Performance Analysis: An Introduction to the Tools and Methodology used in the POP CoE

Jon Gibson and Wadud Miah, NAG

EU H2020 Centre of Excellence (CoE)

1 October 2015 – 31 March 2018

Grant Agreement No 676553

Timetable

- Introduction to Performance Analysis and the POP CoE Methodology
- Performance Analysis Tools for Parallel Codes
  - Scalasca
  - The BSC Tools: Extrae and Paraver
- Coffee Break (3.00 - 3.30pm)
- Parallel I/O Profiling and the Darshan Profiling Tool
- Hands-on Session with the Profiling Tools

Introduction to Performance Analysis and the POP CoE Methodology

Contents

- The POP Service
- Code Performance
- Profiling and Optimisation
- Real Examples of Code Improvements
- The POP Metrics

The POP Service

- **P**erformance **O**ptimisation and **P**roductivity
- A Centre of Excellence
  - Collaborative European project funded by Horizon 2020 programme
  - Runs October 2015 – March 2018
- Providing Free Services within Europe
  - Precise understanding of **parallel** application and system behaviour
  - Across application areas, platforms and scales
  - Suggestions/support on how to rewrite code in the most productive way
  - For academic and industrial codes and users

The POP Team

- Participating institutions:
  - Barcelona Supercomputing Center, Spain (coordinator)
  - HLRS, Germany
  - Jülich Supercomputing Center, Germany
  - NAG, UK
  - RWTH Aachen, IT Center, Germany
  - TERATEC, France
- A team with:
  - Expertise in performance analysis and optimisation
  - Expertise in parallel programming models and practices
  - A research and development background and a proven commitment to real academic and industrial use cases

What does POP do?

? Performance Audit ⇨ Report
- Identify performance issues of customer code
- Small effort (< 1 month)

! Performance Plan ⇨ Report
- Follow-up on the audit service
- Identifies the root causes of issues and qualifies/quantifies fixes
- Longer effort (1-3 months)

✓ Proof-of-Concept ⇨ Software Demonstrator
- Experiments and mock-up tests for customer codes
- Kernel extraction, parallelisation, mini-apps, …
- 6 months effort

Who are POP targeting?

- Code Developers
  - Assessment of detailed behaviour of code
  - Suggestion of most productive directions to refactor code
- Users & Infrastructure Operators
  - Assessment of achieved performance in production conditions
  - Possible improvements from modifying environment setup
  - Evidence to interact with code provider
  - Training of support staff
- Vendors
  - Benchmarking, customer support and system design

Why improve performance?

- Time is money – especially on supercomputers
- To run bigger and/or more complex simulations
- To remain competitive with similar codes

Understanding Performance is Hard

- Scientific Codes
  - Often large codes developed by many people
  - Development driven by functionality rather than performance
  - Difficult to get an overview of the code's behaviour
- HPC machines
  - Complex architectures
    - Many nodes, each consisting of a number of multicore processors
    - An interconnect and a filesystem
    - Vector operations
    - Deep memory hierarchies with a number of levels of cache
  - Not easy to program efficiently

Where do we start?

- Are there any easy wins?
  - Are we using the best performing compiler for our code?
  - With the best choice of compiler flags?
  - And the best performing MPI library?
- We need to be very selective before spending time optimising code
  - *"Premature optimization is the root of all evil."* – Donald Knuth
  - Optimising code is often time-consuming
  - Optimised code is often more difficult to read/understand (hence debug/maintain)
  - Optimising a routine that only takes 2% of the execution time is going to have very little impact on the overall performance
- We therefore need a way to understand the behaviour of a code in order to guide the optimisation process

