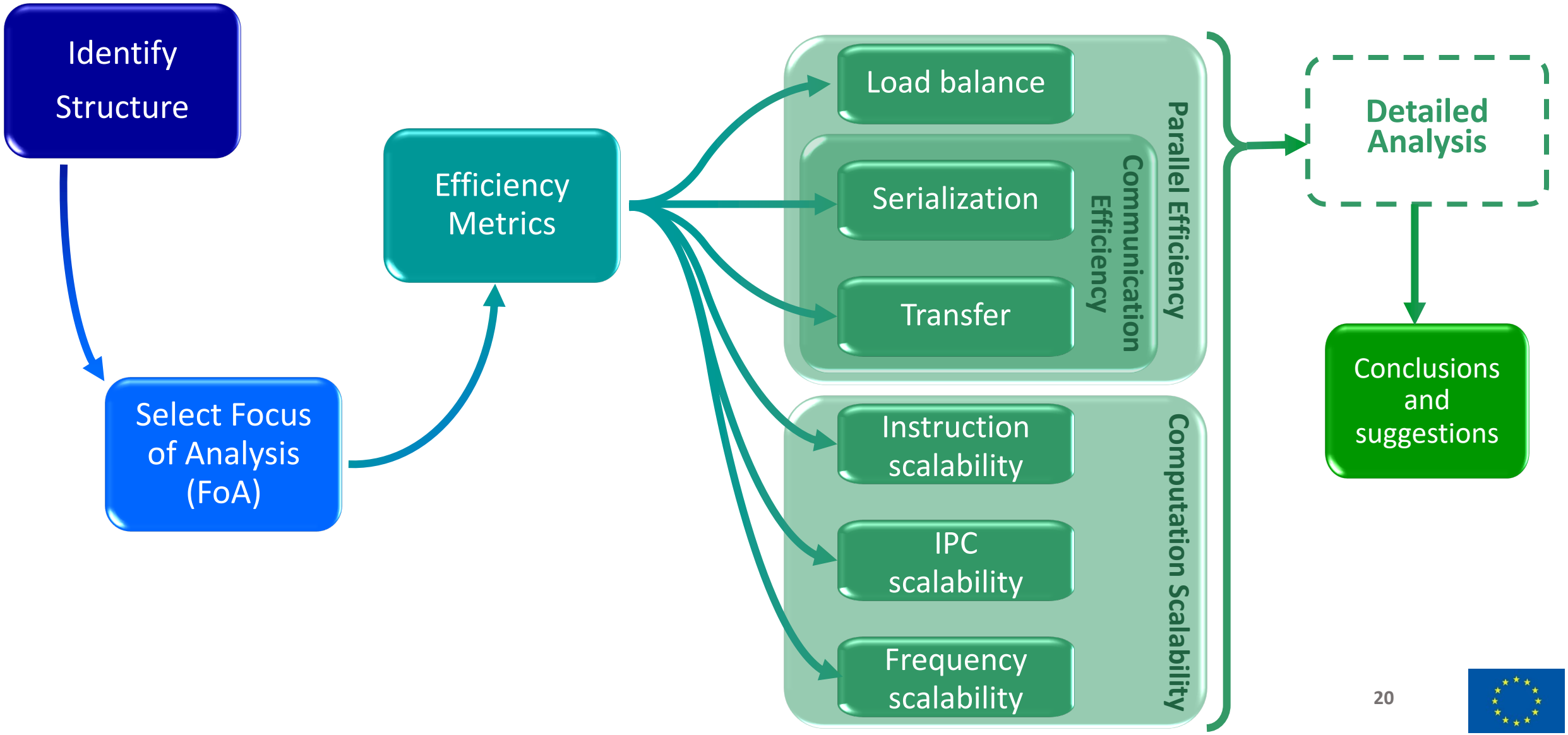
Trace analysis - the journey



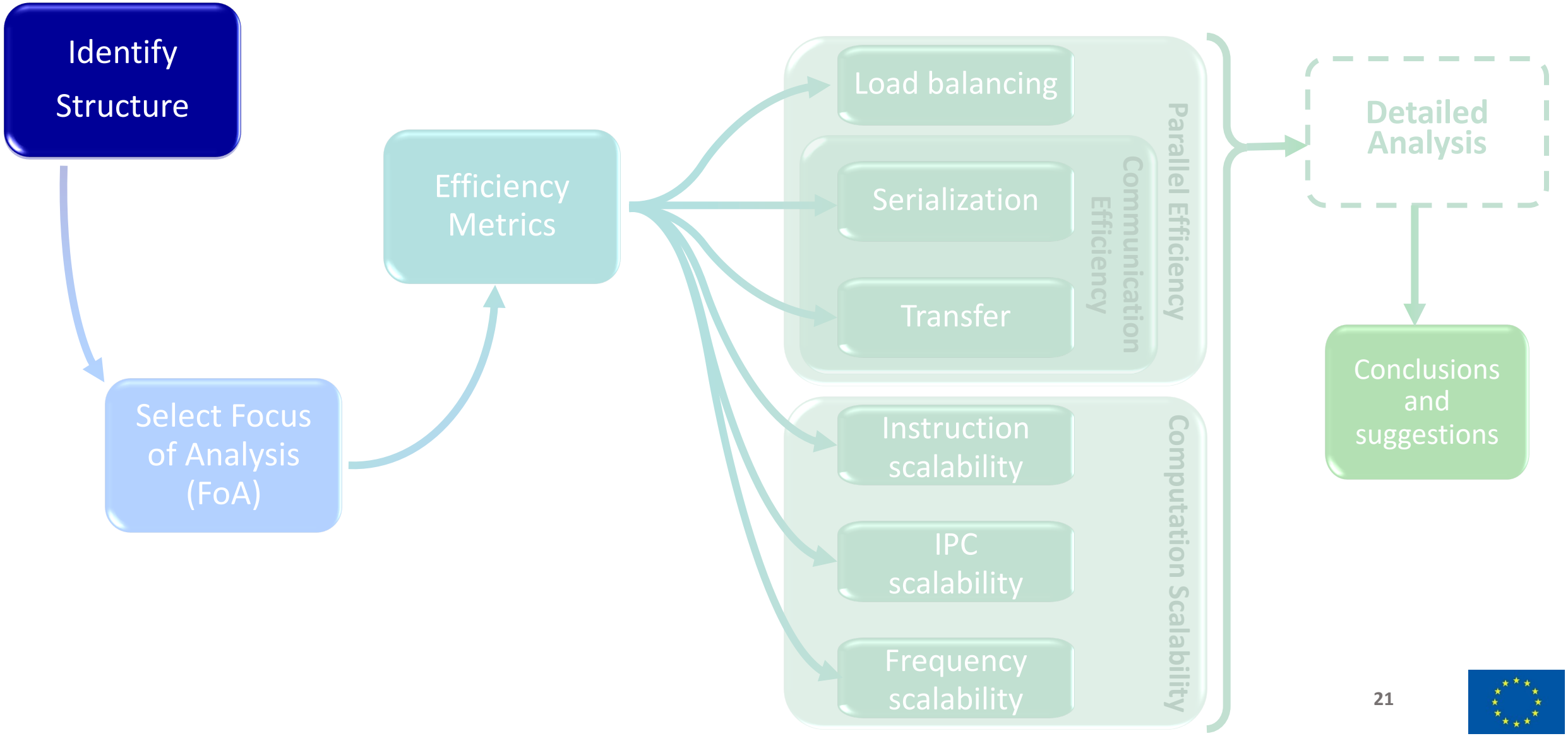
Module 2

Introduction to Parallel Performance Analysis using the POP metrics

The journey


Identify Structure
Select Focus of Analysis (FoA)
Efficiency Metrics
Load balance
Serialization
Transfer
Communication Efficiency
Parallel Efficiency
Instruction scalability
IPC scalability
Frequency scalability
Computation Scalability
Detailed Analysis
Conclusions and suggestions

The journey


Identify Structure
Select Focus of Analysis (FoA)
Efficiency Metrics
Load balancing
Serialization
Transfer
Communication Efficiency
Parallel Efficiency
Instruction scalability
IPC scalability
Frequency scalability
Computation Scalability
Detailed Analysis
Conclusions and suggestions

Identify structure

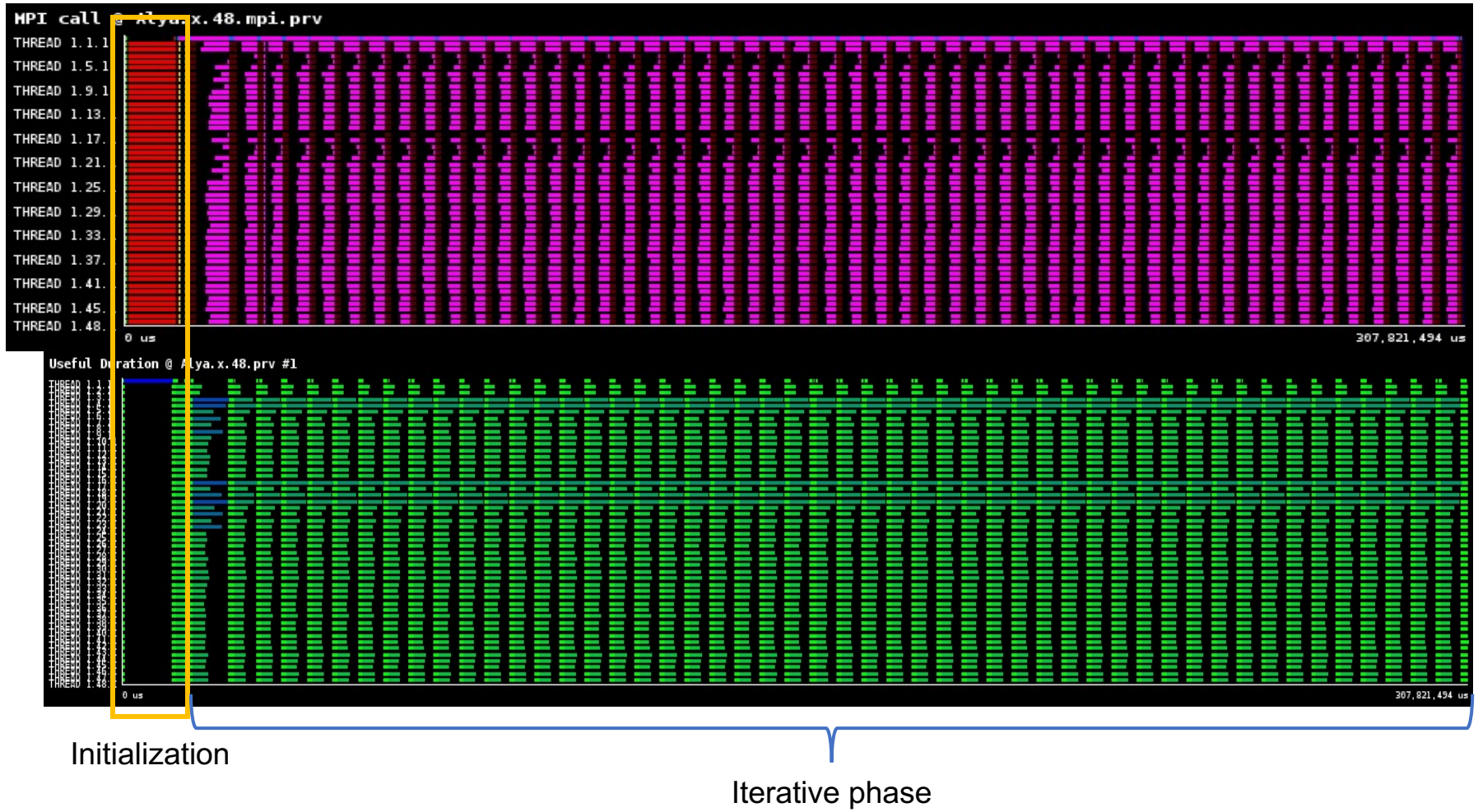
- Objectives:
  - Understand general structure
  - Identify initialization/finalization phases
  - Detect iterative pattern
  - Understand granularity

- Different levels of ‘complexity’ → Different levels of knowledge

- The following examples show trace data collected using BSC’s tools
  - Extrae to collect trace data
  - Paraver to visualize it

Identify timeline structure

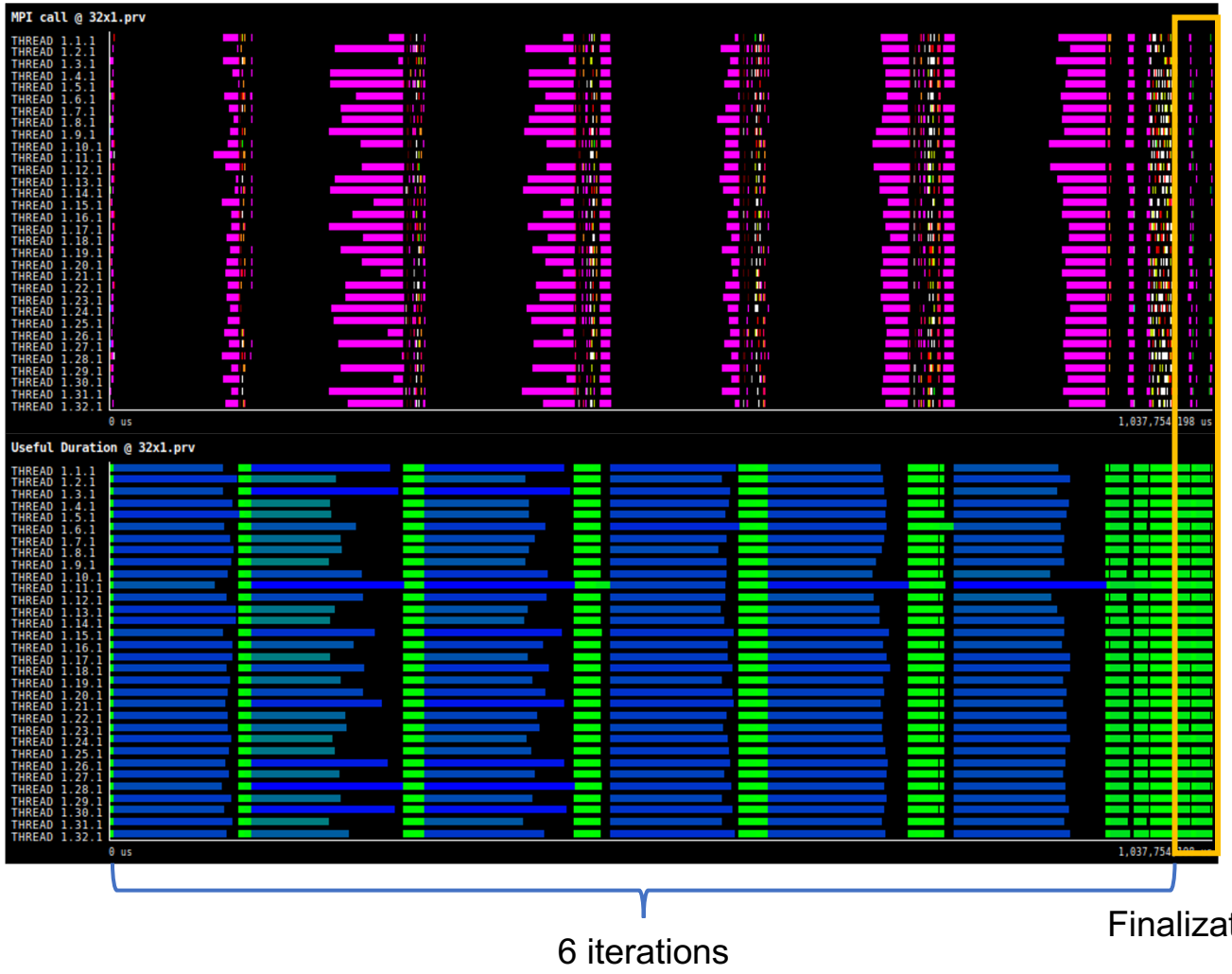
- Usually use “MPI calls” view or “Useful Duration” views



- Clear iterative pattern
- With an initialization phase
- All iterations are similar
  - We can select a few to analyze

