Julian Kunkel

Introduction to performance engineering

Learning Objectives

- Describe basic system characteristics with typical values
- Create (strong/weak) scaling measurements and diagrams
- Utilize a basic system model to assess performance
- Sketch the system optimization cycle
- Define (strong/weak) scalability
- Describe challenges for performance analysis/optimization

Outline

Goals for HPC

HPC

- Empowers users to complete computation that needs vast compute/storage resources.

Requirements to fulfill this goal

- **Usability**: Should empower users to easily compute/store what they need
- **Programmability**: Easy to code applications for developers
- **Efficiency** is important for High-Performance Computing
  - If you obtain only 1% of efficiency, then you need 100x compute nodes!
  - Computation on 1000's of nodes is high, so efficiency is important
- **Cost-efficiency**: Cheap to compute, well-utilize bought hardware
- **Performance-portability**: Allow reuse of code between systems retaining performance
  - Also, if possible, only little code changes/tunings should be neccessary

Efficiency

- We will focus on efficiency in this talk.
- What is efficiency?

System/data center perspective

- Efficiency = Utilization of the capabilities of hardware
- We paid for the porsche, so we want to drive faster than 10 km/h
- Examples:
  - CPU/GPU utilization 100%
  - Network/storage bandwidth = 10 GBit/s, use on average 9 GBit/s
  - Memory/storage capacity 90%
- However, an applications may not need much of a single resource ...

Efficiency (2)

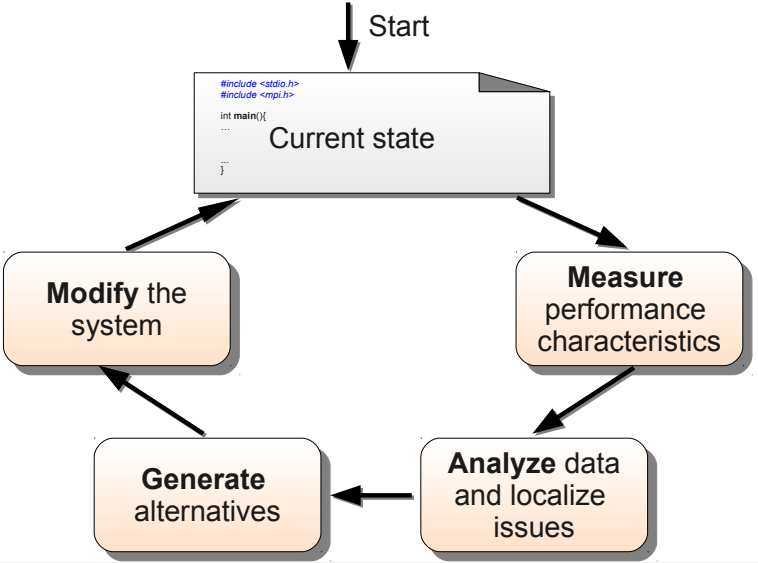
User perspective

- User perception: Execution time of the application should be low
- Users often do not care about system efficiency
- Using 10x nodes/cores should lead to 1/10th of runtime
- Running with 10x input size and 10x compute nodes should lead to same execution time
- If performance isn't sufficient for a science use case, optimize application/system

Performance Engineering

- Definition: Process of analyzing/understanding and optimizing applications
- Requires good understanding of system and application behavior
- Tools and models can help users, centers offer help, too

Optimization Cycle (for Any System)



Understanding of Application and System-Behavior

How can we understand system behavior?

- Theory: Performance models
  - Models - Determine performance for a system or workload
  - Behavioral models - build models based on ensemble of observations
  - **System characteristics** are a basic model of system capabilities
- Observation
  - Measure runs on the system - note measurements perturb behavior
  - Benchmarking: specific applications geared to exhibit certain system behavior
  - Tracing: record relevant operations of the application with their timing
  - Profiling: record operations of the application and create statistics
- Monitoring: system/tool-provided creation of observations
- System/application simulation
  - Based on system and workload models

Code Optimization

Alternatives/Options

1. Run code on a more suitable system - e.g., faster, more memory, different CPUs
2. Tune execution without changing code
3. Increase efficiency by modifying code - this is complex...

Tuning

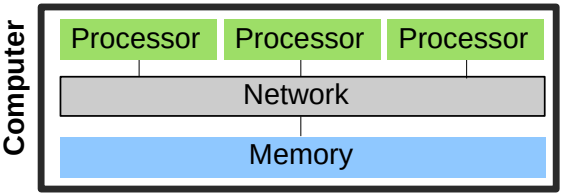
- Definition: Process of analyzing and optimizing system parameters without changing code
- Examples: Compiler options, system settings, changing tunable parameters...
- Any user should have a basic understanding of systems

Outline

Reminder: Parallel & Distributed Architectures

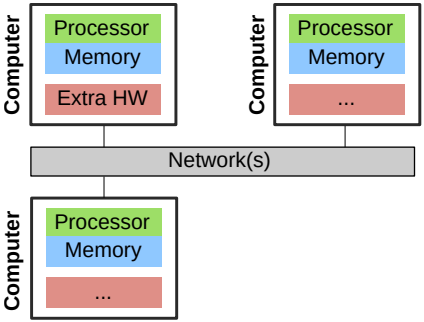
In practice, systems are a mix of two paradigms:

Shared memory



- Processors can access a joint memory
  - Enables communication/coordination
- Cannot be scaled up to any size
- Very expensive to build one big system

Distributed memory systems



- Processor can only see own memory
- Performance of the network is key

Hardware Performance

Computation

- CPU performance (frequency × cores × sockets)
  - E.g.: 2.5 GHz × 12 cores × 2 sockets = 60 Gcycles/s
  - The number of cycles per operation depend on the instruction stream
- Memory (throughput × channels) + latency per access
  - E.g.: 25.6 GB/s per DDR4 DIMM × 3 – L1/L2/L3 caches are somewhat relevant

Communication via the network

- Throughput, e.g., 125 MiB/s with Gigabit Ethernet
- Latency, e.g., 0.1 ms with Gigabit Ethernet

