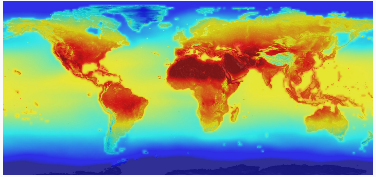
Toward Transparent and Parsimonious Methods for Automatic Performance Tuning

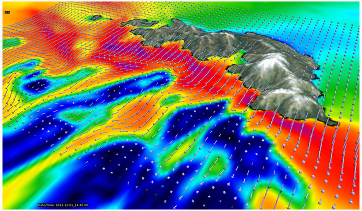
Pedro Bruel
phrb@ime.usp.br

July 9 2021

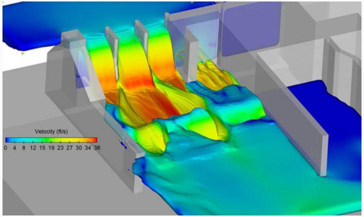
High Performance Computing is Needed at Multiple Scales, ...

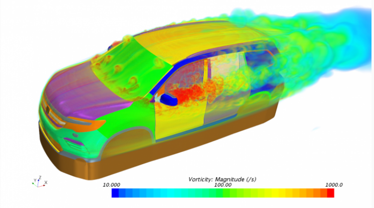
Climate simulation for policies to fight climate change



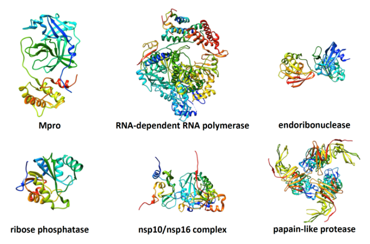


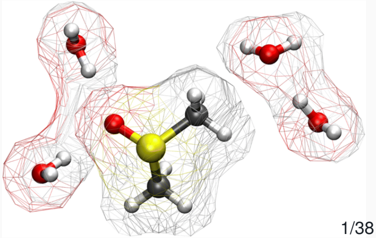
Fluid dynamics for stronger infrastructure and fuel efficiency