Identify structure

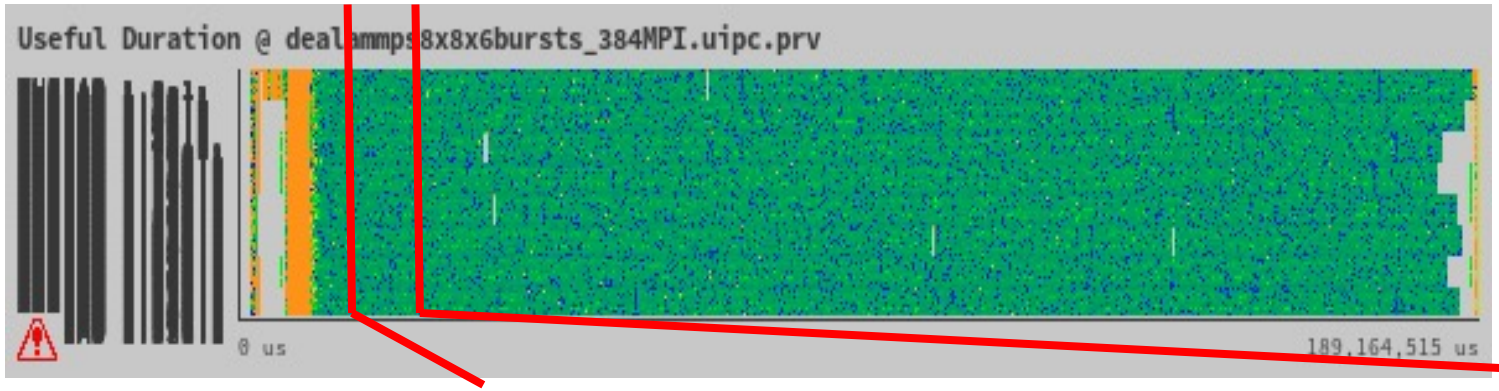
- Not always easy

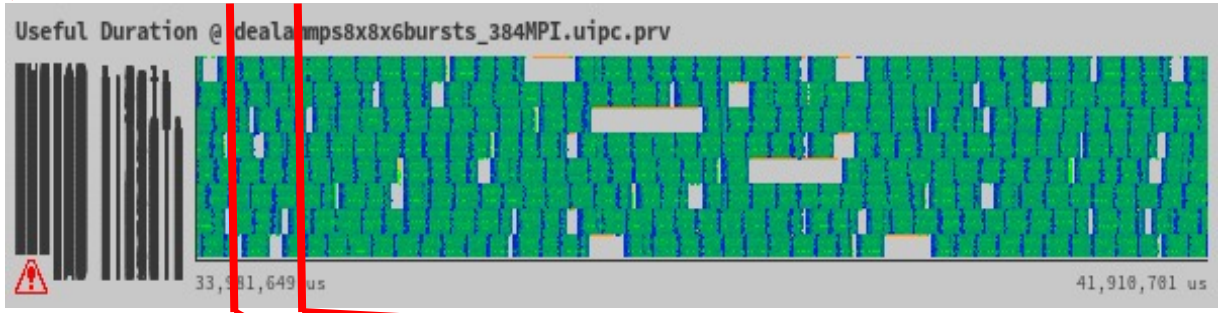


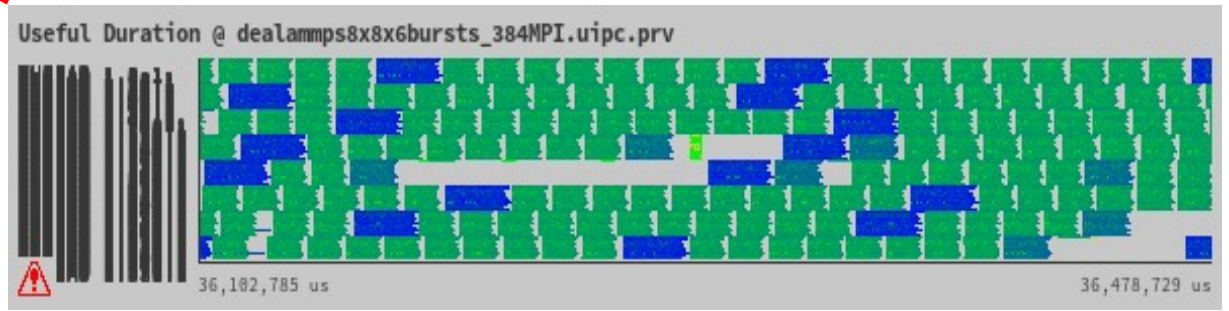
- There are 6 iterations and one finalization phase
- Iterations are not regular along time
  - Different patterns of load balance
  - Different patterns of duration

Identify structure

- Not always easy



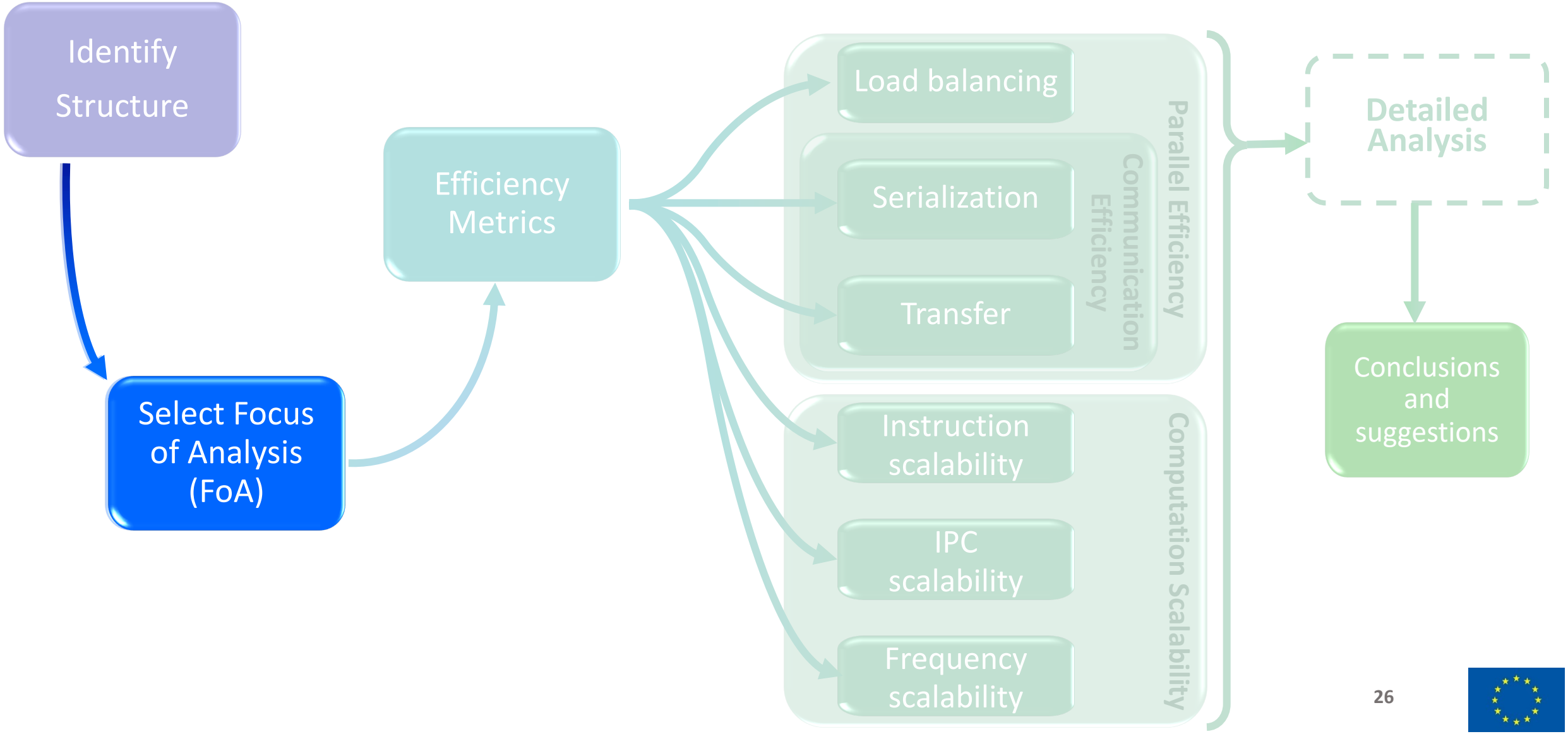




- It is not easy to detect an iterative structure
- No global synchronizations

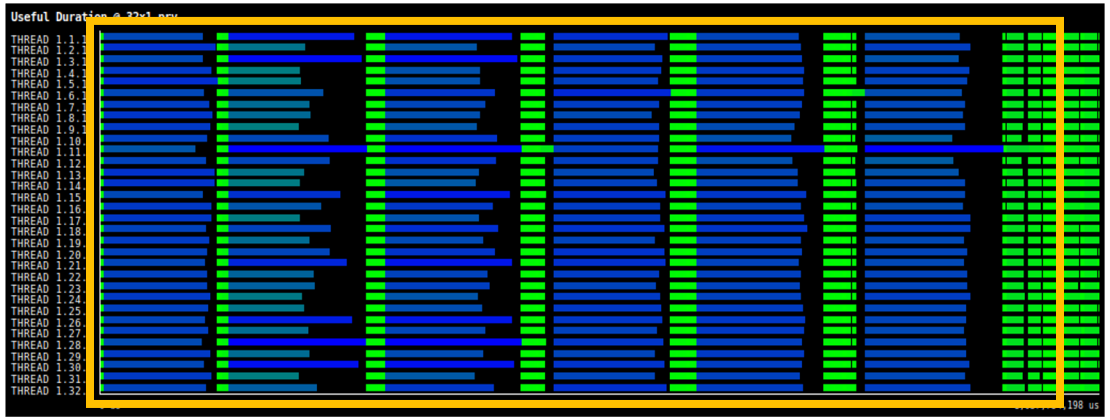
- Zooming in we detect that synchronizations only happen at node level
- Not possible to determine iterations

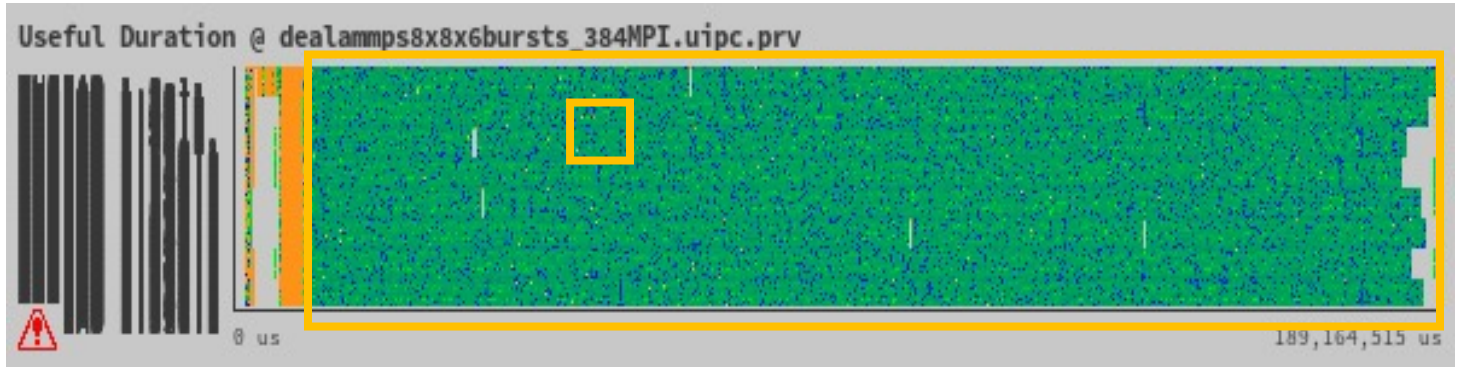
The journey


Identify Structure
Select Focus of Analysis (FoA)
Efficiency Metrics
Load balancing
Serialization
Transfer
Communication Efficiency
Parallel Efficiency
Instruction scalability
IPC scalability
Frequency scalability
Computation Scalability
Detailed Analysis
Conclusions and suggestions

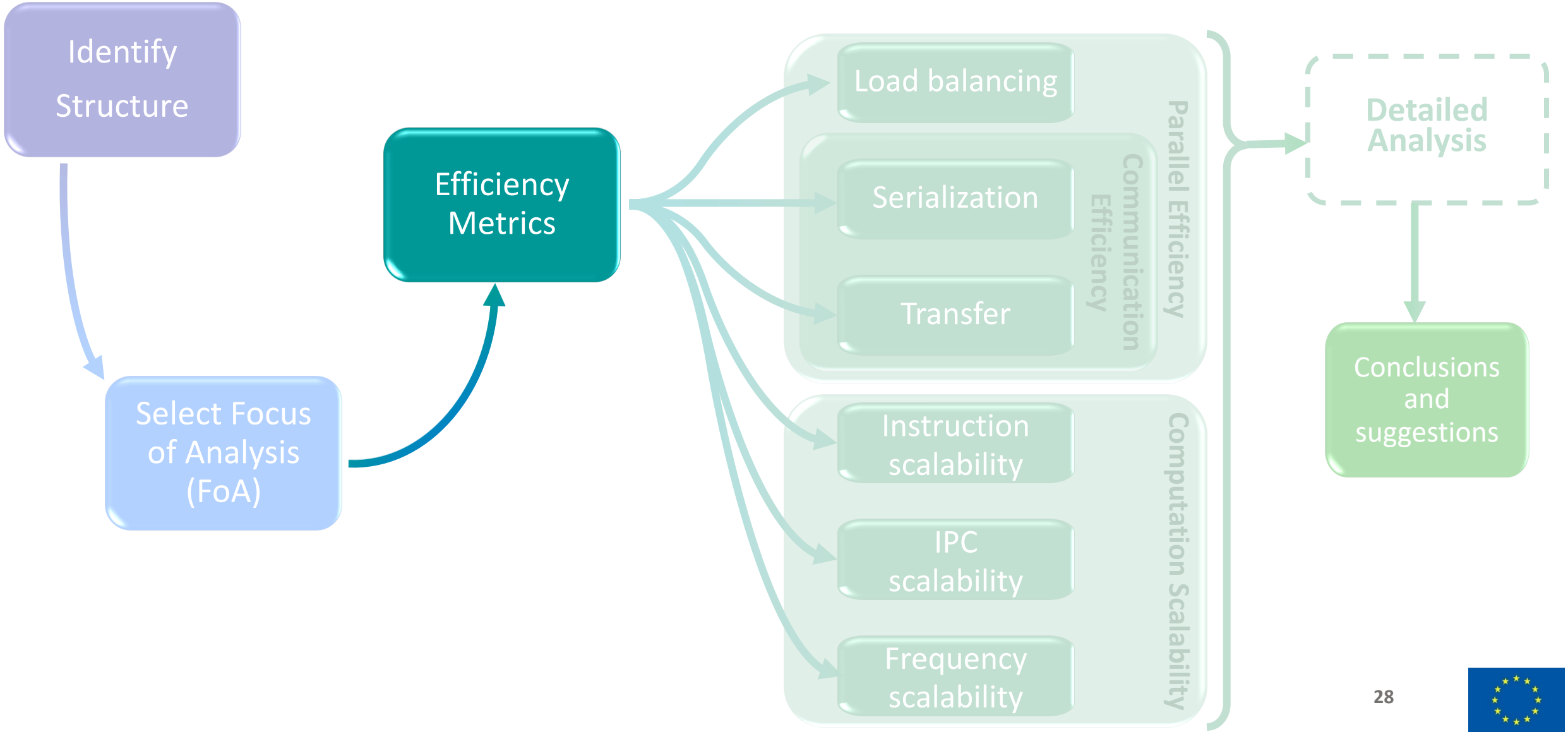
Select Focus of Analysis (FoA)

- **Objective:** Select the region we want to analyze
  - No one correct answer, depends on the context of the analysis
  - For the same trace we may select two different FoAs to perform two different studies with two different objectives