Input/output devices

- HDD mechanical parts (head, rotation) lead to expensive seek

⇒ Access data consecutively and not randomly

⇒ Performance depends on the I/O granularity

- E.g.: 150 MiB/s with 10 MiB blocks

Benchmark for Memory Throughput

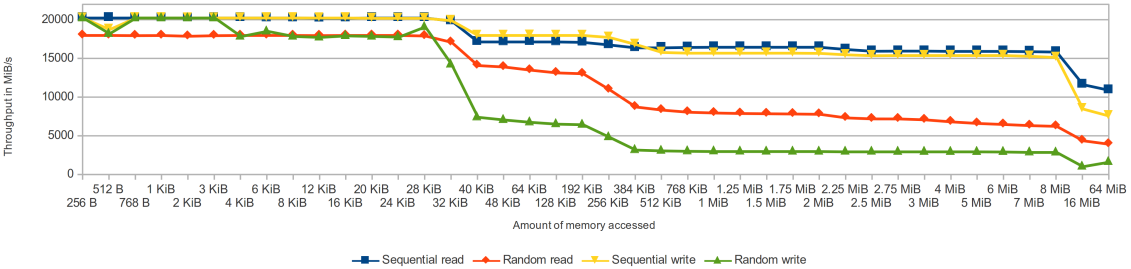


Figure: Memory performance using the fbui benchmark (on an older system)

Performance Varies!

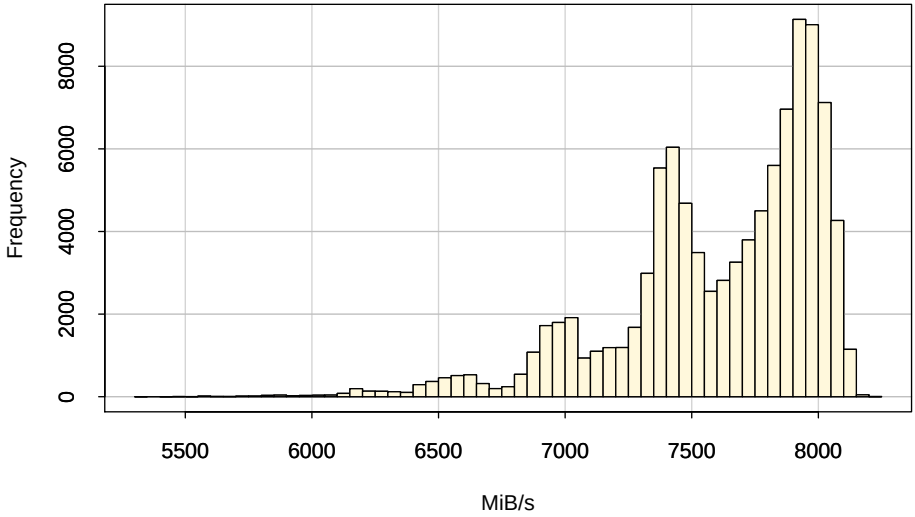


Figure: Histogram for many (identical) 64 Byte accesses

Limitations

- Performance of any parallel application is bound by a ressource
  - Compute, Memory, Network
- Application profiles decide if the app is **compute/network/memory/IO bound**
  - Application demand specific resources more than others
    E.g., the communication is optimized
  - Even within a single compute core, apps utilize ALU differently
    The instruction mix differs...
- Let's first look at a single process and compute node
  - Apps are often memory or compute bound, this can be somewhat easily analyzed
    - Aim: Identify which part of code we must optimize, or shall we move to a different system?
  - We can compute **or** measure memory traffic (Q) and (arithmetic) work (W)
  - Operational intensity $I = \frac{W}{Q}$ is the number of ops per byte memory traffic
    - Often: use number of FLOP (floating point ops) as work

Roofline Model: Illustates Memory and Compute Limitations

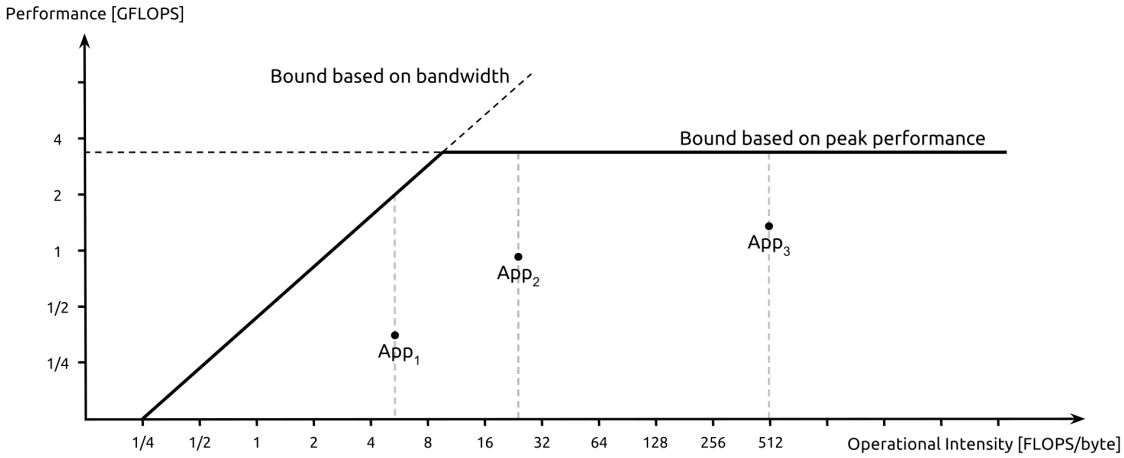
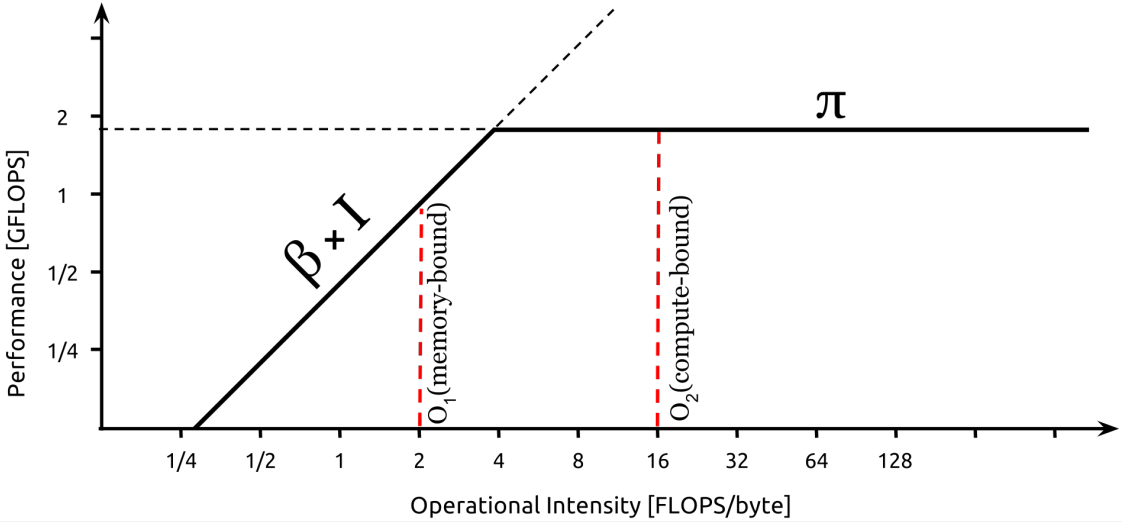


