Parallel Engineering Codes: Performance Optimisation with the POP Methodology

Fouzhan Hosseini, The Numerical Algorithms Group (NAG)

Fouzhan.hosseini@nag.co.uk, Nov 2020

EU H2020 Centre of Excellence (CoE)

1 December 2018 – 30 November 2021

Grant Agreement No 824080

Performance Optimisation and Productivity

A Centre of Excellence in HPC

- Promotes best practices in parallel programming
  - Improving Parallel Software can add a lot of value: Reduced expenditure, faster results, novel solutions
  - **The POP Methodology -** a systematic approach to performance optimization building a quantitative picture of application behavior

- Free services for all EU academic and industrial codes and users
  - Suggestions on improving code performance, described in a ***Performance Assessment***
  - Practical help with code refactoring through a ***Proof of Concept***

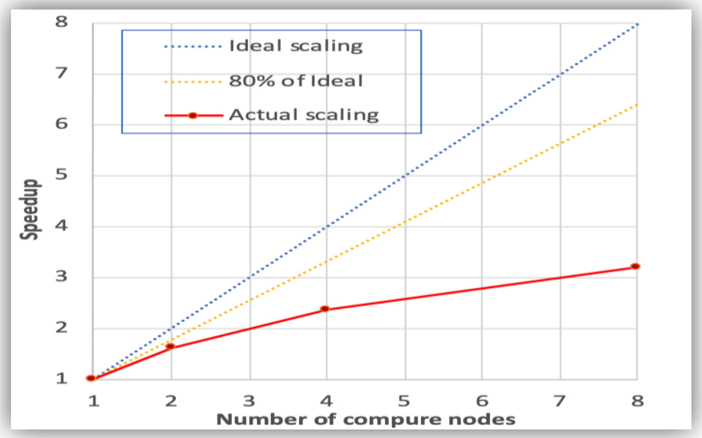
- A Team with
  - Excellence in performance tools and tuning
  - Excellence in programming models and practices
  - R & D background in real academic and industrial use cases

For further information, visit: https://www.pop-coe.eu pop@bsc.es @POP_HPC youtube.com/POPHPC

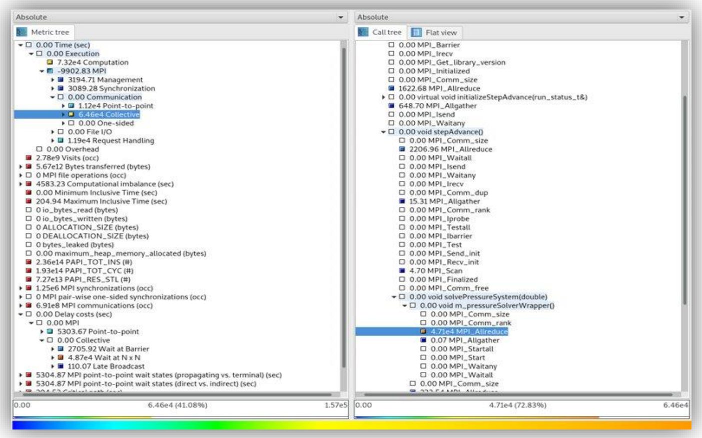
Parallel Performance is hard to understand

How do we measure the performance of our parallel programs?

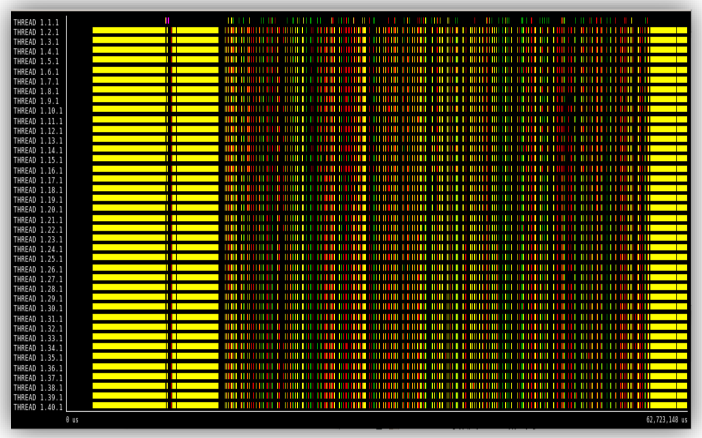
- Traditional speed-up and efficiency plots?
- Profiling & tracing with performance tools?
  - Tracing is powerful, but potentially generates overwhelming amount of data



Speedup plot



Cube, perf. metrics per routines/call stack, data collected by Scalasca/Score-P



Paraver, timeline view of program execution, data collected by Extrae

Difficult to know where to start and what to look for

Main Problem: Lack of quantitative understanding of the actual behavior of a parallel application