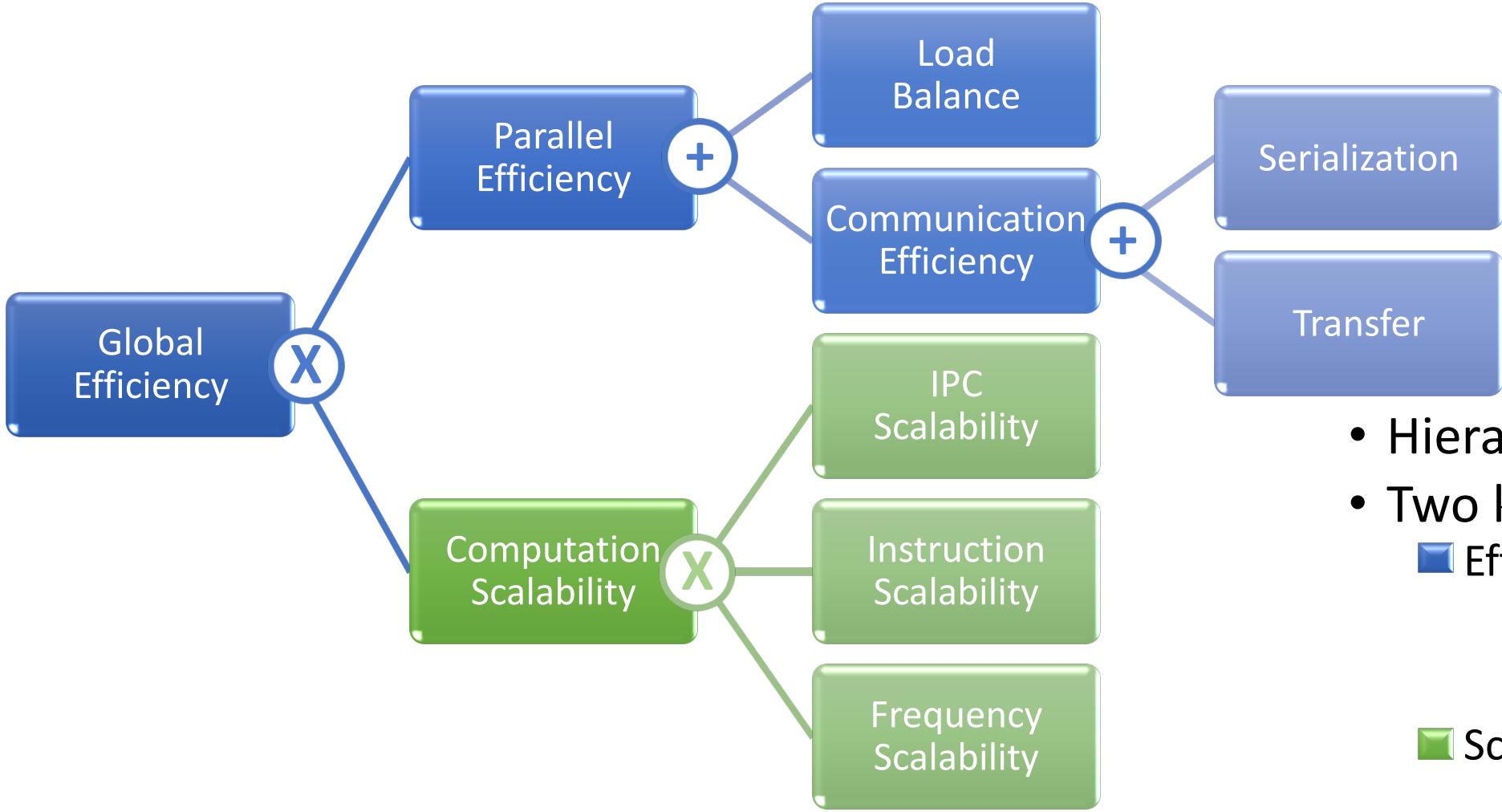




The journey


Identify Structure
Select Focus of Analysis (FoA)
Efficiency Metrics
Load balancing
Serialization
Transfer
Communication Efficiency
Parallel Efficiency
Instruction scalability
IPC scalability
Frequency scalability
Computation Scalability
Detailed Analysis
Conclusions and suggestions

POP MPI 'additive' Efficiency Metrics



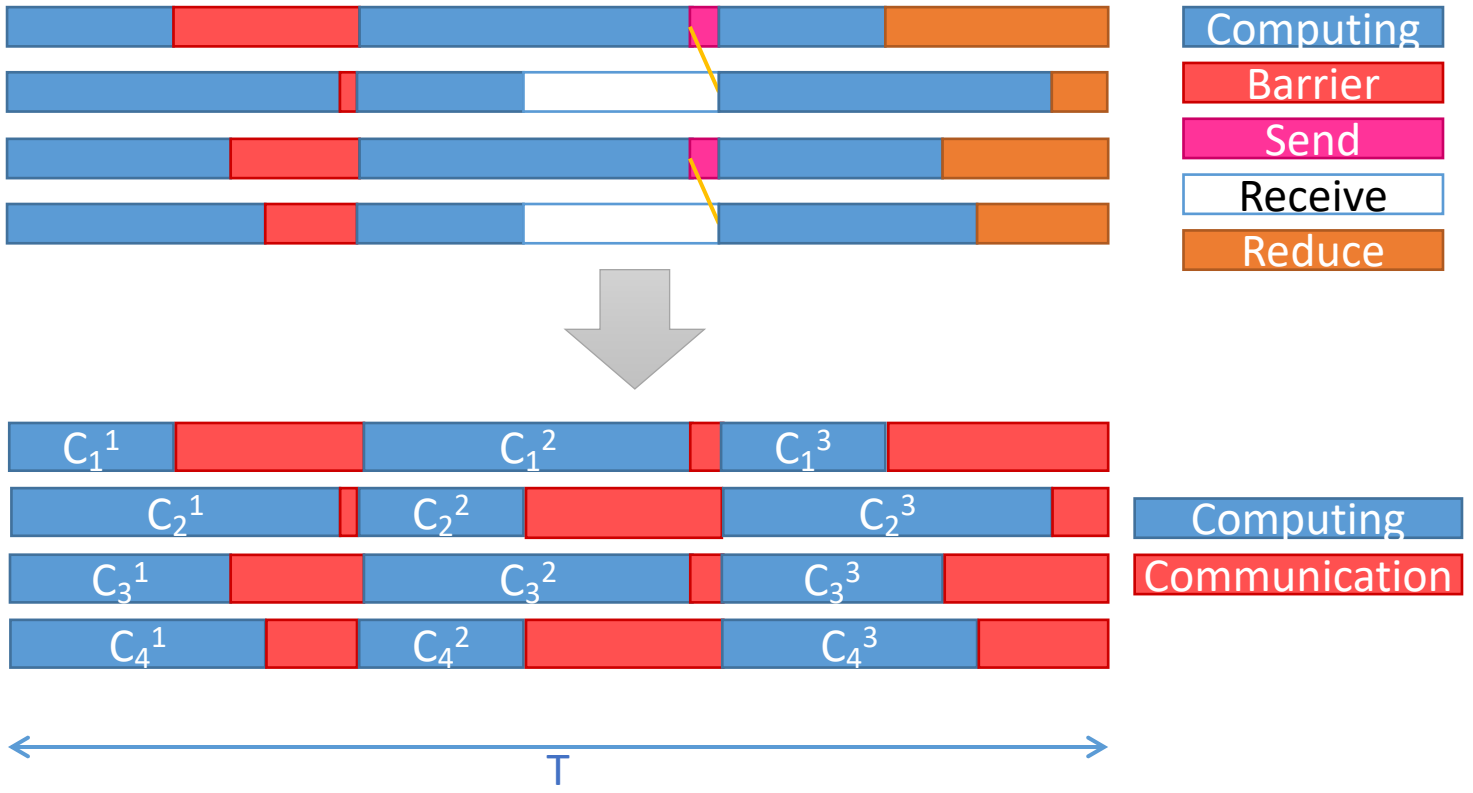
- Hierarchical model
- Two kind of metrics:
  - Efficiency metrics
    - Absolute metrics
    - 1 to 100% scale
    - Additive
  - Scalability metrics
    - Relative to a base case
    - 100% for the base case

The idea

- A simple set of metrics
- Easily calculated from trace data
- Each metric points at a specific cause or causes of poor parallel performance
- A hierarchy
  - Top level metrics give a broad overview
  - Low level metrics allow us to drill down into the details

Some semantics first

- The state of a process is simplified to two values
  - Useful == Computing
  - Not useful == Otherwise



- T = Elapsed time
- P = Number of processes
- c_i = Compute time of process i
  - $c_i = \sum_{j=1}^{n} c_i^j$
- C = Total compute time
  - $C = \sum_{i=1}^{P} c_i$

What do we want to know?

- There are some obvious first questions for any form of parallelism
  1. How good is the parallelism?
  2. Is the total time in useful computation constant?
     - ‘Useful’ means computation outside parallel libraries i.e. executing your code
- These questions apply to any form of parallelism, not just MPI

- Using a trace visualisation tool we can usually quickly find:
  1. **Sum of all time in useful computation**
  2. **Maximum time in useful computation measured over the cores**
- Often we can also find:
  3. **Total number of processor useful cycles**
  4. **Total number of processor useful instructions**

What can we calculate using this data?

How good is the parallelism?

- Ideally we split total useful computation evenly over all cores, with no overheads from parallelism i.e.

'Ideal runtime' = sum of time in useful computation / N_c

= average useful computation

N_c = number of cores

- Hence define:

Parallel Efficiency = Ideal runtime / Runtime

= Average useful computation / Runtime

- Note: this measures an **absolute** efficiency

Is useful computation time constant?

- Ideally the sum of time in useful computation remains constant as we increase the number of cores used
  - But in practice total useful computation often increases

- Define:

Computation scaling =

Reference total useful computation / Total useful computation

- Reference is typically the smallest time in useful computation

Child-metrics for comp. scaling

- Define:

Useful IPC = Useful instructions / Useful cycles

Useful Frequency = Useful cycles / Sum of time in useful computation

- Note: **Time = Instructions / (IPC x Frequency)**

- We can split Computation Scaling into three scaling metrics
  1. Useful IPC Scaling
  2. Useful Instruction Scaling
  3. Useful Frequency Scaling
- Multiplying these three scaling gives us the Computation Scaling

How good is performance overall?

- We can combine Parallel Efficiency and Computation Scaling by multiplying

Global efficiency = Parallel Efficiency x Computation Scaling

- This is the same as the Parallel Efficiency that would be obtained if the time in useful computation had stayed the same as the reference case

A hierarchy of metrics

- We already have a nice set of useful metrics
  - They can be used with MPI, OpenMP, Pthreads, etc.
- There is a hierarchy
  - **Global Efficiency** splits into **Parallel Efficiency** & **Computation scaling**
  - **Computation scaling** splits into **Instruction**, **IPC** and **Frequency Scaling**
- The metrics give us insight into
  - Overall performance
  - Is the problem in the parallelism or the computation?
  - Is poor Computation Scaling due to:
    - Increasing useful instructions
    - Reducing IPC
    - Reducing frequency
- We can use these for benchmarking
  - e.g. to compare performance before and after code modifications

Example of using the POP metrics

#nodes	1	2	4	8
Global Efficiency	0.95	0.38	0.24	0.14
↳ Parallel Efficiency	0.95	0.53	0.42	0.34
↳ Computation Scaling	1.00	0.73	0.57	0.41
↳ IPC Scaling	1.00	0.85	0.67	0.50
↳ Instruction Scaling	1.00	0.94	0.95	0.94
↳ Frequency Scaling	1.00	0.91	0.89	0.89

- We immediately see **Parallel Efficiency is very low** on > 1 compute node
  - Around 2/3 of run time on 8 nodes is overhead from poor parallelism
- **Computation Scaling is also poor**
  - Time in useful computation on 8 nodes > twice that on 1 node
  - Caused mostly by poor IPC scaling
- **Instruction & Frequency Scaling are good**

Example 2

Number of nodes	1	2	4	8
Global Efficiency	0.86	0.70	0.50	0.34
↳ Parallel Efficiency	0.86	0.72	0.60	0.47
↳ Computation Scaling	1.00	0.96	0.84	0.74
↳ IPC Scaling	1.00	0.97	0.94	0.98
↳ Instruction Scaling	1.00	0.96	0.88	0.76
↳ Frequency Scaling	1.00	1.03	1.02	0.99

- The main problem here is the parallelism
  - 50% of the run time is due to parallelism inefficiencies
  - Our next question: what is causing this?
- Computation Scaling is low
  - The instruction count is increasing!
- IPC and Frequency Scaling are good

Example 3

Number of CPU cores	1	4	8	12	16	20	24
Global Efficiency	0.99	0.76	0.53	0.44	0.38	0.36	0.30
↳ Parallel Efficiency	0.99	0.85	0.76	0.73	0.68	0.65	0.58
↳ Computation Scaling	1.00	0.89	0.69	0.60	0.56	0.55	0.51
↳ IPC Scaling	1.00	0.92	0.78	0.74	0.70	0.69	0.65
↳ Instruction Scaling	1.00	1.00	1.00	1.00	1.00	0.99	0.99
↳ Frequency Scaling	1.00	0.97	0.89	0.82	0.81	0.80	0.78

- Poor Parallel Efficiency yet again!
  - This needs further investigation
- And Computation Scaling contributes to 50% of run time on 24 cores
  - Main contribution is reducing IPC
  - But reducing processor frequency also plays a part

Module 3

POP metrics for MPI applications

Q: Causes of low parallel efficiency?

- Let's now extend these metrics for specific parallel methodologies
  - i.e. split Parallel Efficiency into suitable child metrics
    - Ideally one child metric per source of inefficiency
- For **MPI** - we typically want to understand costs due to
  - Load imbalance of useful computation
  - Time inside MPI
    - Is it due to data transfer?
    - Or due to time waiting in MPI caused by dependencies (i.e. synchronisation points)?
- How do we extend the hierarchy further?

The idea of ‘additive’ metrics

- Let’s revisit POP’s **Parallel Efficiency**
  - This measures the efficiency of the parallelisation
- We measured useful ***computation*** and ***runtime*** on ***n*** cores
  - ***comp*** is useful computation per CPU core

Parallel Efficiency = average(*comp*) / *runtime*

- **Average(*comp*)** is the **ideal runtime** when considering parallelism

Parallel Efficiency ideal runtime = average(*comp*) = sum(*comp*) / *n*

- It is the runtime we would get if all useful work (***comp***) is split evenly over the cores with no overheads from the parallelism

The idea……

- Look again at POP's **Global Efficiency**
  - This measures efficiency of the parallelisation combined with inefficiency due to any increase in useful computation
- Define ***comp_ref*** as useful computation on our reference case
- And ***n*** is number of cores for the other cases under consideration

Global Efficiency = [sum(*comp_ref*)/*n*] / *runtime*

- We can think of the 'ideal runtime' for Global Efficiency

Global Efficiency ideal runtime = sum(*comp_ref*)/*n*

- This is the runtime we would get if the work from the reference case was split evenly over ***n*** cores with no overheads

POP's 'additive' metric methodology

- We'll define some performance metrics that can be mapped to **known issues**
- For each issue / performance metric we define an 'ideal runtime'
- Then for each issue

Efficiency = ideal runtime / runtime

- We can also define:

Inefficiency = 1 - efficiency

  - For optimal performance: efficiency = 1, and inefficiency = 0

- This defines a hierarchy where we can **add** 'child' inefficiency values to get the 'parent' inefficiency value
  - Since **Inefficiency = 'time cost of issue(s)' / runtime**
  - Splitting the cost of the bottleneck into the individual contributions is the same as splitting the inefficiency value

Efficiency and inefficiency

- **Efficiency** tells us what fraction of our actual execution time would remain after removing a specific issue or set of issues
  - Efficiency is a nice metric when we want to think in terms of ideal runtimes

- **Inefficiency** tells us what fraction of our actual execution time would be removed by eliminating a specific issues or set of issues
  - We can add child inefficiencies to obtain the parent metric value
    - Inefficiency is a nice metric when we want to split a parent metric into contributions

MPI child metrics

- We split Parallel Efficiency into
  1. Load balance efficiency – this tells us the cost of the computational imbalance that exists independently of the MPI
  2. Communication efficiency – tells us the total cost of MPI

- We next split Communication Efficiency into
  1. Transfer efficiency
     - The cost of time in data transfer
     - This time would vanish if the network had zero latency & infinite bandwidth
  2. Serialization Efficiency
     - The cost of MPI dependencies i.e. time cost of MPI which occurs even on an ideal network
     - This would vanish if MPI dependencies could somehow be removed

Load Balance Efficiency

- The **Load Balance Efficiency** reflects how well the distribution of work over the processes is done in the application.

- The **Load Balance Inefficiency** is measured by the difference between the maximum time the processes spend in computation and the average time the processes spends in computation.

