Korali

High-performance framework for Bayesian uncertainty quantification and optimization

13.12.2019 - CSCS Lugano

Dr. Sergio Martin

CSElab

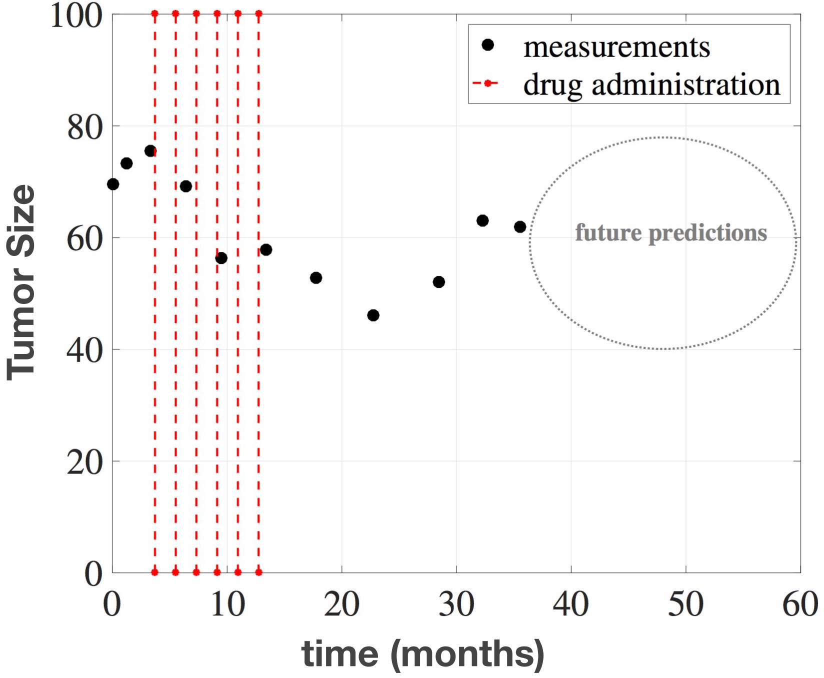
Computational Science & Engineering Laboratory

ETH zürich

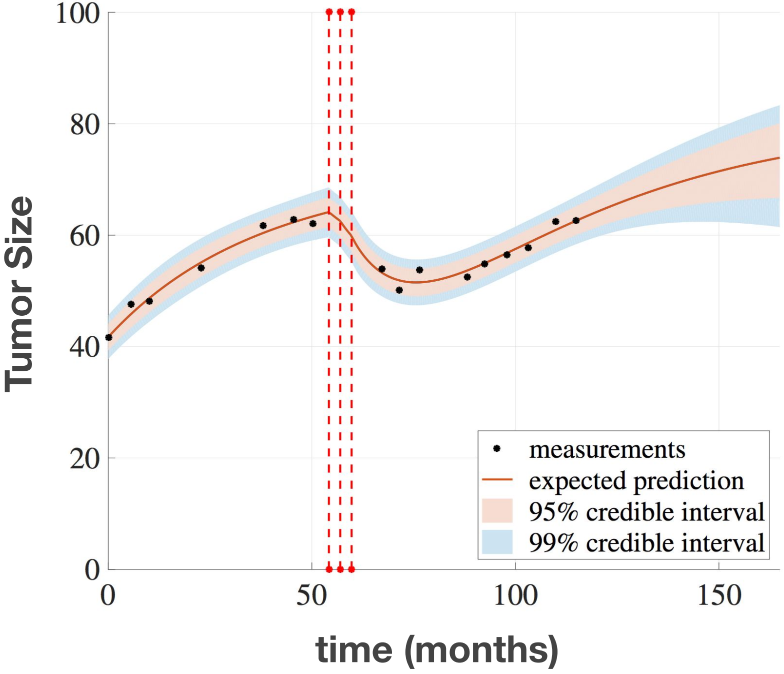
Why Uncertainty Quantification

Medicine: Designing better drugs and treatments for cancer patients.

Measurements



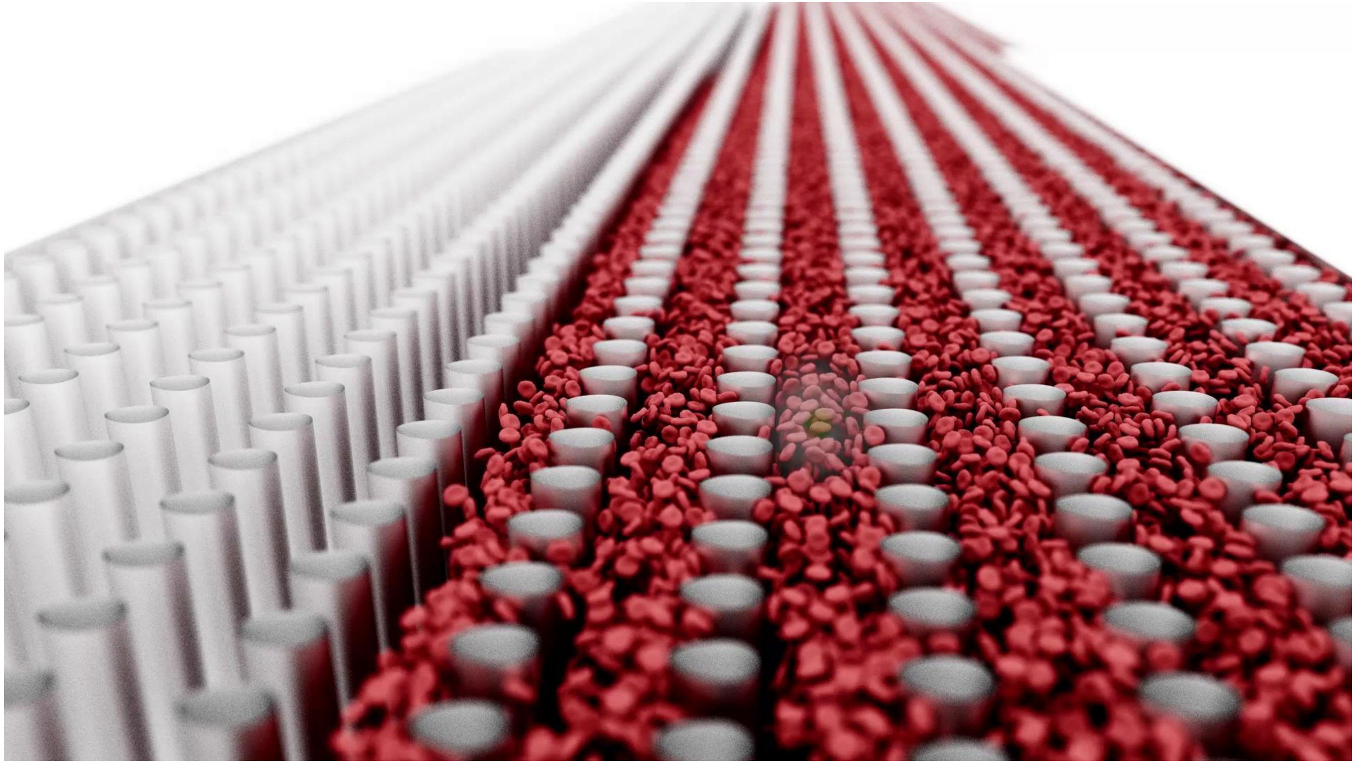
Robust Predictions

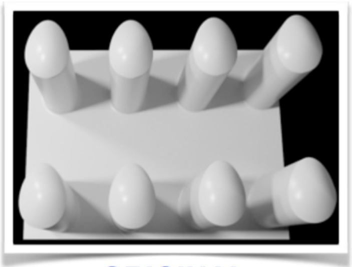


G. Arampatzis, et al. “Langevin diffusion for population based sampling with an application in bayesian inference for pharmacodynamics", 2018

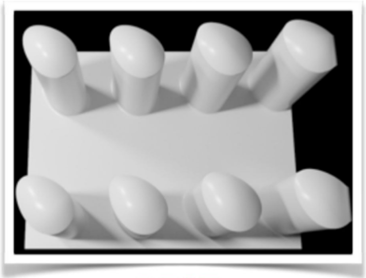
Why Optimization

Improving medical devices for diagnosys.





ORIGINAL

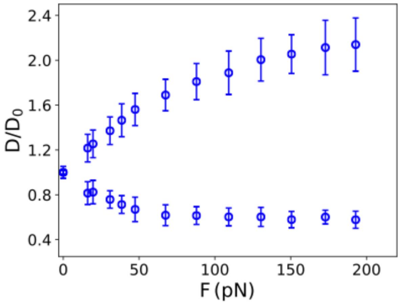


BEST

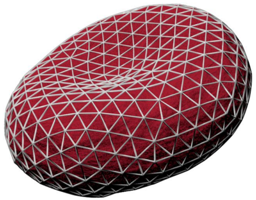
D. Rossinelli et al., “The In-Silico Lab-on-a-Chip: petascale and high-throughput simulations of microfluidics at cell resolution," in Proceedings of Supercomputing (SC) ’15, 2015.