Figure: Giu.natale / Wikipedia

Roofline Model: Naive Model



HPC Cluster Characteristics

- High-end components
- Extra fast interconnect, global/shared storage with dedicated servers
- Network provides high (near-full) bisection bandwidth. Various topologies are possible.

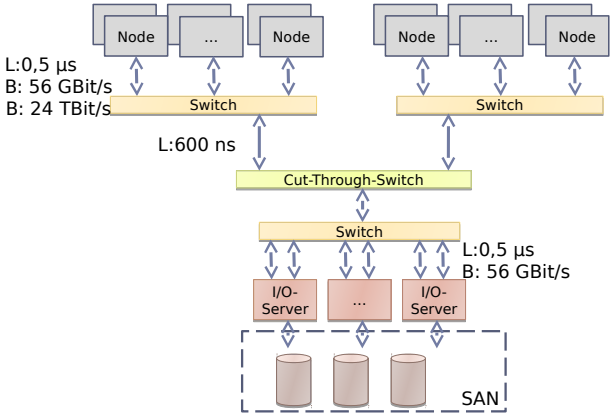


Figure: Architecture of a typical HPC cluster (here fat-tree network topology)

Big Data Cluster Characteristics

- Usually commodity components
- Cheap (on-board) interconnect, node-local storage
- Communication (bisection) bandwidth between different racks is low

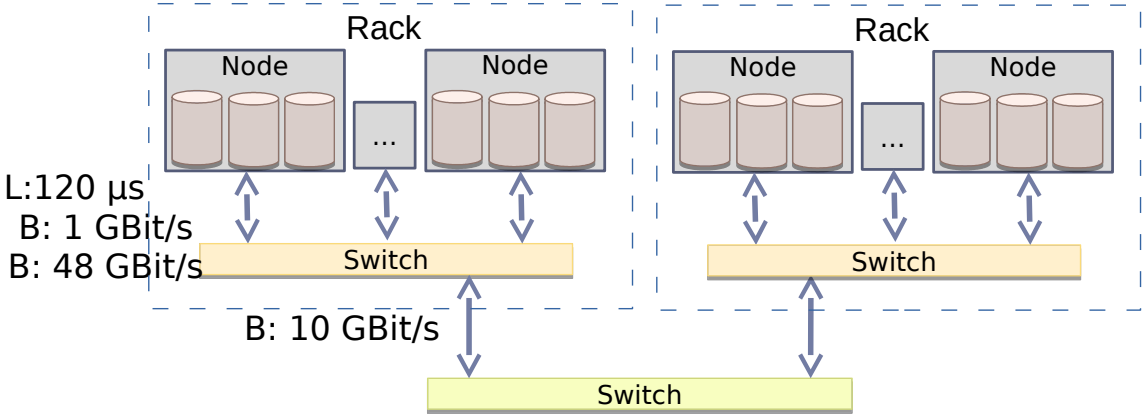


Figure: Architecture of a typical big data cluster

Impact of Software

- Efficiency of programming languages: Java needs 1.2x - 2x of cycles compared to C[1]
- All hardware components should be utilized concurrently, i.e., asynchronous
  - Pipeline computation, I/O, and communication
  - At best hide two of them $\Rightarrow$ 3x speedup vs sequential
  - Asynchronous (ni barriers) – avoid waiting for the slowest component
- Balance and distribute workload among all processes
  i.e., 10 processes, each should compute 10% of the work and finish at the same time
  - Slowest process determines performance
    - Q: if slowest process computes 10% of work, how fast can you be?
  - If only 1 works you cannot benefit from parallelism
- Avoid I/O, if possible (keep data in memory)
- Avoid communication and memory access, if possible

[1] This does not matter much compared to the other factors. But vectorisation matters.

Outline

Amdahl and Speedup

- Amdahl: fraction of parallelizable code
- Speedup is bound by $S = \frac{1}{1-p}$
- p is the proportion of parallelizable code
- Assumption: infinite resources

Speedup

- How much faster is the parallel program?
- Definition: time parallel / time sequential
  Speedup of 1 == as fast as sequential
- Determine speedup by running app
  - Vary parallelism = # procs/threads

Efficiency

- Definition: Speedup / Parallelism
- 100% Efficiency means perfect speedup

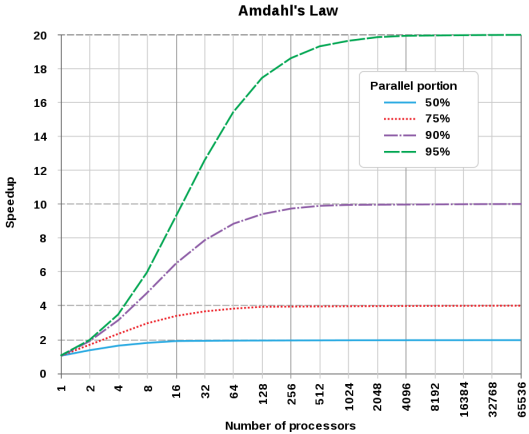


Figure: Source: Daniels220, Wikipedia

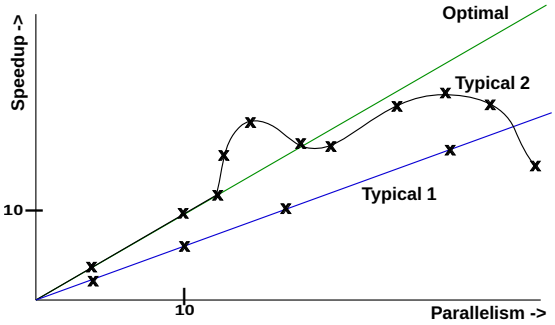
Strong Scaling

- Situation: Keeping the problem size, increase parallelism
- Example: Compute 10 days of weather forecast on 1 node, then on 10 nodes
- Optimal result: 10x resources

