Understand the Performance of your Application with just Three Numbers

Jesus Labarta (BSC)

POP Webinar 3

October 11th 2017

EU H2020 Center of Excellence (CoE)

Agenda

- Motivation and current practices
- Efficiency model
- How to compute it
- Examples
- What’s next

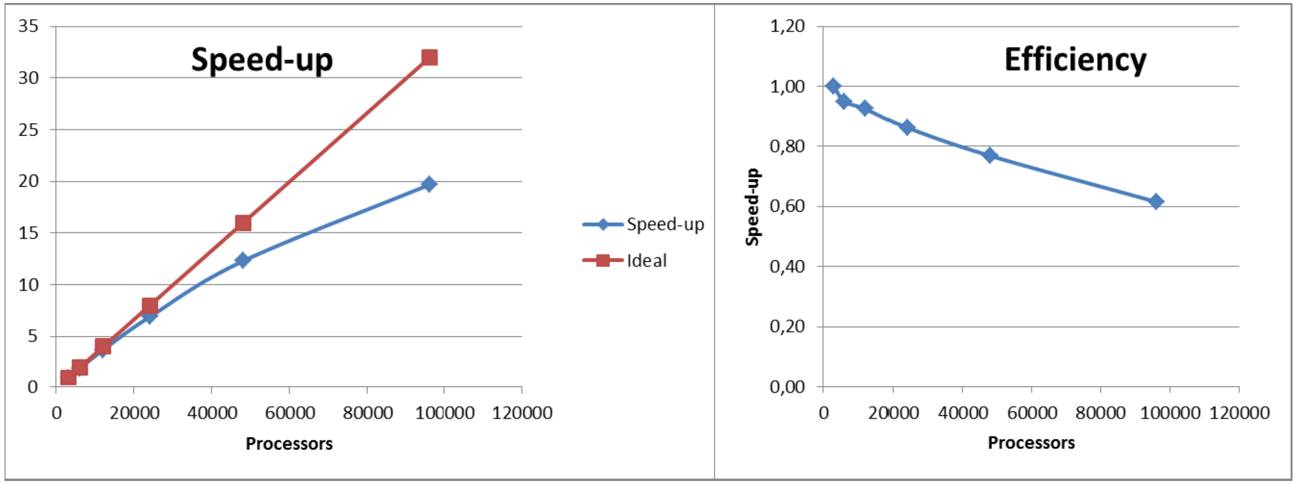
Measuring performance of MPI programs

- How do we measure the performance of our MPI programs?
  - Elapsed time
  - Scaling plots
  - Profiles
  - Traces

- How much insight do we get?
  - Who to blame?
    - Myself? the machine? the programming model? its implementor? the tool developer? The environment and way the program is run?
  - Proper direction to refactor?

Performance and scaling

- Elapsed time
- Scaling plots
  - Speedup, efficiency

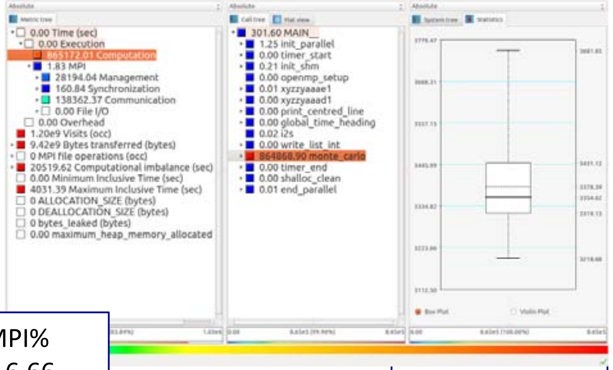


- To consider
  - The global effect
  - Too coarse aggregation
    - Risk of speculating about causes of observed behavior with little capability of verifying hypotheses
  - Reference time for scaling plots

Profiling

- Aggregate metrics (mostly time)
  - During program execution
- For components of syntactic structure
  - routines, call stacks, loops
- Hotspots
  - Code regions dominating the profile where to focus optimization

- To consider
  - Loose information on distributions
  - Many codes flat
  - Keep in mind Amdahl's law



```
Absolute
Metric tree
▾□ 0.00 Time (sec)
  ▾□ 0.00 Execution
      ■ 865172.01 Computation
    ▾■ 1.83 MPI
      ▸■ 28194.04 Management
      ▸■ 160.84 Synchronization
      ▸■ 138362.37 Communication
      ▸□ 0.00 File I/O
   □ 0.00 Overhead
 ■ 1.20e9 Visits (occ)
▸■ 9.42e9 Bytes transferred (bytes)
▸□ 0 MPI file operations (occ)
▸■ 20519.62 Computational imbalance (sec)
 □ 0.00 Minimum Inclusive Time (sec)
 ■ 4031.39 Maximum Inclusive Time (sec)
 □ 0 ALLOCATION_SIZE (bytes)
 □ 0 DEALLOCATION_SIZE (bytes)
 □ 0 bytes_leaked (bytes)
 □ 0.00 maximum_heap_memory_allocated
```

```
Absolute
Call tree   Flat view
▾■ 301.60 MAIN__
  ▸■ 1.25 init_parallel
  ▸■ 0.00 timer_start
  ▸■ 0.21 init_shm
   ■ 0.00 openmp_setup
  ▸■ 0.01 xyzzyaaae1
  ▸■ 0.00 xyzzyaaad1
  ▸■ 0.00 print_centred_line
  ▸■ 0.00 global_time_heading
   ■ 0.02 i2s
  ▸■ 0.00 write_list_int
  ▸■ 864868.90 monte_carlo
  ▸■ 0.00 timer_end
  ▸■ 0.00 shalloc_clean
  ▸■ 0.01 end_parallel
```

