Understanding Application Performance with the POP Metrics

Fabian Orland

IT Center, RWTH Aachen University
Overview

• Motivation

• POP Metrics
  – Standard POP metrics
  – Hybrid POP metrics
    ▪ Additive hybrid metrics
    ▪ Multiplicative hybrid metrics
    ▪ Critical Path based metrics

• Case Studies / Success Stories
  – CalculiX
  – juKKR

• Summary
Motivation

- **Problem:**
  - Why is my code getting inefficient at scale?
  - Multiple fundamental issues of (parallel) programming possible

- **Solution:** POP metrics
  - Standardized performance assessment independent of application / system
  - Easily verify performance improvements
Standard POP metrics

- Hierarchy of metrics
  - Aka *fundamental model factors*
  - Efficiency metrics ranging from 0 to 1
  - Child metrics multiply to yield parent metric by design

- Highlight issues in the parallel structure of an application

- Parallel Efficiency breaks down into
  - Load balance
  - Serialization
  - Transfer

- Computational Scaling captures impact of scaling to node-level performance
Standard POP metrics

Load Balance

- Reflects global imbalance of work between execution units

\[ LB = \frac{\text{avg}(\text{useful time})}{\text{max}(\text{useful time})} \]

- \textit{Useful time}: execution time outside parallel runtimes

Legend

- useful time
- parallel runtime
- load imbalance
Standard POP metrics

Serialization Efficiency

- Reflects moving imbalance of work between execution units, resp., alternating dependencies

- $\text{SerE} = \frac{\text{max(useful time)}}{\text{ideal runtime}}$

- Ideal runtime: execution time on an ideal machine with 0 communication cost (inf. BW / 0 lat)
Standard POP metrics

Transfer Efficiency

- Cost of transfer/communication/synchronization

- $TE = \frac{\text{ideal runtime}}{\text{real runtime}}$

- Real runtime: observed execution time
Standard POP metrics

Computational Scaling

- In scaling experiments, node-level performance impacts the time metrics: useful time, ideal time, real time
- Computational scaling represents this impact
- Computational scaling and its factors are normalized to a baseline
- Impact factors:
  - IPC scaling
  - Instruction scaling
  - Frequency scaling
  - (Vectorization scaling)
Hybrid POP metrics

Additive model

• Currently only support for MPI+OpenMP applications

• Extension to MPI+CUDA(+OpenMP) possible

• Advantage: detailed OpenMP analysis
  – Easily compute metrics per OpenMP region
  – Identify which region(s) are most inefficient

Details on how to compute these metrics can be found here: https://pop-coe.eu/further-information/learning-material
Understanding Application Performance with the POP Metrics

Fabian Orland
aiXcelerate 2022
5th December, 2022

Hybrid POP metrics

Multiplicative model

- Support for hybrid MPI+X applications
  - $X \in \{ \text{OpenMP, CUDA, …} \}$

- Idea:
  - Determine MPI efficiencies
  - Attribute remaining inefficiencies to paradigm X

Details on how to compute these metrics can be found here: https://pop-coe.eu/further-information/learning-material
Hybrid POP metrics

Critical path-based model

- Generalization of multiplicative hybrid metrics
  - Hybrid split of Communication Efficiency into programming models

- Idea:
  - Critical path = event path in program execution with longest duration
  - $\text{runtime}_{\text{ideal}} \approx \text{critical path of useful computation}$

- Prototype tool for "on-the-fly" calculation of hybrid metrics
  - Enables metric calculation for applications with non hierarchical communication (e.g. MPI-Detach with detached tasks)

Hybrid POP metrics

Recommendations

Hybrid code?

- no: Standard POP metrics
- yes: MPI + OpenMP?

MPI + OpenMP?

- no: Multiplicative hybrid metrics
- yes: Additive hybrid metrics
Case Studies

CalculiX - A Free Software Three-Dimensional Structural finite Element Program

- C+Fortran code parallelized with pthreads
- http://www.calculix.de/
Case Study: CalculiX

Performance Assessment

- Issue with load balance identified by POP metrics
- Computations scale super-linear
  - Probably cache effects

| POP standard metrics for region: dgmres1mt CalculiX |
|-----------------|------|------|------|------|
|                 | 2    | 4    | 6    | 8    |
| Global Efficiency | 0.8  | 0.85 | 0.88 | 0.89 |
| Parallel Efficiency | 0.8  | 0.76 | 0.75 | 0.75 |
| Load Balance      | 0.81 | 0.77 | 0.76 | 0.76 |
| Communication Efficiency | 0.99 | 0.99 | 0.99 | 0.99 |
| Serialization Efficiency | 0.99 | 0.99 | 0.99 | 0.99 |
| Transfer Efficiency | 1.0  | 1.0  | 1.0  | 1.0  |
| Computational Scaling | 1.0  | 1.12 | 1.17 | 1.18 |
| Instruction Scaling  | 1.0  | 1.06 | 1.08 | 1.08 |
| IPC Scaling        | 1.0  | 1.06 | 1.08 | 1.09 |
Case Study: CalculiX