A Solution: The POP Metrics

Simple but extremely powerful idea

- Devise a simple set of performance metrics using values easily obtained from the trace data
- Where low values indicate **specific** causes of poor parallel performance

These metrics then are used to understand

- What are the causes of poor performance
- What to look for in the trace data

- Besides, the metrics provide a common ground for discussing performance issues
  - Between developers, users and analysts

POP MPI Parallel Efficiency Metrics

Global Efficiency
Describes how well your parallel application scales.
GE = PE × Comps

Parallel Efficiency
Describe how well the execution of your code is working in parallel
PE = LB × CommE

Communication Efficiency
Reflects the loss of efficiency by communication.
CommE = max (CT/TT) = SerE × TE

Transfer Efficiency
Describes the loss of efficiency due to actual data transfer time.
TE = TT on ideal network / TT

Computational Scaling
Describe how well the computational load of an application scales with the number of processors/threads.
CompS

Load Balance
Measures how well work is distributed to threads/processes in the application.
LB = average (CT) / max (CT)

Serialisation Efficiency
Describes efficiency loss due to dependencies, waiting in MPI calls, ets.
SerE = max(CT / TT on ideal network)

Key
CT Computation Time
TT Total Time
Ideal network data transfer takes zero time

Instruction Scaling
Compares the total number of instructions executed for different numbers of threads/processes.

IPC Scaling
Compares how many instructions per cycle (IPC) are executed for different numbers of threads/processes.

Frequency Scaling
Compares the processor frequency for different numbers of threads/processes.

For more details visit https://pop-coe.eu

POP Metrics Are Easy to Calculate

From any trace data, we only need

- Runtime
- Max computation time over all processes
- Average computation time over all processes
- Total number of useful cycles over all processes
- Total number of useful instructions over all processes
- Runtime on an ideal network (optional)

Or use tools developed and supported by the POP CoE

POP Performance Monitoring Tools

Developing open-source tools

- Extrae (tracing), Paraver (visualisation) & Dimemas
  - https://tools.bsc.es
- Score-P (profiling and tracing), Scalasca (Post Processing) & Cube (visualisation)
  - https://www.scalasca.org
- MAQAO: synthetic reports and hints with a focus on core performance
  - http://www.maqao.org
- PyPOP: automated generation of POP metrics from Extrae traces
  - https://github.com/numericalalgorithmsgroup/pypop

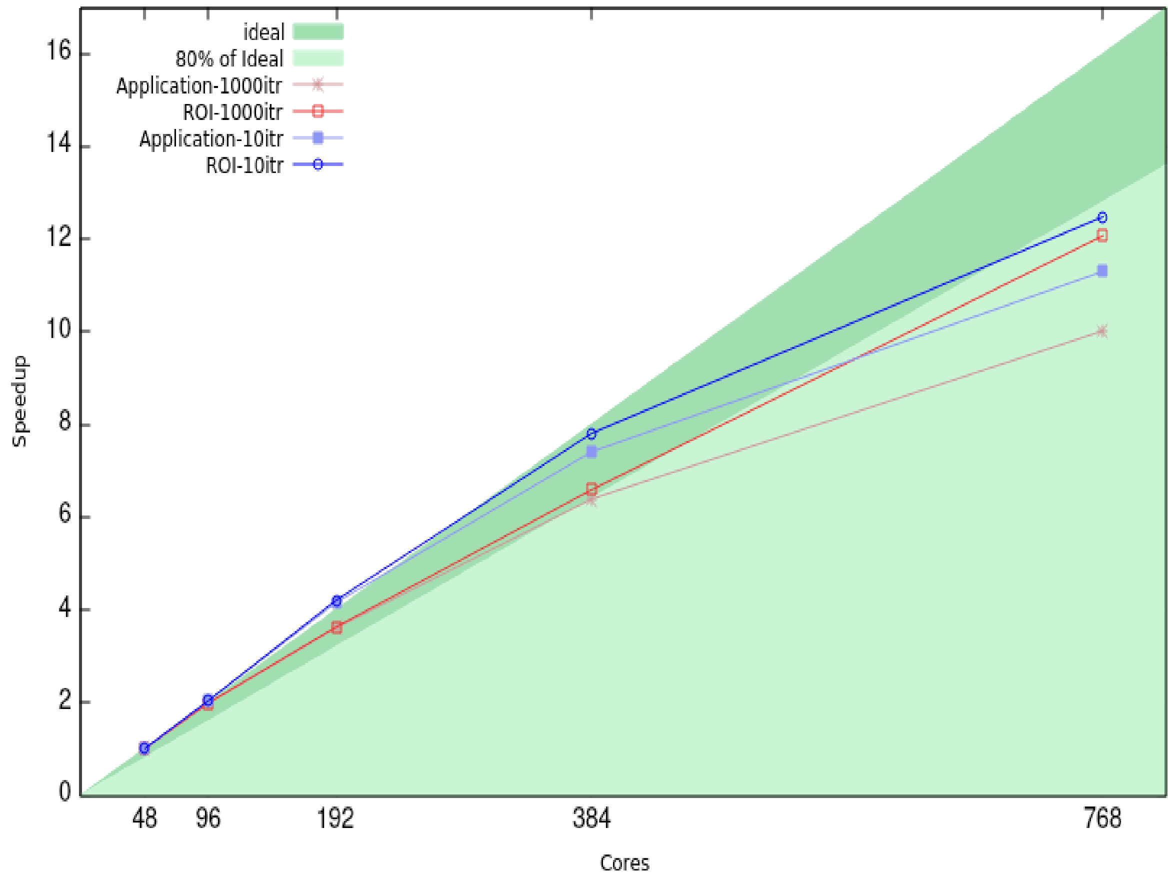
For more help on how to use these tools and calculate the POP metrics