⇒ 1/10th of runtime

- Naturally, there is a limit as work cannot be distributed infinitely

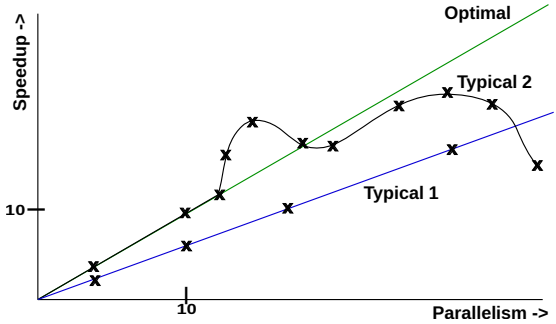
- Two examples of speedup curves
- X mark the measured points
- Note: Typical* are more similar than expected



Groupwork: Assessing Speedup Diagrams

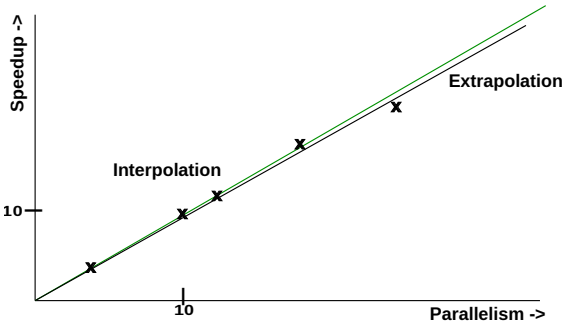
Task: Assess the two strong scaling curves Typical1 and Typical2

- Are the measurements of T1/T2 good?
- T2: What could be the cause for the observed performance changes?
- Is there any relationship between the shape of T1 and T2?
- Time: 5 minutes

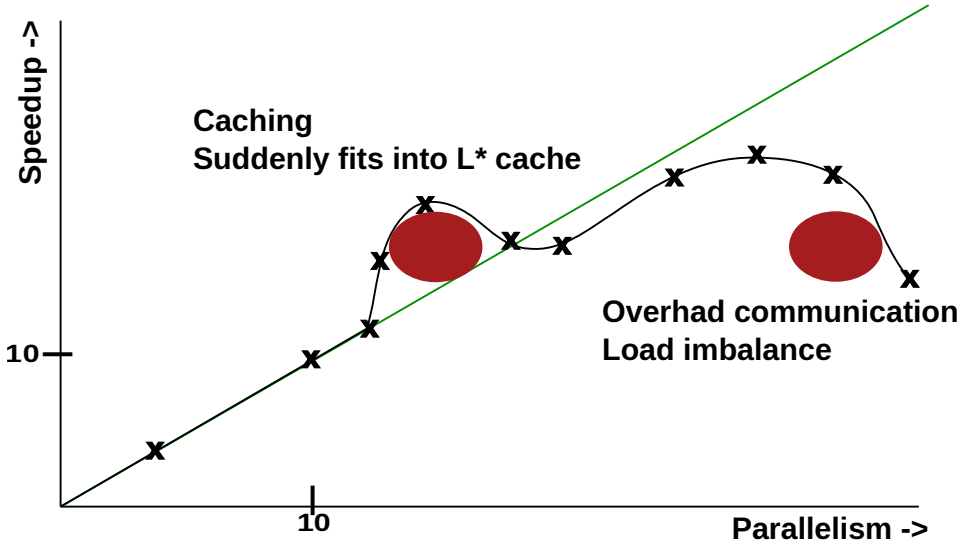


(Self-) Cheating

- Measuring less points
  Typical1 can look like Typical2
- Interpolate the remaining points
- Some people show how their app scales
  - Be careful with the assessment
- *Speedups* $>$ *Parallelism* is suspicious
  - Means efficiency $>$ 100%

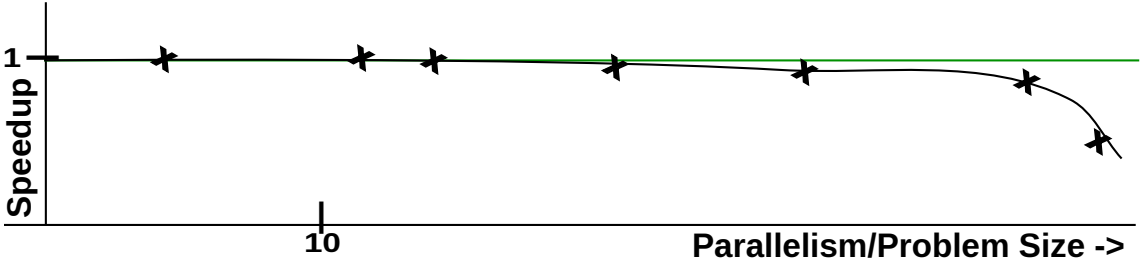


Causes for Observation



Weak Scaling

- Situation: Increase the problem size with parallelism
  - This can scale to large configs as the amount of work per processor stays the same
- Example: 10x number of nodes, 10x problem size
- Optimal result: runtime stays the same