Methodology: Bayesian Inference

Experimental Data
(i.e, Physical Observations)



Computational Model
(e.g. MPI-Based hydrodynamics solver)



Statistical Assumptions
(e.g. Model parameters)

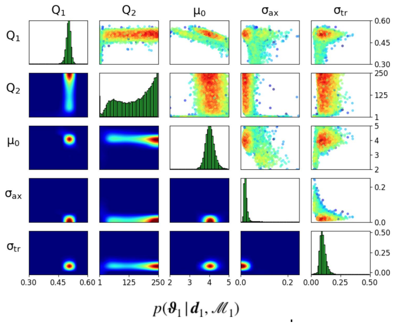
$$d = f(x \,|\, \vartheta) + \epsilon$$

$$\epsilon \sim \mathcal{N}(0, \sigma_n)$$

Applying Bayes' Theorem

$$p(\vartheta \,|\, d) = \frac{p(d \,|\, \vartheta)\, p(\vartheta)}{p(d)}$$

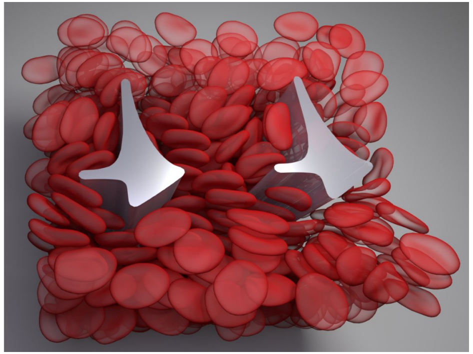
Posterior Distribution of Parameters



Bayesian Inference:
Evidence-based knowledge about the physical reality.

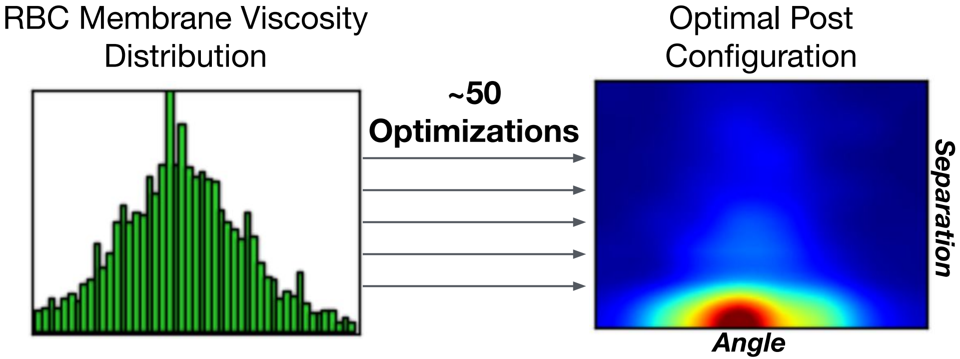
Currently at CSELab @ ETH Zürich

Physical Model
Row of two posts with periodic boundary conditions.



Computational Model
Mirheo: State-of-the-Art GPU-based microfluidics solver.

Statistical Model
Optimization of post configuration over ~50 RBC types.



We need an extreme-Scale UQ/O Framework

Computational Demands Estimation:

GPU-Time per Evaluation: **~7 hours**

50 Optimization Experiments x 400 Evaluations

= 60,000 Model Evaluations

Total usage: ~**140,000 Node Hours**

This represents 100% Piz Daint for a whole day!

State of the art UQ/Opt Libraries

Software	Optimization	Bayesian Inference	Parallelism	Language
APT-MCMC	no	yes	Local (Thread-based)	C++
BCM	no	yes	Local (Thread-based)	C++
EasyVVUQ	no	yes	Fork-Join Concurrency	Python
GAMBIT	yes	yes	no	C++
PSUADE	yes	yes	Job-Scheduler Concurrency	C++
Stan	yes	yes	no	C++
UQLab	yes	yes	no	MATLAB

No existing libraries offer nor have demonstrated:

- Seamless Integration with MPI/CUDA Computational Models
- Efficient execution at at **extreme scales** (thousands of nodes).

The Korali Framework

Mission:

Develop an UQ and optimization framework for extreme-scale studies.

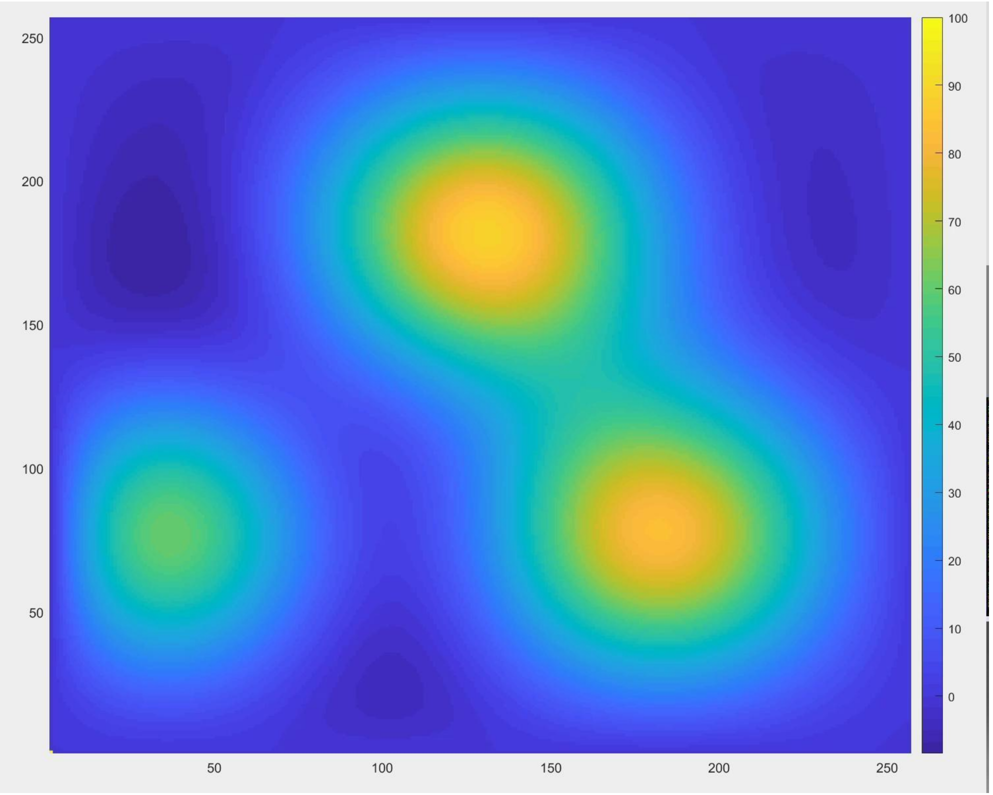
Motivation:

- Ensure a seamless integration with parallel/distributed computational models.
- Maximize node usage.
- Restore jobs in case of failure with minimal loss of progress.
- Highly documented, easy to use, and adopted by the wider community.

About the Project:

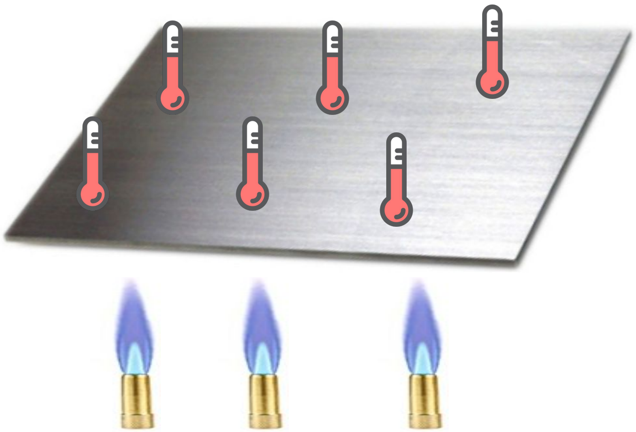
- Development started on early 2019.
- Programmed with C++ and Python.
- Open-Source (github)

Bayesian Inference with Korali (I)



Given:

A square metal plate with 3 sources of heat underneath it.



We have: ~10 temperature measurements at different locations

Can we infer the (x,y) locations of the 3 heat sources?

