ArmoniK: Simplifying Access to Performance at Scale

DP2E-AI 2025

Aneo

Jérôme Gurhem

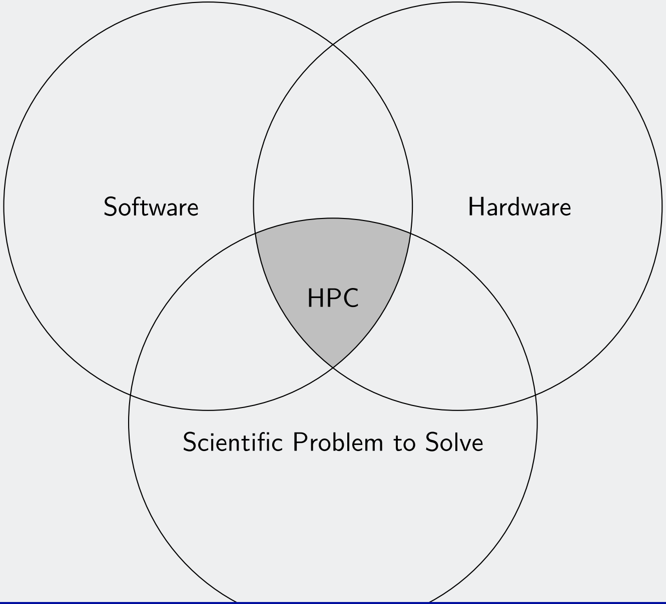
June 23, 2025

Outline

Section 1

Can We Simplify Access to Performance at Scale?

HPC: At the Cutting Edge of Three Disciplines



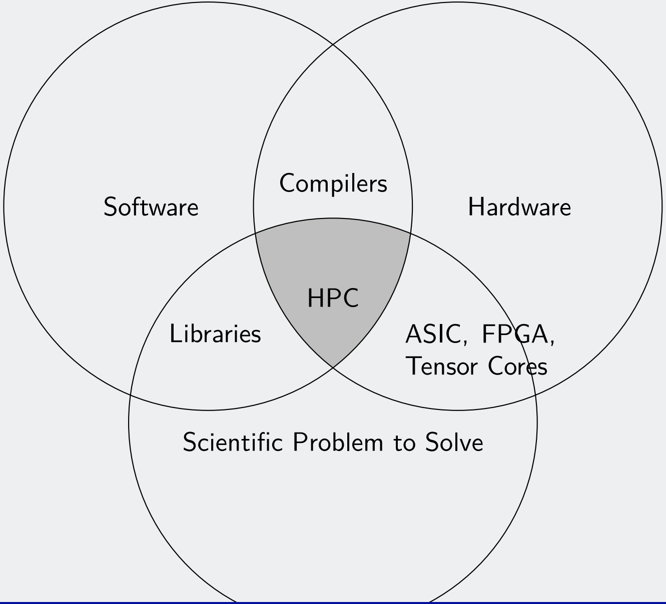
The Decline of the Scientific Polymath

- **17th–18th centuries:** Scientists like Newton and Leibniz contributed across disciplines — science was relatively unified.
- **19th century:** Rapid growth of disciplines (e.g., chemistry, biology, physics); first signs of specialization.
- **20th century:** Explosion of subfields and technical complexity; solo mastery becomes impossible.
- **Today:** Even within a single field (e.g., AI, genetics), experts specialize narrowly — cross-field comprehension is rare.

Result

- No one scientist can read or understand all scientific publications.
- Applicable to HPC : It is increasingly difficult to master all its fields.

Building Interfaces Between Disciplines



Current Interfaces

- **Hardware/Software**
  - **Programming Languages:** C, C++, Python, Fortran, etc.
  - **Compilers:** GCC, Intel, LLVM, etc.
- **Software/Scientific Problem**
  - **Libraries:** BLAS, LAPACK, FFTW, TensorFlow, PyTorch, etc.
  - **Frameworks:** MPI, OpenMP, CUDA, OpenCL, Kokkos, etc.
- **Hardware/Scientific Problem**
  - **AI:** AWS Inferentia, Google TPU, Nvidia Tensor Cores, etc.
  - **Dedicated Accelerators:** FPGAs, ASICs, etc.