- Global Efficiency
  - Parallel Efficiency
    - Load Balance Efficiency
    - Communication Efficiency
      - Serialization Efficiency
      - Transfer Efficiency
  - Computation Efficiency
    - IPC Scaling
    - Instruction Scaling
    - Frequency Scaling

$$\text{Load Balance Efficiency} = \frac{\text{runtime} - \max(\text{comp}) + \text{avg}(\text{comp})}{\text{runtime}}$$

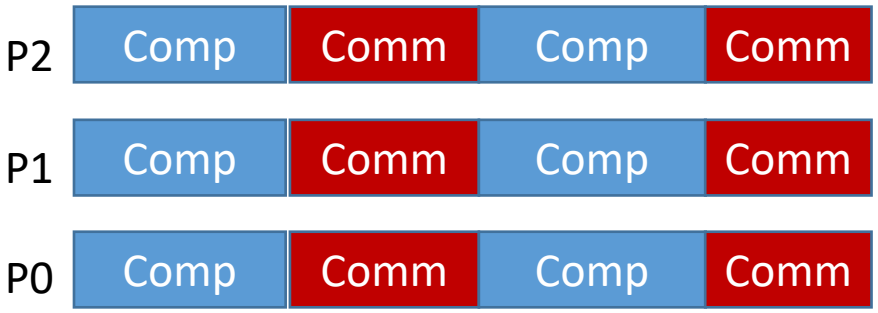
Load Balance Efficiency

- The **Load Balance Efficiency** reflects how well the distribution of work to processes is done in the application

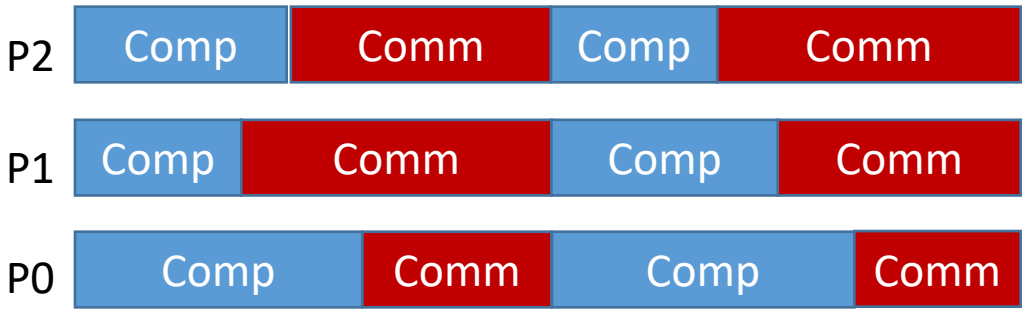
$$\text{Load Balance Efficiency} = \frac{\text{runtime} - \max(\text{comp}) + \text{avg}(\text{comp})}{\text{runtime}}$$

- Global Efficiency
  - Parallel Efficiency
    - Load Balance Efficiency
    - Communication Efficiency
      - Serialization Efficiency
      - Transfer Efficiency
  - Computation Efficiency
    - IPC Scaling
    - Instruction Scaling
    - Frequency Scaling

Example 1: good load balance (LB = 100%)



Example 2: bad load balance (LB = 77%)



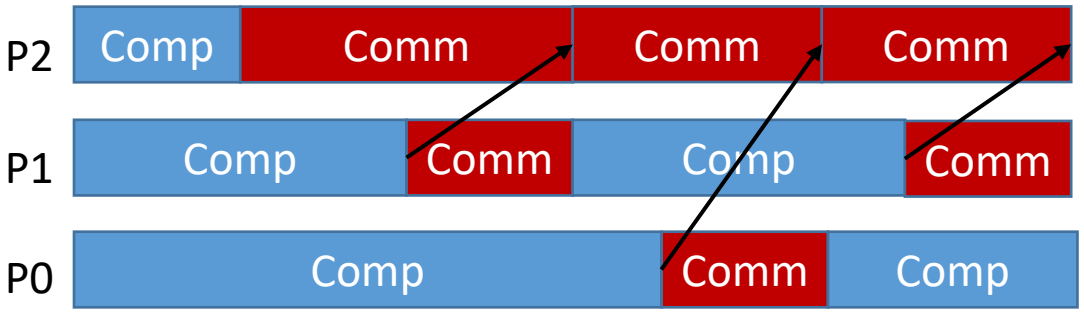
Communication Efficiency

- The **Communication Efficiency** reflects the loss of efficiency by MPI communication
- The **Communication Efficiency** can be computed as

$$\text{Communication Efficiency} = \frac{\max(\text{comp})}{\text{runtime}}$$

- Global Efficiency
  - Parallel Efficiency
    - Load Balance Efficiency
    - Communication Efficiency
      - Serialization Efficiency
      - Transfer Efficiency
  - Computation Efficiency
    - IPC Scaling
    - Instruction Scaling
    - Frequency Scaling

Example:



Compute	Communication	Efficiency
1 sec.	5 sec.	$^1/_6$
4 sec.	2 sec.	$^4/_6$
5 sec.	1 sec.	$^5/_6$

Com.Eff. = $^5/_6$ = 83%

Communication Efficiency

- The **Communication Efficiency** reflects the loss of efficiency by communication.
- The **Communication Efficiency** can be split further into Serialization Efficiency and Transfer Efficiency.

- Global Efficiency
  - Parallel Efficiency
    - Load Balance Efficiency
    - Communication Efficiency
      - Serialization Efficiency
      - Transfer Efficiency
  - Computation Efficiency
    - IPC Scaling
    - Instruction Scaling
    - Frequency Scaling

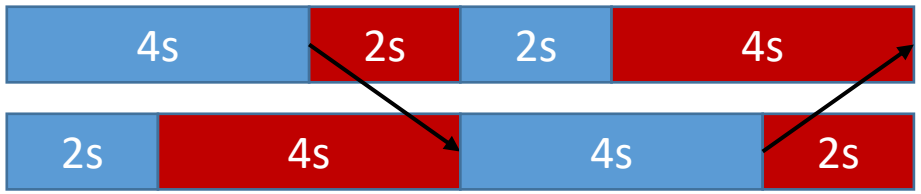
Transfer Efficiency

- The **Transfer Efficiency** describes loss of efficiency due to actual data transfer.
- The **Transfer Efficiency** can be computed as

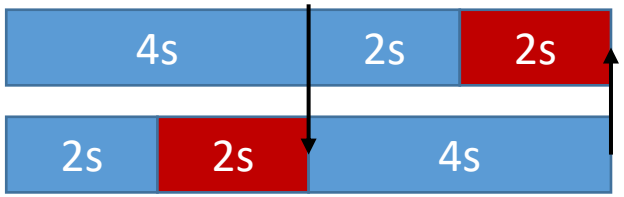
$$\text{Transfer Efficiency} = \frac{\text{ideal network runtime}}{\text{runtime}}$$

- Global Efficiency
  - Parallel Efficiency
    - Load Balance Efficiency
    - Communication Efficiency
      - Serialization Efficiency
      - Transfer Efficiency
  - Computation Efficiency
    - IPC Scaling
    - Instruction Scaling
    - Frequency Scaling

Execution on a real network



Simulation on an ideal network



Tr.Eff. = $\frac{8}{12}$ = 67%

= Computation

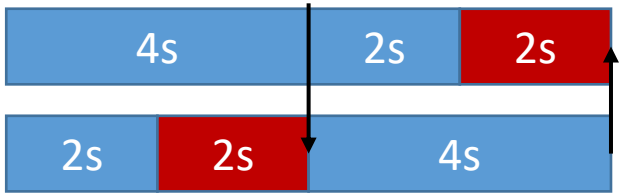
= Communication

Serialization Efficiency

- The **Serialization Efficiency** describes loss of efficiency due to dependencies between processes.
- Dependencies can be observed as waiting time in MPI calls where no data is transferred, i.e. where one process has not yet arrived at a communication call.
- On an ideal network with instantaneous data transfer these inefficiencies are still present.

- Global Efficiency
  - Parallel Efficiency
    - Load Balance Efficiency
    - Communication Efficiency
      - Serialization Efficiency
      - Transfer Efficiency
  - Computation Efficiency
    - IPC Scaling
    - Instruction Scaling
    - Frequency Scaling

Simulation on an ideal network



= Computation

= Communication

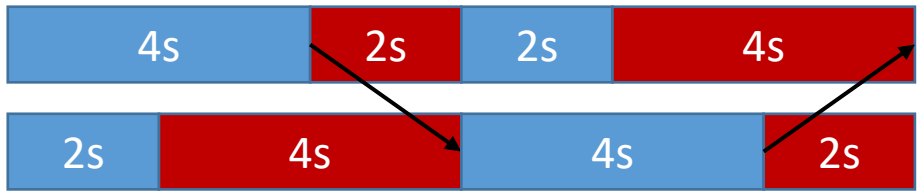
Serialization Efficiency

- Serialization Inefficiency cost is the difference between the ideal network runtime and the maximum time in useful computation

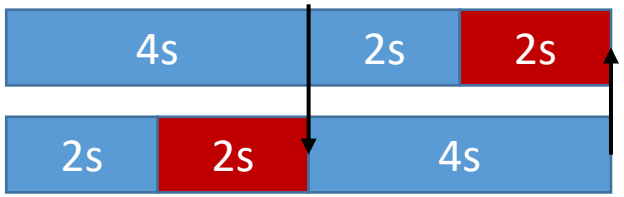
$$\text{Serialization Efficiency} = \frac{\text{runtime} - \text{ideal network runtime} + \max(\text{comp})}{\text{runtime}}$$

- Global Efficiency
  - Parallel Efficiency
    - Load Balance Efficiency
    - Communication Efficiency
      - Serialization Efficiency
      - Transfer Efficiency
  - Computation Efficiency
    - IPC Scaling
    - Instruction Scaling
    - Frequency Scaling

Execution on a real network



Simulation on an ideal network



Ser.Eff. = $\frac{10}{12}$ = 83%

= Computation

= Communication

MPI metrics - Example 1

Number of Processes	48	96	192	384	768
Global Efficiency	0.94	0.84	0.73	0.54	0.34
➥ Parallel Efficiency	0.94	0.89	0.81	0.71	0.55
➥Load Balance	0.97	0.97	0.96	0.97	0.97
➥Communication Efficiency	0.96	0.92	0.84	0.74	0.58
➥Transfer Efficiency	0.98	0.94	0.87	0.80	0.70
➥Serialization Efficiency	0.99	0.98	0.97	0.93	0.87
➥Computation Scaling	1.00	0.94	0.90	0.77	0.63
➥Instruction Scaling	1.00	0.95	0.88	0.74	0.60
➥IPC Scaling	1.00	1.00	1.03	1.04	1.05
➥Frequency Scaling	1.00	1.00	1.00	1.00	0.99

- Load balance, IPC Scaling and Frequency Scaling are good
- The main issues are Instruction Scaling and time in MPI data transfer
  - Some efficiency is lost due to serialization

MPI metrics - Example 2

	Original code				Refactored code		
#nodes	1	2	4	8	2	4	8
Global Efficiency	0.95	0.38	0.24	0.14	0.79	0.62	0.45
➥ Parallel Efficiency	0.95	0.53	0.42	0.34	0.80	0.68	0.57
➥ Load balance Efficiency	0.97	0.89	0.94	0.91	0.92	0.90	0.91
➥ Communication Efficiency	0.97	0.59	0.45	0.38	0.87	0.75	0.63
➥ Serialisation Efficiency	0.98	0.93	0.86	0.85	0.98	0.96	0.92
➥ Transfer Efficiency	1.00	0.64	0.53	0.45	0.88	0.79	0.69
➥ Computation Scaling	1.00	0.73	0.57	0.41	1.00	0.92	0.79
➥ IPC Scaling	1.00	0.85	0.67	0.50	0.99	0.94	0.84
➥ Instruction Scaling	1.00	0.94	0.95	0.94	0.96	0.95	0.95

- In the original code MPI data transfer and IPC scaling are very low
- In the new version of the code, we see big improvements in efficiency

How to calculate

- Most of the values can be easily extracted by hand using trace visualisation tools, but this is painful for large traces
- With Extrae traces Dimemas is required to find ideal network runtime
  - Intel's ITAC can also calculate this with ITAC traces
- Hardware counter data usually requires PAPI
  - Or VTtune
- NAG-PyPOP can be used to run the entire analysis i.e.
  - Extracts all the values needed
    - Uses Paramedir, another BSC tool (included with Paraver's Linux version)
  - Runs Dimemas
  - PyPOP can also chop traces for a region of interest if the Extrae API is used to switch tracing on & off

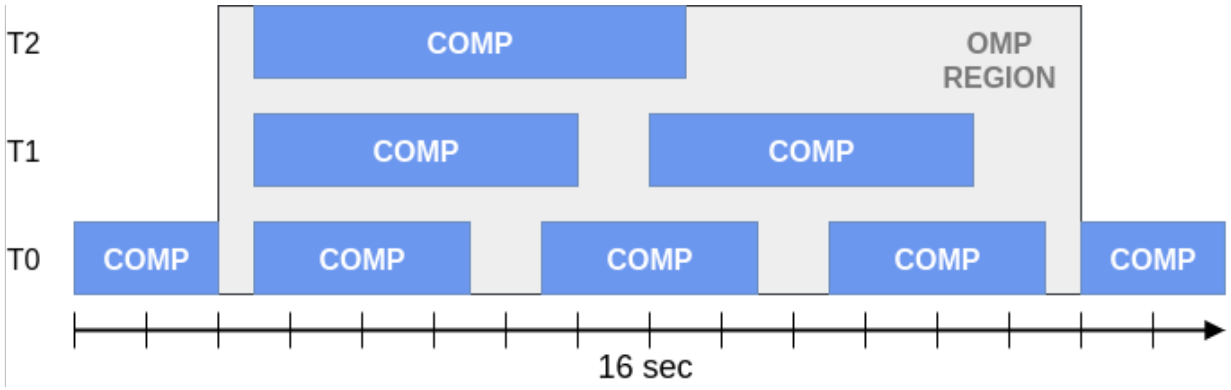
Module 4

POP metrics for OpenMP and Hybrid codes

Additive metrics for OpenMP

- What do we want to know for OpenMP code?
  1. How much inefficiency is due to execution outside OpenMP regions
  2. How much inefficiency is within OpenMP regions
- We can easily define metrics to calculate efficiencies for these
  - Serial region efficiency
    - Measures the cost of Amdahl's Law
  - OpenMP efficiency
- We can split the OpenMP Efficiency further e.g.
  - A contribution per OpenMP region
  - A contribution per source of inefficiency
    - e.g. load imbalance, etc
  - VTune does this very well

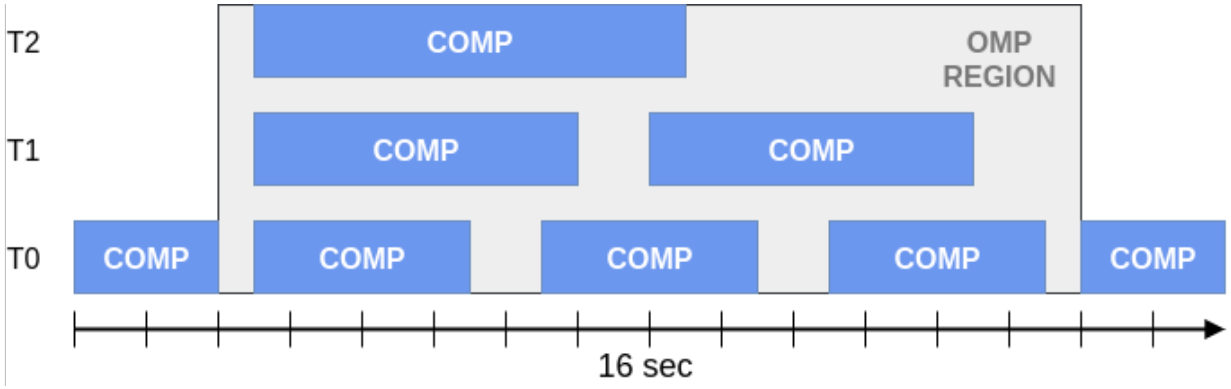
Additive metrics for OpenMP



- T2: OMP Computation = 1 x 6 sec
- T1: OMP Computation: 2 x 4.5 sec.
- T0: OMP Computation: 3 x 3 sec.

- OpenMP inefficiency is the difference between
  - **Time in OpenMP** (12s)
  - **Average OpenMP computation** (8s)
- The ideal runtime in this case is 16 - 12 + 8 = 12s
- OpenMP Efficiency is 12 / 16 = 0.75
  - i.e. ideal runtime / actual runtime
- 25% of the run time is caused by inefficiency within the OpenMP

Additive metrics for OpenMP



- T2: OMP Computation = 1 x 6 sec
- T1: OMP Computation: 2 x 4.5 sec.
- T0: OMP Computation: 3 x 3 sec.