Bayesian Inference with Korali (II)

To use Korali, users define an ***Experiment.***

Experiment

Model:
2D Heat Equation (MPI)

Problem:
Parameter Inference

Solver:
Sampler

Run

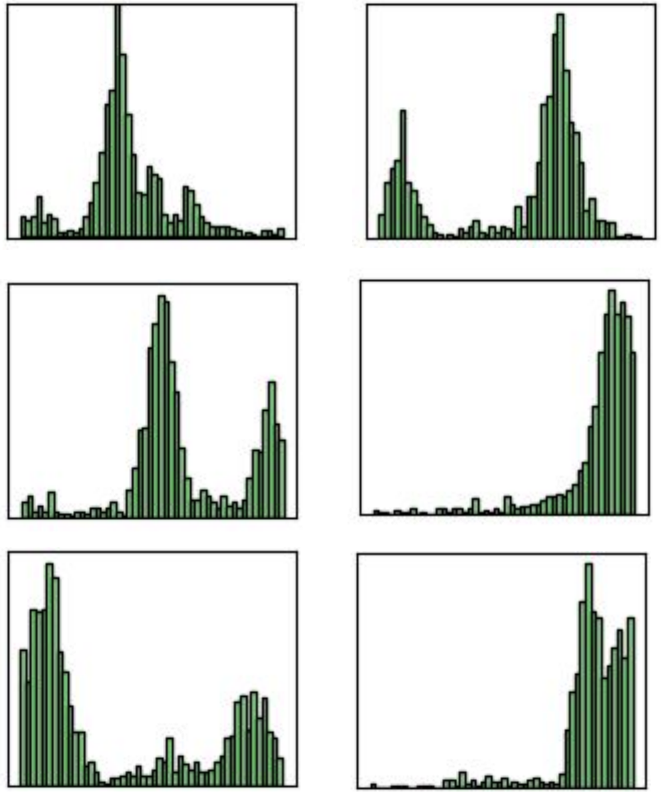
Likelihood Probability Distributions

X **Y**

Heat Source 1

Heat Source 2

Heat Source 3

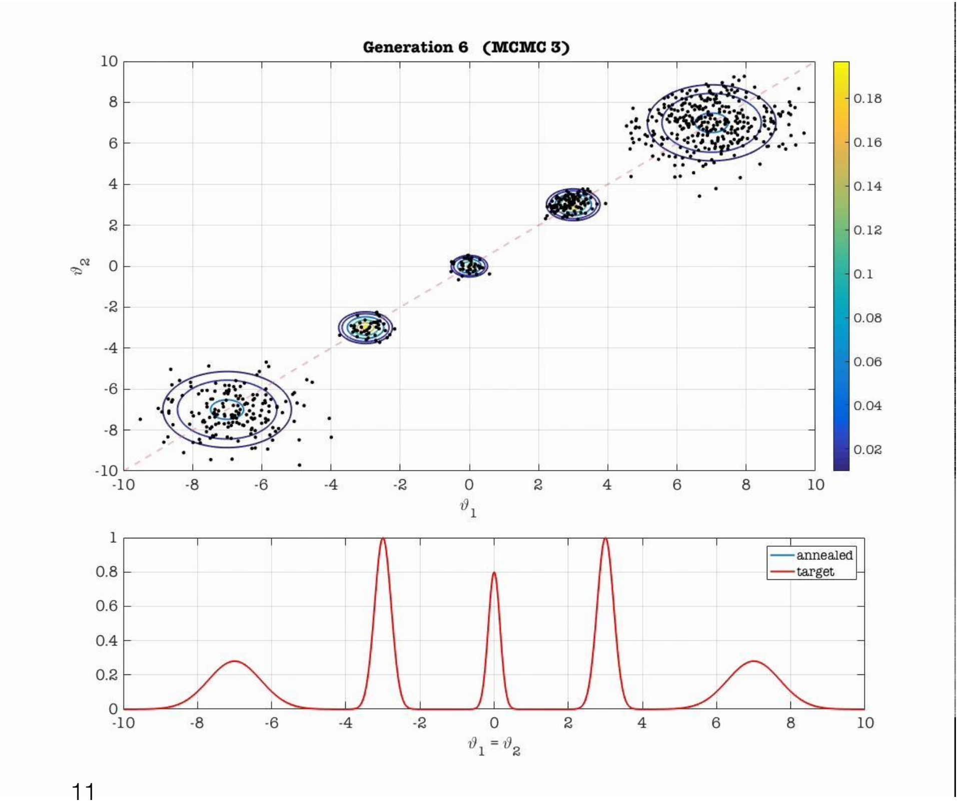


Korali Generation-Based Engine (I)

Example: Sampling Parameter Probability Distribution.

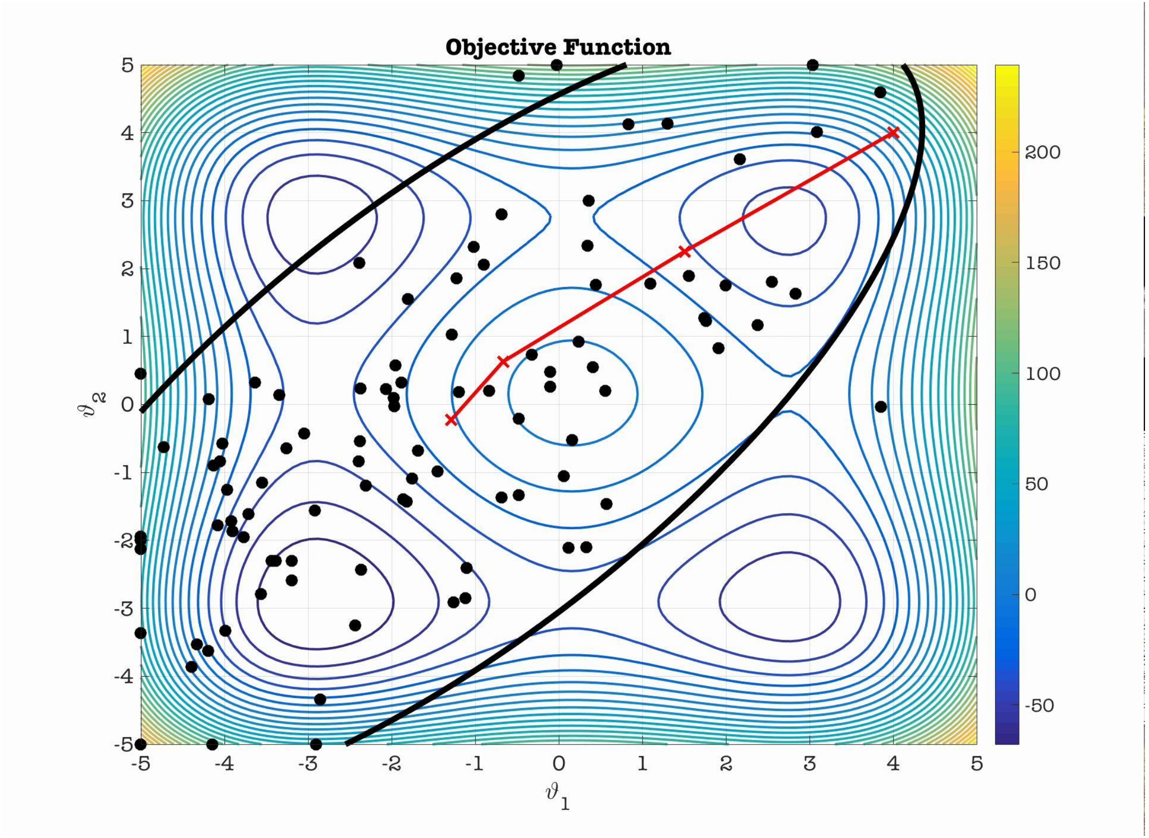
Parameter Space

Approximation to the real Distribution



Korali Generation-Based Engine (II)

Example: Parameter Optimization.



Korali’s 7 Design Goals

+ **Software Engineering Goals**
  + Usability
  + Extensibility
  + Self-Enforced Engineering

+ **High-Performance Goals**
  + Heterogeneous Model Support
  + Scalable Distributed Sampling
  + Self-Enforced Fault Tolerance
  + Efficiency at extreme scale.

Usability

Approach: We use a **descriptive** interface. Specifies the **what**, not the how.

```python
from myModels import myModel
e = korali.Experiment()

# Configuring problem
e["Problem"]["Type"] = "Evaluation/Direct"
e["Problem"]["Objective Function"] = myModel

e["Variables"][0]["Name"] = "Mu"
e["Variables"][0]["Minimum"] = 0.0
e["Variables"][0]["Maximum"] = 100000.0

e["Variables"][1]["Name"] = "Sigma"
e["Variables"][1]["Minimum"] = 0.0
e["Variables"][1]["Maximum"] = 100000.0

# Configuring Solver
e["Solver"]["Type"] = "Sample/MCMC"
e["Solver"]["Population Size"] = 3
e["Solver"]["Burn In"] = 5
e["Solver"]["Max Samples"] = 10000

korali.run(e)
```

Problem

Variables

Solver

Minimal programming knowledge required.

No function calls used, other than *run*()

User does not need to know how Korali operates.

Only describe the innate characteristics of the problem.

Independent from implementation.

This same interface could be used by other libs.

Mostly Language-independent.

Add semicolons for C++ or load from config file.