Outline

Our Basic Network Model and Observations

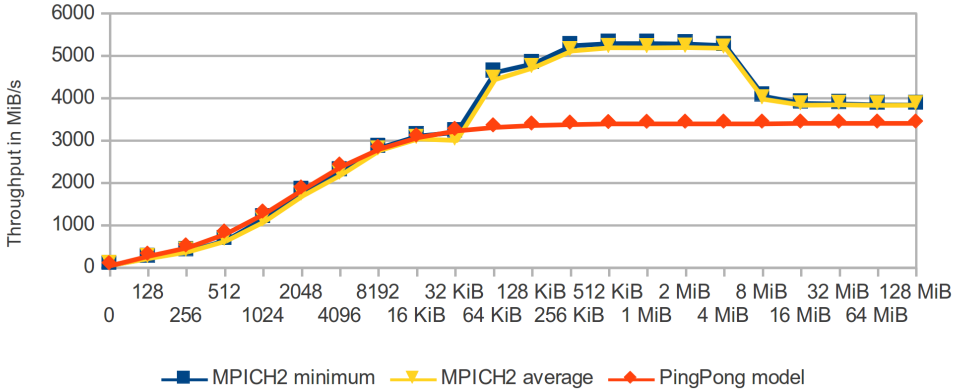


Figure: Inter-socket Communication

- Utilizing the basic hardware model: latency + throughput

Collective Algorithms: Simulated and Measurements

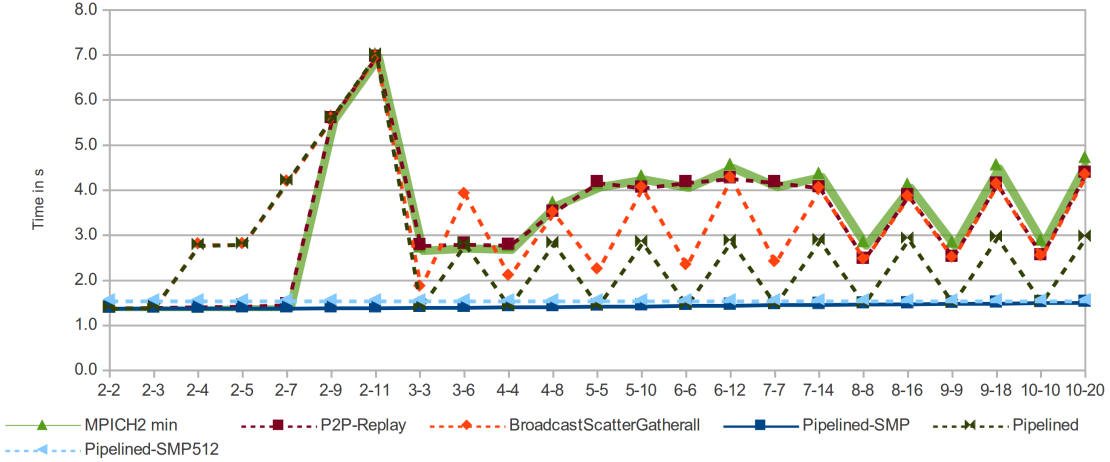


Figure: `MPI_Bcast()`, 100 MiB Data, Inter-Node Communication (1), X-Axis (Nodes-Procs)

Strategy

Guiding question

Is the observed performance *acceptable*?

- My observation: often a simple approximative model is sufficient
  - Knowing that something is 100x slower than it should be...
- You must understand the basic architecture of the software system
- You must understand most important hardware characteristics
- Advice
  - Start with simple models for workload and hardware performance
  - Refine the model as needed, e.g., include details about intermediate steps

Approximation – Simple Example on Computation

Example: Summing up data in an array of 10M ints

- Workload: 10M integers
- System: 3.7 GHz PC
- Python (for loop): 0.39s = 98 MB/s, 144 cycles per op
  $(10 \cdot 1000 \cdot 1000) \cdot 4$ bytes $/$ $0.39s = 98MiB/s$
  $3700 \cdot 1000 \cdot 1000 cycles \cdot 0.39s / (10 \cdot 1000 \cdot 1000 op) = 144$ cycles/op
- Numpy: 0.0055s, 7000 MB/s, 2 cycles per op
- Python (sum up numbers): 0.14s, 272 MB/s, 52 cycles per op
- One line to measure the performance in Python using Numpy:

```
1    timeit.timeit(stmt="np.sum(d)", setup="import numpy as np; d =
         ↪ np.array(range(1,10*1000*1000))", number=1)
2    # Just sum up numbers: sum(range(1,10*1000*1000))
```

Methodology

1. Measure time for the execution of your workload
2. Quantify the workload with some metrics
   - E.g., amount of tuples or data processed, computational operations needed
   - E.g., you may use the statistics output for each Hadoop job
3. Compute W, the workload you process per time
4. Compute expected performance P based on system's hardware characteristics
5. Compare W with P, the efficiency is $E = \frac{W}{P}$
   - If $E << 1$, e.g., 0.01, you are using only 1% of the potential!

Example: Object Storage

Scenario: Accessing data on object storage

1. Time: 0.1s (3x measured, between 0.09 and 0.11s)
2. Workload: 100 MiB of data fetched from object storage
3. $W = 100MiB/0.1s = 1000MiB/s$
4. System: Client and server are interconnected via a 100 GbE network
   Characteristics: $P = 12,500GiB/s$ throughput
   Latency doesn't matter for large files
5. Efficiency: $E = 1,000/12,500 = 8\%$

For a 10 GbE interconnect, 80% efficiency would have been achieved!

Groupwork: Assessing Performance (Compute Only)

Task: Aggregating 10 Million integers with 1 thread/process

- Vendor-reported performance from [14] indicates improvements

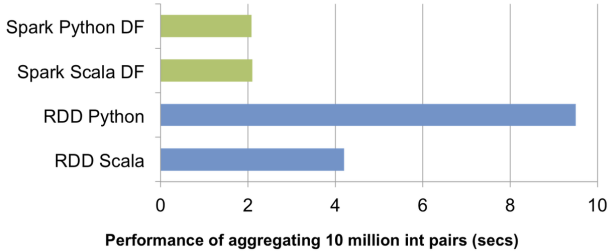


Figure: Source: Reference [14]

- These are the advancements when using “Spark DF” instead of “RDD”
- Can we trust in such numbers? Are these numbers good?
- Discuss these numbers in your group (Time: 5 minutes)

Assessing Performance of In-Memory Computing

Measured performance numbers and theoretic considerations

- Spark [14]: 160 MB/s, 500 cycles per operation[2]
  - Invoking external programming languages is even more expensive!
- Python (raw): 0.44s = 727 MB/s, 123 cycles per operation
- Numpy: 0.014s = 22.8 GB/s, 4 cycles per operation (memory BW limit)
- One line to measure the performance in Python using Numpy:

```
1  timeit.timeit(stmt="np.sum(d)", setup="import numpy as np; d =
       ↪ np.array(range(1,10*1000*1000))", number=1)
```

- Hence, the big data solution is 125x slower in this example than expected!