Scalasca (JSC)

```
Task  AppTime    MPITime     MPI%
  0     15.3        1.02     6.66
  1     15.3       0.293     1.91
  2     15.3       0.607     3.95
  3     15.3       0.239     1.56
  ....
  *      123        6.37     5.19
```

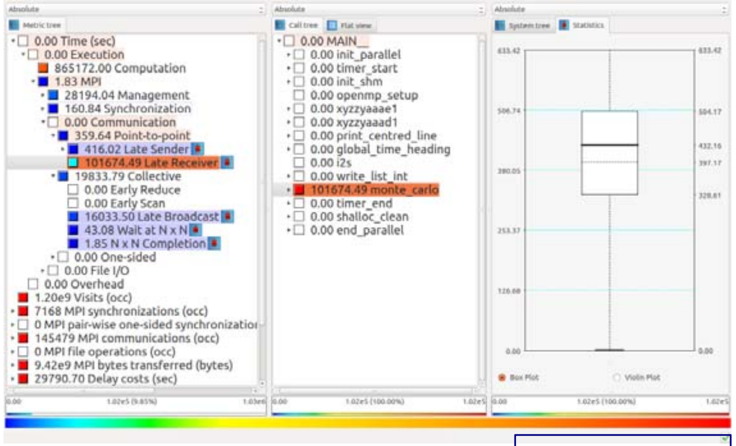
mpiP

gprof

```
Each sample counts as 0.01 seconds.
  %   cumulative   self     name
22.80    20.82    20.82    LagrangeNodal(Domain&)
18.72    37.92    17.10    CalcFBHourglassForceForElems(Domain&, …)
17.15    53.58    15.66    EvalEOSForElems(Domain&, double*, …)
12.68    65.16    11.58    CalcKinematicsForElems(Domain&, double*, …)
10.87    75.09     9.93    IntegrateStressForElems(Domain&, double*, …)
6.53     81.05     5.96    CalcMonotonicQGradientsForElems(Domain&, …)
4.80     85.43     4.38    CalcQForElems(Domain&, double*)
….
```

Tracing

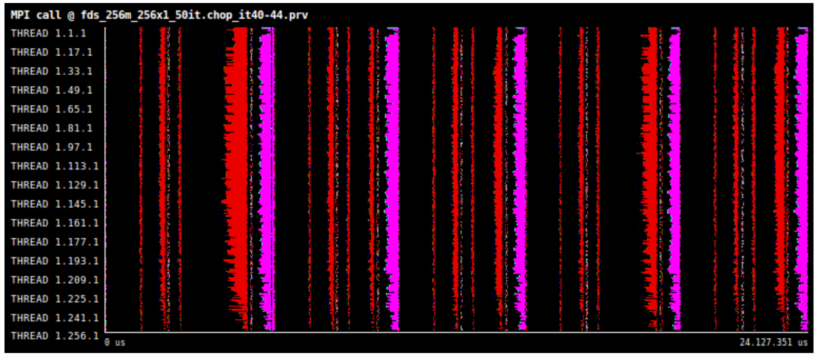
- Emitting all events for later analysis or visualization



```
□ 0.00 Time (sec)
  □ 0.00 Execution
    865172.00 Computation
    1.83 MPI
      28194.04 Management
      160.84 Synchronization
      □ 0.00 Communication
        359.64 Point-to-point
          416.02 Late Sender
          101674.49 Late Receiver
        19833.79 Collective
          □ 0.00 Early Reduce
          □ 0.00 Early Scan
          16033.50 Late Broadcast
          43.08 Wait at N x N
          1.85 N x N Completion
        □ 0.00 One-sided
      □ 0.00 File I/O
  □ 0.00 Overhead
1.20e9 Visits (occ)
7168 MPI synchronizations (occ)
□ 0 MPI pair-wise one-sided synchronizatior
145479 MPI communications (occ)
□ 0 MPI file operations (occ)
9.42e9 MPI bytes transferred (bytes)
29790.70 Delay costs (sec)
```

```
□ 0.00 MAIN__
  □ 0.00 init_parallel
  □ 0.00 timer_start
  □ 0.00 init_shm
  □ 0.00 openmp_setup
  □ 0.00 xyzzyaaae1
  □ 0.00 xyzzyaaad1
  □ 0.00 print_centred_line
  □ 0.00 global_time_heading
  □ 0.00 i2s
  □ 0.00 write_list_int
  101674.49 monte_carlo
  □ 0.00 timer_end
  □ 0.00 shalloc_clean
  □ 0.00 end_parallel
```