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

Other tools can also be used

Example 1: A Computational Fluid Dynamics Code

- Code: **C++, MPI**
- Platform: MareNostrum-IV(@BSC)
  - Dual Intel Xeon Platinum 8160 Skylake 48-core nodes
- Performance data collected using ***Score-P/Scalasca***
  - Using compiler instrumentation filter and hardware counters
- Scale:
  - ***48-768 cores (1-16 nodes)***



Example 1- POP Metrics

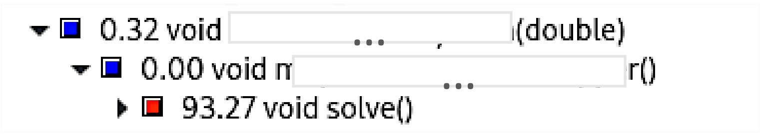
Number of cores	48	96	192	384	768
Global Efficiency	0.93	0.94	0.93	0.84	0.76
↳ Parallel Efficiency	0.93	0.91	0.87	0.77	0.68
↳ Load balance	0.99	0.98	0.98	0.97	0.95
↳ Communication Efficiency	0.94	0.92	0.89	0.79	0.72
↳ Serialisation	0.95	0.94	0.92	0.85	0.81
↳ Transfer efficiency	0.99	0.99	0.97	0.94	0.89
↳ Computational Scaling	1.00	1.03	1.07	1.09	1.12
↳ Instruction Scaling	1.00	0.99	0.97	0.95	0.92
↳ IPC Scaling	1.00	1.05	1.10	1.18	1.27
↳ Frequency Scaling	1.00	1.00	1.00	0.98	0.96

- We immediately see that **Serialisation** is the main factor that limits the scalability
- Efficiency values are between 0 to 1, and
  - metric values above 0.8 represent acceptable performance

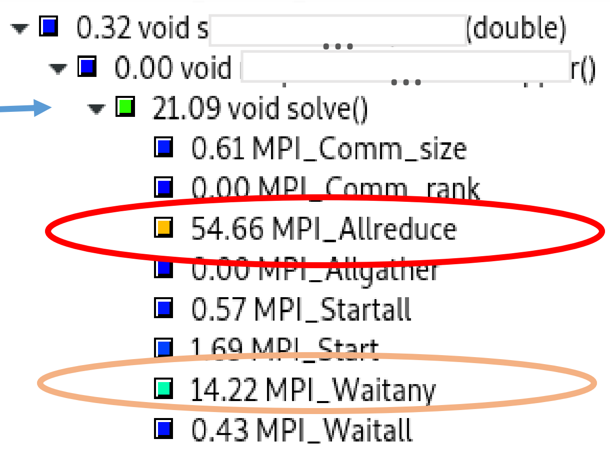
Example 1: Cause of Low Serialisation Efficiency

- Serialisation
  - typically happens due to at least one process arriving early/late at synchronization point

- Scalasca calculates a delay cost metric
  - This metric highlights the root causes of serialization
  - Attributes processes' waiting time to the routines causing serialization