- Serial Region inefficiency (for n_t threads) is time in serial computation multiplied by $(n_t-1) / n_t$
  - Time in serial computation is 4s
  - $(n_t-1) / n_t$ is 2/3
- The idea runtime would be 16 – 4 x (2/3)
- Serial Region Efficiency = [16 – 4(2/3)] / 16 = 0.83
- 17% of the run time is due to serial execution outside OpenMP

OpenMP metrics example

Number of cores	1	10	18	30	45
Global Efficiency	1.00	0.80	0.58	0.36	0.26
Parallel Efficiency	1.00	0.86	0.69	0.60	0.55
OpenMP Efficiency	1.00	0.95	0.81	0.74	0.70
Serial Region Efficiency	1.00	0.91	0.88	0.86	0.85
Computational Scaling	1.00	0.94	0.84	0.60	0.48
Instruction Scaling	1.00	1.01	1.02	1.00	1.00
IPC Scaling	1.00	0.92	0.82	0.61	0.50
Frequency Scaling	1.00	1.00	1.00	0.98	0.95

- Low Parallel Efficiency has 2 causes
  - Inefficiency withing OpenMP regions
  - Too much serial computation outside OpenMP
- IPC scaling is also a problem

Additive metrics for MPI + OpenMP

- We can start with Parallel Efficiency and split this into child metrics
- With additive metrics we have complete freedom about how to define child metrics

- One obvious option is to split Parallel Efficiency into
  1. Process Efficiency (ignores all thread inefficiencies)
  2. Thread Efficiency (ignores process inefficiencies)

Process Efficiency

- To assess Process Efficiency we want to know
  1. How evenly ‘useful work’ is distributed over the processes
  2. How much time the **processes** spend in MPI
     - Also cost of data transfer and serialisation due to dependencies
- These are the same performance issues as pure MPI code

- If we ignore the threading there are only three states, i.e.
  1. A process is in serial computation on the master thread: ***serial_comp***
  2. A process is inside an OpenMP parallel region: ***omp***
  3. A process is inside MPI and outside OpenMP

- When ignoring the threading we will assume only **time outside OpenMP and inside MPI** is non-useful

Process Efficiency

- We define ***serial_comp*** and ***omp*** as useful
  - Time in MPI and outside OpenMP is considered the performance issue

useful* = *serial_comp* + *omp

- And we define an 'ideal runtime' as **average(*useful*)**

Process Efficiency = average(*useful*) / *runtime*

- We can use the MPI additive methodology to define ideal run times that split Process Efficiency into
  - **Load Balance Efficiency** – cost of imbalance of ***useful*** over processes
  - **Transfer Efficiency** – cost of MPI time due to data transfer over network
  - **Serialization Efficiency** – remaining cost of time in MPI

Process Efficiency child metrics

- We define ‘ideal runtime’ values per issue that allow us to split Process Efficiency into child metrics
  1. **Process Communication Efficiency**

     = max(*useful*) / runtime

  2. **Process Load Balance Efficiency**

     = [*runtime* - max(*useful*) + average(*useful*)] / runtime

- We can similarly split Process Communication Efficiency
  - **Process Transfer Efficiency** measures cost of data transfer
  - **Process Serialization Efficiency** measures cost of time in MPI without transfer

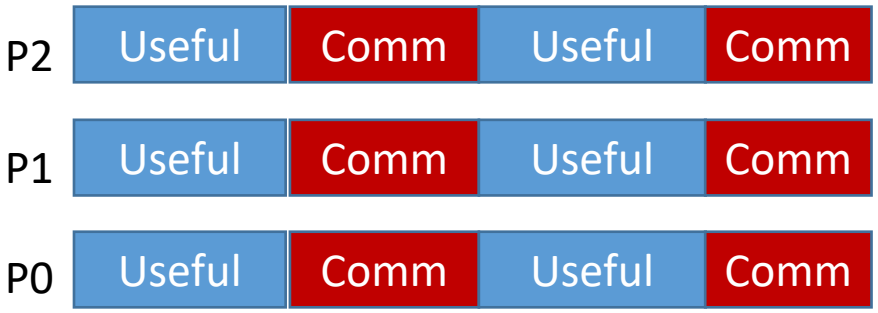
Process Load Balance Efficiency

- **Process Load Balance Efficiency** reflects how well the distribution of useful work over processes is done, where ***useful = serial_comp + omp***

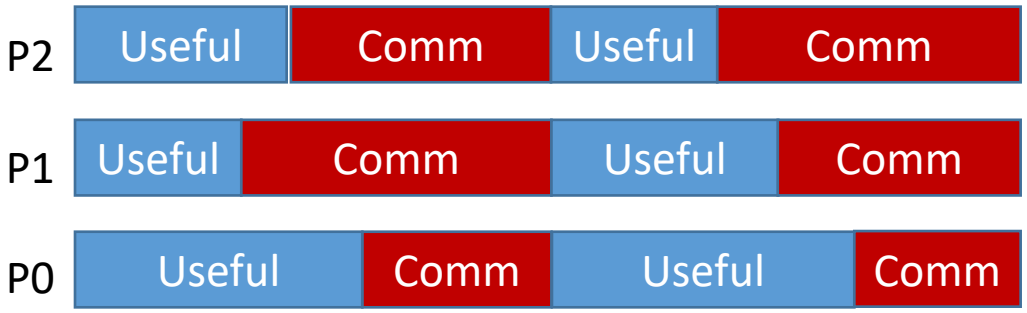
$$\text{Process Load Balance Efficiency} = \frac{\text{runtime} - \max(\text{useful}) + \text{avg}(\text{useful})}{\text{runtime}}$$

- Parallel Efficiency
  - Thread Efficiency
  - Process Efficiency
    - Process Load Balance Efficiency
    - Process Comm. Efficiency
      - Process Serialization Efficiency
      - Process Transfer Efficiency

Example 1: good load balance (LB = 100%)



Example 2: bad load balance (LB = 77%)



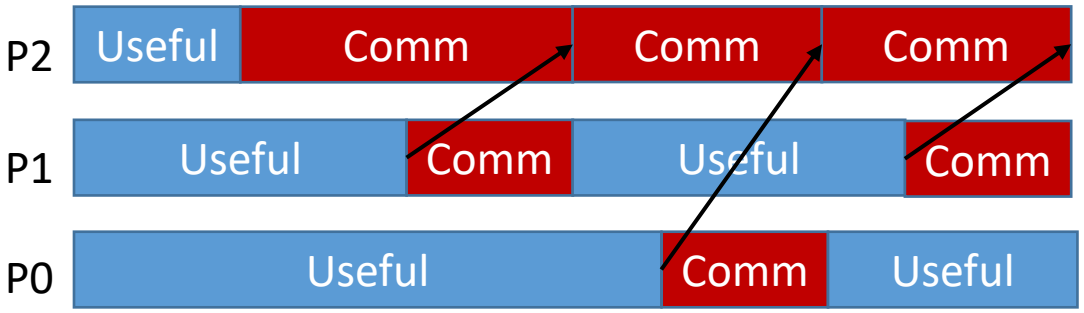
Process Communication Efficiency

- The **Process Communication Efficiency** reflects the loss of efficiency by communication.
- The **Communication Efficiency** can be computed as

$$\textbf{Communication Eff} = \frac{\max(\text{useful})}{\text{runtime}}$$

- Parallel Efficiency
  - Thread Efficiency
  - Process Efficiency
    - Process Load Balance Efficiency
    - Process Comm. Efficiency
      - Process Serialization Efficiency
      - Process Transfer Efficiency

Example:



Useful	Communication	Efficiency
1 sec.	5 sec.	$^1/_6$
4 sec.	2 sec.	$^4/_6$
5 sec.	1 sec.	$^5/_6$

CommE = $^5/_6$ = 83%

Thread Efficiency

- We still need to account for thread inefficiencies, i.e.
  1. Serial computation outside OpenMP (due to Amdahl's Law)
  2. Time not doing useful computation inside OpenMP regions
- For Thread Efficiency we can define

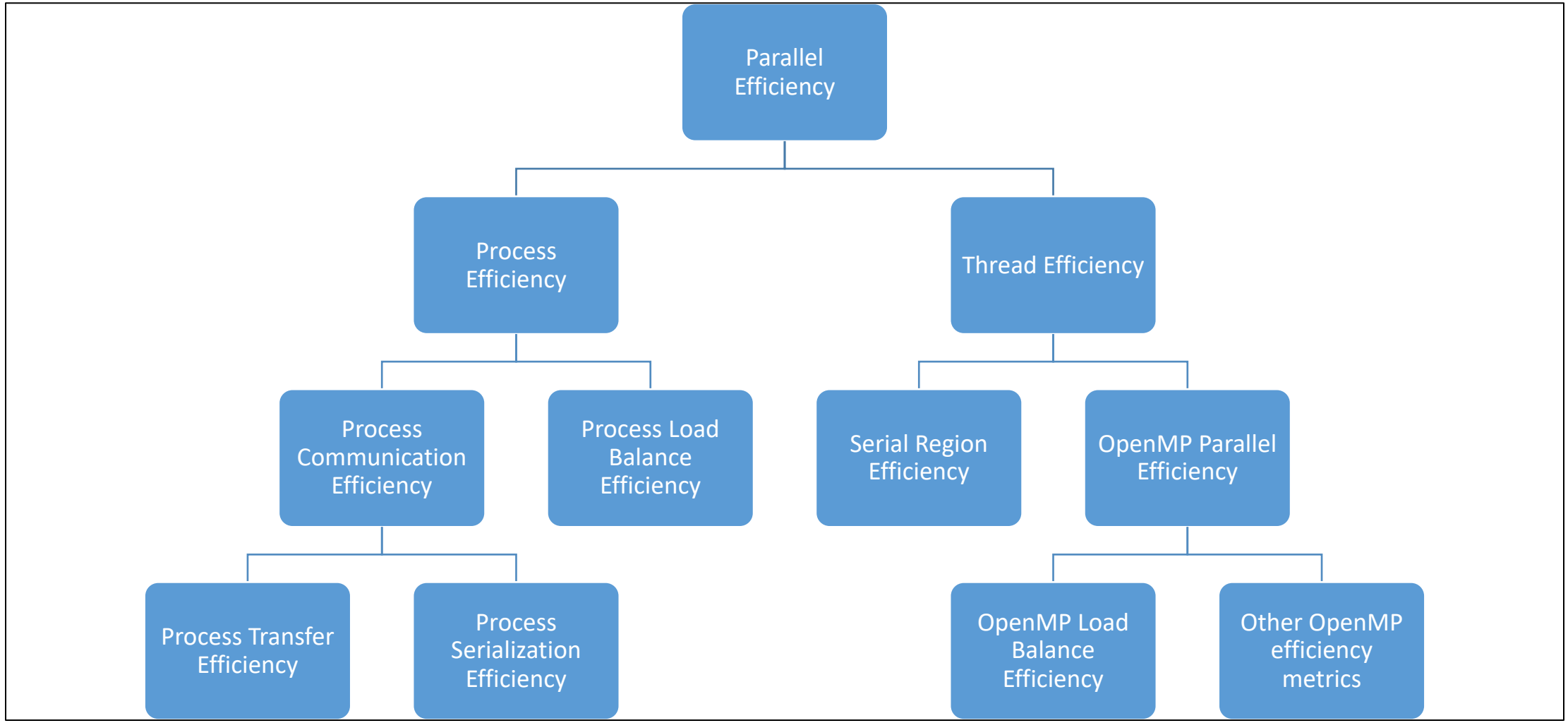
Ideal runtime = runtime - avg(*serial_comp*) - avg(*omp*) + avg(comp)

- As per OpenMP we use the methodology to split Thread Efficiency into
  - **Average** cost of serial computation outside OpenMP (i.e. Amdahl's Law)
  - **Average** cost of inefficiencies within OpenMP parallel regions
- We can then split OpenMP Parallel Efficiency
  - Contribution per bottleneck (e.g. load balance within OpenMP)
  - Contribution per parallel region

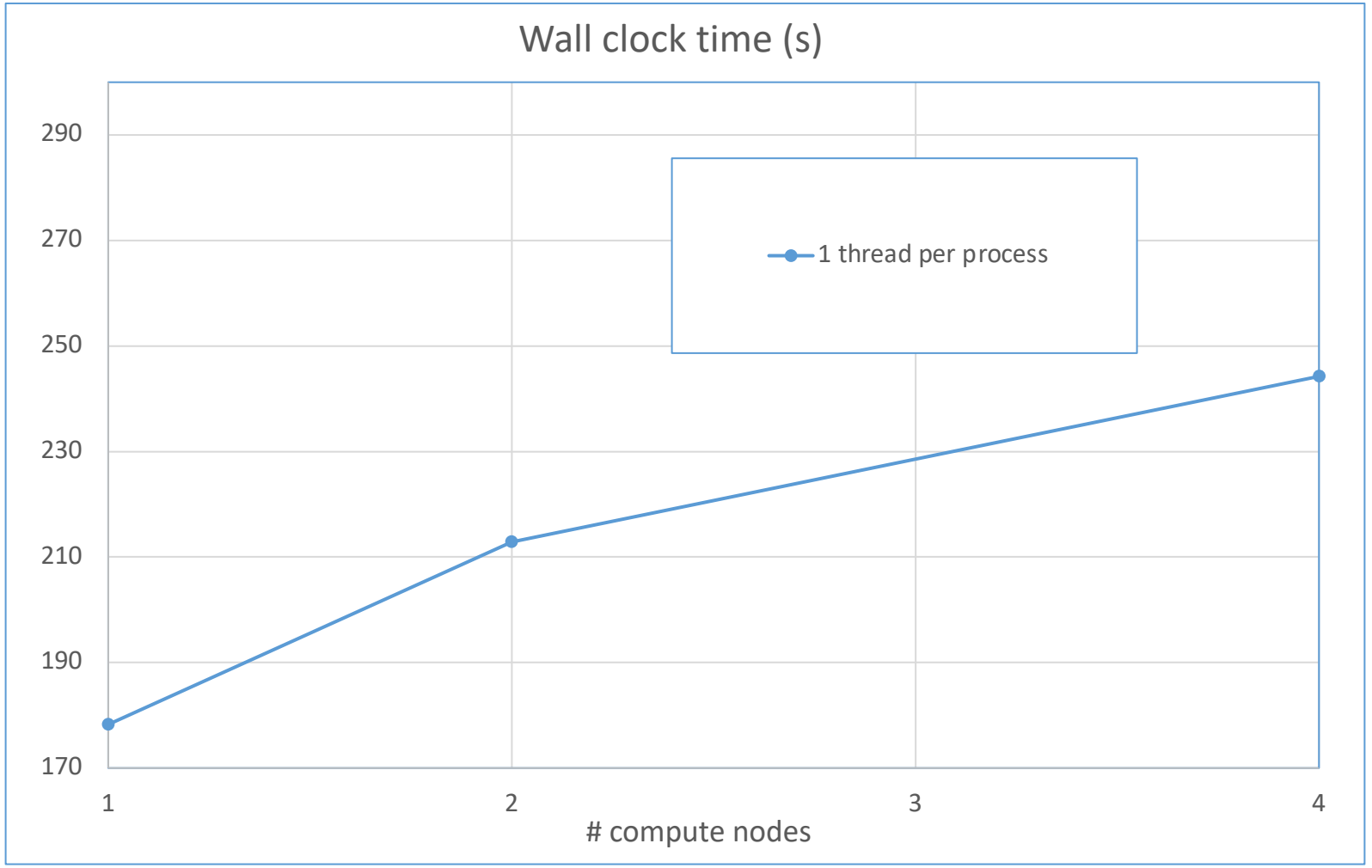
Thread Efficiency child metrics

- **Serial Region Efficiency**
  - Measures cost of serial computation on master threads outside OpenMP
    - i.e. cost of Amdahl's Law
- **OpenMP Parallel Efficiency**
  - Measures how well the OpenMP regions are parallelised
- And we can split OpenMP Parallel Efficiency various ways
  1. We can calculate a contribution per OpenMP parallel region
  2. Or we can calculate a contribution per bottleneck class e.g.
     - Imbalance of computation within OpenMP
     - Average time outside useful computation in MPI, synchronisation, etc.
- This is subject to availability of suitable trace data & tools!
  - e.g. VTune or the NAG-PyPOP Python package
  - **PyPOP is a good tool for automatically calculating these metrics from Extrae data**

