Today's Lecture

High Throughput Computing:
- Comparison with High-Performance Computing
- Rationale and relation to sampling-based methods

Introduction To Korali
- Basic Principles and Usage
- Examples

Sample Distribution
- Scheduling and Workload Division Strategies
- A discussion of MPI for sampling
High Throughput vs High Performance
**HPC vs. HTC**

**High-Performance Computing**

Build efficient, resilient, and scalable algorithms to obtain results as fast as possible.

**Examples:**

- Simulation of physical systems
  - Earthquakes
  - Fusion Reactors
  - Fluid Dynamics
  - Climate Modeling

- Artificial systems
  - Machine Learning / AI
  - Big-Data Analysis
  - Traffic / Stock Exchange Modeling

**High-Throughput Computing**

Execute as many tasks as possible, given the allotted time and computational resources.

**Examples:**

- High-Volume Operations
  - Internet / Gaming Server Requests
  - Database Transactions
  - Banking Operations

- Sampling-Based Analysis
  - Monte-Carlo Methods
  - Uncertainty Quantification
  - Optimization

**Continuum**
High Performance Computing

Large-Scale simulation of Earthquake at the San Andreas fault.

In HPC we strive to optimize the computation and reduce communication costs.
High Throughput Computing

Inferring Parameters Probability Distributions.

Parameter Space

Annealing to the real Distribution
High Throughput Computing

- Perfectly (Embarrassingly) Parallel - Independent samples
- Minimal communication costs: only for the distribution of sample parameters to nodes.

In HTC we strive to maximize the number of sample evaluations per unit of time.
The Korali Framework
The Korali Framework

High-Performance / High-Throughput Uncertainty Quantification and Optimization Framework

What it does:

Solves a variety of statistical problems:
- Bayesian Inference
- Optimization
- Probability Distribution Sampling
- Surrogate Modeling

Implements a variety of solver methods:
- MCMC, TMCMC
- CMAES, rProp, DEA
- Gaussian Processes

Useful Features:

- It has a simple interface.
- It can run C++/Python/Fortran applications.
- It can also run pre-compiled applications.

Tailored for Large-Scale Computing:

- Its sampling engine is fully-parallel.
- Implements fault-tolerance mechanisms.

Installation / Tutorials / Documentation: https://www.cse-lab.ethz.ch/korali/
Korali Experiment

Represents a complete description of a problem and how to solve it.

What are we need to know?

- Sample the shape of a Probability Density Function
- Infer the parameters of a model using Bayes' Theorem
- Optimize a decision (sales strategy, weights of a NN)
Korali Experiment

Represents a complete description of a problem and how to solve it.

- Problem
- Variable 1
- Variable 2
- Variable n
- Solver
- Model

Represents the parameter space of a Sample

How many variables?
What are their Prior Distribution?
What are their lower/upper bounds?

Image Source: http://www.andysfriedman.net/AndysWebPage/BSJ/ParameterSpace.html
Korali Experiment

Represents a complete description of a problem and how to solve it.

- Problem
- Solver
- Model

Optimization/Sampling Method

Optimization: CMA-ES, Conjugate Gradient, etc...

Sampling: MCMC, TMCMC

Reinforcement Learning: (coming)
Korali Experiment

Represents a complete description of a problem and how to solve it.

Problem

Solver

Model

Experiment

Computational Model to Run

Blood Flow Simulation
Airplane Wing Drag Simulation
Heat Distribution Model
Korali uses a **descriptive** interface. Specifies the **what**, not the **how**.

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

e["Problem"]['Type'] = "Probability Sampling"
e["Problem"]['Probability Function'] = myPDF

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

e['Solver']['Type'] = "MCMC"
e['Solver']['Population Size'] = 32
e['Solver']['Burn In'] = 5

k = korali.Engine(); k.run(e)
```

**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.
Generation-Based Sampling

Step 1) **Solver** creates new set of samples $S$ to evaluate containing a value for each variable.

Step 2) **Problem** preprocesses samples, adding necessary information

Step 3) **Evaluate** samples with a computational model ($f$).

Step 4) **Post-process** samples and return derivative result $L(S)$ to the solver

Step 5) **Update internal state**, with a better approximation of the results.

Step 6) **Store results** and repeat until convergence
Optimization with Korali

**Goal:** Find the global minimum of the Ackley Function

\[
f(x, y) = -20 \exp \left[ -0.2 \sqrt{0.5 \left( x^2 + y^2 \right)} \right] \\
- \exp \left[ 0.5 \left( \cos 2\pi x + \cos 2\pi y \right) \right] + e + 20
\]

double ackley(double *x)
{
    const double a = 20, b = 0.2, c = 2\pi;

    double s1 = x[0]*x[0] + x[1]*x[1];
    double s2 = cos(c*x[0]) + cos(c*x[1]);

    return -a*exp(-b*sqrt(s1/2)) - exp(s2/2) + a + exp(1.);
}

**Variable x1:** [-40; 40]

**Variable x2:** [-40; 40]
Inferring Parameters with Korali

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?
Inferring Parameters with Korali

**Experiment**

**Problem:** Bayesian Inference

**Variables:**
(X,Y) Positions of 3 Candles = 6 Variables

**Model:**
2D Heat Equation (MPI)

**Solver:**
TMCMC

**Posterior Distributions**

Heat Source 1

Heat Source 2

Heat Source 3
Sample Scheduling
Sequential Sampling

Evaluating a single sample at a time.

Sequential Sample Scheduler
Sequential Sampling

Core usage Timeline

Samples may require different times
(Paramater-based runtime variance)

Wasted Resources
Sequential Sampling in Korali

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