Numbers report percentage of total delay cost for Example 2 - ROI on 768 cores

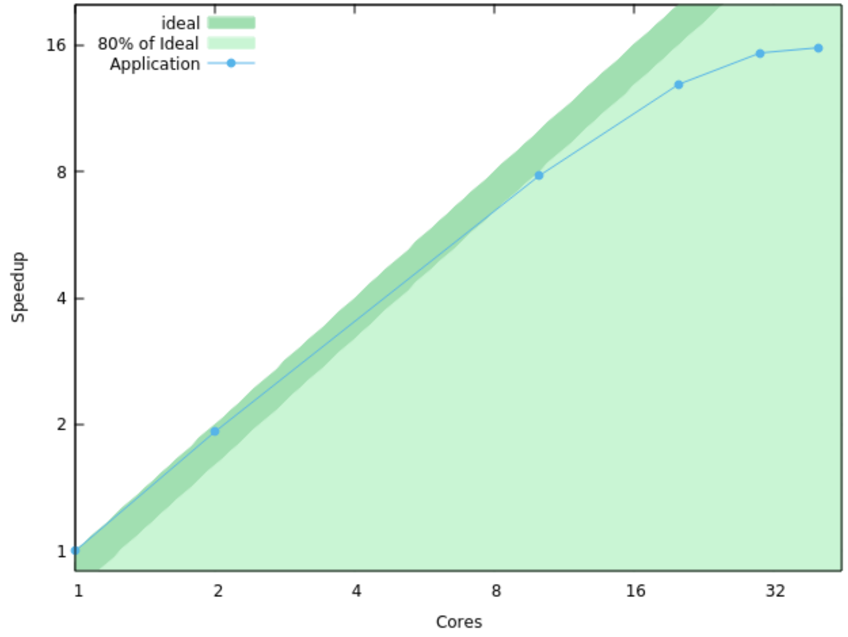


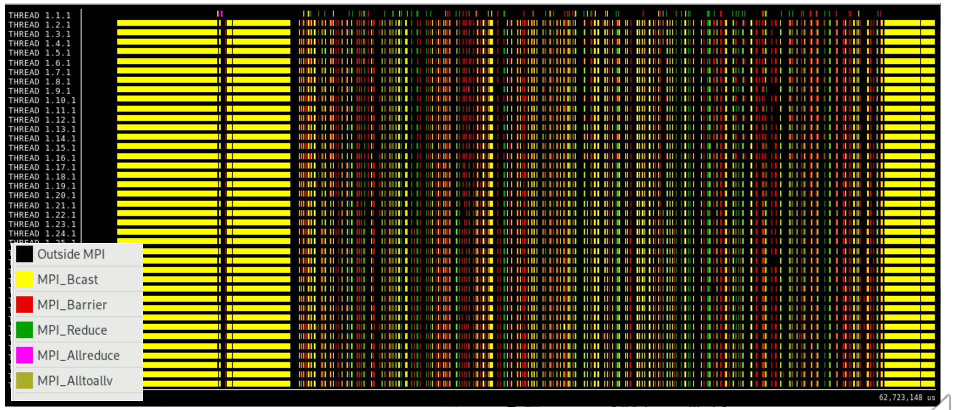
Inclusive values (▶)
Exclusive values (▼)

- **The MPI collective calls and imbalanced computation regions within a Library call were the main causes of the serialisation on 768 cores**

Example 2: A Molecular Dynamic Simulation Code

- Code: C++, Fortran, MPI
  - No access to the source code
- Platform:
  - Dual Intel Xeon Gold 6248 CPU @ 2.50GHz – 40 cores
  - Intel Fortran and C++ compiler with MKL and MPI Library (2019 version)
- Performance data collected using **Extrae**
- Scale:
  - **2-40 cores**