Current Interfaces

- **Hardware/Software**
  - **Programming Languages:** C, C++, Python, Fortran, etc.
  - **Compilers:** GCC, Intel, LLVM, etc.
- **Software/Scientific Problem**
  - **Libraries:** BLAS, LAPACK, FFTW, TensorFlow, PyTorch, etc.
  - **Frameworks:** MPI, OpenMP, CUDA, OpenCL, Kokkos, etc.
- **Hardware/Scientific Problem**
  - **AI:** AWS Inferentia, Google TPU, Nvidia Tensor Cores, etc.
  - **Dedicated Accelerators:** FPGAs, ASICs, etc.

Are they good enough?

- Yes ! For the trade-off they were designed for.
- Still, they need deep expertise to be used effectively.

Why Look Beyond Current Interfaces?

- Complexity of heterogeneous systems is growing rapidly.
- Experts are required to bridge hardware/software/application gaps.
- Scientific applications need:
  - Scalability without re-architecting code.
  - Transparent access to diverse computing resources.
  - Efficient resource utilization and orchestration.
- Current solutions lack:
  - Flexibility across architectures.
  - Seamless integration and automation.
  - User-friendly interfaces for scientists and engineers.

Why Look Beyond Current Interfaces?

- Complexity of heterogeneous systems is growing rapidly.
- Experts are required to bridge hardware/software/application gaps.
- Scientific applications need:
  - Scalability without re-architecting code.
  - Transparent access to diverse computing resources.
  - Efficient resource utilization and orchestration.
- Current solutions lack:
  - Flexibility across architectures.
  - Seamless integration and automation.
  - User-friendly interfaces for scientists and engineers.

The Question

Can we simplify access to performance at scale?

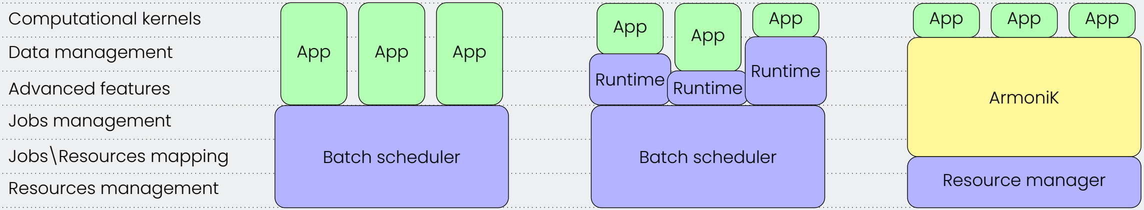
Section 2

ArmoniK: A User-Friendly Trade-Off

ArmoniK's Goals

- Simplify the development of distributed applications
- Provide a user-friendly interface for scientists and engineers
- Adress the challenges of HPC without requiring deep expertise in parallel programming
- Getting reasonable performances

ArmoniK: a Hybrid Framework



- Computational kernels: User computations
- Data management: Reads, Writes, Communications between processes
- Advanced features: Overlapping, load balancing
- Jobs management: Job queues, resource allocation, job lifecycle
- Jobs / Resources mapping: Determine job execution on which machines
- Resources management: Machine pool update, node addition or removal

Task-based Programming in ArmoniK

Definition

Paradigm focusing on the decomposition of complex operations into smaller tasks

- Expression of complex data-driven dependencies
- ArmoniK's **distributed scheduler** responsible for:
  - Task distribution and load balancing
  - Dependency resolution
  - Tasks execution
  - Data management (overlapping, prefetching, and checkpointing)

```python
from pymonik import Pymonik, Task

if __name__ == "__main__":
    with Pymonik(endpoint="localhost:5001", partition="pymonik"):
        my_task = Task(lambda a, b: a+b, func_name="add")
        result = my_task.invoke(1, 2).wait().get()
        print(f"Result of add task: {result}") # 3
```

ArmoniK Programming Model

- **Task Graph**: Bipartite DAG whose node sets are tasks and data
- **Task node**: Single-node computation (possibly multi-threaded) taking one or several data inputs and outputting one or several data
- **Data node**: Immutable piece of data depending on only one task at most
- **Dependencies**: Dependencies between tasks are expressed as data dependencies
- **Dynamic**: The graph may not be entirely known in advance (tasks can append tasks and replace edges with subgraphs)

Dynamic Graph

Definition

Dependency graph is not fully known when scheduling starts.