Korali’s 7 Design Goals

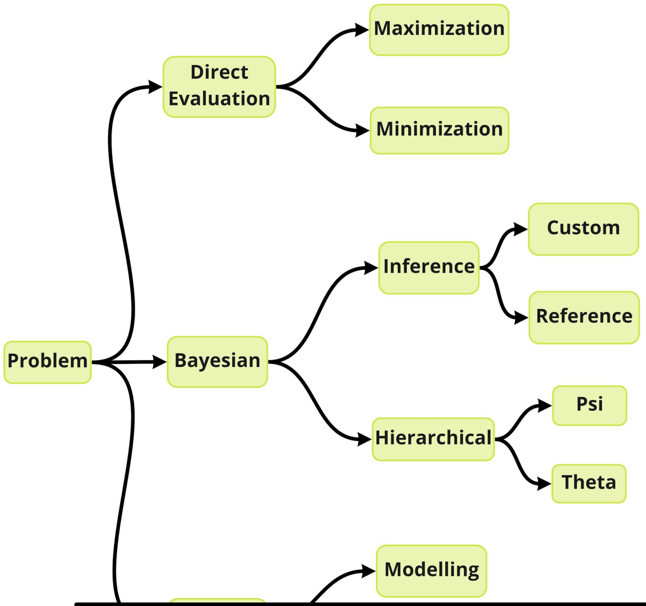
+ **Software Engineering Goals**
    + Usability
    + Extensibility
    + Self-Enforced Engineering

+ **High-Performance Goals**
    + Heterogeneous Model Support
    + Scalable Distributed Sampling
    + Self-Enforced Fault Tolerance
    + Efficiency at extreme scale.

Korali Modular Design

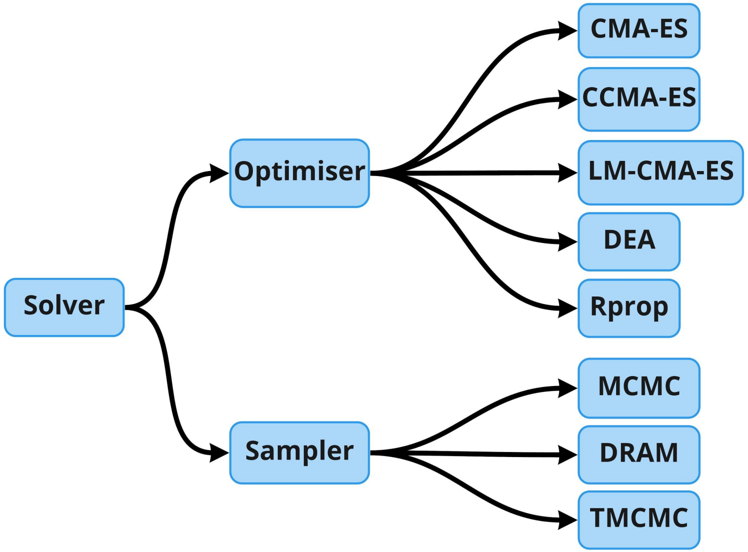
Three problem families

Total: 8 different problem types.



Two solver families

Total: 8 different solver methods.



Several more modules are currently in development.

Extending Korali

Anyone can add a new solver or problem into Korali.

- Allow users to develop and test new methods at scale.
- Create a user community that develops and extends Korali organically.

- **Requirements:** Basic object-based C++ knowledge.
- **Strategy:** Plug-and-Play (automatic module detection).

Example: Adding a new optimizer.

```
/solvers/optimizer/CMA-ES
                  /CCMA-ES
                  /LM-CMA-ES
                  /DEA
                  /Rprop
                  /myOptimizer
```

Adding 3 Files…

`/myOptimizer._hpp`

Defines the myOptimizer class.
Inherits responsibilities from the parent (optimizer) class

`/myOptimizer._cpp`

Defines how this class satisfies these responsibilities

`/myOptimizer.config`

Specifies and documents user-configurable settings
Uses JSON (JavaScript Object Notation) format.

Korali's 7 Design Goals

- **Software Engineering Goals**
  - Usability
  - Extensibility
  - Self-Enforced Engineering

- **High-Performance Goals**
  - Heterogeneous Model Support
  - Scalable Distributed Sampling
  - Self-Enforced Fault Tolerance
  - Efficiency at extreme scale.

Self-Enforced Software Engineering (I)

We want Korali to be community-driven. Therefore…

We need to **enforce** good SW practices systematically.

1) Every configuration item **shall** be documented.

/myOptimizer.config

```
"Name": [ "Population Size" ]
"Type": "size_t"
"Description": "Specifies the number of
                samples to evaluate per generation..."

"Name": [ "Mu Value" ]
"Default": "32"
"Type": "size_t"
"Description": "Number of samples used
                to update the covariance matrix"
```

Automatic Web-based Documentation

Module Configuration

["Population Size"]

Specifies the number of samples to evaluate per generation (preferably $4 + 3 * log(N)$, where N is the number of variables).

- Default Value: *none*
- Datatype: size_t
- Syntax:

```
korali["Variables"][i]["CMAES"]["["Population Size"]"] = *value*
```

["Mu Value"]

["Mu Type"]

Self-Enforced Software Engineering (II)

2) Every new module needs a **tutorial**.

```
/tutorial/a1-myOptimizer/run-myOptimizer.py
/tutorial/a1-myOptimizer/README.md
```

Uploaded automatically to our Webpage

Must be a representative Python or C++ application

A.10 - Optimizing a problem with MyOptimizer

In this tutorial we show how to **optimize** and **sample** the posterior distribution of a Bayesian inference problem.

Problem Setup

In this example we will solve the inverse problem of estimating the Variables of a linear model using noisy data. We consider the computational model,

$$f(x;\vartheta) = \vartheta_0 + \vartheta_1 x\,,$$

for $x \in \mathbb{R}$. We assume the following error model,

$$y = f(x;\vartheta) + \varepsilon\,,$$

with ε a random variable that follows normal distribution with zero mean and σ standard deviation. This assumption leads to the likelihood function,

$$p(y|\varphi, x) = \mathcal{N}(\,y\,|\,f(x;\vartheta), \sigma^2\,)\,.$$

Self-Enforced Software Engineering (II)

3) Korali automatically converts all tutorials into **CircleCI** regression tests:

Test Collection

Type	Code	Description
Regression Test	REG-000	Check for a correct installation of Korali and its modules.
Regression Test	REG-001	Re-run all example applications for basic sanity check.
Regression Test	REG-002	Run the korali.plotter for all example application results.
Regression Test	REG-003	Test correct execution of solvers with non 0815 parametrization.
Regression Test	REG-004	Run the korali.plotter for all example application results.

All tests **must pass** before accepting the new module:

Build Status

Status	Branch	URL
PASSED	master	https://github.com/cselab/korali/tree/master
PASSED	development	https://github.com/cselab/korali/tree/development

Test Architectures

System	Compiler	Python
Debian GNU/Linux 9	gcc version 6.3.0	Python 3.7.3
macOS 10.13.6 (Darwin 17.7.0)	Apple LLVM version 10.0.1 (clang-1001.0.46.4)	Python 3.7.3

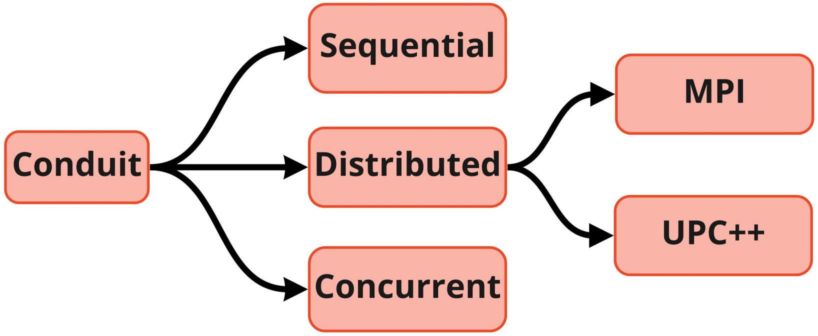
Korali's 7 Design Goals

- **Software Engineering Goals**
  - Usability
  - Extensibility
  - Self-Enforced Engineering

- **High-Performance Goals**
  - Heterogeneous Model Support
  - Scalable Distributed Sampling
  - Self-Enforced Fault Tolerance
  - Efficiency at extreme scale.

Heterogeneous Model Support

Korali exposes multiple **"Conduits"**: ways to run computational models.