Timeline of the program execution on 40 cores

Example 2: POP Metrics

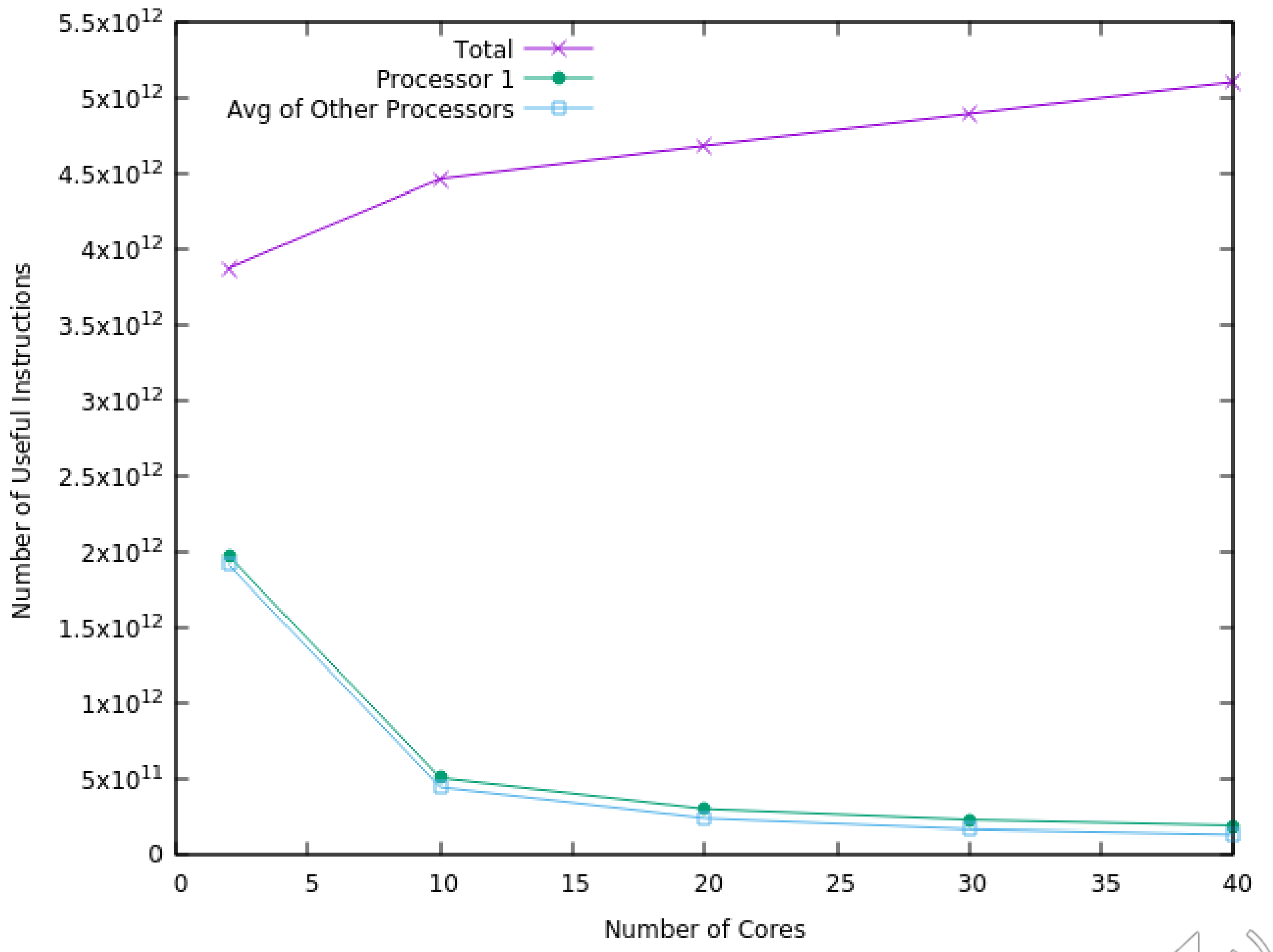
Number of cores	2	10	20	30	40
Global Efficiency	0.95	0.73	0.60	0.47	0.36
↳ Parallel Efficiency	0.95	0.89	0.81	0.75	0.68
↳ Load balance	0.95	0.92	0.85	0.81	0.80
↳ Communication Efficiency	0.99	0.97	0.95	0.92	0.85
↳ Serialisation	1.00	0.99	0.99	0.98	0.94
↳ Transfer efficiency	0.99	0.98	0.96	0.94	0.91
↳ Computational Scaling	1.00	0.82	0.74	0.63	0.53
↳ Instruction Scaling	1.00	0.87	0.83	0.79	0.76
↳ IPC Scaling	1.00	0.99	0.95	0.90	0.83
↳ Frequency Scaling	1.00	0.95	0.94	0.88	0.84

Poor scalability of the code is due to multiple factors:

- **Load imbalance** and **increasing instruction count** are major limiting factors,
- Resulting in, respectively, poor Parallel Efficiency and poor Computational scaling

Example 2: Useful Instructions

- Total number of useful instructions increases with increasing number of processes
  - Low Instruction scaling
- Process 1 always executes more instructions compared with other processes
  - Load imbalance
- With 40 processes, Processor 1 executes 46% more instructions with respect to average number of instruction per process
  - Amdahl's law



Example 3: A Computational Fluid Dynamics Code

- Code: **Fortran, OpenMP**
  - POP *Performance Assessment* followed by *Proof of Concept* service
- Platform:
  - MareNostrum-IV(@BSC)
    - Dual Intel Xeon Platinum 8160 Skylake 48-core nodes
- Tools used:
  - **Extrae & Paraver**
  - **Vtune**
  - **MAQAO**
- Scale:
  - **1-45 threads**

POP metrics from the *Performance Assessment*

# threads	1	10	30	45
Global Efficiency	1.00	0.80	0.36	0.26
↳ Parallel Efficiency	1.00	0.86	0.60	0.55
↳ OpenMP Region Efficiency	1.00	0.95	0.74	0.70
↳ Serial Region Efficiency	1.00	0.91	0.86	0.85
↳ Computational Scaling	1.00	0.94	0.60	0.48
↳ Instruction Scaling	1.00	1.01	1.00	1.00
↳ IPC Scaling	1.00	0.92	0.61	0.50
↳ Frequency Scaling	1.00	1.00	0.98	0.95