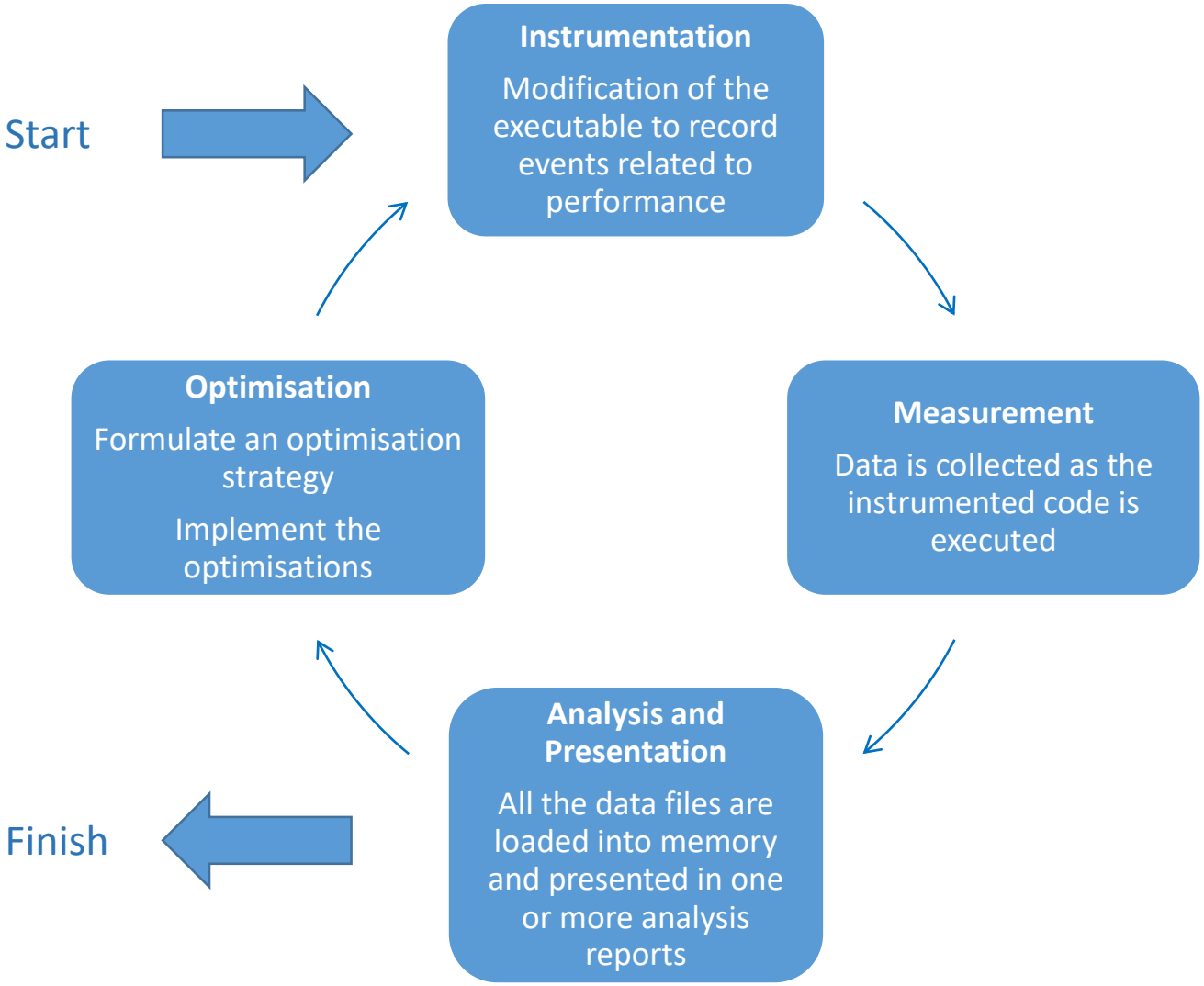
Profiling and Performance Analysis

- *Profiling* refers to the monitoring of a code's behaviour as it executes
- There are a number of profiling tools available, which by helping to answer a number of key questions, allow us to optimise effectively
  - What are the most time-consuming routines?
  - What are the most time-consuming lines in those routines?
  - Is it easy to optimise or is the efficiency already high?
  - What needs to be optimised, i.e. what is the bottleneck?
    - Cache efficiency, vectorisation, etc
  - For a parallel code, is it load-balanced?
    - Essential if the code is to scale
  - How many MPI messages are there and what size are they?