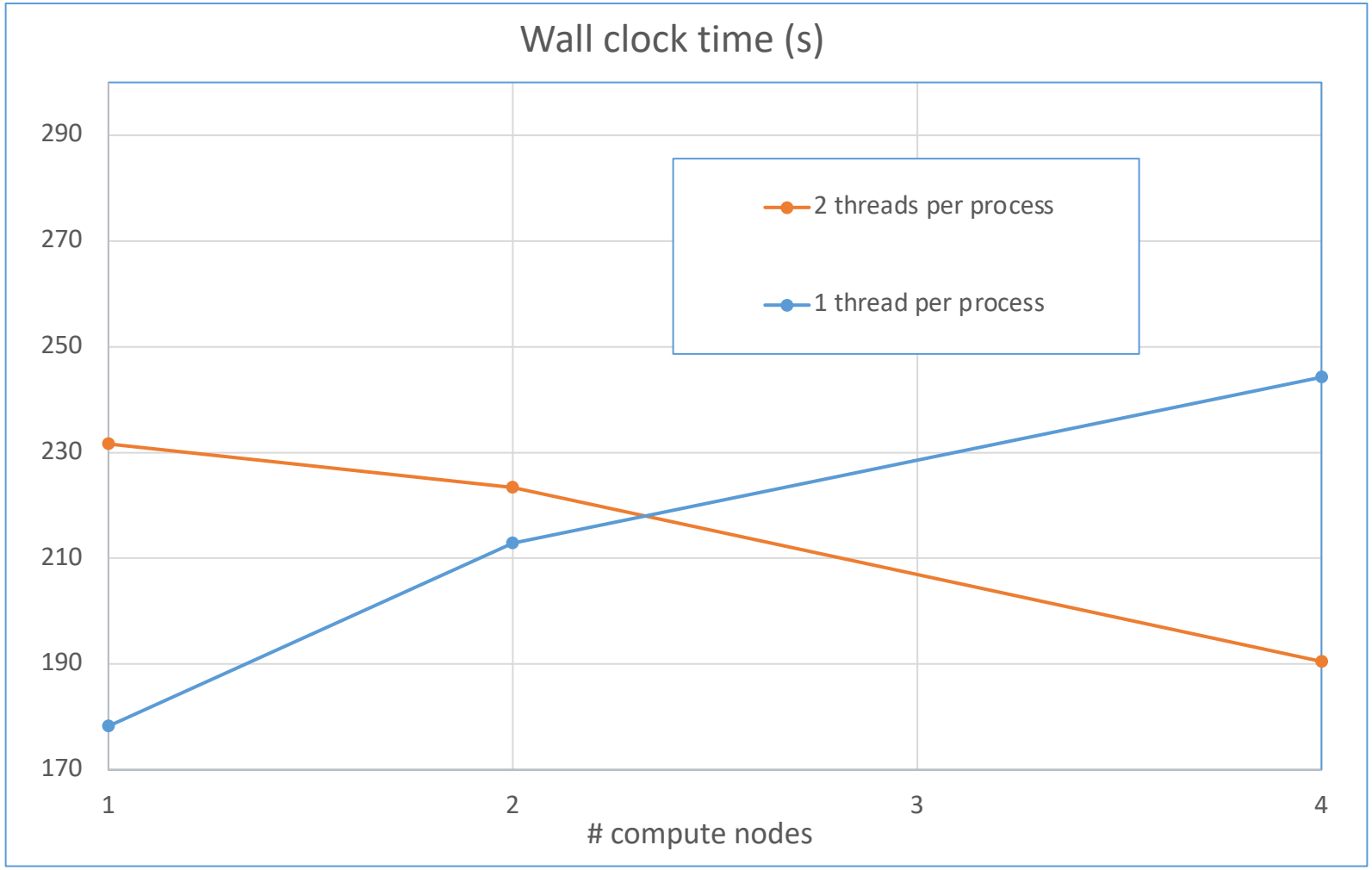
Additive metrics hierarchy



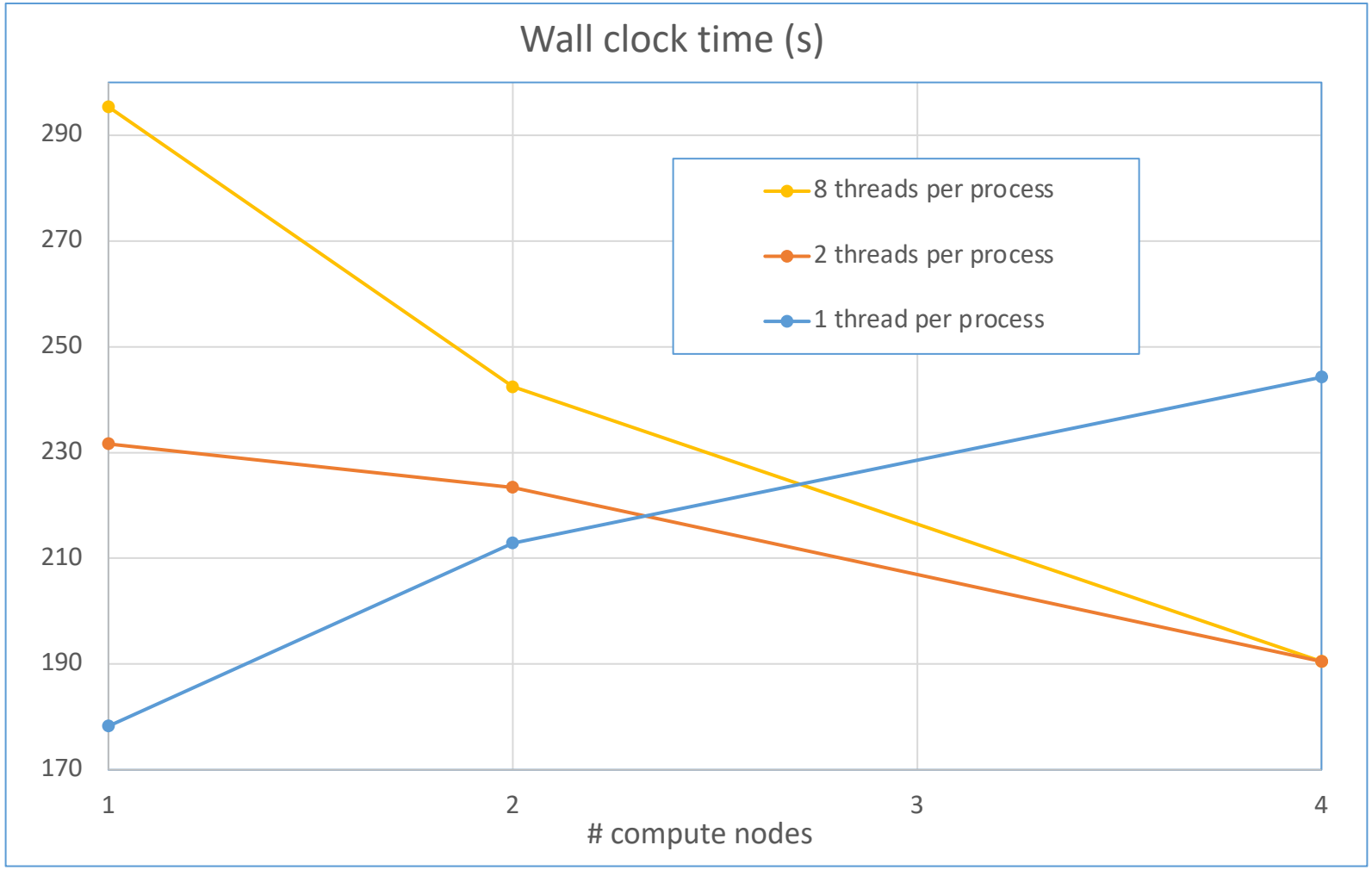
Example (strong scaling)



Mixed results with 2 threads/proc!!



What is going on?



POP metrics to the rescue!

Number of compute nodes	1			2			4		
Number of Processes	48	24	6	96	48	12	192	96	24
Threads per Process	1	2	8	1	2	8	1	2	8
Total Threads	48	48	48	96	96	96	192	192	192
Speedup	1.00	0.77	0.60	0.84	0.80	0.74	0.73	0.94	0.94
Global Efficiency	0.50	0.39	0.30	0.21	0.20	0.19	0.09	0.12	0.12
↳ Parallel Efficiency	0.50	0.32	0.22	0.24	0.18	0.13	0.13	0.12	0.09
↳ Process Efficiency	0.51	0.42	0.57	0.24	0.24	0.33	0.13	0.17	0.22
↳ Process Load balance	0.97	0.98	0.99	0.98	0.99	0.99	0.99	0.99	0.99
↳ Process Communication Eff.	0.54	0.44	0.58	0.26	0.26	0.35	0.13	0.18	0.24
↳ Process Transfer Efficiency	0.55	0.45	0.59	0.29	0.27	0.36	0.16	0.20	0.25
↳ Process Serialisation Eff.	0.98	0.99	0.99	0.97	0.98	0.99	0.97	0.98	0.99
↳ Thread Efficiency	1.00	0.90	0.65	1.00	0.93	0.80	1.00	0.95	0.86
↳ OpenMP Parallel Efficiency	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
↳ Serial Region Efficiency	1.00	0.90	0.65	1.00	0.94	0.80	1.00	0.95	0.86
↳ Computational Scaling	1.00	1.22	1.37	0.88	1.12	1.39	0.73	0.98	1.36
↳ Instruction Scaling	1.00	1.04	1.05	0.91	1.00	1.04	0.75	0.91	1.02
↳ IPC Scaling	1.00	1.16	1.29	0.98	1.11	1.32	0.99	1.07	1.33
↳ Frequency Scaling	1.00	1.02	1.01	0.99	1.02	1.01	0.98	1.00	1.01

What needs investigating further?

Number of compute nodes	1			2			4		
Number of Processes	48	24	6	96	48	12	192	96	24
Threads per Process	1	2	8	1	2	8	1	2	8
Total Threads	48	48	48	96	96	96	192	192	192
Speedup	1.00	0.77	0.60	0.84	0.80	0.74	0.73	0.94	0.94
Global Efficiency	0.50	0.39	0.30	0.21	0.20	0.19	0.09	0.12	0.12
↳ Parallel Efficiency	0.50	0.32	0.22	0.24	0.18	0.13	0.13	0.12	0.09
↳ Process Efficiency	0.51	0.42	0.57	0.24	0.24	0.33	0.13	0.17	0.22
↳ Process Load balance	0.97	0.98	0.99	0.98	0.99	0.99	0.99	0.99	0.99
↳ Process Communication Eff.	0.54	0.44	0.58	0.26	0.26	0.35	0.13	0.18	0.24
↳ Process Transfer Efficiency	0.55	0.45	0.59	0.29	0.27	0.36	0.16	0.20	0.25
↳ Process Serialisation Eff.	0.98	0.99	0.99	0.97	0.98	0.99	0.97	0.98	0.99
↳ Thread Efficiency	1.00	0.90	0.65	1.00	0.93	0.80	1.00	0.95	0.86
↳ OpenMP Parallel Efficiency	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
↳ Serial Region Efficiency	1.00	0.90	0.65	1.00	0.94	0.80	1.00	0.95	0.86
↳ Computational Scaling	1.00	1.22	1.37	0.88	1.12	1.39	0.73	0.98	1.36
↳ Instruction Scaling	1.00	1.04	1.05	0.91	1.00	1.04	0.75	0.91	1.02
↳ IPC Scaling	1.00	1.16	1.29	0.98	1.11	1.32	0.99	1.07	1.33
↳ Frequency Scaling	1.00	1.02	1.01	0.99	1.02	1.01	0.98	1.00	1.01

Single thread issues

Number of compute nodes	1			2			4		
Number of Processes	48	24	6	96	48	12	192	96	24
Threads per Process	1	2	8	1	2	8	1	2	8
Total Threads	48	48	48	96	96	96	192	192	192
Speedup	1.00	0.77	0.60	0.84	0.80	0.74	0.73	0.94	0.94
Global Efficiency	0.50	0.39	0.30	0.21	0.20	0.19	0.09	0.12	0.12
↳ Parallel Efficiency	0.50	0.32	0.22	0.24	0.18	0.13	0.13	0.12	0.09
↳ Process Efficiency	0.51	0.42	0.57	0.24	0.24	0.33	0.13	0.17	0.22
↳ Process Load balance	0.97	0.98	0.99	0.98	0.99	0.99	0.99	0.99	0.99
↳ Process Communication Eff.	0.54	0.44	0.58	0.26	0.26	0.35	0.13	0.18	0.24
↳ Process Transfer Efficiency	0.55	0.45	0.59	0.29	0.27	0.36	0.16	0.20	0.25
↳ Process Serialisation Eff.	0.98	0.99	0.99	0.97	0.98	0.99	0.97	0.98	0.99
↳ Thread Efficiency	1.00	0.90	0.65	1.00	0.93	0.80	1.00	0.95	0.86
↳ OpenMP Parallel Efficiency	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
↳ Serial Region Efficiency	1.00	0.90	0.65	1.00	0.94	0.80	1.00	0.95	0.86
↳ Computational Scaling	1.00	1.22	1.37	0.88	1.12	1.39	0.73	0.98	1.36
↳ Instruction Scaling	1.00	1.04	1.05	0.91	1.00	1.04	0.75	0.91	1.02
↳ IPC Scaling	1.00	1.16	1.29	0.98	1.11	1.32	0.99	1.07	1.33
↳ Frequency Scaling	1.00	1.02	1.01	0.99	1.02	1.01	0.98	1.00	1.01

Single node hybrid performance

Number of compute nodes	1			2			4		
Number of Processes	48	24	6	96	48	12	192	96	24
Threads per Process	1	2	8	1	2	8	1	2	8
Total Threads	48	48	48	96	96	96	192	192	192
Speedup	1.00	0.77	0.60	0.84	0.80	0.74	0.73	0.94	0.94
Global Efficiency	0.50	0.39	0.30	0.21	0.20	0.19	0.09	0.12	0.12
↳ Parallel Efficiency	0.50	0.32	0.22	0.24	0.18	0.13	0.13	0.12	0.09
↳ Process Efficiency	0.51	0.42	0.57	0.24	0.24	0.33	0.13	0.17	0.22
↳ Process Load balance	0.97	0.98	0.99	0.98	0.99	0.99	0.99	0.99	0.99
↳ Process Communication Eff.	0.54	0.44	0.58	0.26	0.26	0.35	0.13	0.18	0.24
↳ Process Transfer Efficiency	0.55	0.45	0.59	0.29	0.27	0.36	0.16	0.20	0.25
↳ Process Serialisation Eff.	0.98	0.99	0.99	0.97	0.98	0.99	0.97	0.98	0.99
↳ Thread Efficiency	1.00	0.90	0.65	1.00	0.93	0.80	1.00	0.95	0.86
↳ OpenMP Parallel Efficiency	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
↳ Serial Region Efficiency	1.00	0.90	0.65	1.00	0.94	0.80	1.00	0.95	0.86
↳ Computational Scaling	1.00	1.22	1.37	0.88	1.12	1.39	0.73	0.98	1.36
↳ Instruction Scaling	1.00	1.04	1.05	0.91	1.00	1.04	0.75	0.91	1.02
↳ IPC Scaling	1.00	1.16	1.29	0.98	1.11	1.32	0.99	1.07	1.33
↳ Frequency Scaling	1.00	1.02	1.01	0.99	1.02	1.01	0.98	1.00	1.01

Single versus multiple node hybrid

Number of compute nodes	1			2			4		
Number of Processes	48	24	6	96	48	12	192	96	24
Threads per Process	1	2	8	1	2	8	1	2	8
Total Threads	48	48	48	96	96	96	192	192	192
Speedup	1.00	0.77	0.60	0.84	0.80	0.74	0.73	0.94	0.94
Global Efficiency	0.50	0.39	0.30	0.21	0.20	0.19	0.09	0.12	0.12
↳ Parallel Efficiency	0.50	0.32	0.22	0.24	0.18	0.13	0.13	0.12	0.09
↳ Process Efficiency	0.51	0.42	0.57	0.24	0.24	0.33	0.13	0.17	0.22
↳ Process Load balance	0.97	0.98	0.99	0.98	0.99	0.99	0.99	0.99	0.99
↳ Process Communication Eff.	0.54	0.44	0.58	0.26	0.26	0.35	0.13	0.18	0.24
↳ Process Transfer Efficiency	0.55	0.45	0.59	0.29	0.27	0.36	0.16	0.20	0.25
↳ Process Serialisation Eff.	0.98	0.99	0.99	0.97	0.98	0.99	0.97	0.98	0.99
↳ Thread Efficiency	1.00	0.90	0.65	1.00	0.93	0.80	1.00	0.95	0.86
↳ OpenMP Parallel Efficiency	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
↳ Serial Region Efficiency	1.00	0.90	0.65	1.00	0.94	0.80	1.00	0.95	0.86
↳ Computational Scaling	1.00	1.22	1.37	0.88	1.12	1.39	0.73	0.98	1.36
↳ Instruction Scaling	1.00	1.04	1.05	0.91	1.00	1.04	0.75	0.91	1.02
↳ IPC Scaling	1.00	1.16	1.29	0.98	1.11	1.32	0.99	1.07	1.33
↳ Frequency Scaling	1.00	1.02	1.01	0.99	1.02	1.01	0.98	1.00	1.01

How to calculate

- The metrics are defined in documents on the POP website
  - https://pop-coe.eu/further-information/learning-material
- The data needed to calculate these can be extracted by hand from Extrae traces using Paraver
  - This is very tedious
  - It requires advanced expertise in Paraver
  - It is easy to make mistakes
- Using NAG-PyPOP is highly recommended
- Note: MPI inside OpenMP is not fully supported

Module 5

BSC tracing tools & PyPOP

Choosing the correct tracing tool

- Must capture data as efficiently and accurately as possible
- Look for something that can
  - Gather the relevant data (architecture, parallelism)
  - Gather *just* the relevant data (configurable filters)
  - Provide useful information (postprocessing tools, trace explorer)
- Common to use multiple tools
  - Different views of the problem
  - Can check unexpected results
  - In depth info on specific issues (GPU, vectorisation, I/O etc.)