Poor scalability of the code is due to multiple factors:

- **OpenMP Region Efficiency** and **reducing IPC** are major limiting factors,
- Resulting in, respectively, poor Parallel Efficiency and poor Computational scaling

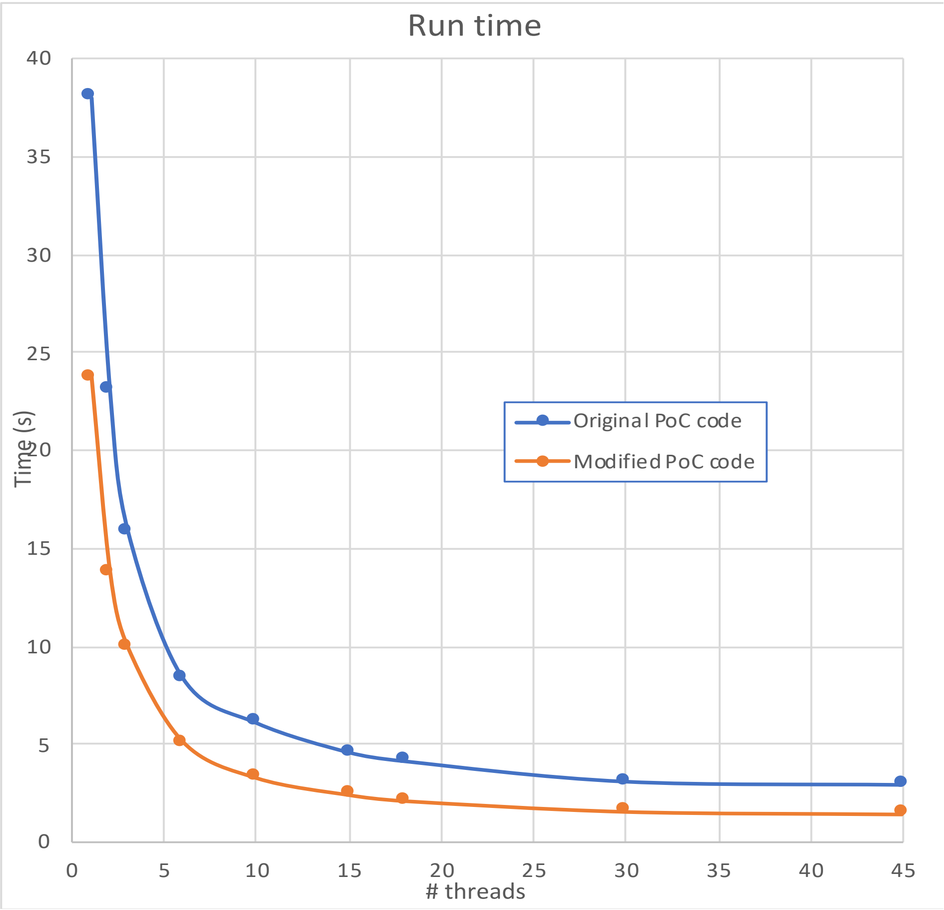
Example 3: Improving the Performance

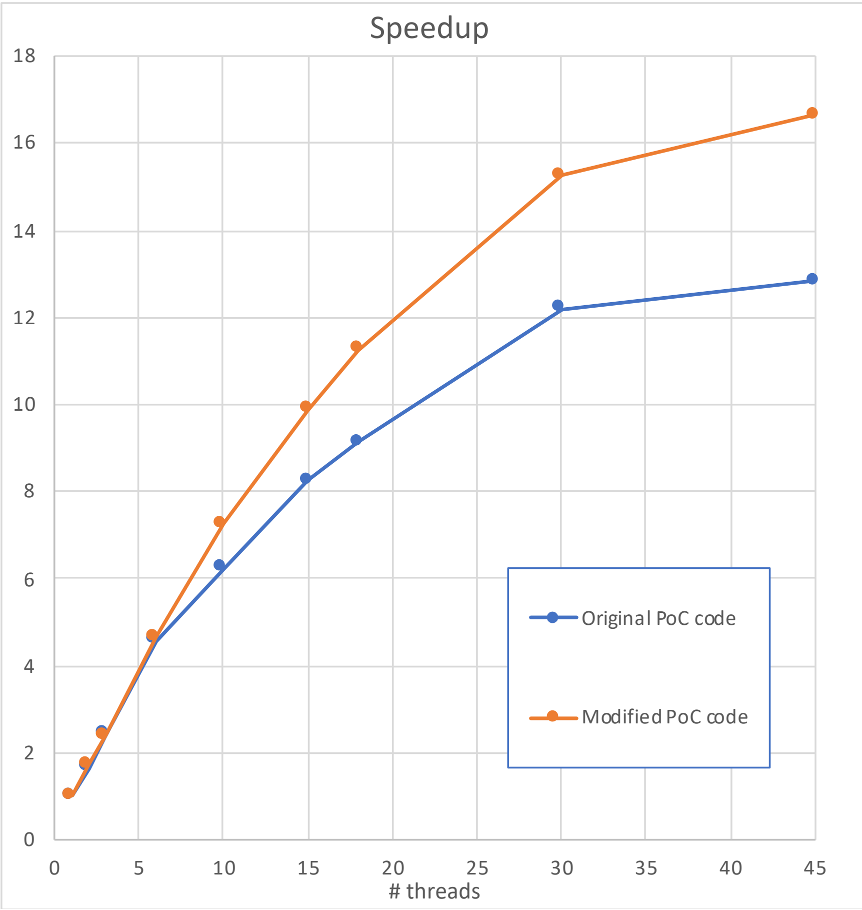
- Refactoring the code to address performance issues via POP Proof of Concept
  - Use of OpenMP COLLAPSE clause to improve load balance
  - Move some calculations outside the loops & remove unnecessary calculations
  - Use optimal loop ordering with nested loops