Scalasca (JSC)



```
MPI call @ fds_256m_256x1_50it.chop_it40-44.prv
THREAD 1.1.1
THREAD 1.17.1
THREAD 1.33.1
THREAD 1.49.1
THREAD 1.65.1
THREAD 1.81.1
THREAD 1.97.1
THREAD 1.113.1
THREAD 1.129.1
THREAD 1.145.1
THREAD 1.161.1
THREAD 1.177.1
THREAD 1.193.1
THREAD 1.209.1
THREAD 1.225.1
THREAD 1.241.1
THREAD 1.256.1
0 us                                   24.127.351 us
```

Paraver (BSC)

- To consider
  - Lots of data
    - The “Big Performance Data” challenge: how to handle
    - The “Performance Analytics” challenge: flexibility, analysis power, interpretation

Insight on performance

- Understanding performance isn't easy (Jon Gibson POP 1st webinar)
  - Many factors and interaction between them
  - Potentially overwhelming amount of data. How to get real insight ?

- Can we report performance …
  - Few numbers ?
  - Fundamental concepts ?
- … pointing to "strategic" directions on how to refactor the code ?

- Having a common ground, abstracted from program specificities, on which to discuss between developers, users and analysts would be extremely useful

Characterizing MPI application performance

- Parallel Efficiency Model
  - 0..1
  - Multiplicative

$$ParEff = LB * \overbrace{Ser * Trf}^{CommEff}$$

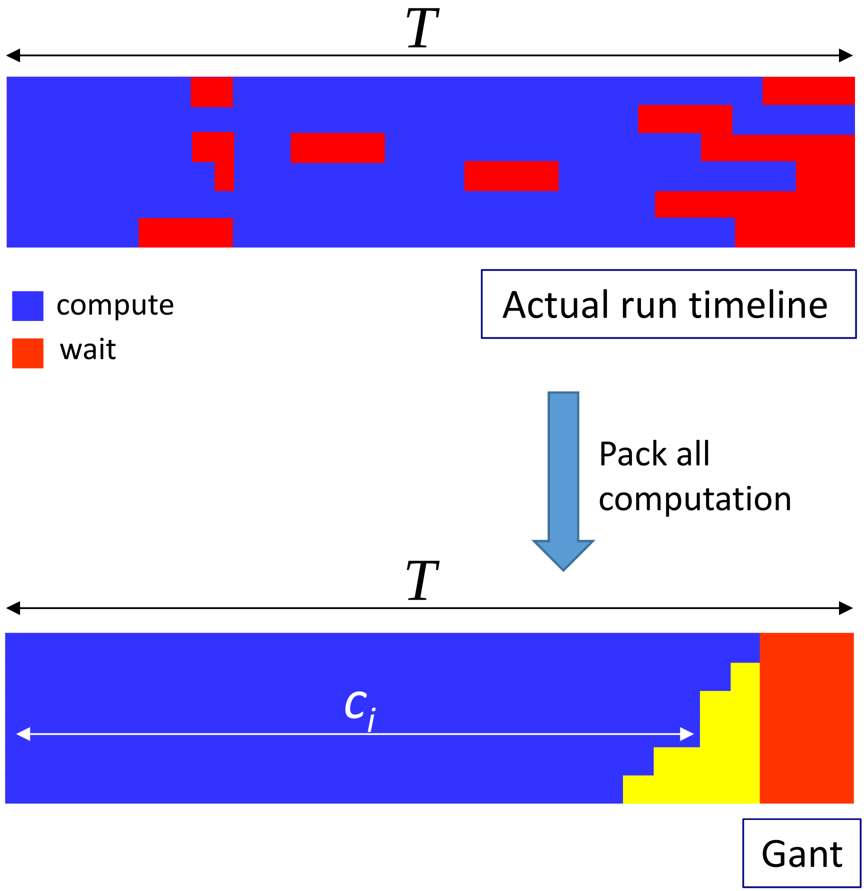
- Efficiency factors
  - Load balance
    - Globally uneven distribution of work
  - Serialization
    - Synchronization. “Circular” wait for “slow” processes
    - Dependencies or dynamic imbalances propagated through synchronizations
  - Transfer
    - Actual limitation caused by data transfer

A bit on load balance

- Load balance efficiency
  - Account for variability in amount of work between processes
  - Directly reflecting impact of such variability in performance (parallel efficiency)