Collecting trace data - instrumentation

- Measurement code is inserted so every event of interest is captured e.g.
  - Hardware counters e.g. number of cycles or instructions
  - Function names
  - Function arguments e.g. size of MPI calls
- Instrumentation can occur before/during compilation or at run time
- There are various ways to do this e.g.
  - Manual instrumentation of source code by the user
  - Automatic compiler instrumentation
  - Intercepting calls to parallel libraries e.g. using LD_PRELOAD

Tracing Pros & Cons

- Tracing advantages
  - Event traces preserve relationships between individual events
  - Allows reconstruction of **dynamic** application behaviour on any required level of abstraction
  - Most general measurement technique
  - Profile data can be reconstructed from event traces
- Disadvantages
  - Traces can very quickly become extremely large
  - Tracing can increase run time significantly
    - If it does, is the data representative of the untraced execution?
  - Writing events to file at runtime may causes perturbation
- Sampling is an alternative to tracing
  - Much less data & performance information

Knowing how to trust trace data

- Read the tools documentation and check what is and isn't supported e.g.
  - Is nested OpenMP threading supported?
  - Is parallelism captured when linking against precompiled libraries?
- Understand the mechanism of instrumentation
  - What events are and aren't captured?
- Check output logs after running
  - Errors? Warnings?
- View the trace data
  - Is it complete?
    - Missing processes or threads?
    - Missing events?
- Sanity check as much as possible
  - e.g. check for agreement between run times with and without tracing
    - Tracing may add a significant time overhead

POP tracing tools

We use the following tools (developed by some of the POP partners)

- Extrae (tracing) + Paraver (visualisation) + Dimemas (simulation)
  - PyPOP for automated generation of POP metrics from Extrae traces
- Score-P (tracing) + Scalasca (post processing) + Cube (visualisation)

To understand how to generate trace files & calculate POP metrics

- See POP website learning material & online training
  - https://pop-coe.eu/further-information/learning-material

Other tracing tools can be used e.g. Intel's VTune

Score-P + Scalasca + Cube

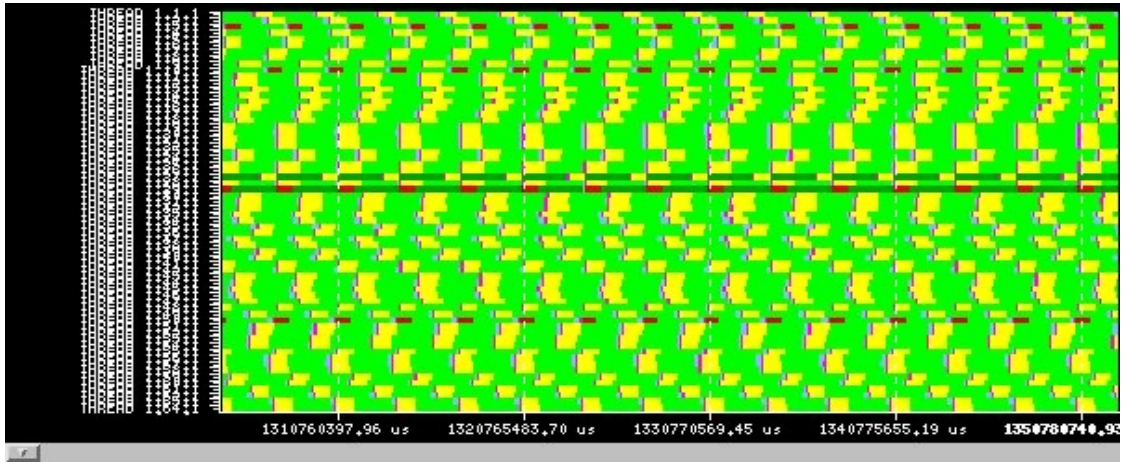
- Today we've only time to introduce
  - Extrae, Paraver, Dimemas & PyPOP

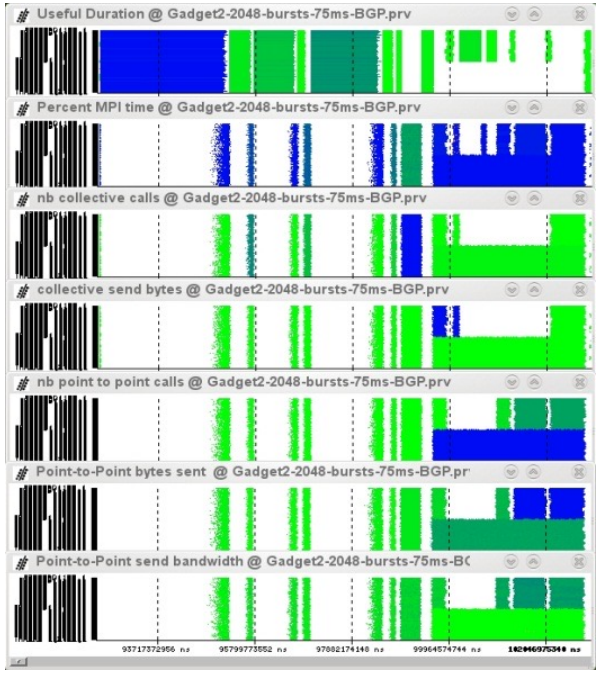
- For more information on Score-P, Scalasca & Cube
  - **21st POP Webinar recording - The Scalasca Scalable Parallel Performance Analysis Toolset - For POP Assessments and Beyond**

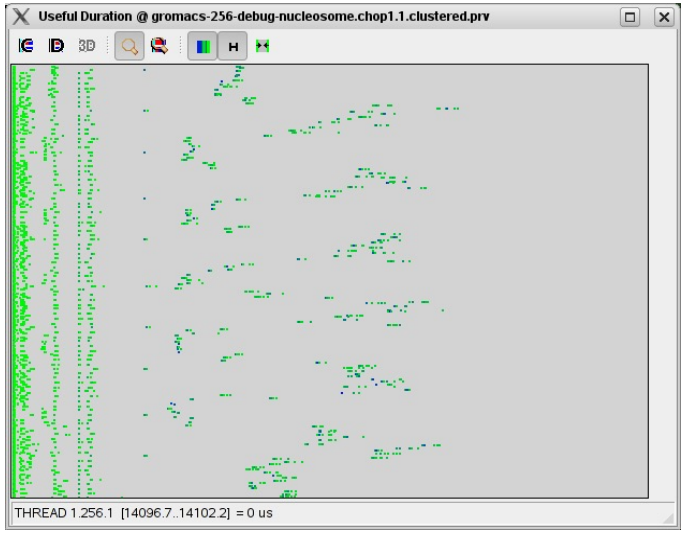
- Also see the learning material on the POP CoE website

The BSC Tools

- Since 1991
- Based on traces
- Open Source
  - http://tools.bsc.es
- Core tools:
  - Extrae – instrumentation
  - Paraver (& paramedir) – trace analysis
  - Dimemas – message passing simulator
- Focus
  - Detail, variability, flexibility
  - Behavioral structure vs. syntactic structure
  - Intelligence: Performance Analytics







Extrae

How does Extrae work?

- Symbol substitution through **LD_PRELOAD**
  - Specific libraries for each combination of runtimes
    - MPI
    - OpenMP
    - OpenMP+MPI
    - …
- Dynamic instrumentation
  - Based on Dyninst (developed by U.Wisconsin/U.Maryland)
    - Instrumentation in memory
    - Binary rewriting
- Static link (i.e., PMPI, Extrae API)

Extrae Features

- Parallel programming models
  - MPI, OpenMP, pthreads, OmpSs, CUDA, OpenCL, Java, Python…
- Platforms
  - Intel, Cray, BlueGene, MIC, ARM, Android, Fujitsu, Sparc…
- Performance Counters
  - Using PAPI interface
- Link to source code
  - Callstack at MPI routines
  - OpenMP outlined routines
  - Selected user functions (Dyninst)
- Periodic sampling
- User events (Extrae API)

No need to recompile / relink!

Using Extrae in 3 steps

1. Adapt the job submission script
2. (Optional) Tune the Extrae XML configuration file
   - Examples distributed with Extrae at $EXTRAE_HOME/share/example
3. Run it!

- For further reference check the **Extrae User Guide:**
  - Also distributed with Extrae at $EXTRAE_HOME/share/doc
  - https://tools.bsc.es/tools_manuals

Step 1: Adapt the job script to load Extrae (LD_PRELOAD)

train.sh

```
(...)

### Application path
PISVM=../../bin/pisvm-train

### Input path
TRAINDATA=../../input/sdap_area_all_training.el

### Extrae path
export EXTRAE_HOME=/homec/deep/deep83/tools/extrae/
export EXTRAE_WORK_DIR=/work/$USER/pisvm/romeraw

### Run the application
srun ./trace.sh $PISVM -D -o 1024 -q 512 -c 10000
                -g 16 -t 2 -m 1024 -s 0 $TRAINDATA

### Generate the trace
export TRACE_NAME=pisvm-romeraw-train.prv
${EXTRAE_HOME}/bin/mpi2prv
   -f ${EXTRAE_WORK_DIR}/TRACE.mpits
   -o ${TRACE_NAME}
```

trace.sh

```
#!/bin/bash

### Load Extrae
source ${EXTRAE_HOME}/etc/extrae.sh
export EXTRAE_CONFIG_FILE=../../../../config/extrae.xml

### Load the tracing library (choose C/Fortran)
export LD_PRELOAD=$EXTRAE_HOME/lib/libmpitrace.so
#export LD_PRELOAD=$EXTRAE_HOME/lib/libmpitracef.so

# Run the program
$*
```

Select tracing library

Step 1: Which tracing library?

- Choose depending on the application type

Library	Serial	MPI	OpenMP	pthread	CUDA
libseqtrace	✓				
libmpitrace[f][1]		✓			
libomptrace			✓		
libpttrace				✓	
libcudatrace					✓
libompitrace[f] [1]		✓	✓		
libptmpitrace[f] [1]		✓		✓	
libcudampitrace[f] [1]		✓			✓

[1] include suffix “f” in Fortran codes

Step 2: Extrae XML configuration

```
<mpi enabled="yes">
  <counters enabled="yes" />
</mpi>

<openmp enabled="yes">
  <locks enabled="no" />
  <counters enabled="yes" />
</openmp>

<pthread enabled="no">
  <locks enabled="no" />
  <counters enabled="yes" />
</pthread>

<callers enabled="yes">
  <mpi enabled="yes">1-3</mpi>
  <sampling enabled="no">1-5</sampling>
</callers>
```

Trace MPI calls + HW counters

Trace call-stack events @ MPI calls

Step 2: Extrae XML configuration (II)

```
<counters enabled="yes">
  <cpu enabled="yes" starting-set-distribution="cyclic">
    <set enabled="yes" domain="all" changeat-time="0">
      PAPI_TOT_INS, PAPI_TOT_CYC, PAPI_L1_DCM, PAPI_L2_DCM
    </set>
    <set enabled="yes" domain="all" changeat-time="500000us">
      ...
    </set>
    <set enabled="yes" domain="all" changeat-time="500000us">
      ...
    </set>
    <set enabled="yes" domain="all" changeat-time="500000us">
      ...
    </set>
  </cpu>
  <network enabled="no" />
  <resource-usage enabled="no" />
  <memory-usage enabled="no" />
</counters>
```

Select which HW counters are measured

Step 2: Extrae XML configuration (III)

```
<buffer enabled="yes">
  <size enabled="yes">5000000</size>
  <circular enabled="no" />
</buffer>

<sampling enabled="no" type="default" period="50m" variability="10m" />

<merge enabled="yes"
       synchronization="default"
       tree-fan-out="16"
       max-memory="512"
       joint-states="yes"
       keep-mpits="yes"
       sort-addresses="yes"
       overwrite="yes"
>
  $TRACE_NAME$
</merge>
```

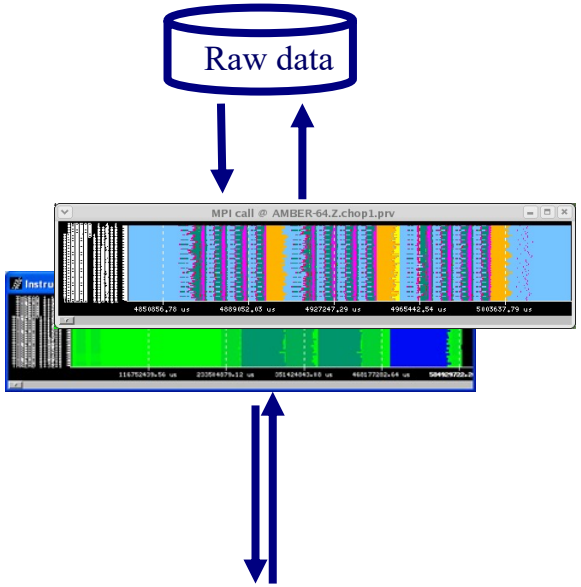
Trace buffer size

Collect more measurements with sampling

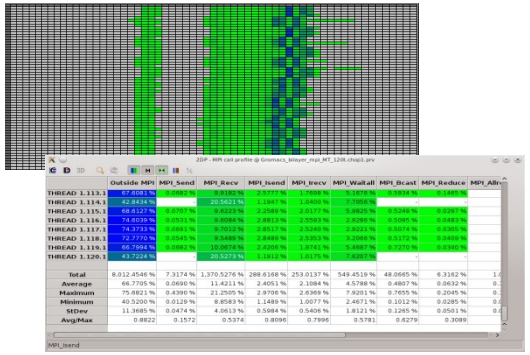
Merge intermediate files into Paraver trace

Paraver

Paraver – Performance data browser



Timelines



2/3D tables (Statistics)

Trace visualization/analysis

+ trace manipulation

Goal = Flexibility

No semantics

Programmable

Comparative analyses

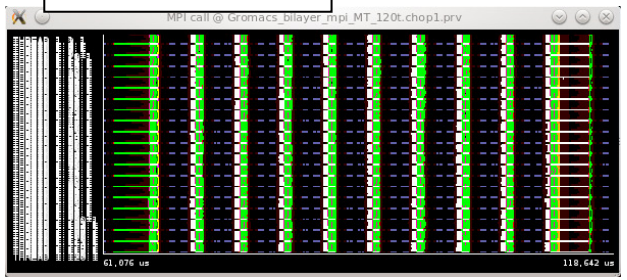
Multiple traces

Synchronize scales

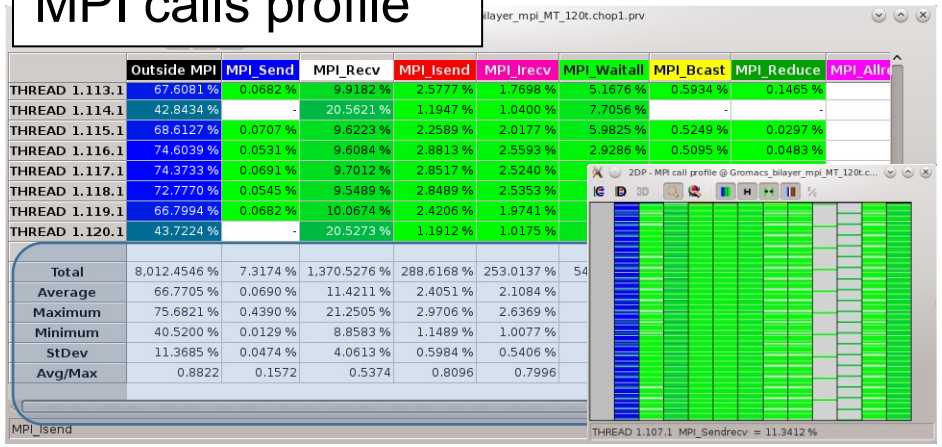
Tables: Profiles, histograms, correlations