```
Korali Main Process

Sample
Sample
Sample
Sample

Computational Model

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

Korali Application

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

Running Application

$ ./myKoraliApp.py
```
Concurrent Sampling

Evaluating many samples in the cores of a single node.
High Throughput Computing

Node 0 Core 0
Sample 0
Sample 8
Sample 9
Node 0 Core 1
Sample 1
Sample 10
Node 0 Core 2
Sample 2
Sample 11
Node 0 Core 3
Sample 3

Node 1 Core 0
Node 1 Core 1
Node 1 Core 2
Node 1 Core 3

Time
Concurrent Sampling in Korali

Uses fork/join to create multiple concurrent worker processes.

### 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)
```

Running Application

```
$ ./myKoraliApp.py
```
Distributed Sampling

Evaluating samples on cores from multiple (distributed) nodes.
High Throughput Computing

Core Usage Timeline

Node 0 Core 0
- Sample 0
- Sample 8

Node 0 Core 1
- Sample 1
- Sample 9

Node 0 Core 2
- Sample 2
- Sample 10

Node 0 Core 3
- Sample 3
- Sample 11

Node 1 Core 0
- Sample 4
- Sample 12

Node 1 Core 1
- Sample 5
- Sample 13

Node 1 Core 2
- Sample 6
- Sample 14

Node 1 Core 3
- Sample 7
- Sample 15

Time
Distributed Conduit

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

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

Computational Model

```python
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