LB = 🟦 / (🟦 + 🟨)

$$LB = \frac{avg(...)}{\max(...)}$$



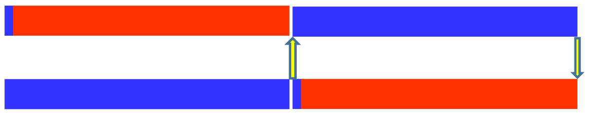
Actual run timeline

Pack all computation

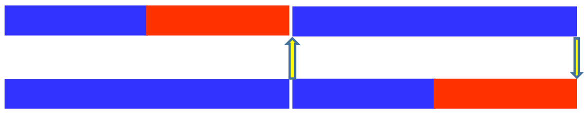
Gant

A bit more on serialization

- Actual dependence chains

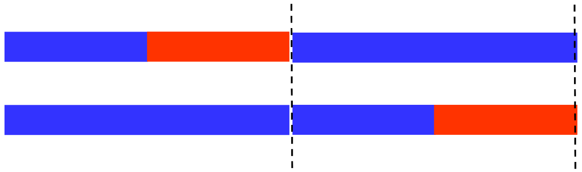


p2p message

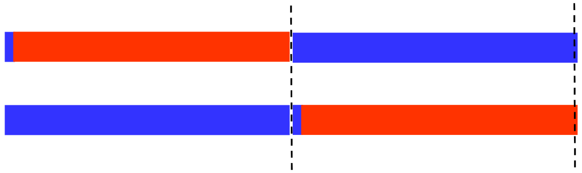


p2p message

- Alternating load imbalances



Collective call



Collective call

🟦 compute

🟥 wait

Why are these metrics important ?

- They quantify fundamental parallel programming concepts …
  - Other metrics do not:
    - Lot of time in MPI →
      - Blame MPI vendor?
      - Pack messages ?
      - Overlap communication and computation ?
      - Improve domain decomposition ?
      - Work on numbering algorithm ?
- … providing deep insight/awareness …
  - Of known characteristic of the program … even if not properly quantified
  - Exposing unexpected behaviors
- … and a common ground for discussion