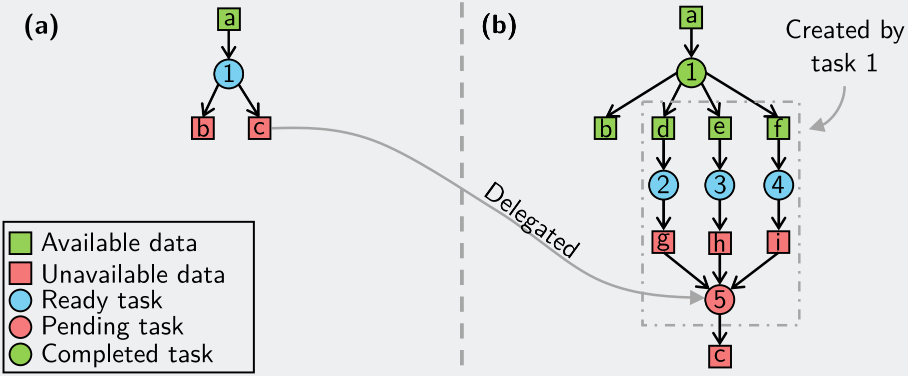
- Task dependencies not known before submission
- Submissions can happen anytime
- Tasks can submit new tasks
- Tasks can delegate the production of their output to their new tasks

```python
from pymonik import Pymonik, task

@task
def add_one(a:int) -> int:
    return a + 1

@task
def add(a: int, b: int) -> int:
    if a <= 0:
        return b
    b_plus_one = add_one.invoke(b)
    return add.invoke(a-1, b_plus_one, delegate=True)
```

Dynamic Graph Example

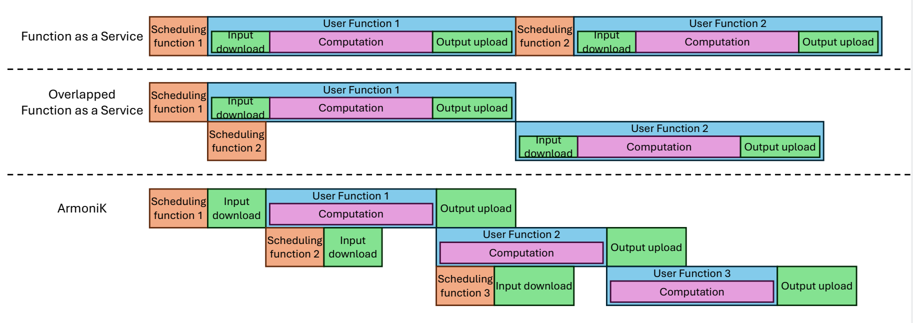


Computations/Comm Overlapping

Data Management

Responsibility for data allocation, transfer, and storage between computational operations

- ArmoniK is responsible for tasks input and output data management
- Allows for automatic communication + scheduling/task execution overlapping
- Automatic Uncoordinated Checkpointing