```python
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)
```
Combining HPC+HTC

Study the variability and uncertainties of e.g., Earthquake Ground Motion models.

Vp peak velocity of coherent pulse, cm/s  NS
Vp peak velocity of coherent pulse, cm/s  EW
Tp period of coherent pulse, s
Nc cycles in coherent pulse
Tpk time to the peak of the pulse
phi phase angle of the pulse
Vr peak velocity of incoherent ground motion, cm/s  NS
Vr peak velocity of incoherent ground motion, cm/s  EW
Tau1 envelope rise time, s
Tau2 constant time, s
Tau3 envelope decay time, s
power spectrum central frequency, Hz
power spectrum bandwidth factor,

Credit: http://people.duke.edu/~hpgavin/groundmotions/

Image: https://www.brgm.eu/project/earthquake-simulations-can-uncertainties-be-quantified-more-accurately
Combining HPC+HTC

Core Usage Timeline

Sample 0
- Node 0 Core 0
  - Rank 0
  - Rank 1
  - Rank 2
  - Rank 3

Sample 1
- Node 1 Core 0
  - Rank 0
  - Rank 1
  - Rank 2
  - Rank 3

Node 0 Core 1
- Rank 0
- Rank 1
- Rank 2
- Rank 3

Node 0 Core 2
- Rank 0
- Rank 1
- Rank 2
- Rank 3

Node 0 Core 3
- Rank 0
- Rank 1
- Rank 2
- Rank 3

Node 1 Core 1
- Rank 0
- Rank 1
- Rank 2
- Rank 3

Node 1 Core 2
- Rank 0
- Rank 1
- Rank 2
- Rank 3

Node 1 Core 3
- Rank 0
- Rank 1
- Rank 2
- Rank 3

Time
Let's Discuss

Suppose:

We run CMAES with 16 Samples per Generation.
We have 16 nodes with 24 cores each.
We have an iterative MPI-based solver as computational model.

What is the best strategy?

1) Run 1 sample per node, with each model using 24 cores.
2) Run 1 sample at a time, with the model using all 384 cores.
3) Run 1 sample per core.
Sample Distribution Strategies
Task Distribution

How do we distribute samples to cores?
Divide-And-Conquer Strategy

Distribute samples equally (in number) among cores at the start of every generation.

Regular communication:
- Happens at the beginning of each generation.
- Message Sizes Well-known.
- Can use separate messages or a Broadcast

Only applicable when the entire workload is known from the beginning
Load Imbalance

- Happens when cores receive uneven workloads.
- Represents a waste of computational power.

Parallel Sampler - Single-Core Model

Node 0 Core 0
- Sample 0
- Sample 1
- Idle

Node 0 Core 1
- Sample 2
- Sample 3

Node 0 Core 2
- Sample 4
- Sample 5
- Idle

Node 0 Core 3
- Sample 6
- Sample 7
- Idle

Total Running Time = \( \text{Max(Core Time)} \)

Load Imbalance Ratio = \( \frac{\text{Max(Core Time)} - \text{Average(Core Time)}}{\text{Max(Core Time)}} \)
Assign workload opportunistically, as cores/work become available.

**Asynchronous Behavior:**
- Producer sends samples to workers as soon as they become available.
- Workers *report back* finished sample and its result.
- Producer keeps a queue of available workers.

Does not require the entire workload in advance.
Load Imbalance

Parallel Sampler - Single-Core Model

Node 0 Core 0
Node 0 Core 1
Node 0 Core 2
Node 0 Core 3

Total Running Time \approx \text{Mean(Core Time)}, \text{ as sample size and cores } \rightarrow \text{ Infinite}

Lost Performance \% = \frac{\#\text{ProducerCores}}{\#\text{TotalCores}}

Pop Quiz: Why do we need to 'sacrifice' one worker node?
Pop Quiz: What's the impact on large multi-core systems? (Euler = 24 cores)
Generation-Based Methods

**CMA-ES**

All samples for the next generation are known at the end of the previous generation.

**TMCMC**

Samples for the current generation are determined in real-time, based on the evaluation of previous chain steps.
Let's Discuss

Q1: Is the Divide and Conquer Strategy Good for CMA-ES? What about TMCMC?

Q2: Is the Producer/Consumer Strategy Good for CMA-ES? And for TMCMC?
Producer/Consumer in Korali

Korali Engine

Supercomputer

Experiment

Start Experiment

Run Next Generation

Engine

Generate Samples

Solver

Update State

Preprocess Samples

Problem

Postprocess Results

Distribute Samples

Conduit

Collect Results

Worker 0

Rank 0 (Core 0)

Worker 1

Rank 0 (Core 1)

Worker 2

Rank 0 (Core 2)

Worker N

Rank 0 (Core M-2)

Worker N

Rank 1 (Core M-1)

Worker N

Rank 1 (Core M-1)

Worker N

Rank 1 (Core M-1)
Importance of Load Balancing in HTC

Full extent of Load Imbalance:

With imbalance-avoiding techniques and strategies.

Average Efficiency
73.9%

Average Efficiency
97.8%
MPI and Sample Distribution: A Discussion
Two-sided Communication: A sender and a receive process explicitly participate in the exchange of a message.

A message encodes two pieces of information:
1. The actual message payload (data)
2. The fact that two ranks reached the exchange point (synchronization).

It does not encode semantics: the receiver needs to know what to do with the data.
One-sided Communication: A process can directly access a shared partition in another address space

It only encodes one piece of information: **data**.

Allows passing/receiving data without a corresponding send/recv request.

The other end is not notified of the operation (concurrency hazards)

Good for cases in which synchronization / ordering is not necessary.
A Good Case for MPI: Iterative Solvers

Structured Grid Stencil Solver
- Iteratively approaches a solution.

2D Grid

Traditional Decomposition
1 Process (Rank) per Core.

Ranks Exchange Halo (Boundary) Cells

Node

Rank 0

Rank 1

Rank 2

Rank 3
Most HPC applications are programmed under the **Bulk-Synchronous Model**.  
- Iterates among separate *computation* and *communication* phases.
A NOT so Good Case for MPI: Genome Assembly

Original DNA

- Construct a genome (chromosome) from a pool of short fragments produced by sequencers
- Analogy: shred many copies of a book, and reconstruct the book by examining the pieces
- Complications: shreds of other books may be intermixed, can also contain errors
- Chop the reads into fixed-length fragments (k-mers)
- K-mers form a De Bruijn graph, traverse the graph to construct longer sequences
- Graph is stored in a distributed hash table

Re-assembled DNA

Slide Credit: Scott B. Baden (Berkeley Lab)
Image Credit: http://people.mpi-inf.mpg.de/~sven/images/assembly.png
**A NOT so Good Case for MPI: Genome Assembly**

Build k-mer graphs from independent segments, sharing their hash numbers.

**Initial Segment of DNA:** \textbf{ACTCGATGCTCAATG}

- **Rank 0**
  - GATG->ATGC
  - ACTC->CTCG->TCGA
  - TGTC->GCTC-CTCA-TCAA
  - Hash Table for Rank 1

- **Rank 1**
  - TGCT->GCTC
  - TCAA->CAAT->AATG
  - Hash Table for Rank 0

**Detect new edge**

**Update Hash Table**

**Detect coinciding hash**

**Align K-mers**

**Completely Asynchronous:**
- Detection of coincident hashes
- Asynchronous Hash Updates

**Irregular Communication:**
- K-mer chain size can vary
- Need to allocate hash entries in real time (cannot pre-allocate)

**Difficult to implement on MPI due to its asynchronicity**
Let's Discuss

Q1: Is MPI a good model for the divide-and-conquer strategy?

Q2: Is MPI a good model for the Producer/Consumer strategy?

We need a communication model that is completely asynchronous -> e.g., UPC++