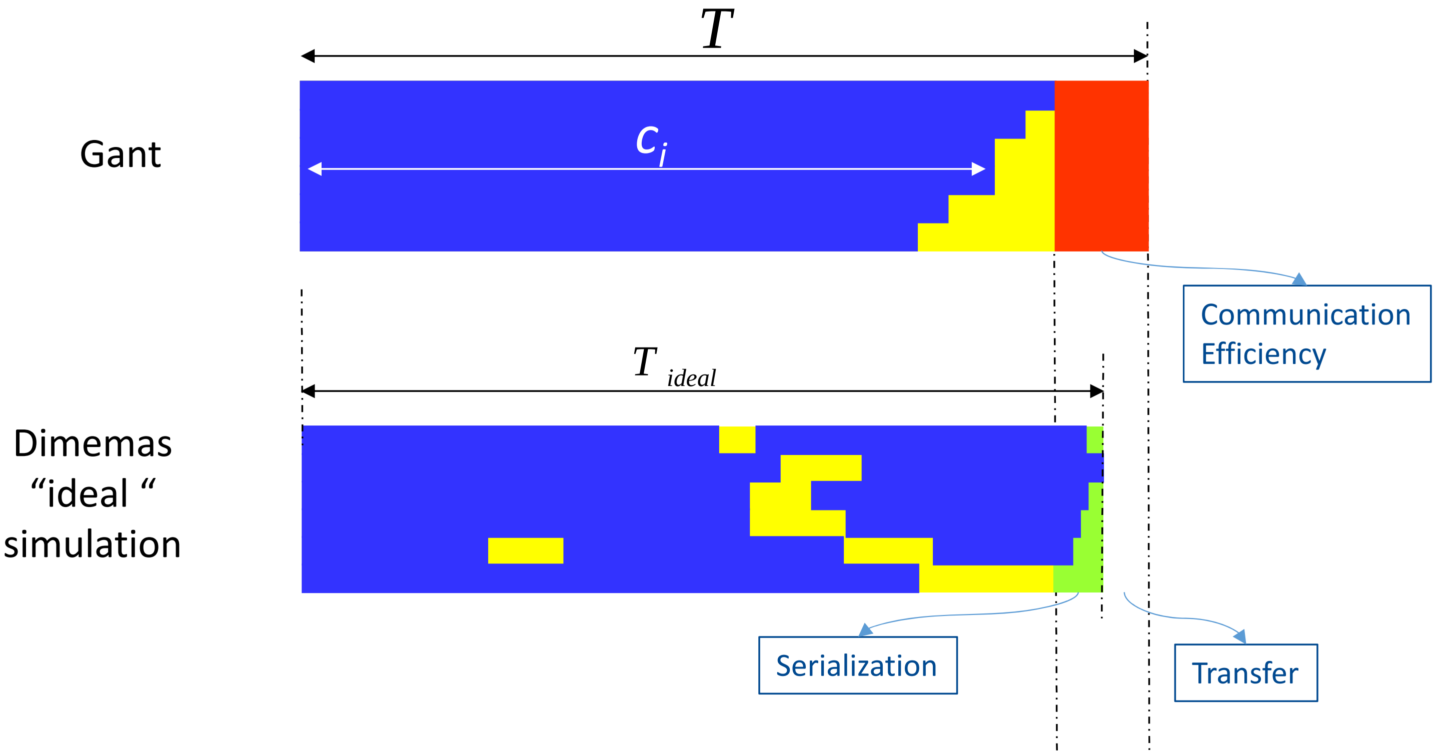
Example

	32	48	64	96	128	256
Parallel Efficiency	0.9174	0.9056	0.8874	0.8466	0.8641	0.7895
Load Balance	0.9460	0.9249	0.9340	0.8584	0.8705	0.8132
Comm. Efficiency	0.9697	0.9792	0.9501	0.9863	0.9926	0.9708
Serialization	0.9699	0.9795	0.9505	0.9870	0.9937	0.9754
Transfer	0.9998	0.9997	0.9996	0.9993	0.9989	0.9953

- Even if “fairly good” numbers, it gives important indications on relevance of individual factors, coupling effects, …
- Can point to “outliers” which may be studied in detail
    - Where in the timeline? Cause ?

How to compute

- BSC tools based on traces



Communication Efficiency

Serialization

Transfer

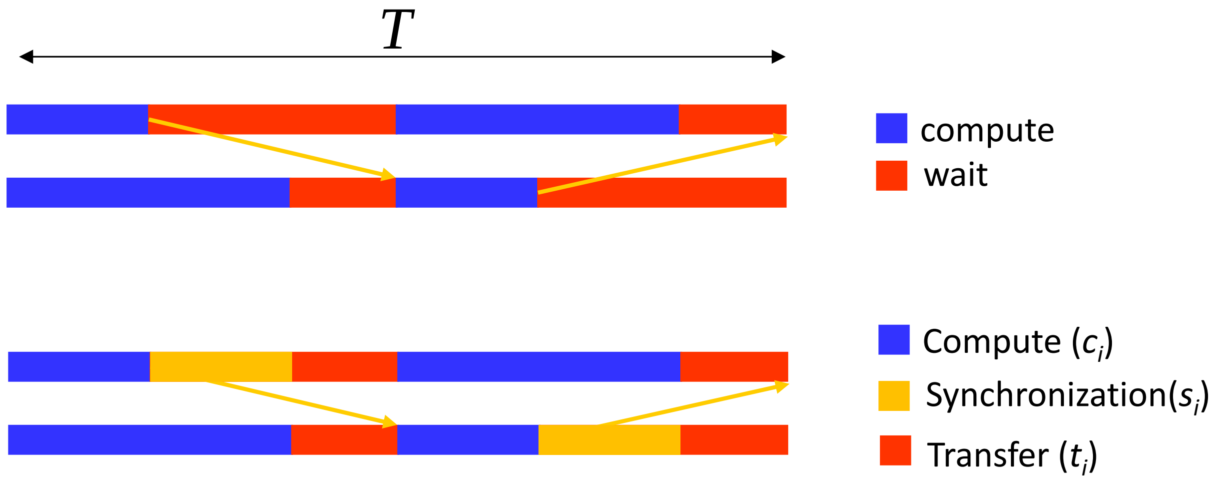
$$LB = \frac{\frac{1}{P}\sum_{i=1}^{P} c_i}{\max(c_i)}$$

$$Ser = \frac{\max(c_i)}{T_{ideal}}$$

$$Trf = \frac{T_{ideal}}{T}$$

How to compute

- Scalasca (JSC): based on traces



🟦 compute

🟥 wait

🟦 Compute (c_i)

🟨 Synchronization(s_i)

🟥 Transfer (t_i)

How to compute

- With standard profile data per process:
  - Should have precise profiling of the MPI activity

$$LB = \frac{\frac{1}{P}\sum_{i=1}^{P} c_i}{\max(c_i)}$$

```
Task  AppTime   MPITime    MPI%
 0     15.3       1.02     6.66
 1     15.3      0.293     1.91
 2     15.3      0.607     3.95
 3     15.3      0.239     1.56
 4     15.3      0.873     5.69
 5     15.3       1.01     6.58
 6     15.3      0.646     4.21
 7     15.3       1.68    10.94
 *      123       6.37     5.19
```

mpiP output

Where:

$$c_i = AppTime_i - MPITime_i$$

$$T = \max(AppTime_i)$$

  - Communication efficiency:
    - Can not separate serialization and transfer effects