Assessing Compute and Storage Workflow

- Daytona GraySort: Sort at least 100 TB data in files into an output file
  - Generates 500 TB of disk I/O and 200 TB of network I/O [12]
  - Drawback: Benchmark is not very compute intense
- Data record: 10 byte key, 90 byte data
- Performance Metric: Sort rate (TBs/minute)

	Hadoop MR Record	Spark Record	Spark 1 PB
Data Size	102.5 TB	100 TB	1000 TB
Elapsed Time	72 mins	23 mins	234 mins
# Nodes	2100	206	190
# Cores	50400 physical	6592 virtualized	6080 virtualized
Cluster disk throughput	3150 GB/s (est.)	618 GB/s	570 GB/s
Sort Benchmark Daytona Rules	Yes	Yes	No
Network	dedicated data center, 10Gbps	virtualized (EC2) 10Gbps network	virtualized (EC2) 10Gbps network
Sort rate	**1.42 TB/min**	**4.27 TB/min**	**4.27 TB/min**
Sort rate/node	**0.67 GB/min**	**20.7 GB/min**	**22.5 GB/min**

Figure: Source: Reference [12]

Assessing Performance of In-Memory Computing

Hadoop

- 102.5 TB in 4,328 seconds [13]
- Hardware: 2100 nodes, dual 2.3Ghz 6cores, 64 GB memory, 12 HDDs
- Sort rate: 23.6 GB/s = 11 MB/s per Node $\Rightarrow$ 1 MB/s per HDD
- Clearly this is suboptimal!

Apache Spark (on disk)

- 100 TB in 1,406 seconds [13]
- Hardware: 207 Amazon EC2, 2.5Ghz 32vCores, 244GB memory, 8 SSDs
- Sort rate: 71 GB/s = 344 MB/s per node
- Performance assessment
  - Network: 200 TB $\Rightarrow$ 687 MiB/s per node
    Optimal: 1.15 GB/s per Node, but we cannot hide (all) communication
  - I/O: 500 TB $\Rightarrow$ 1.7 GB/s per node = 212 MB/s per SSD
  - Compute: 17 M records/s per node = 0.5 M/s per core = 4700 cycles/record

Executing the Optimal Algorithm on Given Hardware

An utopic algorihm

Assume 200 nodes and well known key distribution

1. Read input file once: 100 TB
2. Pipeline reading and start immediately to scatter data (key): 100 TB
3. Receiving node stores data in likely memory region: 500 GB/node
   Assume this can be pipelined with the receiver
4. Output data to local files: 100 TB

Estimating optimal runtime

Per node: 500 GByte of data; I/O: keep 1.7 GB/s per node

1. Read: 294s
2. Scatter data: 434s $\Rightarrow$ Reading can be hidden
3. One read/write in memory (2 sockets, 3 channels): 6s
4. Write local file region: 294s

Total runtime: $434 + 294 = 728 \Rightarrow$ 8.2 T/min $\Rightarrow$ The Spark record is quite good!

Outline

Understanding of Application and System-Behavior

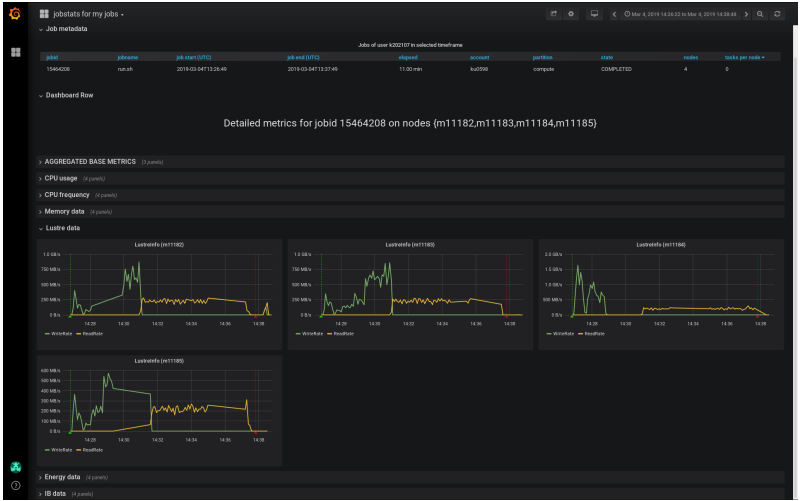
How can we understand system behavior?

- Theory: Performance models
- System/application simulation
- **Observation**
- **Monitoring**: system/tool-provided creation of observations

Observation and monitoring of behavior

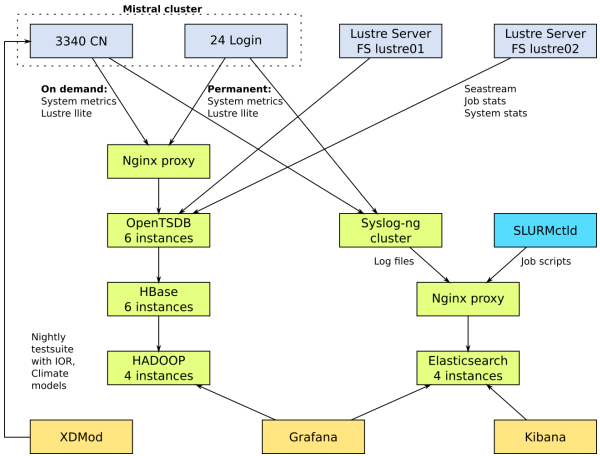
- System-level, i.e., observable statistics such as CPU utilization, bytes read
- Application-level, record individual operations performance
- There are many interesting metrics that can be recorded
- Many tools exists that aid this analysis

System-Wide Monitoring



- Center tools
  - Example: Grafana
- Various metrics for
  - Compute
  - Network
  - I/O
- Here: Focus I/O