Some Profiling Tools

- Gprof – GNU Profiler
- PAPI – Performance Application Programming Interface
- TAU – Tuning and Analysis Utilities
- Scalasca
- Extrae and Paraver
- Darshan
- Allinea MAP
- HPCToolkit
- OpenSpeedShop
- Vampirtrace and Vampir
- …and many others.

The Profiling-Optimisation Cycle

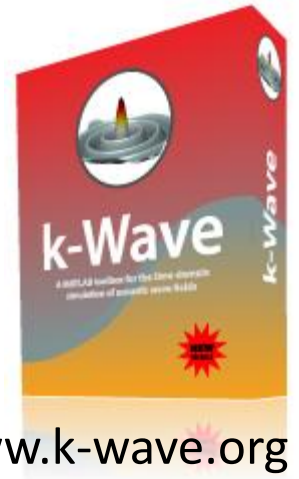


The Details of the Run

- The Input Data
  - Profiling results, and therefore possible bottlenecks, are likely to change with different input files.
  - Ideally, therefore, we want to profile a typical production run rather than a trivial test case.
- The Number of Cores
  - Profiling results are likely to change when the job is run on different numbers of cores.
  - When a code does not scale well, profiling it on different numbers of cores will help identify the cause of the poor scaling.
  - Ideally, profile on the number of cores you aim to scale up to.