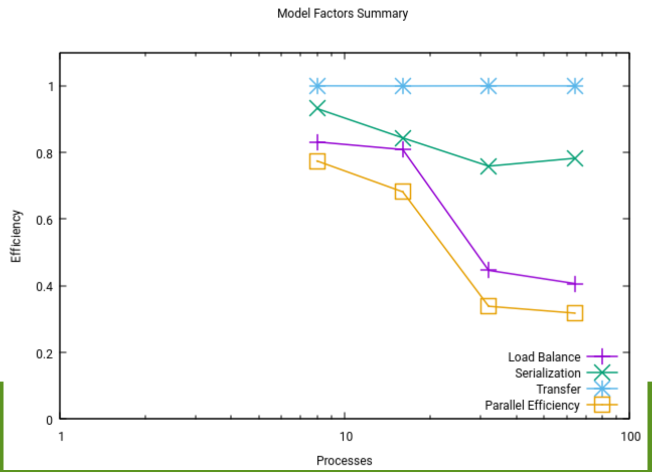
$$CommEff = \frac{\max(c_i)}{T}$$

Methodology on BSC infrastructure

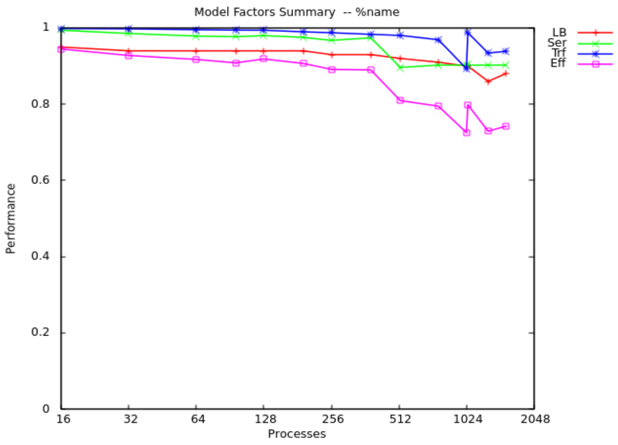
- Obtain traces
  - Extrae (https://tools.bsc.es/extrae)
- Might want to generate cuts of the “Focus Of Analysis” area
  - Paravver/paramedir (https://tools.bsc.es/paraver)
- Perform automated scaling analysis

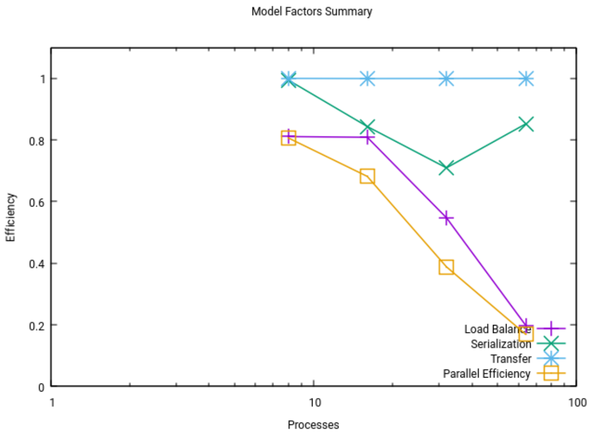
```
$model_factors.py –sc strong -t 8.prv 16.prv 32.prv 64.prv
```