	Original code for *Proof of Concept*						Modified code					
# threads	1	2	10	18	30	45	1	2	10	18	30	45
Global Efficiency	1.00	0.86	0.65	0.41	0.31	0.15	1.00	0.86	0.72	0.62	0.51	0.37
↳ Parallel Efficiency	1.00	0.97	0.80	0.69	0.62	0.59	1.00	0.97	0.90	0.83	0.78	0.75
↳ OpenMP Region Efficiency	1.00	0.97	0.81	0.69	0.63	0.60	1.00	0.97	0.91	0.85	0.80	0.78
↳ Serial Region Efficiency	1.00	1.00	0.99	0.99	0.99	0.99	1.00	1.00	0.99	0.98	0.98	0.98
↳ Computational Scaling	1.00	0.89	0.81	0.60	0.49	0.26	1.00	0.88	0.81	0.75	0.65	0.49
↳ Instruction Scaling	1.00	1.00	1.00	1.00	0.99	0.97	1.00	1.00	1.00	0.99	0.99	0.98
↳ IPC Scaling	1.00	0.87	0.80	0.60	0.51	0.36	1.00	0.89	0.82	0.77	0.67	0.56
↳ Frequency Scaling	1.00	1.02	1.02	1.00	0.97	0.74	1.00	1.00	0.98	0.98	0.98	0.89

Example 3: Performance of modified code

- The modified code
  - is 1.6x faster on 1 thread due to reduced instruction count
  - is 2.1x faster than original on 45 threads
  - shows better parallel scaling with a speedup of 16.7 on 45 threads relative to 1 thread

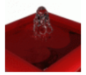




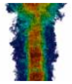
Some Success Stories

- More than 350 services since 2015 across all domains
  - With a significant number of services (about 30%) for engineering
- See ⇨ https://pop-coe.eu/blog/tags/success-stories

- Performance Improvements for SCM's ADF Modeling Suite
- **3x Speed Improvement** for zCFD Computational Fluid Dynamics Solver
- **25% Faster time-to-solution** for Urban Microclimate Simulations
- **2x performance improvement** for SCM ADF code
- Proof of Concept for BPMF leads to around **40% runtime reduction**
- POP audit helps developers **double their code performance**
- **10-fold scalability improvement** from POP services



- POP performance study improves performance **up to a factor 6**



- POP Proof-of-Concept study leads to **nearly 50% higher performance**

- POP Proof-of-Concept study leads to **10X performance improvement** for customer