Parallel GMRES Implementation

Structure of GMRES solver:
- Matrix vector multiplication (orange)
- Orthogonalization (pink)
- Restart after every 10 inner iterations
- Each thread solves a small independent linear subsystem

Observation:
- Thread 4 has worse convergence of its subsystem solution compared to other threads
- If only 1/8 threads is actually computing ➔ load balance issue
Case Study: CalculiX

Proof of Concept

- We extracted a single GMRES call from the application as a test kernel
- Idea: use nested parallelism to schedule additional threads on idling cores

<table>
<thead>
<tr>
<th>Time</th>
<th>Original Implementation</th>
<th>PoC Implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.00s</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.05s</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.10s</td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.50s</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.55s</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.60s</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Result:
  - Load balance improvement: 74% -> 89% [orthogonalization routine (pink) not parallelized]
  - Speedup for the kernel: 1.18x, should be 1.09x for the real application (approx. 50% of runtime is GMRES solver)
Final result

- Runtime improvement by implementing nested tasks
- We also implemented buffered I/O (not covered in this talk)
- POP metrics verify performance improvements!
Case Studies

JuKKR – The Juelich Korringa-Kohn-Rostoker DFT code family

- Fortran code parallelized with hybrid MPI+OpenMP
- https://jukkr.fz-juelich.de/
Case Study: JuKKR

aiXcelerate 2019

Before aixcelerate 2019

After aixcelerate 2019

• Load balance issue on MPI level identified
Observations:
- (+) MPI communication is almost optimal
- (+) OpenMP regions are very balanced
- (-) lot of serial code on thread level
- (-) decreasing IPC in serial calculations

Case Study: JuKKR

Performance Assessment

<table>
<thead>
<tr>
<th>Threads per Process</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>8</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>Global Efficiency</td>
<td>0.94</td>
<td>0.64</td>
<td>0.40</td>
<td>0.19</td>
<td>0.13</td>
</tr>
<tr>
<td>← Parallel Efficiency</td>
<td>0.94</td>
<td>0.76</td>
<td>0.59</td>
<td>0.44</td>
<td>0.39</td>
</tr>
<tr>
<td>← Process Level Efficiency</td>
<td>0.94</td>
<td>0.93</td>
<td>0.91</td>
<td>0.88</td>
<td>0.94</td>
</tr>
<tr>
<td>← Load balance</td>
<td>0.97</td>
<td>0.97</td>
<td>0.95</td>
<td>0.94</td>
<td>0.98</td>
</tr>
<tr>
<td>← MPI Communication Efficiency</td>
<td>0.97</td>
<td>0.96</td>
<td>0.96</td>
<td>0.94</td>
<td>0.96</td>
</tr>
<tr>
<td>← MPI Transfer Efficiency</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>← MPI Serialisation Efficiency</td>
<td>0.97</td>
<td>0.96</td>
<td>0.96</td>
<td>0.94</td>
<td>0.96</td>
</tr>
<tr>
<td>← Thread Level Efficiency</td>
<td>1.00</td>
<td>0.83</td>
<td>0.68</td>
<td>0.56</td>
<td>0.45</td>
</tr>
<tr>
<td>← OpenMP Region Efficiency</td>
<td>1.00</td>
<td>0.98</td>
<td>0.98</td>
<td>0.97</td>
<td>0.92</td>
</tr>
<tr>
<td>← Serial Region Efficiency</td>
<td>1.00</td>
<td>0.85</td>
<td>0.70</td>
<td>0.59</td>
<td>0.52</td>
</tr>
<tr>
<td>← Computational Scaling</td>
<td>1.00</td>
<td>0.84</td>
<td>0.67</td>
<td>0.44</td>
<td>0.34</td>
</tr>
<tr>
<td>← Instruction Scaling</td>
<td>1.00</td>
<td>0.97</td>
<td>0.94</td>
<td>0.90</td>
<td>0.86</td>
</tr>
<tr>
<td>← IPC Scaling</td>
<td>1.00</td>
<td>0.96</td>
<td>0.88</td>
<td>0.74</td>
<td>0.67</td>
</tr>
</tbody>
</table>