DKRZ Monitoring System



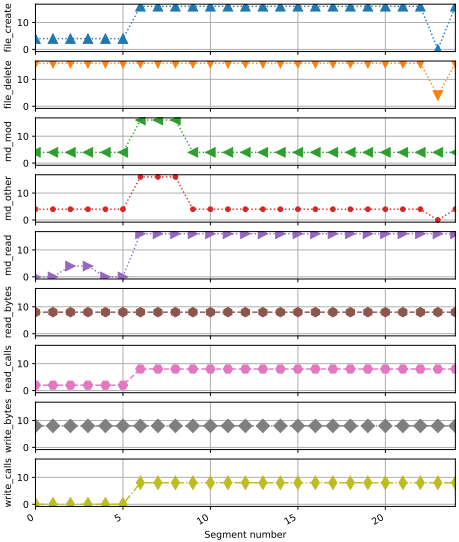
Details

- Periodicity: 10s
- Record metrics
  - From /proc
- Jobs are linked to the data

Mistral Supercomputer

- 3,340 Nodes
- 2 Lustre file systems
- 52 PByte capacity
- 100+ OSTs per fs

Visualizing Job Behavior and Comparing different jobs



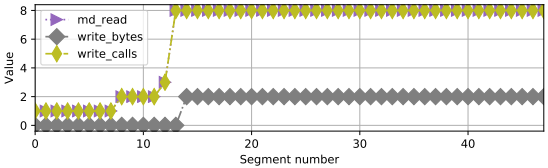
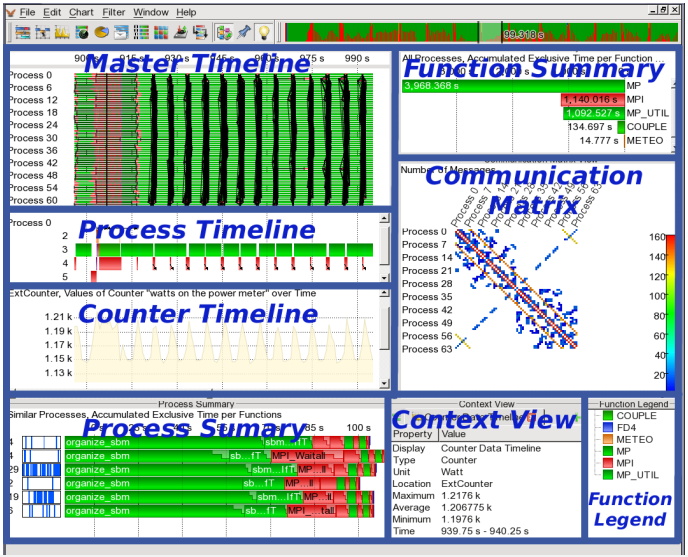


Figure: For this job, other metrics == 0

- Different jobs differ significantly
- We can compare jobs based on metrics
- Example:
  - I/O metrics
  - Segments represent 10 min

Vampir: Analyzing Application Performance



Outline

How Can Benchmarks Help to Analyze Systems and Workloads?

- A benchmark exhibits a specific behavior in order to analyze it
  More information in the next talk!
- Benefits of benchmarks
  - Can use simple/understandable sequence of operations
    - Ease comparison with theoretic values (that requires understandable metrics)
  - May use a pattern like a realistic workloads
    - Provides performance estimates or bounds for workloads!
  - Sometimes only possibility to understand hardware capabilities
    - Because the theoretic analysis may be infeasible
- Benefits of benchmarks vs. applications
  - Are easier to code/understand/setup/run than applications
  - Come with less restrictive "license" limitations
- Flexible testing (strategies)
  - Single-shot: e.g., acceptance test
  - Periodically: regression tests

Benchmarks

- Benchmarks measure system behavior and implement (simple) well-known behavior
- Many benchmarks exist covering various aspects of the system
  - Low-level hardware: CPU, Memory, Storage
  - Software: MPI
  - Application kernels: Linpack, HPCC
  - Mini-apps representing application behavior
- Might be synthetic or inspired by a real workload

Predictability and Latency Matters

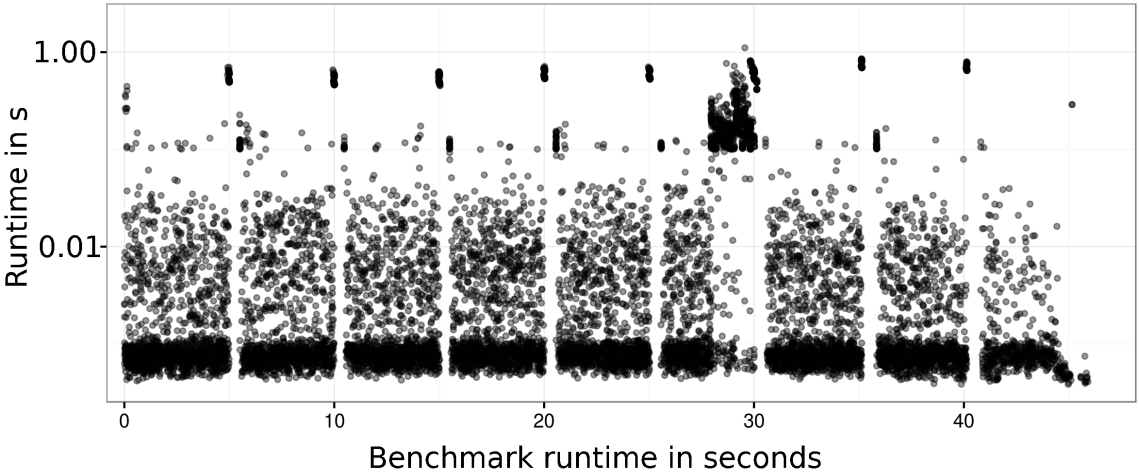
Performance Predictability

- How long does an I/O / metadata operation take?
- Important to predict runtime
- Important for bulk-synchronous parallel applications
  - The slowest straggler defines the performance

Measurement

- In the following, we plot the timelines of metadata create operations
  - Sparse plot with randomly selected measurements
  - Every point above 0.1s is added
- All results obtained on 10 Nodes using MD-Workbench
  `https://github.com/JulianKunkel/md-workbench`
  - Options: 10 PPN, D=1, I=2000, P=10k, precreation phase

Latencies: Lustre / Mistral at DKRZ



Probing Approach

- Many sites run periodic regression tests, e.g., nightly
  - Helps to identify performance regressions with updates
- Instead, we run a non-invasive benchmark (a probe) with a high frequency
  - Mimic the user-visible client behavior
  - Measuring latency for metadata and data operations
- Generate and analyze generated statistics
- Derive a slowdown factor (file system load)

Probing: Performance Measurement

Preparation

- Data: Generate a large file (e.g., $>$ 4x main memory of the client)
- Metadata: Pre-create a large pool of small files (e.g., 100k+ files)