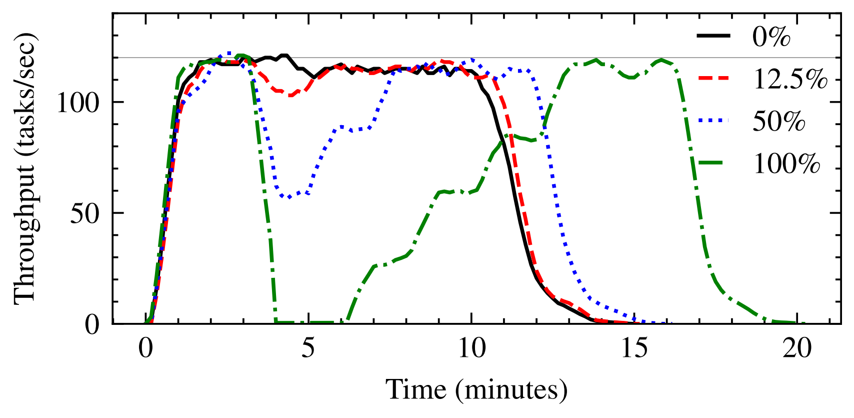


ArmoniK Built-in Features

- **Open Source**: `https://github.com/aneoconsulting/ArmoniK`
- **Observability**: Extensive GUIs, CLIs, monitoring APIs, metrics, logs, and traces
- **Portability**: Effort to transfer an application from one environment to another
  - Officially supported languages: C#, C++, Python, Rust, Java, and JavaScript
  - Tasks on different architectures (x86, ARM, GPU, Linux, Windows), applications, environments
- **Malleability**: Dynamic reconfiguration of the number of allocated resources during execution without interruption
- **Resource Sharing**: Share resources between applications to maximize resource utilization
- **Modularity**: Modules can be swapped without modifying ArmoniK's code to suit user needs and constraints
- **Production Ready**: Designed to be used in production environments, with a focus on stability, security, scalability, and maintainability
  - Used by our clients in their critical systems
  - Drives our needs for validation and guarantees, monitoring, stability

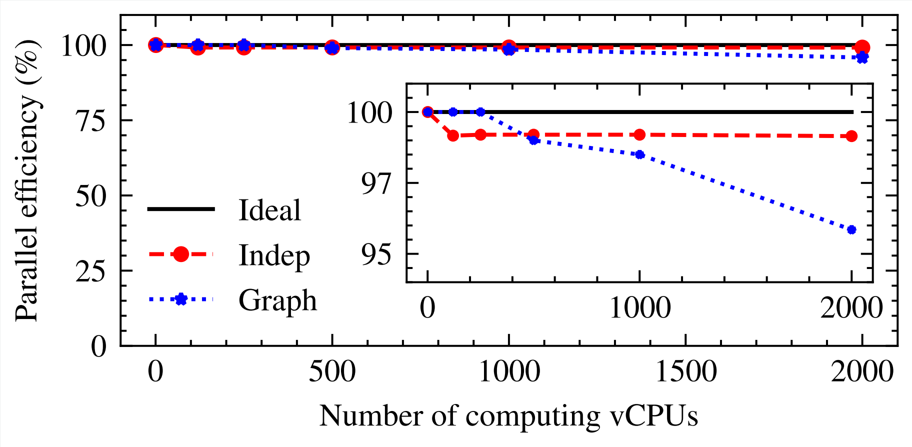
Fault Tolerance

- Works without interruption even when one or more nodes fail
- Allow support for preemptible computing resources
- Automatic and efficient task retry on failure
- Each curve represents a percentage of preempted instances



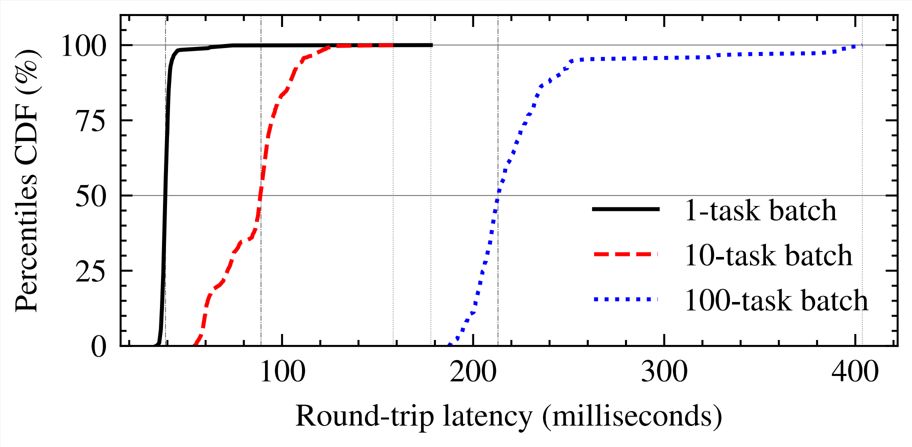
Throughput Scalability

- Indep : independent tasks workload
- Graph : nested fork-join workload
- 1-second long tasks
- Empty input/output data



Low Round-trip Latency

- Cumulative distribution functions (CDFs) of round-trip latency
- Batched submissions of 1, 10, and 100 independent zero-work tasks



Section 3

Towards AI with ArmoniK and JAX

JAX: A Modern Tool for Scientific Computing and AI

JAX is a Python library that enables high-performance scientific computing by combining:

- **NumPy-Compatible API** for familiar, concise numerical code
- **Automatic differentiation** for optimization and ML
- **Just-In-Time (JIT) compilation** for accelerating performance
- **Seamless CPU, GPU, and TPU execution**
- **Composable Transformations** Combine transformations easily: e.g., `jit(grad(f))`, `vmap(grad(f))`.