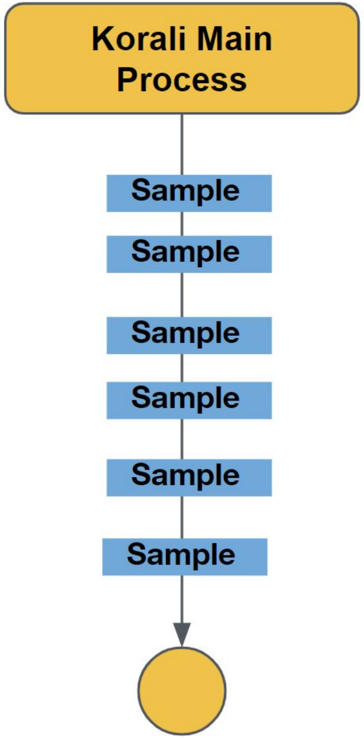
+ **Sequential (default):**
  For simple function-based Python/C++ models (e.g., **f(x) = x**^{2}).

+ **Concurrent:**
  For legacy code or pre-compiled applications (e.g., LAMMPS, Matlab, Fortran).

+ **Distributed:**
  For MPI/UPC++ distributed models (e.g., Mirheo).

Sequential Conduit

Links to the model code and runs the model sequentially via function call:



Computational Model

```python
def myModel(sample):
  x = sample["Parameters"][0]
  y = sample["Parameters"][1]
  # ... computation...
  sample["Evaluation"] = result
```

Korali Application

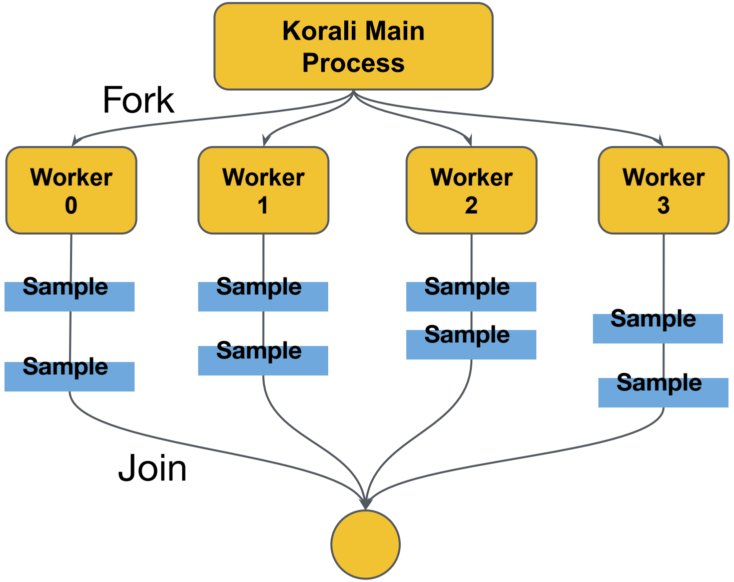
```python
e = korali.Experiment()
k = korali.Engine()
...
e["Problem"]["Objective Function"] = myModel
k["Conduit"]["Type"] = "Sequential"
k.run(e)
```

Running Application

```
$ ./myKoraliApp.py
```

Concurrent Conduit

Uses fork/join to create multiple concurrent worker processes.



Running Application

```
$ ./myKoraliApp.py
```

Computational Model

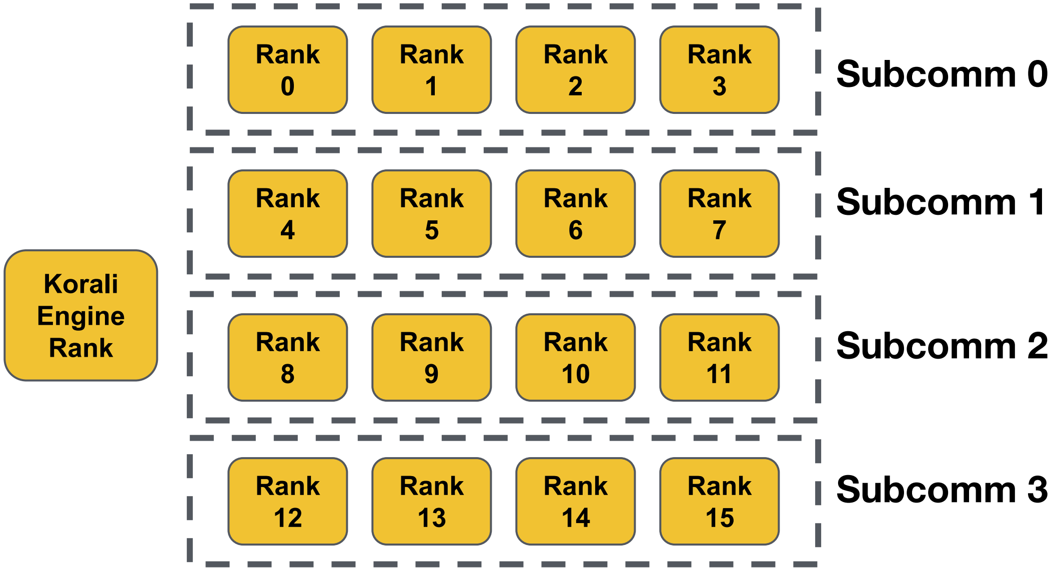
```python
def myModel(sample):
  x = sample["Parameters"][0]
  y = sample["Parameters"][1]
  os.shell.run("srun -n 32 ./myModel" + x + y)
  result = parseResults('ResultFile.out')
  sample["Evaluation"] = result
```

Korali Application

```python
e = korali.Experiment()
k = korali.Engine()
...
e["Problem"]["Objective Function"] = myModel
k["Conduit"]["Type"] = "Concurrent"
k["Conduit"]["Concurrent Jobs"] = 4
k.run(e)
```

Distributed Conduit

Links to and runs distributed MPI/UPC++ applications through sub-communicators.



Computational Model

```
def myModel(sample, MPIComm):
  x = sample["Parameters"][0]
  y = sample["Parameters"][1]
  myRank    = comm.Get_rank()
  rankCount = comm.Get_size()
  # ... Distributed Computation...
  sample["Evaluation"] = result
```

Korali Application

```
e = korali.Experiment()
k = korali.Engine()
...
e["Problem"]["Objective Function"] = myModel
k["Conduit"]["Type"] = "Distributed"
k["Conduit"]["Backend"] = "MPI"
k["Conduit"]["Ranks Per Sample"] = 4
k.run(e)
```