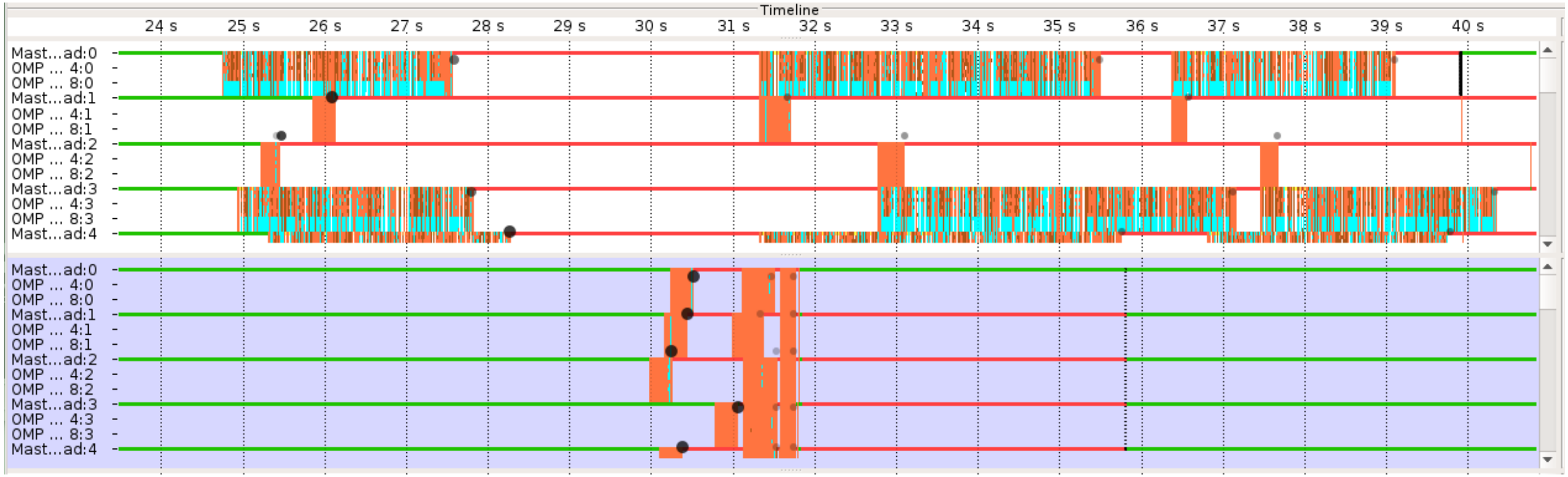
k-Wave – Brno Uni. of Technology

- Toolbox for time domain acoustic and ultrasound simulations in complex and tissue-realistic media
- C++ code parallelised with Hybrid MPI and OpenMP (+ CUDA)
- Profiling showed that
  - 3D domain decomposition suffered from major load imbalance: exterior MPI processes with fewer grid cells took much longer than interior
  - OpenMP-parallelised FFTs were much less efficient for grid sizes of exterior, requiring many more small and poorly-balanced parallel loops
- Using a periodic domain with identical halo zones for each MPI rank reduced overall runtime by a factor of 2



www.k-wave.org

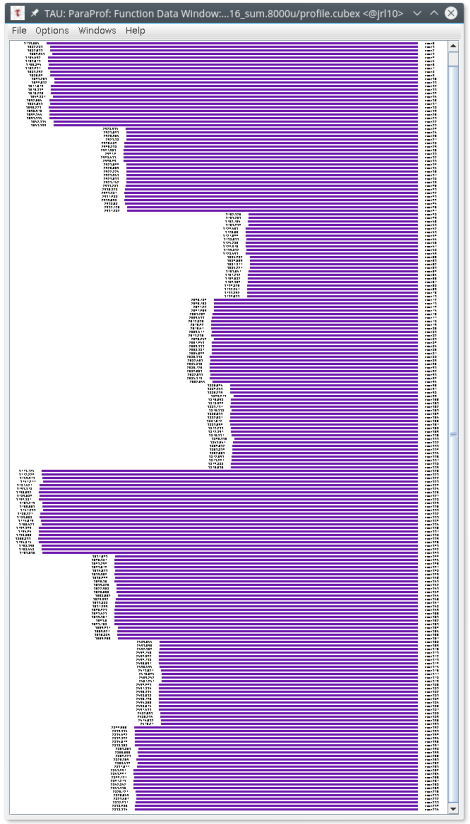
k-Wave – Brno Uni. of Technology



- Comparison time-line before (top) and after (bottom) balancing, showing exterior MPI ranks (0,3) and interior MPI ranks (1,2)
  - MPI synchronization in red; OpenMP synchronization in cyan

EPW – University of Oxford

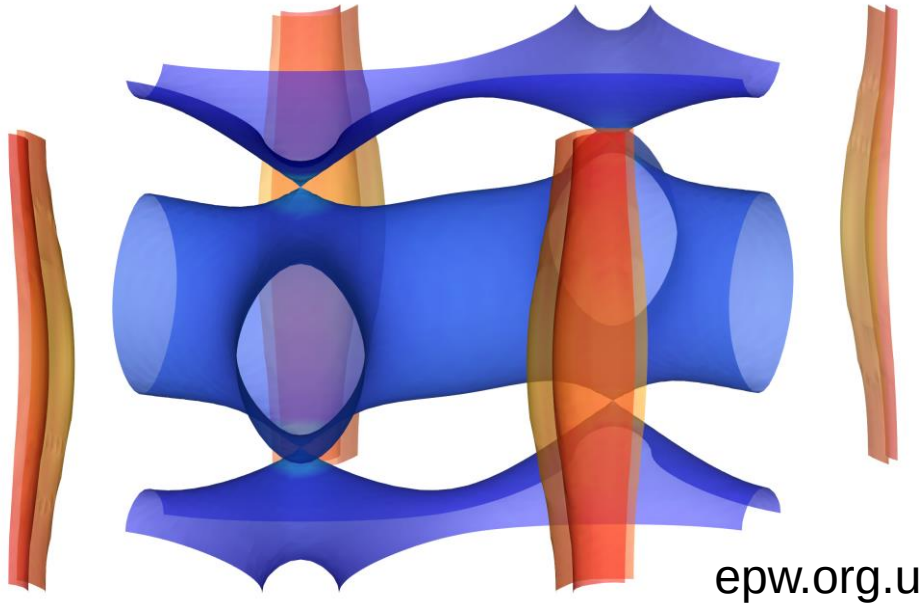
- Electron-Phonon Wannier (EPW) materials science DFT code; part of the Quantum ESPRESSO suite
- Fortran code parallelised with MPI
- Profiling showed
  - Poor load balance
  - Large variations in runtime, likely caused by I/O
  - Final stage spends a great deal of time writing output to disk