(add.) hybrid POP2 metrics for KKRhost, 16 MPI ranks á 12 OpenMP threads
K-point integration (main1b)

Performance issues:

1. 1.5 Mio. matrices $M_k \in \mathbb{R}^{32 \times 32}$
2. Lots of serial code parts
   - $\rightarrow$ Serial Region Efficiency
3. Very short OpenMP parallel regions (avg 55 $\mu$s)
4. Matrices fit into L1 cache
   - False Sharing on L1? $\rightarrow$ IPC scaling
PoC Results

Scalability - k-loop miniapp

- k-loop miniapp now scales to one NUMA domain
  - Speedup of 6x compared to original code
- Additive hybrid POP metrics verify efficiency improvements
  - Serial code parts are much smaller
  - IPC scaling is now perfect

<table>
<thead>
<tr>
<th>OpenMP threads</th>
<th>runtime-ref</th>
<th>runtime-poc</th>
<th>efficiency-ref</th>
<th>efficiency-poc</th>
<th>efficiency-ideal</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1.2</td>
<td>1.9</td>
<td>1.2</td>
<td>1.9</td>
<td>1.2</td>
</tr>
<tr>
<td>2</td>
<td>1.1</td>
<td>1.8</td>
<td>1.1</td>
<td>1.8</td>
<td>1.1</td>
</tr>
<tr>
<td>4</td>
<td>1.0</td>
<td>1.7</td>
<td>1.0</td>
<td>1.7</td>
<td>1.0</td>
</tr>
<tr>
<td>8</td>
<td>1.0</td>
<td>1.7</td>
<td>1.0</td>
<td>1.7</td>
<td>1.0</td>
</tr>
<tr>
<td>12</td>
<td>1.0</td>
<td>1.7</td>
<td>1.0</td>
<td>1.7</td>
<td>1.0</td>
</tr>
</tbody>
</table>

POP additive hybrid metrics for k-loop miniapp (PoC)

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>8</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>Global Efficiency</td>
<td>0.99</td>
<td>0.78</td>
<td>0.73</td>
<td>0.65</td>
<td>0.51</td>
</tr>
<tr>
<td>Parallel Efficiency</td>
<td>0.99</td>
<td>0.89</td>
<td>0.89</td>
<td>0.89</td>
<td>0.83</td>
</tr>
<tr>
<td>Process Level Efficiency</td>
<td>0.99</td>
<td>0.99</td>
<td>0.99</td>
<td>0.99</td>
<td>0.99</td>
</tr>
<tr>
<td>MPI Load Balance</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
</tr>
<tr>
<td>MPI Communication Efficiency</td>
<td>0.99</td>
<td>0.99</td>
<td>0.99</td>
<td>0.99</td>
<td>0.99</td>
</tr>
<tr>
<td>Thread Level Efficiency</td>
<td>1.0</td>
<td>0.89</td>
<td>0.89</td>
<td>0.89</td>
<td>0.83</td>
</tr>
<tr>
<td>OpenMP Region Efficiency</td>
<td>1.0</td>
<td>0.99</td>
<td>0.99</td>
<td>0.99</td>
<td>0.99</td>
</tr>
<tr>
<td>Serial Region Efficiency</td>
<td>1.0</td>
<td>0.89</td>
<td>0.89</td>
<td>0.89</td>
<td>0.83</td>
</tr>
<tr>
<td>Computational Scaling</td>
<td>1.0</td>
<td>0.89</td>
<td>0.82</td>
<td>0.73</td>
<td>0.62</td>
</tr>
<tr>
<td>Instruction Scaling</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
</tr>
<tr>
<td>IPC Scaling</td>
<td>1.0</td>
<td>1.0</td>
<td>1.02</td>
<td>1.02</td>
<td>1.01</td>
</tr>
<tr>
<td>Frequency Scaling</td>
<td>1.0</td>
<td>0.88</td>
<td>0.8</td>
<td>0.71</td>
<td>0.61</td>
</tr>
</tbody>
</table>
Summary

- POP metrics provide a standardized of assessing application performance
- Hierarchy of metrics allows to identify root causes of inefficiencies
- Different sets of metrics for different application requirements
  - Standard POP metrics for single paradigm applications (e.g. MPI xor OpenMP)
  - Hybrid POP metrics for multi paradigm applications (e.g. MPI+OpenMP, MPI+CUDA, …)

- POP metrics in practice (case studies)
  - CalculiX:
    - Load balance issue identified
    - ≈1.10x speedup after optimization
  - JuKKR:
    - Inefficient OpenMP parallelization of k-point integration identified
    - 6x speedup after optimization
Thank you for your attention!