XLA: The Compiler Behind the Speed

- **XLA (Accelerated Linear Algebra)** is a domain-specific compiler developed by Google.
- It transforms JAX Python functions into highly optimized machine code.
- Performs **operation fusion, memory planning, and hardware-specific optimizations**.
- Enables JAX to deliver **hardware-level performance** from high-level Python code.
- Produces efficient code for CPUs, GPUs, and TPUs.

Impact for Scientists and Engineers

- Write portable, optimized code without low-level programming
- Accelerate AI models, PDE solvers, and simulation kernels
- Integrate HPC workflows with modern ML toolchains
- Reduces boilerplate in high-performance computing
- Enables reproducible, portable, and efficient scientific models

ArmoniK + JAX: A Powerful Combination

- **ArmoniK** provides a distributed task scheduler and data management layer.
- **JAX** offers high-performance numerical computing capabilities.
- Together, they enable:
  - Scalable, fault-tolerant AI and scientific computing
  - Automatic data management and task scheduling
  - Efficient execution on heterogeneous hardware
- Ideal for large-scale AI training, scientific simulations, and data analysis.
- Supports dynamic task graphs and overlapping computations.
- Provides a user-friendly interface for scientists and engineers.

What It Looks Like

- Ideal for large-scale AI training, scientific simulations, and data analysis.

```python
from pymonik import Pymonik, task
from jax import numpy as jnp

@task
def cholesky(A):
    return jnp.linalg.cholesky(A)

if __name__ == "__main__":
    x = jnp.array([[2., 1.],[1., 2.]])
    with Pymonik(endpoint="localhost:5001", partition="pymonik",
    ↪   environment={"pip":["jax"]}):
        L = cholesky.invoke(x).wait().get()
        print(jnp.allclose(x, L @ L.T))
```

- Use GPUs instead of CPUs, just swap dependencies and partition.

```python
    with Pymonik(endpoint="localhost:5001", partition="pymonik-gpu",
    ↪   environment={"pip":["jax[gpu]"]}):
```

Section 4

The Right Tool for the Right Job

The Right Tool for the Right Job

- No single scientist can master all fields in HPC.
- Interfaces between disciplines are crucial.
- And the trade-off between performance and usability is essential.
- However, the current trade-off is speed-oriented, requiring deep expertise.
- ArmoniK aims to shift this trade-off towards usability while maintaining reasonable performance.
- It allows scientists to focus on their research without needing deep expertise in parallel programming.
- ArmoniK is not a replacement for existing HPC tools but a complementary framework that simplifies the development of distributed applications.
- We are looking for more use cases: `https://github.com/aneoconsulting/ArmoniK`

Questions and Discussions

- What missing interoperability layers (software, standard, or abstraction) would most accelerate convergence between traditional HPC linear algebra workflows and today's extreme-scale AI workloads?
  - There are tentatives between hardware and software (Kokkos, JAX, etc.), but what about applications that does not fit in these frameworks?
  - In companies, we see that over time, organizational interfaces tends to match the interfaces in the IT systems.
  - We still have a lot of work to build scientific and technology interfaces that allows users to cooperate without a deep expertise in all HPC/AI-related fields.
- Looking ahead to 2030, do you expect the principal bottleneck for extreme-scale AI to be data, algorithms, resilience or energy, and how does that prediction shape your research priorities today?
  - **Energy**: It will probably be an issue in Europe, probably less true elsewhere.
  - It will be a trade-off between the three others.
- Given the different developments in architecture processors for AI and "computational science", do you think we'll see a convergence or divergence of roadmaps?
  - HPC will use whatever will be available on the market, as in the last 30 years.

Thank you for your attention!

Do you have any questions?