EPW – University of Oxford

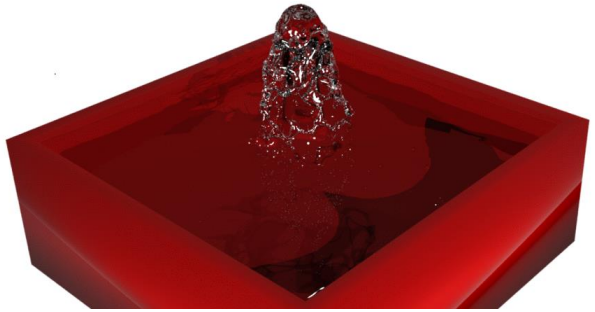
- Original code had all MPI processes writing result to disk at the end
- This was modified this so that only one rank wrote the output
- On 480 MPI processes, time taken to write results fell from over 7 hours to just 56 seconds: a 450-fold speed-up!

- Combined with other improvements, this enabled simulations to scale to a previously impractical 1920 MPI processes



epw.org.uk

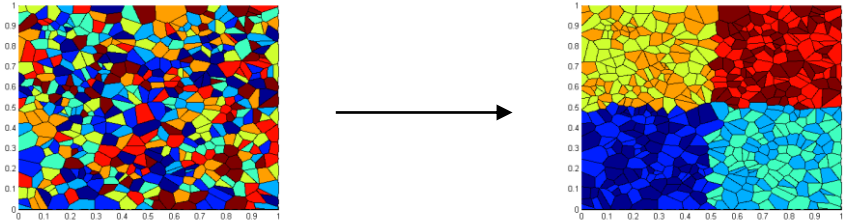
sphFluids – Stuttgart Media University



- Smoothed particle hydrodynamics code
  - C++ with OpenMP
- Profiling identified several issues
  - Definitions of variables in inner loops
  - Unnecessary operations caused by indirection in code design
  - Frequently-used non-inlined functions
  - High cache misses, which could be reduced by reordering the processing of particles
- The developers decided to completely rewrite the code based on their new knowledge, leading to an overall performance improvement of 5x - 6x

GraGLeS2D – RWTH Aachen

- Simulation of microstructure evolution in polycrystalline materials
- After profiling, the following optimisations were implemented
  - Memory allocation library optimised for multi-threading
  - Reordering the work distribution to threads



  - Algorithmic optimisation in the convolution calculation
  - Code restructuring to enable vectorisation
- An improvement of over 10x was demonstrated for the region concerned, with an overall application speed-up of 2.5x

Efficiency Metrics in a POP Performance Audit

Efficiencies

The following metrics are used in a POP performance audit.

- Global Efficiency (GE)
  - Parallel Efficiency (PE)
    - Load Balance Efficiency (LB)
    - Communication Efficiency (CommE)
      - Serialization Efficiency (SerE)
      - Transfer Efficiency (TE)
  - Computation Efficiency (CompE)
    - IPC Scaling
    - Instruction Scaling

Global Efficiency (GE)

- The **Global Efficiency** describes how well the parallelization of your application is working.