Running Application

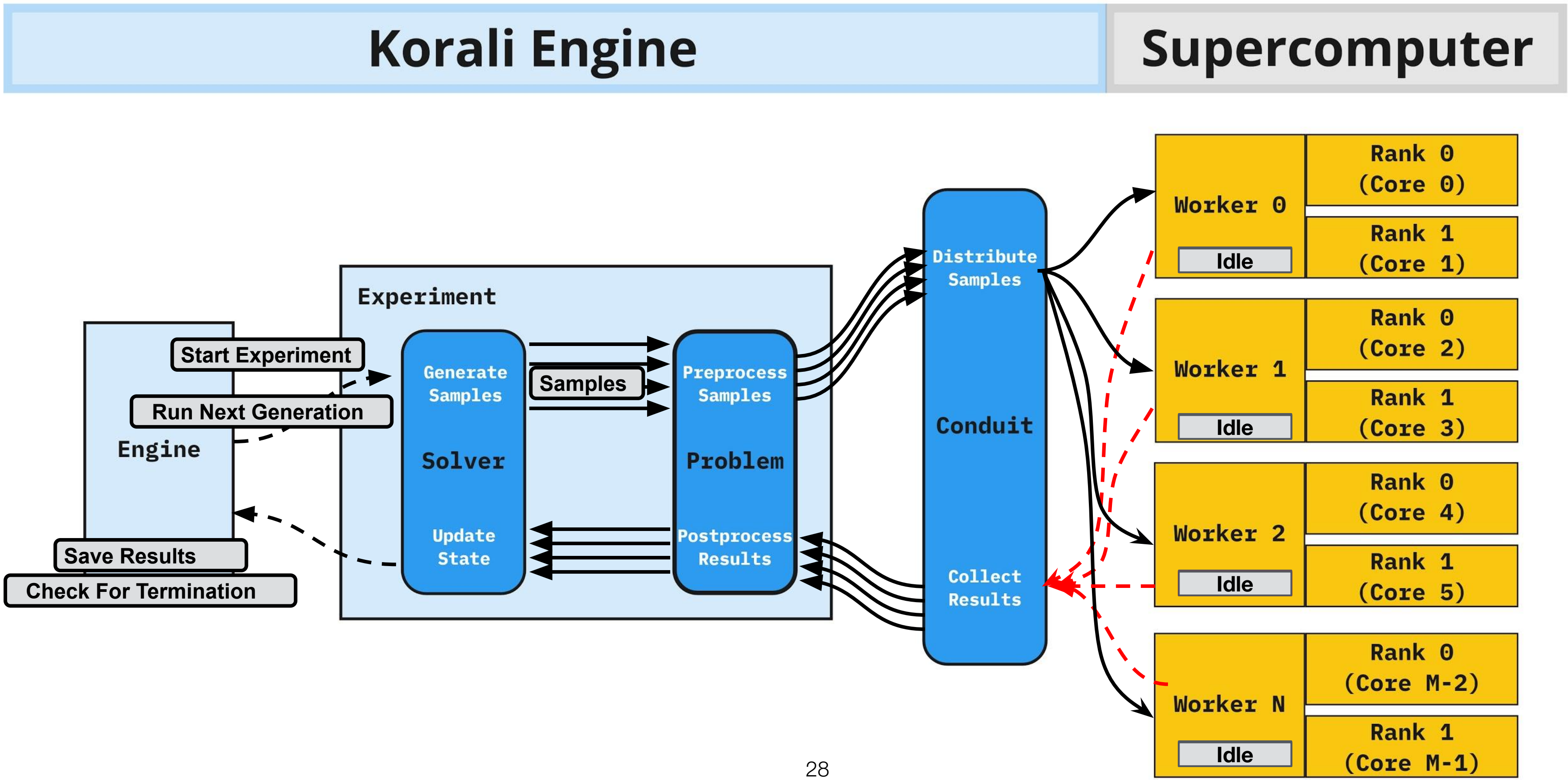
```
$ mpirun -n 17 ./myKoraliApp.py
```

Korali's 7 Design Goals

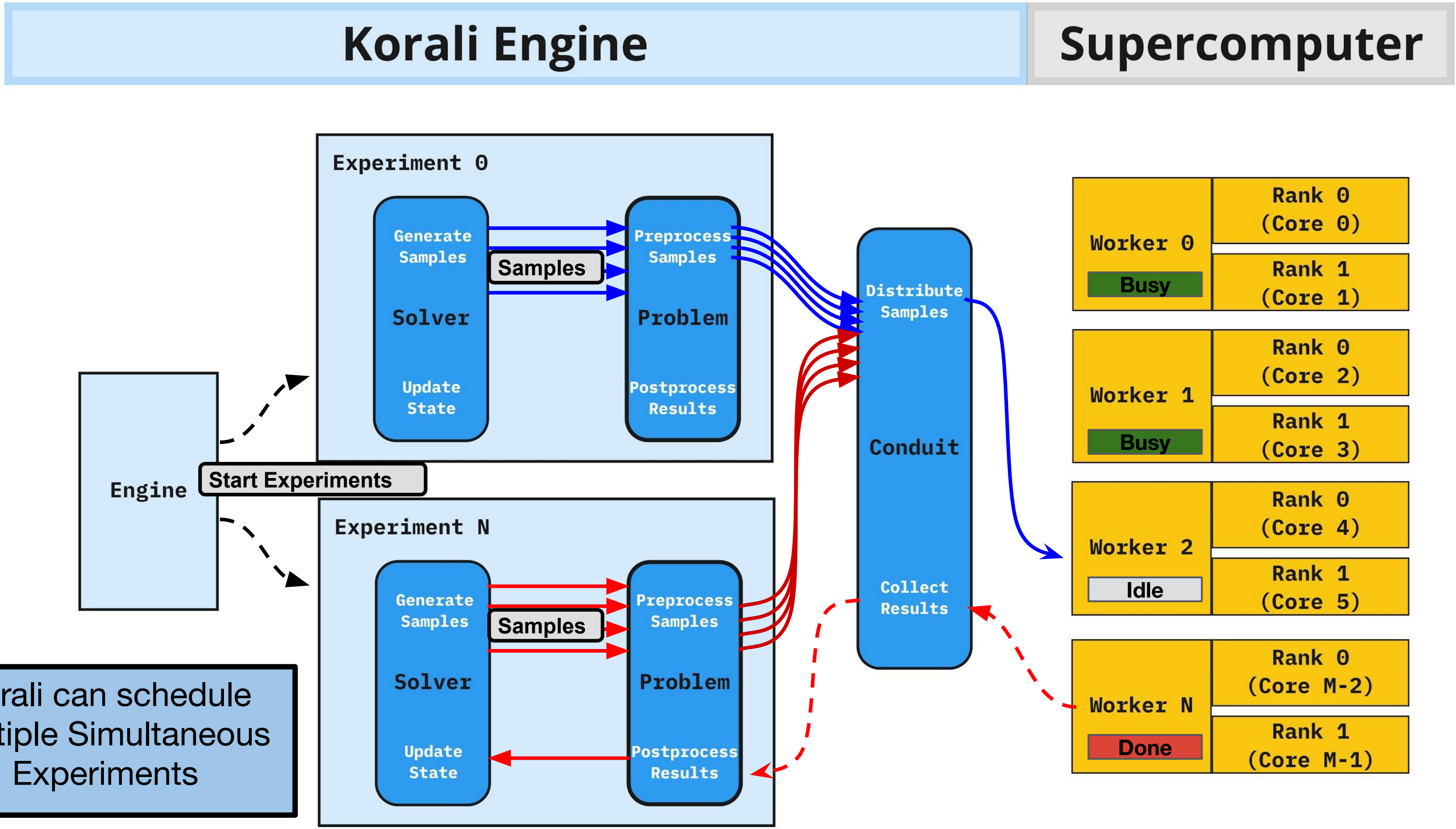
+ **Software Engineering Goals**
  + Usability
  + Extensibility
  + Self-Enforced Engineering

+ **High-Performance Goals**
  + Heterogeneous Model Support
  + Scalable Distributed Sampling
  + Self-Enforced Fault Tolerance
  + Efficiency at extreme scale.

Korali's Scalable Sampler



Scheduling Multiple Experiments



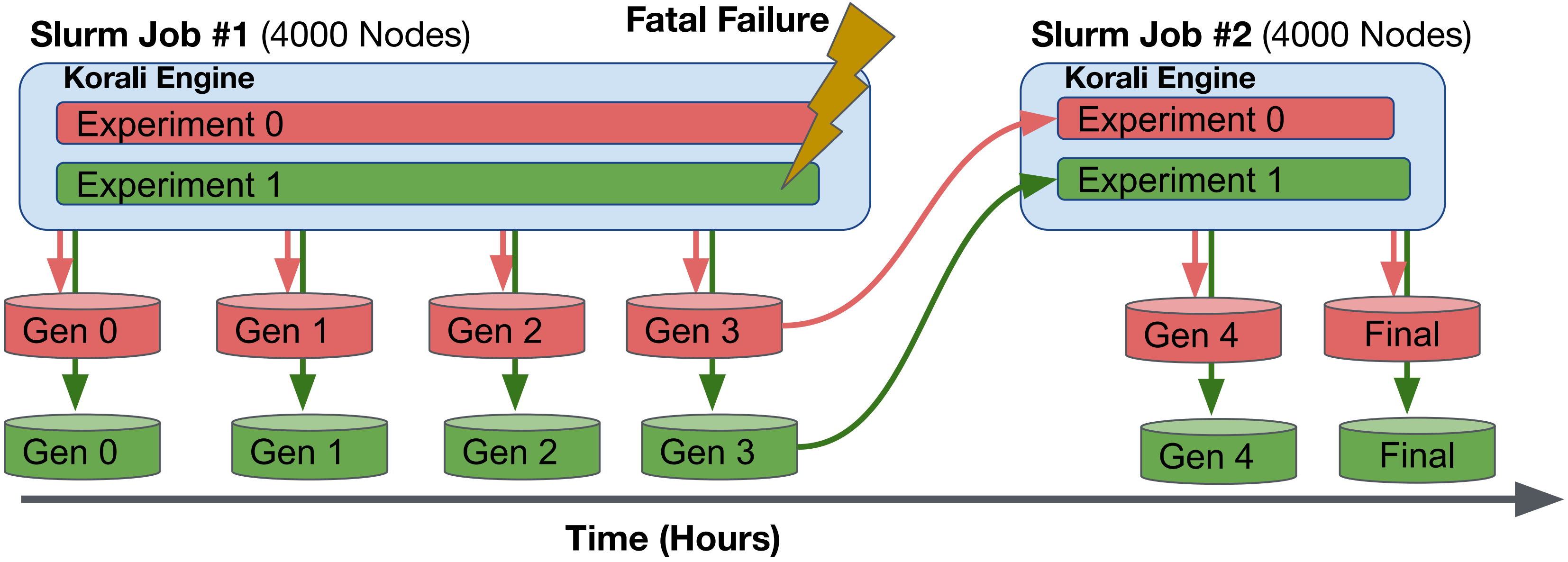
Korali can schedule Multiple Simultaneous Experiments

Korali’s 7 Design Goals

- **Software Engineering Goals**
  - Usability
  - Extensibility
  - Self-Enforced Engineering
- **High-Performance Goals**
  - Heterogeneous Model Support
  - Scalable Distributed Sampling
  - Self-Enforced Fault Tolerance
  - Efficiency at extreme scale.

Self-Enforced Fault Tolerance (I)

Korali saves the entire state of the experiment(s) at **every** generation.