- From timelines to tables

MPI calls



MPI calls profile

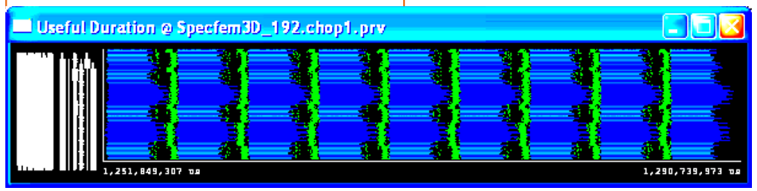


	Outside MPI	MPI_Send	MPI_Recv	MPI_Isend	MPI_Irecv	MPI_Waitall	MPI_Bcast	MPI_Reduce
THREAD 1.113.1	67.6081 %	0.0682 %	9.9182 %	2.5777 %	1.7698 %	5.1676 %	0.5934 %	0.1465 %
THREAD 1.114.1	42.8434 %	-	20.5621 %	1.1947 %	1.0400 %	7.7056 %	-	-
THREAD 1.115.1	68.6127 %	0.0707 %	9.6223 %	2.2589 %	2.0177 %	5.9825 %	0.5249 %	0.0297 %
THREAD 1.116.1	74.6039 %	0.0531 %	9.6084 %	2.8813 %	2.5593 %	2.9286 %	0.5095 %	0.0483 %
THREAD 1.117.1	74.3733 %	0.0691 %	9.7012 %	2.8517 %	2.5240 %			
THREAD 1.118.1	72.7770 %	0.0545 %	9.5489 %	2.8489 %	2.5353 %			
THREAD 1.119.1	66.7994 %	0.0682 %	10.0674 %	2.4206 %	1.9741 %			
THREAD 1.120.1	43.7224 %	-	20.5273 %	1.1912 %	1.0175 %			
Total	8,012.4546 %	7.3174 %	1,370.5276 %	288.6168 %	253.0137 %	54		
Average	66.7705 %	0.0690 %	11.4211 %	2.4051 %	2.1084 %			
Maximum	75.6821 %	0.4390 %	21.2505 %	2.9706 %	2.6369 %			
Minimum	40.5200 %	0.0129 %	8.8583 %	1.1489 %	1.0077 %			
StDev	11.3685 %	0.0474 %	4.0613 %	0.5984 %	0.5406 %			
Avg/Max	0.8822	0.1572	0.5374	0.8096	0.7996			

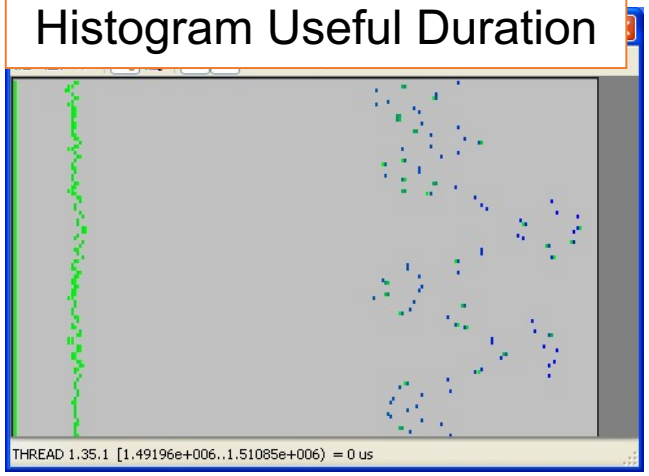
MPI_Isend

THREAD 1.107.1 MPI_Sendrecv = 11.3412 %

Useful Duration



Histogram Useful Duration

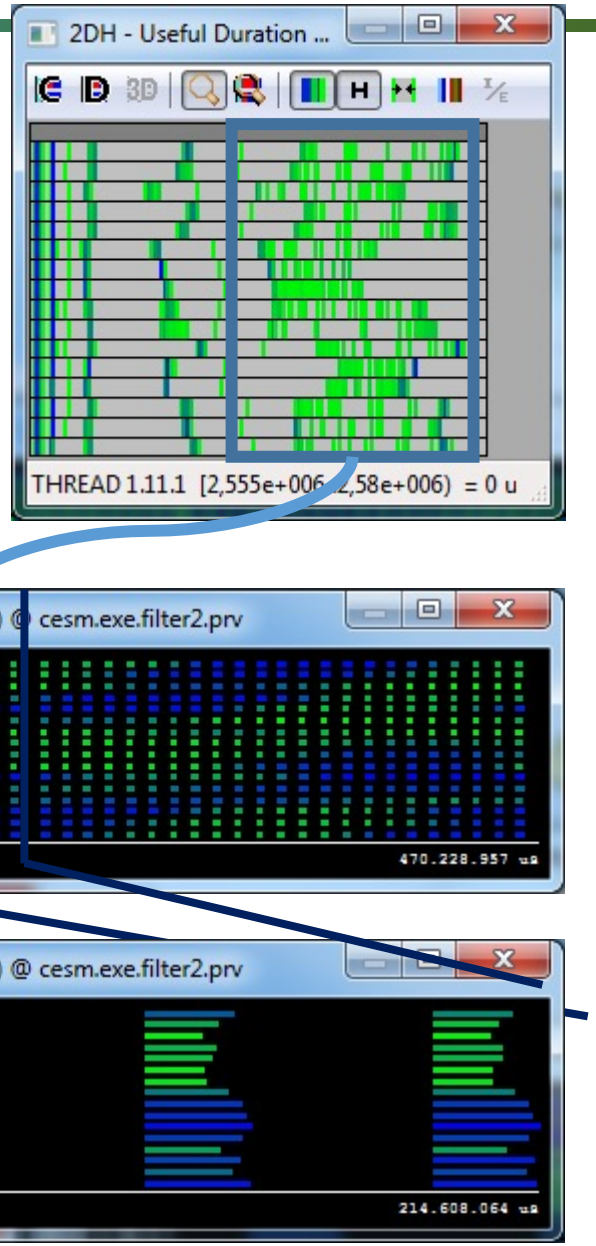


From tables to timelines

CESM: 16 processes, 2 simulated days

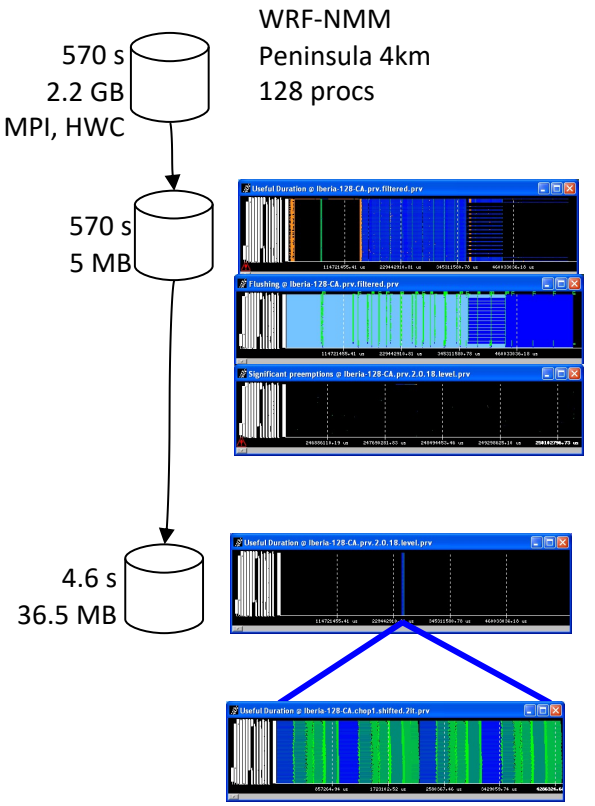
- Histogram useful computation duration shows high variability
- How is it distributed?

- Dynamic imbalance
  - In space and time
  - Day and night
  - Season ? ☺



Trace manipulation

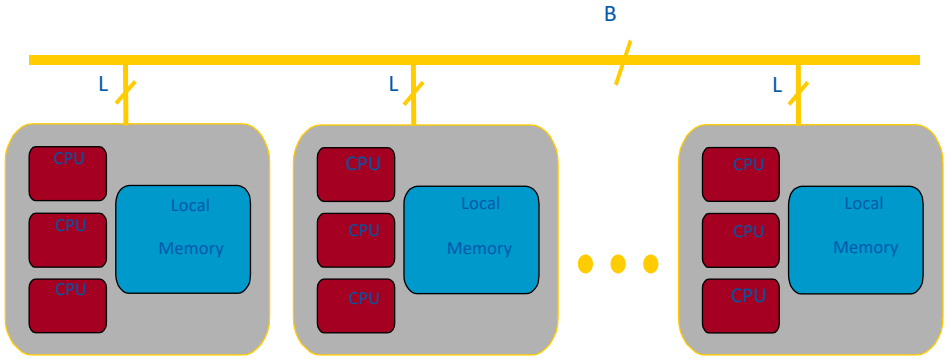
- Data handling/summarization capability
  - Filtering
    - Subset of records in original trace
    - By duration, type, value,…
    - Filtered trace is still a Paraver trace and can be analysed with the same cfgs (as long as the data required has been kept)
  - Cutting
    - All records in a given time interval
    - Only some processes
  - Software counters
    - Summarized values computed from those in the original trace emitted as new even types
    - #MPI calls, total hardware count,…

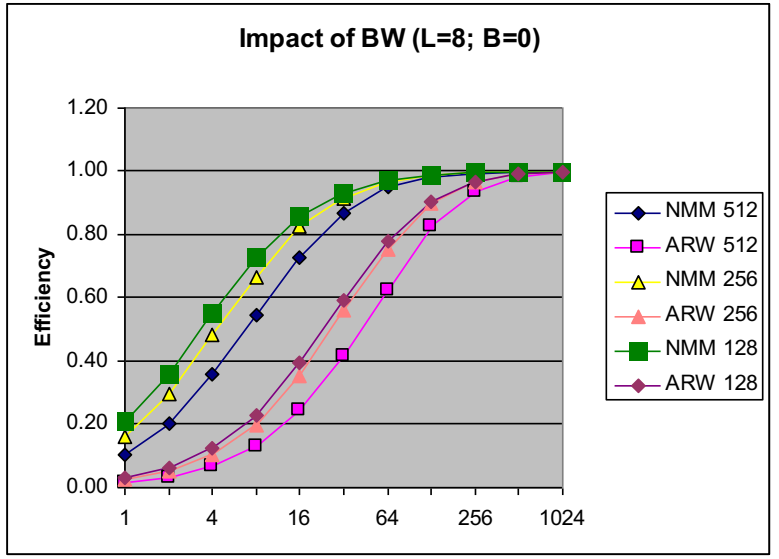


Dimemas

Dimemas: Coarse grain, Trace driven simulation

- Simulation: Highly non-linear model
  - MPI protocols, resource contention…
- Parametric sweeps
  - On abstract architectures
  - On application computational regions
- What-if analysis
  - Ideal machine (instantaneous network)
  - Estimating impact of ports to MPI+OpenMP/CUDA/…
  - Should I use asynchronous communications?
  - Are all parts equally sensitive to network?
- MPI sanity check
  - Modeling nominal
- Paraver – Dimemas tandem
  - Analysis and prediction
  - What-if from selected time window





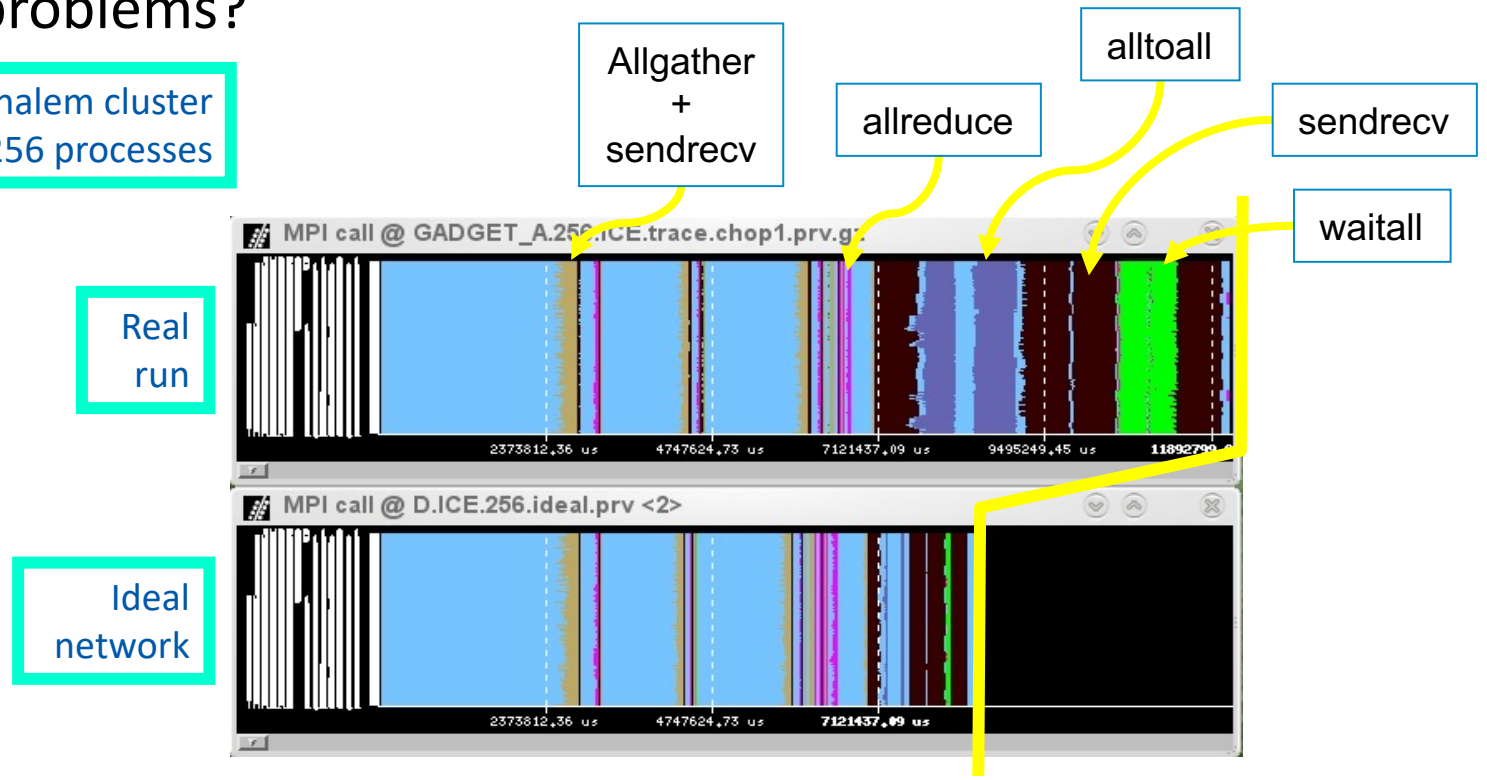
Detailed feedback on simulation (trace)

Ideal machine

The impossible machine: BW = ∞, L = 0

- Actually describes/characterizes intrinsic application behavior
  - Load balance problems?
  - Dependence problems?

GADGET @ Nehalem cluster
256 processes



PyPOP

What is PyPOP?

What PyPOP is:

- Simple tool to automate common tasks in performance profiling
- Rapidly analyse traces and compute POP Metrics
- Easily generate high quality plots and reports
- User friendly and tool agnostic
  - Extrae and manual input currently supported, other formats in development
- Backend framework to build custom analyses using Python

But. . .

- Only one part of the performance analysis workflow
- Not a substitute for manually inspecting traces
- Still under development

What is PyPOP?

PyPOP design choices and philosophy

- Python ≥3.6 with Numpy, Pandas, Bokeh
  - Widely used in science/industry — minimise barrier to entry
  - Plugin-based architecture for extensibility
- Jupyter notebook based interface
  - “Literate programming” encourages self-documentation and reproducibility
  - Easily mix text/code/plotting/GUI elements
  - Generate reports directly from an analysis notebook
- “Wizard”-like GUI for non Python-programmers
  - Generate POP metrics and reports without writing Python code

Typical PyPOP Workflow

POP Metrics from Extrae traces

1. Collect traces with Extrae — required data:
   - MPI Events
   - OpenMP Events
   - Hardware counters: Total Instructions, Total Cycles
2. Optionally pre-process traces
   - Pre-process large (GBs) traces into small (KBs) summary files
   - Speeds up notebook loading and minimizes data downloads

Typical PyPOP Workflow

POP Metrics from Extrae traces

3. Open Jupyter Notebook with “Wizard” GUI
   - Select and process traces or summary files using the GUI
   - Quick initial analysis to generate POP metrics
4. Generate report notebook from GUI
   - Report template notebook containing metric table and scaling plot
   - Add description/discussion text to notebook
   - Customise analysis using Python code
   - Convert notebook to PDF to create shareable report

PyPOP summary

A tool for efficient performance analysis workflows

- Python based with Jupyter notebook interface
- Quickly analyse traces and compute POP metrics
- Plot metric tables and scaling graphs
- Output fully annotated PDF reports

PyPOP Github

- https://github.com/numericalalgorithmsgroup/pypop

Performance Optimisation and Productivity

A Centre of Excellence in HPC

Contact:

- https://www.pop-coe.eu
- pop@bsc.es
- @POP_HPC
- youtube.com/POPHPC

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 676553 and 824080.