- The **Global Efficiency** can be split into Parallel Efficiency and Computation Efficiency.

$$GE = PE * CompE$$

- Global Efficiency (GE)
  - Parallel Efficiency (PE)
    - Load Balance Efficiency (LB)
    - Communication Efficiency (CommE)
      - Serialization Efficiency (SerE)
      - Transfer Efficiency (TE)
  - Computation Efficiency (CompE)
    - IPC Scaling
    - Instruction Scaling

Parallel Efficiency (PE)

- The **Parallel Efficiency** describes how well the execution of the code in parallel is working.

- The **Parallel Efficiency** can be split into Load Balance Efficiency and Communication Efficiency.

$$PE = LB * CommE$$

- Global Efficiency (GE)
  - Parallel Efficiency (PE)
    - Load Balance Efficiency (LB)
    - Communication Efficiency (CommE)
      - Serialization Efficiency (SerE)
      - Transfer Efficiency (TE)
  - Computation Efficiency (CompE)
    - IPC Scaling
    - Instruction Scaling

Load Balance Efficiency (LB)

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

- The **Load Balance Efficiency** is the ratio between the average time of a process spend in computation and the maximum time a process spends in computation.

$$\text{LB} = \frac{avg(tcomp)}{\max(tcomp)}$$

- Global Efficiency (GE)
  - Parallel Efficiency (PE)
    - Load Balance Efficiency (LB)
    - Communication Efficiency (CommE)
      - Serialization Efficiency (SerE)
      - Transfer Efficiency (TE)
  - Computation Efficiency (CompE)
    - IPC Scaling
    - Instruction Scaling

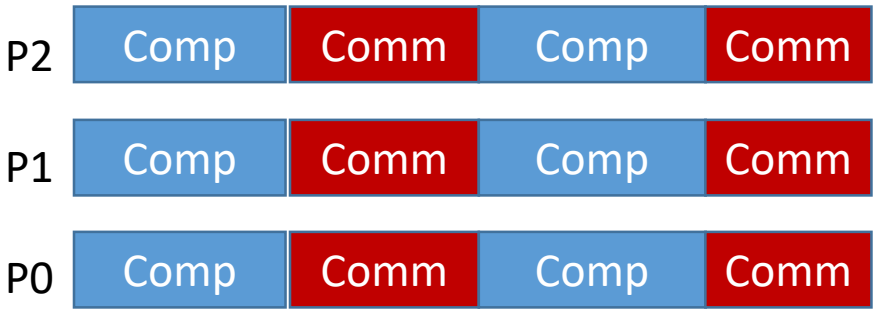
Load Balance Efficiency (LB)

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

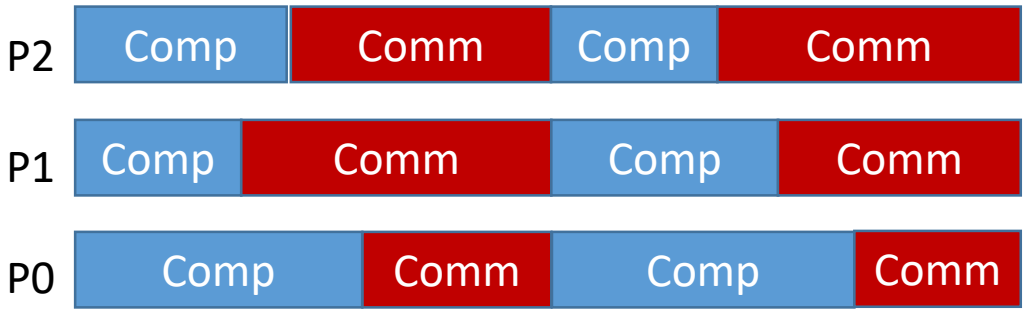
$$\text{LB} = \frac{avg(tcomp)}{\max(tcomp)}$$

- Global Efficiency (GE)
  - Parallel Efficiency (PE)
    - Load Balance Efficiency (LB)
    - Communication Efficiency (CommE)
      - Serialization Efficiency (SerE)
      - Transfer Efficiency (TE)
    - Computation Efficiency (CompE)
      - IPC Scaling
      - Instruction Scaling