Korali can resume **any** Solver / Problem / Conduit combination.
How? Enforced Serialization

Enforced Serialization (I)

Class members in Korali are defined in the config file.

`/myOptimizer._hpp`

`(Class Declaration and Methods Only)`

`/myOptimizer._cpp`

`(Algorithm-relevant methods only)`

`/myOptimizer.config`

`Documents the configuration and state variables of the module.`

Korali Source Preprocessor

`/myOptimizer.hpp`

`Class Declaration and Methods`
`+ Class Members`
`+ Serialization Declarations`

`/myOptimizer.cpp`

`Algorithm-relevant methods`
`+ Serialization method`
`+ Deserialization method`

Benefit: Collaborating users need not worry about serialization.

Korali's 7 Design Goals

- **Software Engineering Goals**
  - Usability
  - Extensibility
  - Self-Enforced Engineering

- **High-Performance Goals**
  - Heterogeneous Model Support
  - Scalable Distributed Sampling
  - Self-Enforced Fault Tolerance
  - Efficiency at extreme scale.

Korali Benchmark

Study: Red Blood Cell - Strain and bending energy inference

Platform: CSCS Piz Daint (GPU)

+ **Processor:** Intel® Xeon® E5-2690 v3 @ 2.60GHz
+ **GPU:** NVIDIA® Tesla® P100 16GB DRAM

Method: Single-Parameter Bayesian Inference with TMCMC

Computational Model: RBC Stretching

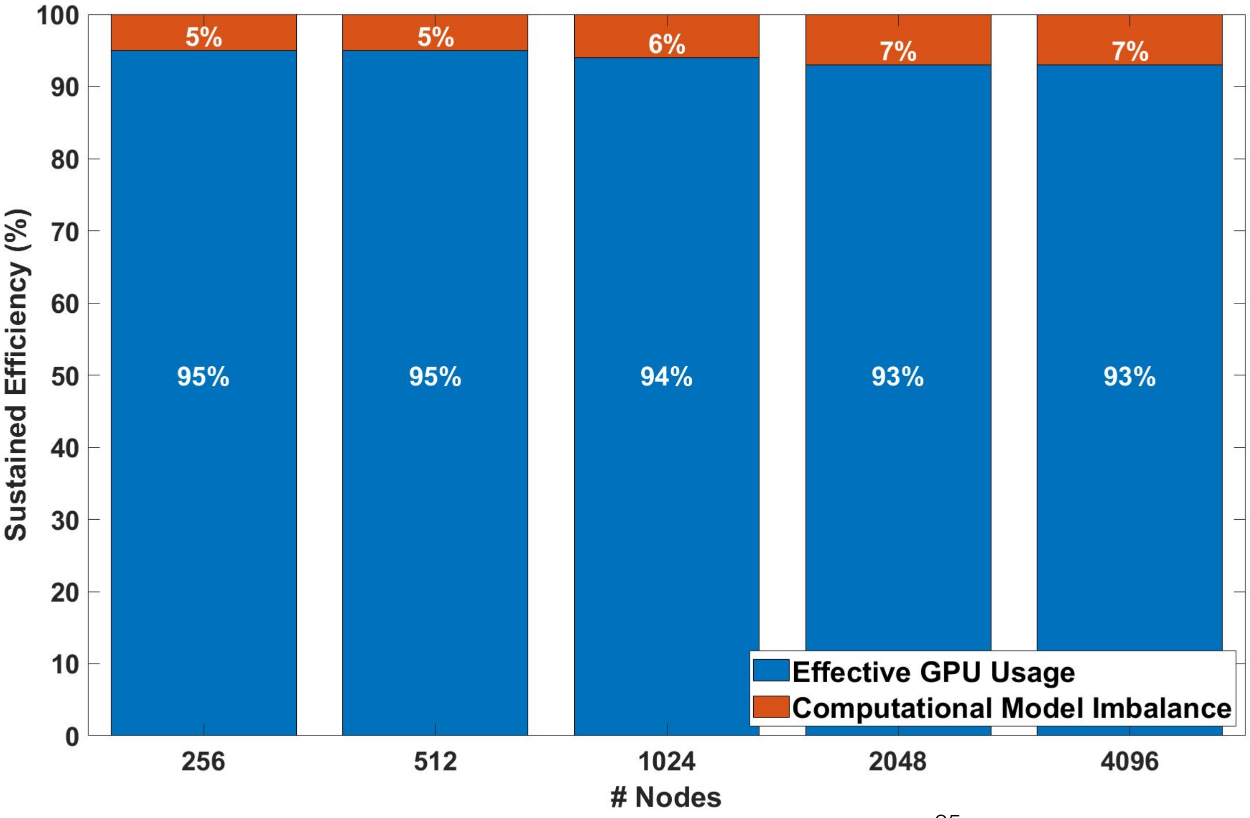
+ Mirheo, 1 GPU x ~15 minutes per sample.

Scaling: Weak Scaling (1 Sample, 1 Node)

+ From 256 to 4096 Nodes **(71% of GPU Piz Daint)**

A. Economides et al. “Hierarchical Bayesian Uncertainty Quantification for a Red Blood Cell Model” [On Review]

Korali Benchmark (Results)

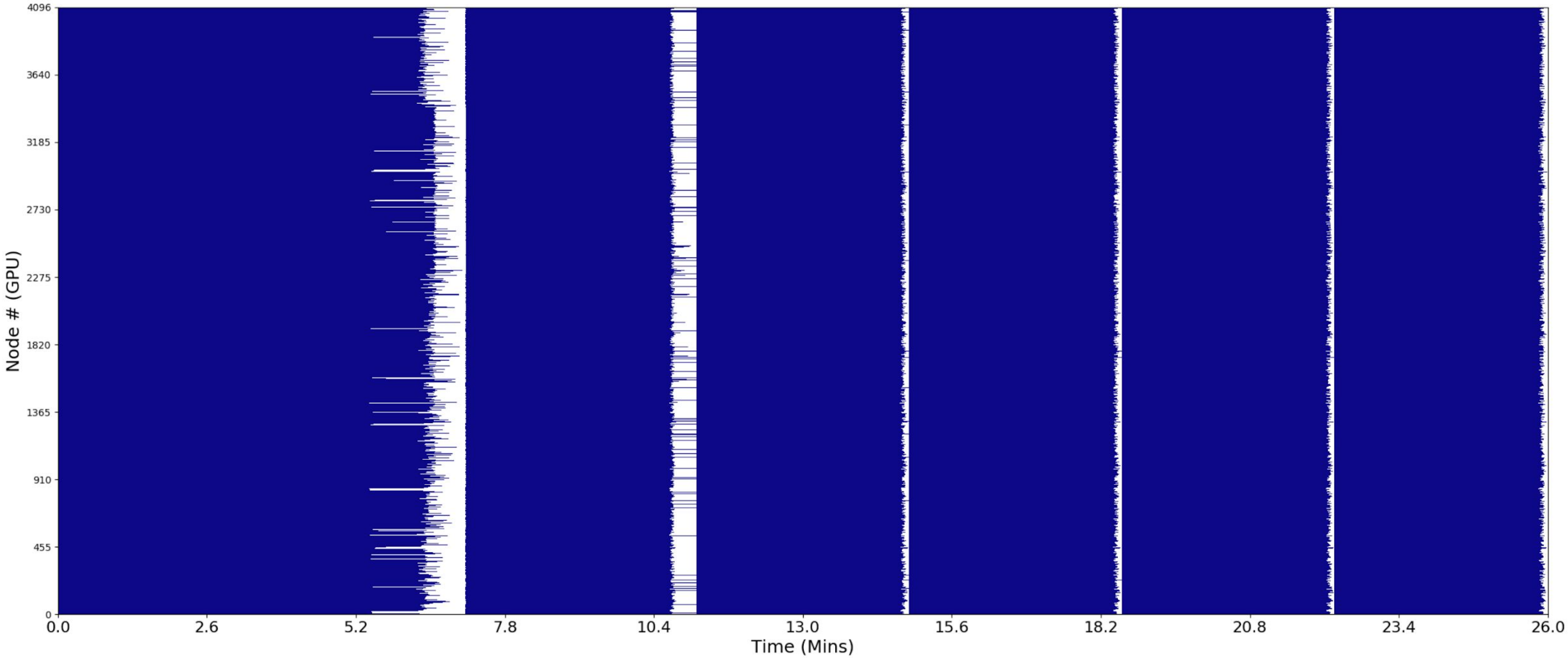


Model Imbalance can reduce efficiency

Korali introduces negligible scheduling or method overheads.