Benchmarks

- Repeat the execution of the two patterns every second
- DD: Read/Write a random 1 MB block
- MD-Workbench: stat, read, delete, write a single file per iteration
  - Allows regression testing, i.e., retain the number of files
  - *J. Kunkel, G. Markomanolis. Understanding Metadata Latency with MDWorkbench.*

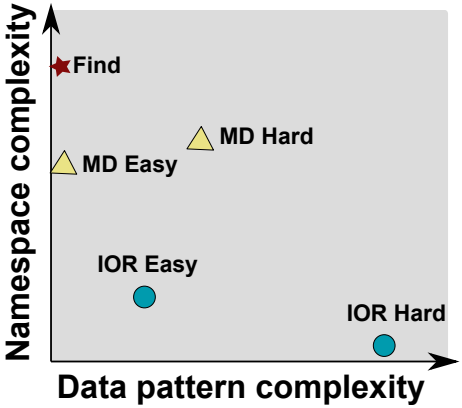
Executed as Bash script or an integrated tool: `https://github.com/joobog/io-probing`

Goals of the IO-500 Benchmarking Effort

- Bound performance expectations for realistic workloads
- Track storage system characteristics behavior over the years
  - Foster understanding of storage performance development
  - Support to identify potent architectures for certain workloads
- Document and share best practices
  - Tuning of the system is encouraged
  - Submitters must submit detailed run parameters
- Support procurements, administrators and users

https://io500.org

Covered Access Patterns



- IOR-easy: large seq on file(s)
- IOR-hard: small random shared file
- MD-easy: mdtest, per rank dir, empty files
- MD-hard: mdtest, shared dir, 3900 byte
- find: query and filter files based on name and creation time
- Executing concurrent patterns not covered (another dimension)

Probing Response Time on Archer when Running IO-500

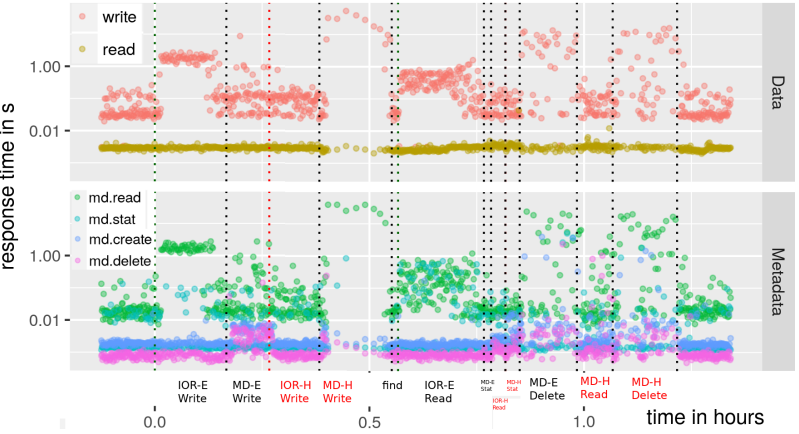


Figure: Response time (all measurements)

- Run on 100 nodes score 8.45
- The IO-500 various phases Data and metadata heavy
- First, all measurements

Validating Slowdown on All Measurements

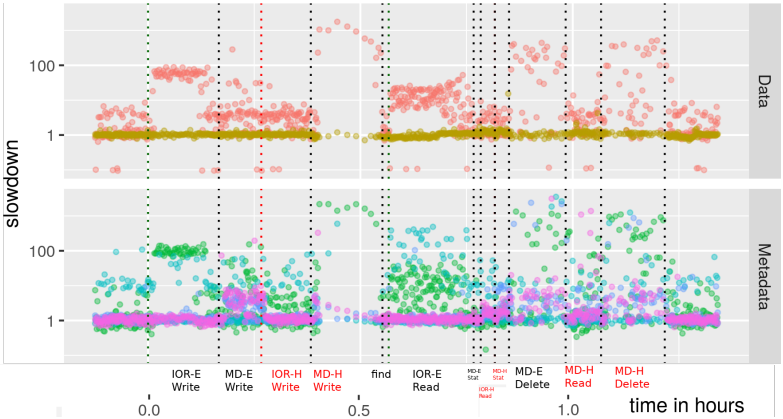


Figure: Slowdown (all measurements)

- Computed median slowdown
  Expected: median of 30 days
- Influence of phases is visible
- MDHard 1000x slowdown
  Influences data latency!
  10s of seconds latency
- IOREasy 100x slowdown
- IORHard not too much
- Data read is stable

Slowdown for 4h Statistics

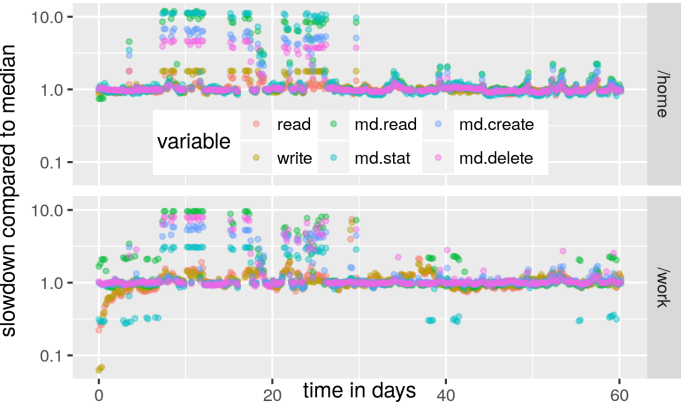


Figure: JASMIN, computed on 4 hour intervals

- Slowdown: Using the median
- Typically value is 1
- Sometimes a system is 10x slower
  - Due to user interactions
  - Concurrent application execution
- Values below 1, unusual (caching)
- Performance can vary significantly!

Summary

- Performance
  - Goal (user-perspective): Optimise the time-to-solution
  - Understanding a few HW throughputs help to assess the performance
  - Linear scalability of the architecture is the crucial performance factor
  - Basic performance analysis
    1. Estimate the workload
    2. Compute the workload throughput per node
    3. Compare with hardware capabilities
- Achieving performance is challenging due to
  - complex systems, deep software stack, performance variability, optimizations
- Monitoring, performance analysis and benchmarking is needed
- We will analyze HPC applications using some of the techniques introduced