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



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



Communication Efficiency (CommE)

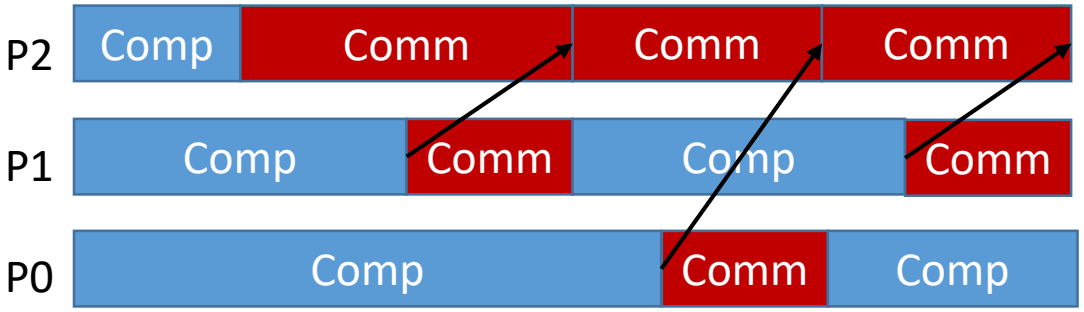
- The **Communication Efficiency** reflects the loss of efficiency by communication.

- The **Communication Efficiency** can be computed as

$$\max_{processes}\left(\frac{computation\ time}{total\ runtime}\right)$$

- Global Efficiency (GE)
  - Parallel Efficiency (PE)
    - Load Balance Efficiency (LB)
    - Communication Efficiency (CommE)
      - Serialization Efficiency (SerE)
      - Transfer Efficiency (TE)
  - Computation Efficiency (CompE)
    - IPC Scaling
    - Instruction Scaling

Example:



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

CommE = $^5/_6$ = 83%

Communication Efficiency (CommE)

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

$$CommE = SerE * TE$$

- Global Efficiency (GE)
  - Parallel Efficiency (PE)
    - Load Balance Efficiency (LB)
    - Communication Efficiency (CommE)
      - Serialization Efficiency (SerE)
      - Transfer Efficiency (TE)
  - Computation Efficiency (CompE)
    - IPC Scaling
    - Instruction Scaling

Serialization Efficiency (SerE)

- 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, because one required process did not arrive at the communication call yet.
- On an ideal network with instantaneous data transfer these inefficiencies are still present, as no real data transfer happens.

- Global Efficiency (GE)
  - Parallel Efficiency (PE)
    - Load Balance Efficiency (LB)
    - Communication Efficiency (CommE)
      - Serialization Efficiency (SerE)
      - Transfer Efficiency (TE)
  - Computation Efficiency (CompE)
    - IPC Scaling
    - Instruction Scaling

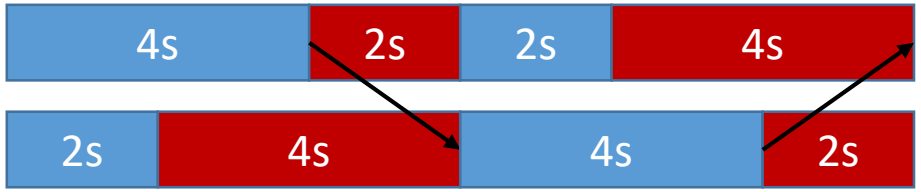
Serialization Efficiency (SerE)

- On an ideal network with instantaneous data transfer these inefficiencies are still present, as no real data transfer happens.
- Serialization Efficiency is computed as

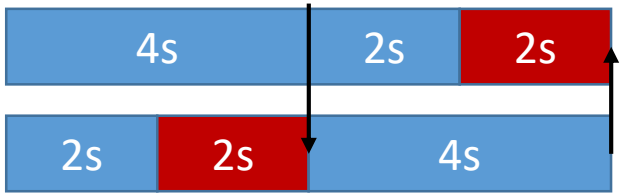
$$\max_{processes}\left(\frac{computation\ time\ on\ ideal\ network}{total\ runtime\ on\ ideal\ network}\right)$$