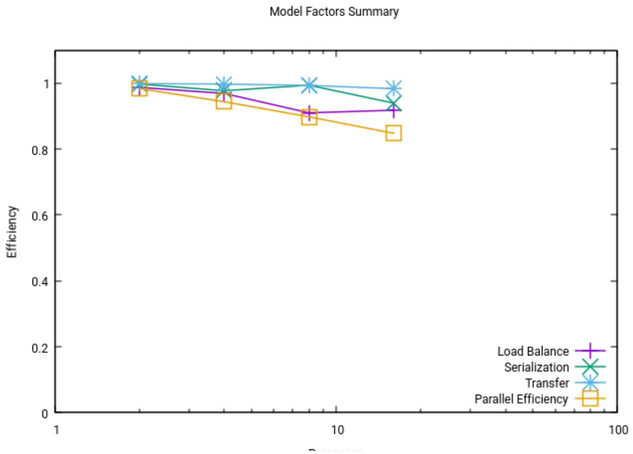
  - Generates several csv, gnuplots

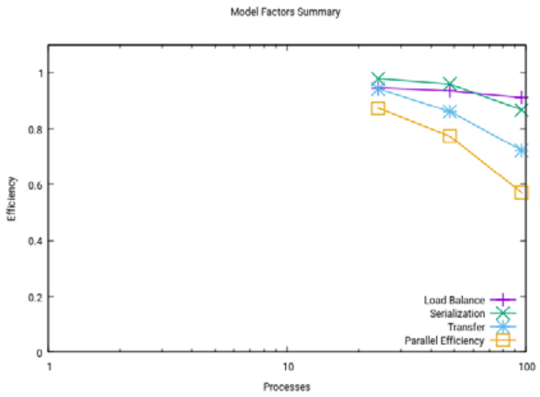


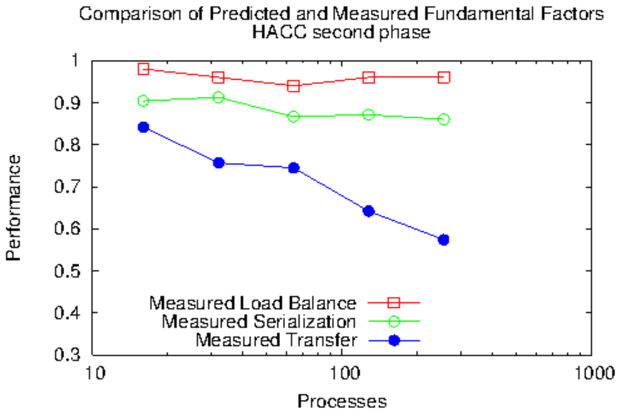
Examples

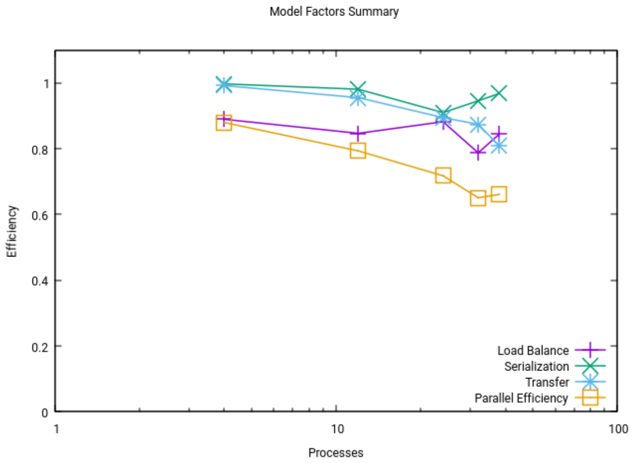

Model Factors Summary -- %name
Performance
1
0.8
0.6
0.4
0.2
0
16
32
64
128
256
512
1024
2048
Processes
LB
Ser
Trf
Eff


Model Factors Summary
Efficiency
1
0.8
0.6
0.4
0.2
0
1
10
100
Processes
Load Balance
Serialization
Transfer
Parallel Efficiency


Model Factors Summary
Efficiency
1
0.8
0.6
0.4
0.2
0
1
10
100
Load Balance
Serialization
Transfer
Parallel Efficiency


Model Factors Summary
Efficiency
1
0.8
0.6
0.4
0.2
0
1
10
100
Processes
Load Balance
Serialization
Transfer
Parallel Efficiency