Execution Timeline (4096 Nodes)

Imbalance causes loss in efficiency:



Addressing Model Imbalance with Korali

Study: Red Blood Cell - Membrane viscosity inference

Platform: CSCS Piz Daint (GPU)

+ **Processor:** Intel® Xeon® E5-2690 v3 @ 2.60GHz
+ **GPU:** NVIDIA® Tesla® P100 16GB DRAM

Method: Five Inference Experiments with TMCMC

+ 5 Datasets from *[Henon 1999] and [Hochmuth 1979]*
+ Apply Hierarchical Bayesian Inference on the results

Computational Model: RBC Relaxation

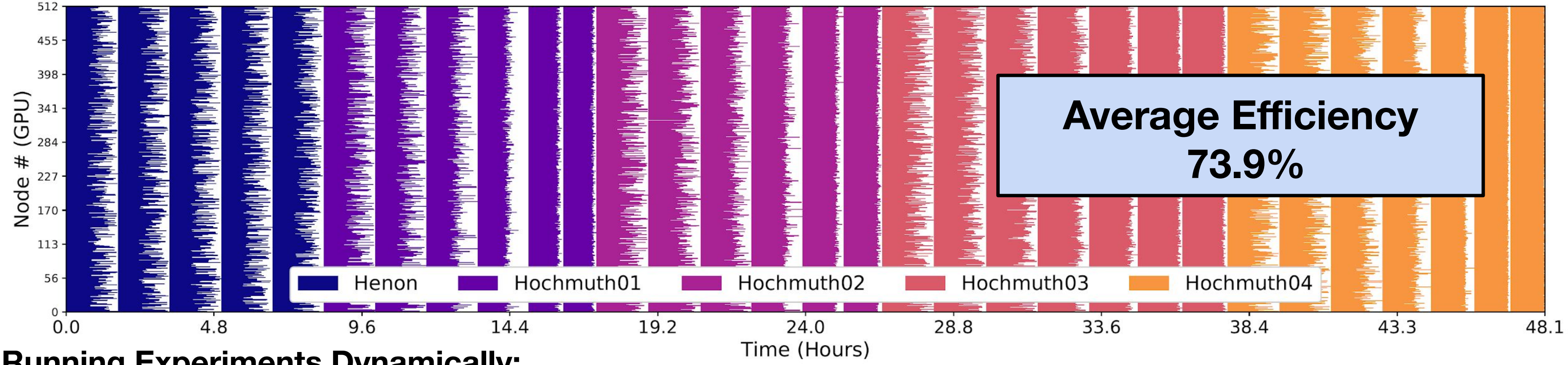
+ Mirheo, 1 GPU x ~45 minutes per sample.

Scale: Single 512-node run.

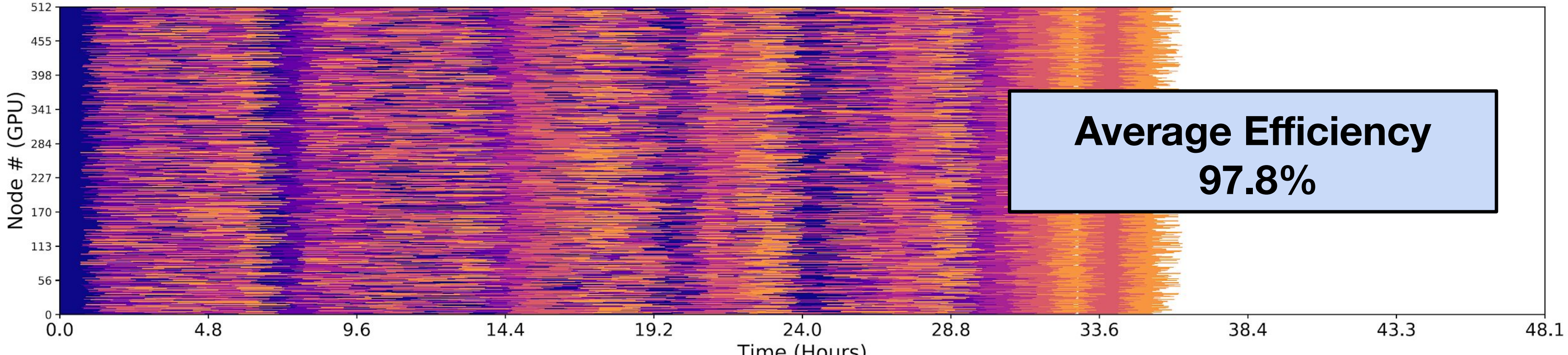
S. Hénon, et al. "A new determination of the shear modulus of the human erythrocyte membrane using optical tweezers." Biophysical Journal, 1999
R. Hochmuth, et al**. "Red cell extensional recovery and the determination of membrane viscosity." Biophysical journal, 1979.**

Execution Timeline (512 Nodes)

Running Experiments Sequentially:

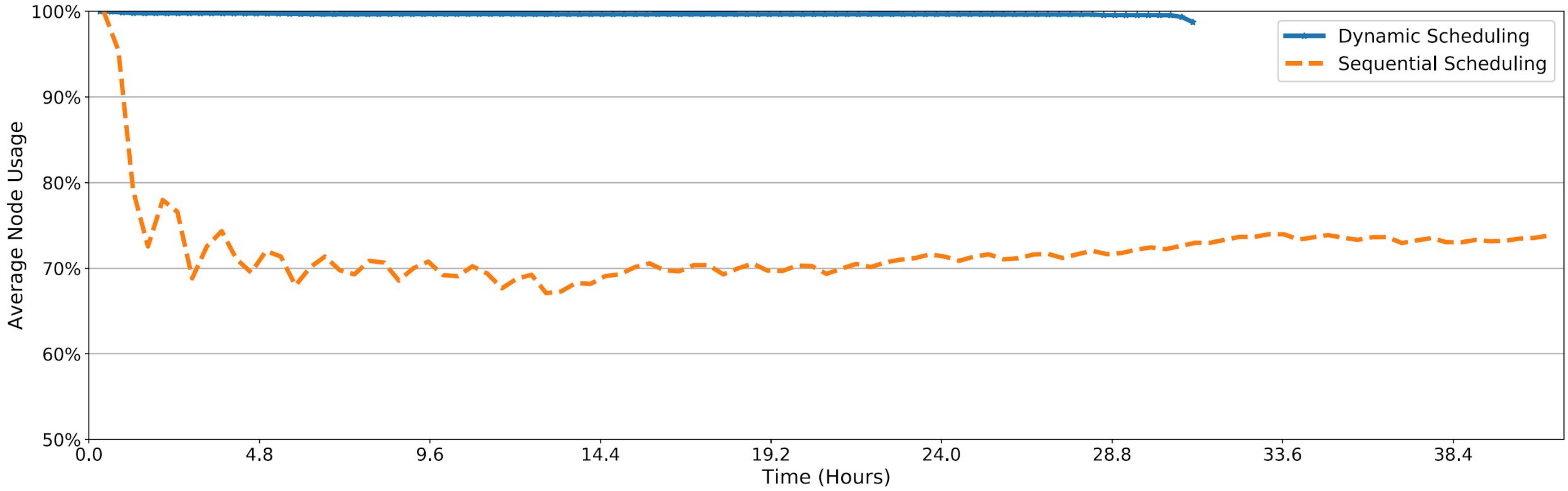


Running Experiments Dynamically:



Execution Timeline (512 Nodes)

Efficiency Timeline:

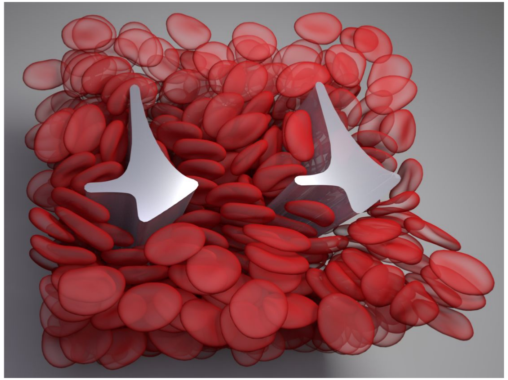


Scheduling multiple experiments in a job realizes sustained efficiency even with model imbalance.

[We are preparing these results for publication.]

Next Steps (I)

Applying Korali to the Hydrodynamic Cell Sorting Study



Current Situation:

Computational demands exceed our budget.

Opportunities for improvement:

+ High Model Imbalance (~70%).
+ Early detection of failing samples (no separation).

Goal: ~140,000 Node Hours ⇨ ~60.000 Node Hours

Next Steps (II)

Extend Korali's Scope:

- Reinforcement Learning
- Surrogate Modelling
- Gaussian Processes (Interpolation)
- Optimal Sensor Placement (Robotics)

About us

Visit our Website: cse-lab.ethz.ch/korali

Source Code: github.com/cselab/korali

Twitter: twitter.com/ethkorali

The Korali Team:

George Arampatzis
Postdoc @ ETHZ

Sergio Martin
Postdoc @ ETHZ

Daniel Wälchli
PhD Student @ ETHZ

Prof. Petros Koumoutsakos
Principal Investigator

Student Assistants:

- Mark Martori (MSc Student @ UZH)
- Susanne Keller (MSc Student @ ETHZ)