- Global Efficiency (GE)
  - Parallel Efficiency (PE)
    - Load Balance Efficiency (LB)
    - Communication Efficiency (CommE)
      - Serialization Efficiency (SerE)
      - Transfer Efficiency (TE)
  - Computation Efficiency (CompE)
    - IPC Scaling
    - Instruction Scaling

Execution on a real network



Simulation on an ideal network



$TE = \frac{8}{12} = 75\%$

= Computation

= Communication

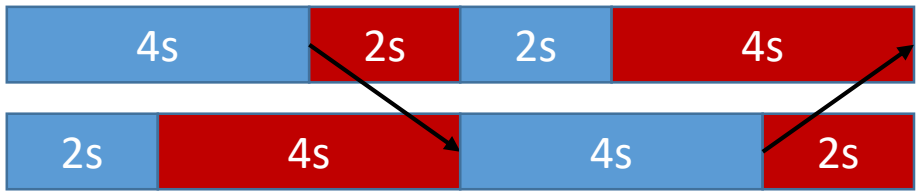
Transfer Efficiency (TE)

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

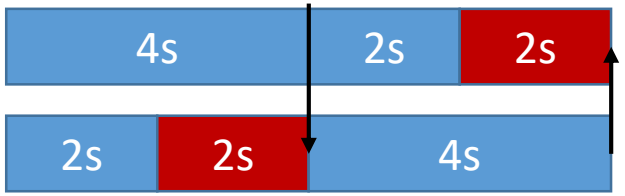
$$\text{TE} = \frac{total\ runtime\ on\ ideal\ network}{total\ measured\ runtime}$$

- Global Efficiency (GE)
  - Parallel Efficiency (PE)
    - Load Balance Efficiency (LB)
    - Communication Efficiency (CommE)
      - Serialization Efficiency (SerE)
      - Transfer Efficiency (TE)
  - Computation Efficiency (CompE)
    - IPC Scaling
    - Instruction Scaling

Execution on a real network



Simulation on an ideal network



$\text{TE} = \frac{8}{12} = 75\%$

= Computation

= Communication

Computation Efficiency (CompE)

- The **Computation Efficiency** describes how well the computational load of an application scales with the number of processes.
- The **Computation Efficiency** is computed by comparing the total time spend in computation for a different number of threads/processes.
- For a linearly-scaling application the total time spend in computation is constant and thus the Computation efficiency is one.

- Global Efficiency (GE)
  - Parallel Efficiency (PE)
    - Load Balance Efficiency (LB)
    - Communication Efficiency (CommE)
      - Serialization Efficiency (SerE)
      - Transfer Efficiency (TE)
  - Computation Efficiency (CompE)
    - IPC Scaling
    - Instruction Scaling

IPC Scaling / Instruction Scaling

- A low computation efficiency can have two reasons:
  1. With more processes more instructions are executed, e.g. some extra computation for the domain decomposition is needed.

     Instruction Scaling compares the total number of instructions executed for a different number of threads/processes.
  2. The same number of instructions is computed but the computation takes more time, this can happen e.g. due to shared recourses like memory channels.

     IPC Scaling compares how many instructions per cycle are executed for a different number of threads/processes.

- Global Efficiency (GE)
  - Parallel Efficiency (PE)
    - Load Balance Efficiency (LB)
    - Communication Efficiency (CommE)
      - Serialization Efficiency (SerE)
      - Transfer Efficiency (TE)
  - Computation Efficiency (CompE)
    - IPC Scaling
    - Instruction Scaling

Performance Optimisation and Productivity

A Centre of Excellence in Computing Applications

Contact:

https://www.pop-coe.eu

mailto:pop@bsc.es

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