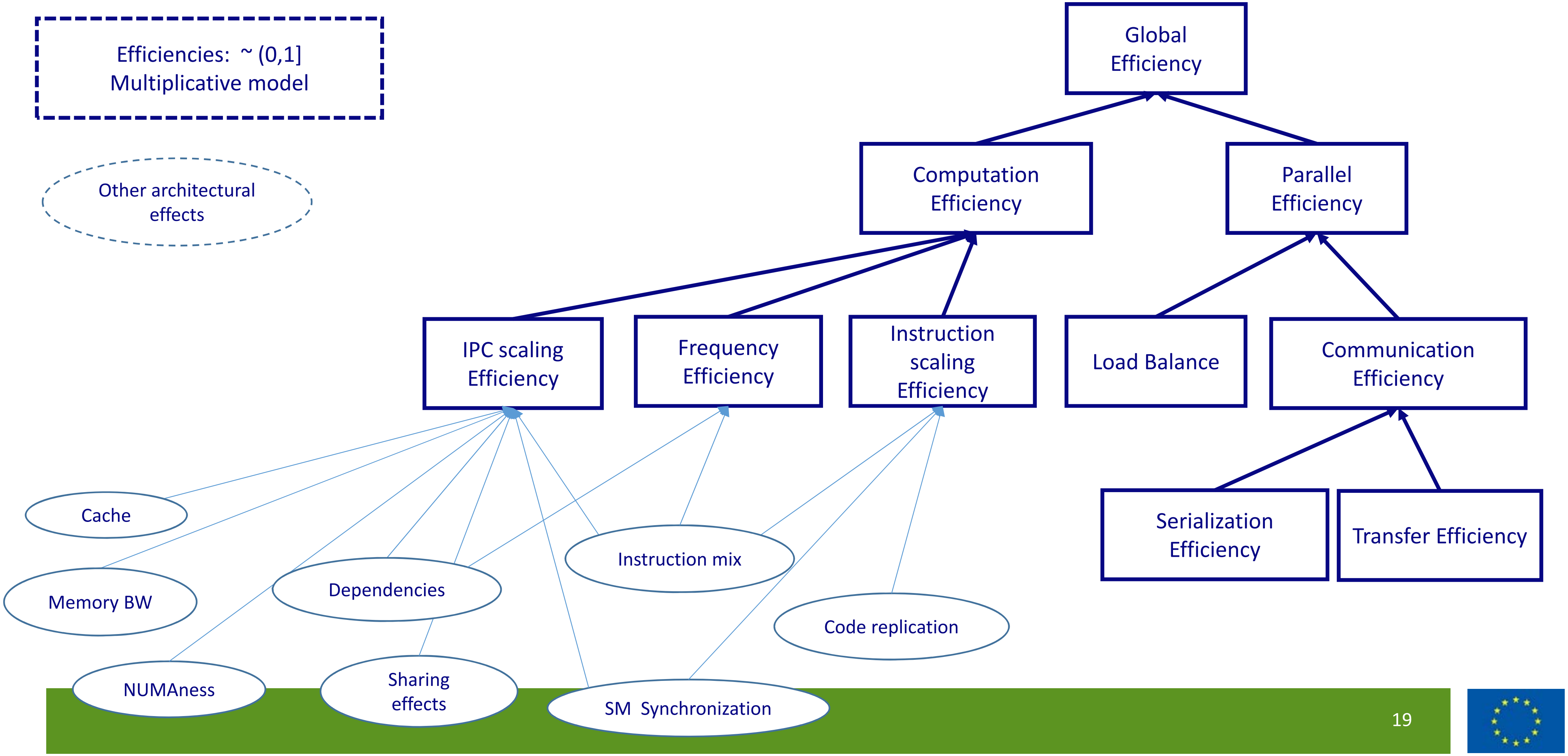

Comparison of Predicted and Measured Fundamental Factors
HACC second phase
Performance
1
0.9
0.8
0.7
0.6
0.5
0.4
0.3
10
100
1000
Processes
Measured Load Balance
Measured Serialization
Measured Transfer


Model Factors Summary
Efficiency
1
0.8
0.6
0.4
0.2
0
1
10
100
Processes
Load Balance
Serialization
Transfer
Parallel Efficiency

Interested in causes ?

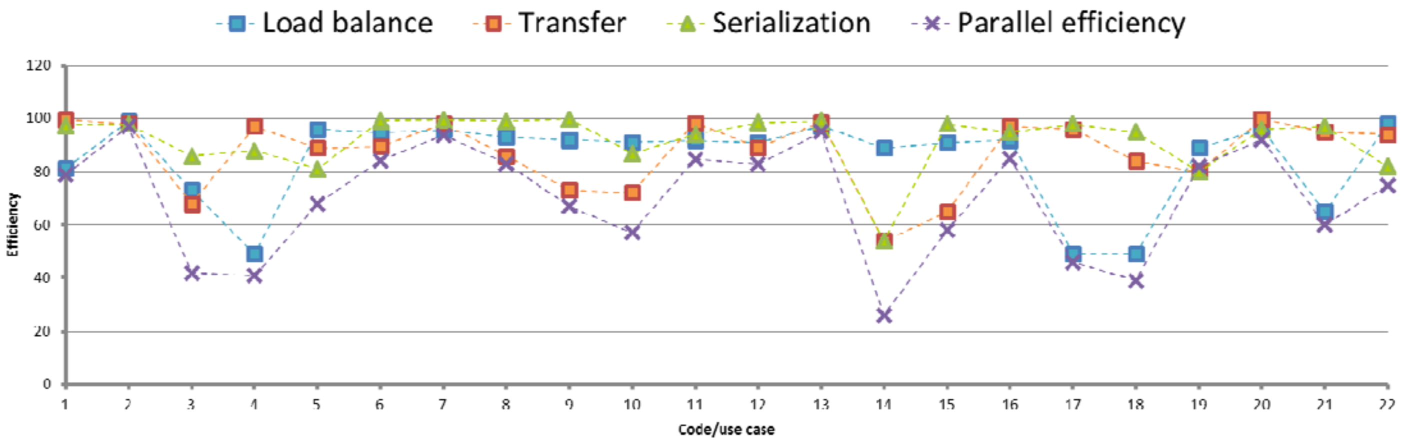
- Possible causes:
  - Load balance: work distribution, IPC (locality, NUMAness…), core frequency, …
  - Serialization: dependencies, dynamic load imbalances within multiple phases separated by synchronization, core frequency, OS scheduling issues (oversubscriptions, noise, …)
  - Transfer: actual data transfer, MPI internal implementation issues (progression engine), network contention, yield policy, OS scheduling issues
- Dig down into actual causes
  - Further Model detail to characterize application
    - Computational efficiencies
  - Detailed trace analysis

Application characterization



Interested in approaches to address

- Specific proposals for each POP customer



- Generic mechanism useful in many cases (Developed@BSC)
  - Taskified MPI + OpenMP (OmpSs) +
  - + Dynamic Load Balance library
  - + MPI+OpenMP/OmpSs interoperability library

Further material

- Follow the “Learning material” link within our web page

  https://www.pop-coe.eu

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.