Online Content

POP Website

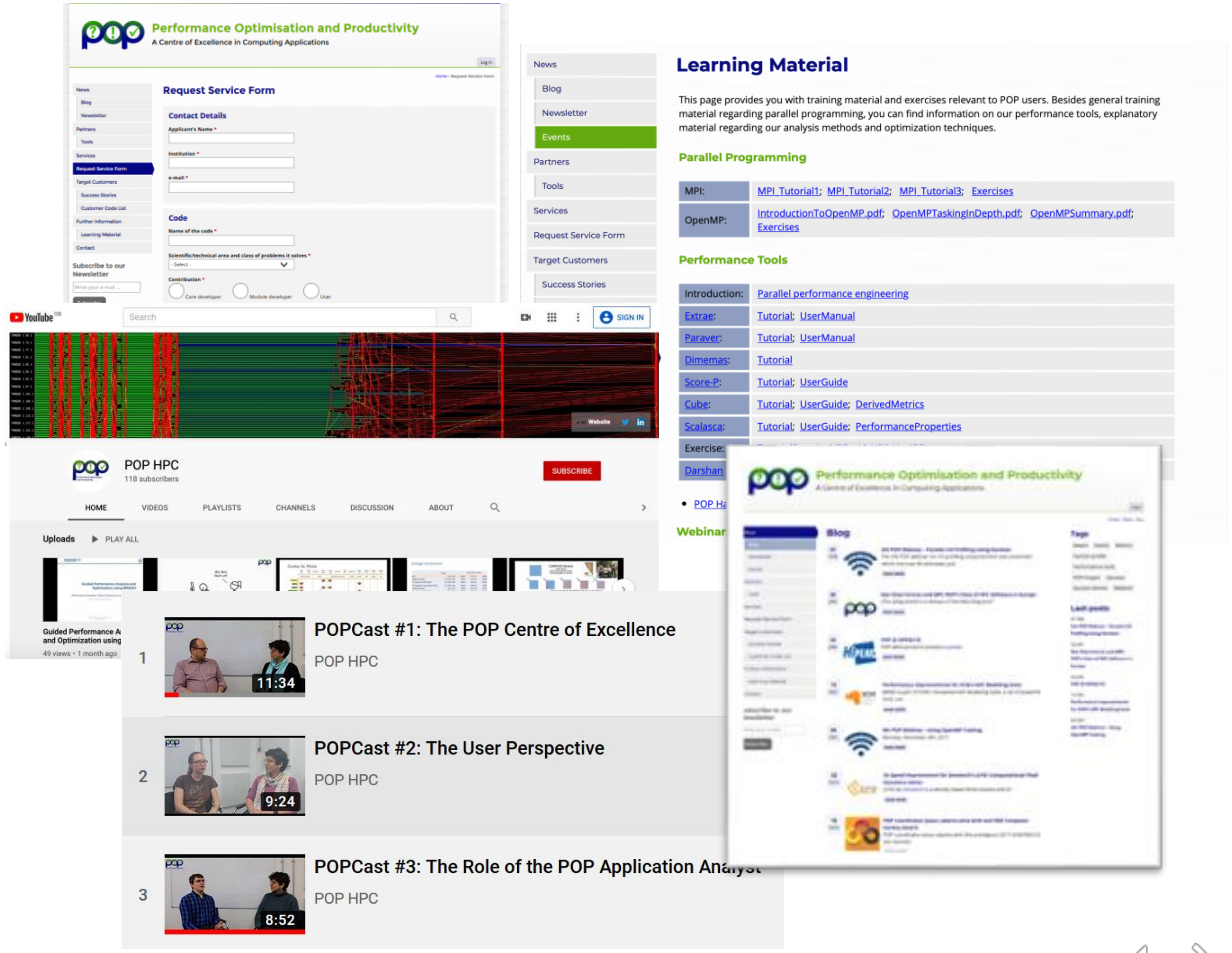
www.pop-coe.eu

- All the information you need to access POP services
  - https://pop-coe.eu/services
- Blogs
- More Learning Materials
- Newsletter
  - subscribe and see past issues

YouTube Channel

https://www.youtube.com/pophpc

- Past Webinars
- POPCasts



POP Online training course

NEW

- A series of self-study modules
  - For those with limited experience in performance analysis of HPC applications
- Learning Objectives:
  - The challenges involved in HPC performance analysis
  - How the POP Metrics aid understanding of application performance
  - How to calculate the POP Metrics for your own HPC applications
  - What POP tools are available and how they can be installed
  - How to capture and analyse performance data with the POP tools

Summary

POP Performance Metrics

- Build a quantitative picture of application behavior
- Allow quick diagnosis of performance problems in parallel codes
- Identify strategic directions for code refactoring
- So far metrics for **MPI**, **OpenMP** and **Hybrid** (OpenMP + MPI) codes

POP works

- Across application domains, platforms, scales
- With (EU) academic and industrial customers including code developers, code users, HPC service providers and vendors
  - To apply for a POP service go to https://pop-coe.eu/services

POP CoE

- Promotes **best practices in parallel programming**
- Encourages a systematic approach to performance optimization
- Facilitates and invests in training HPC users

Performance Optimisation and Productivity

Improving the performance of CAE codes

POP achieves three times speed up in Computational Fluid Dynamics code

POP worked with Zenotech on their computational fluid dynamics solver zCFD. The software is written in Python and C++.

As a result of a POP Proof-of-Concept study zCFD ran 3x faster on a representative input case. This was achieved by:

- Parallelising serial portions of code, specifically those which were not already running in parallel because of a compiler problem.
- Improving computational load balance across OpenMP threads by avoiding very slow mathematical function calls
- Changing execution environment settings to boost CPU performance.

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.