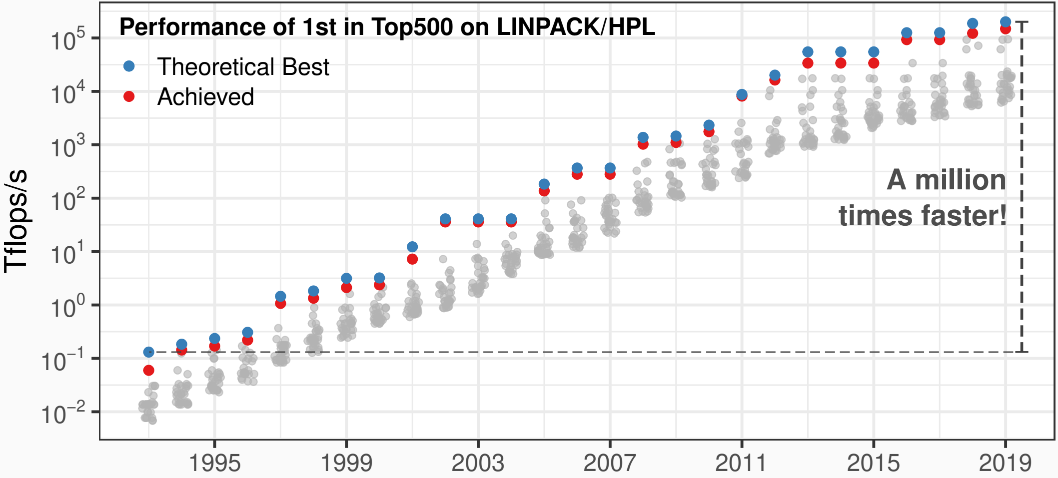


Molecular dynamics for virtual testing of drugs and vaccines

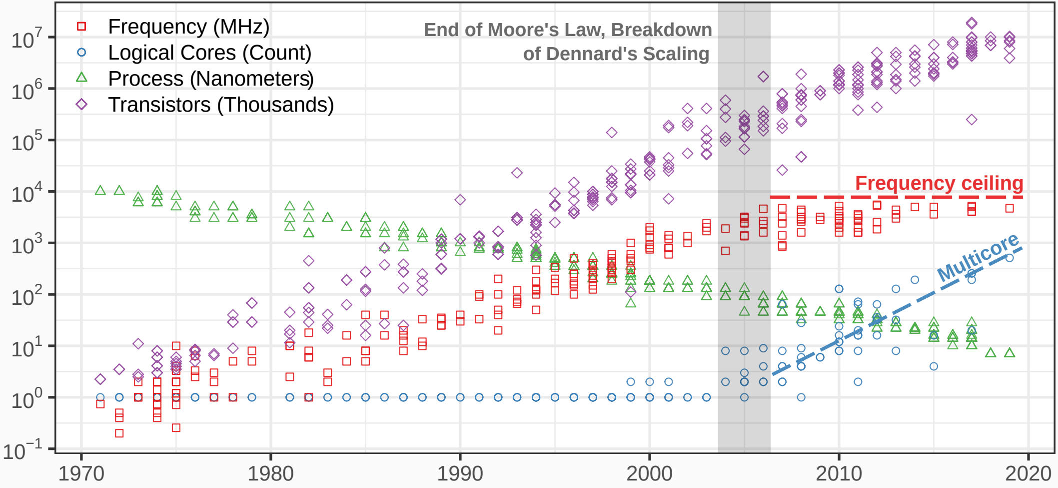




… and the Performance of Supercomputers has so far Improved Exponentially



Software must Improve to Leverage Complexity, and Autotuning can Help



An Autotuning Example: Loop *Blocking* and *Unrolling* for Matrix Multiplication

Optimizing Matrix Multiplication

How to restructure memory accesses in loops to increase throughput by leveraging cache locality?

```
int N = 256;

float A[N][N], B[N][N], C[N][N];
int i, j, k;
// Initialize A, B, C
for(i = 0; i < N; i++){ // Load A[i][]
  for(j = 0; j < N; j++){
    // Load C[i][j], B[][j] to fast memory
    for(k = 0; k < N; k++){




      C[i][j] += A[i][k] * B[k][j];
    }


    // Write C[i][j] to main memory
  }
}
```

An Autotuning Example: Loop *Blocking* and *Unrolling* for Matrix Multiplication

Optimizing Matrix Multiplication

How to restructure memory accesses in loops to increase throughput by leveraging cache locality?

```
int N = 256;
int B_size = 4;
float A[N][N], B[N][N], C[N][N];
int i, j, k, x, y;
// Initialize A, B, C
for(i = 0; i < N; i += B_size){
  for(j = 0; j < N; j += B_size){
    // Load block (i, j) of C to fast memory
    for(k = 0; k < N; k++){
      // Load block (i, k) of A to fast memory
      // Load block (k, y) of B to fast memory
      for(x = i; x < min(i + B_size, N); x++){
        for(y = j; y < min(j + B_size, N); y++){
          C[x][y] += A[x][k] * B[k][y];
        }
      }
    }
    // Write block (i, j) of C to main memory
  }
} // One parameter: B_size
```

An Autotuning Example: Loop *Blocking* and *Unrolling* for Matrix Multiplication

Optimizing Matrix Multiplication

How to restructure memory accesses in loops to increase throughput by leveraging cache locality?

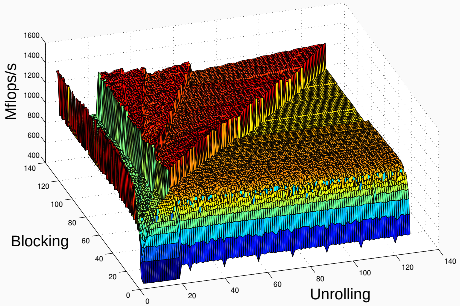
```
int N = 256;
int B_size = 4;
float A[N][N], B[N][N], C[N][N];
int i, j, k; // int U_size = 16;
// Initialize A, B, C
for(i = 0; i < N; i += B_size){
  for(j = 0; j < N; j += B_size){
    // Load block (i, j) of C to fast memory
    for(k = 0; k < N; k++){
      // Load block (i, k) of A to fast memory
      // Load block (k, y) of B to fast memory
      C[i + 0][j + 0] += A[i + 0][k] * B[k][j + 0];
      C[i + 0][j + 1] += A[i + 0][k] * B[k][j + 1];
      // Unroll the other 13 iterations
      C[i + Bsize - 1][j + B_size - 1] += A[i +
      ↪  Bsize - 1][k] * B[k][j + B_size - 1];
    }
    // Write block (i, j) of C to main memory
  }
} // Two parameters: B_size and U_size
```

An Autotuning Example: Loop *Blocking* and *Unrolling* for Matrix Multiplication

Optimizing Matrix Multiplication

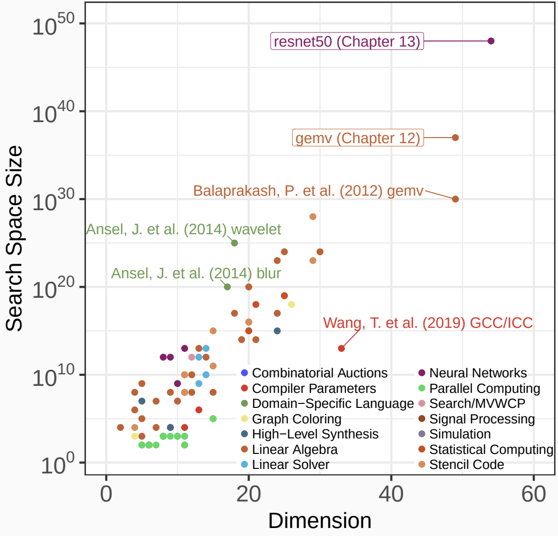
How to restructure memory accesses in loops to increase throughput by leveraging cache locality?

Resulting Search Space

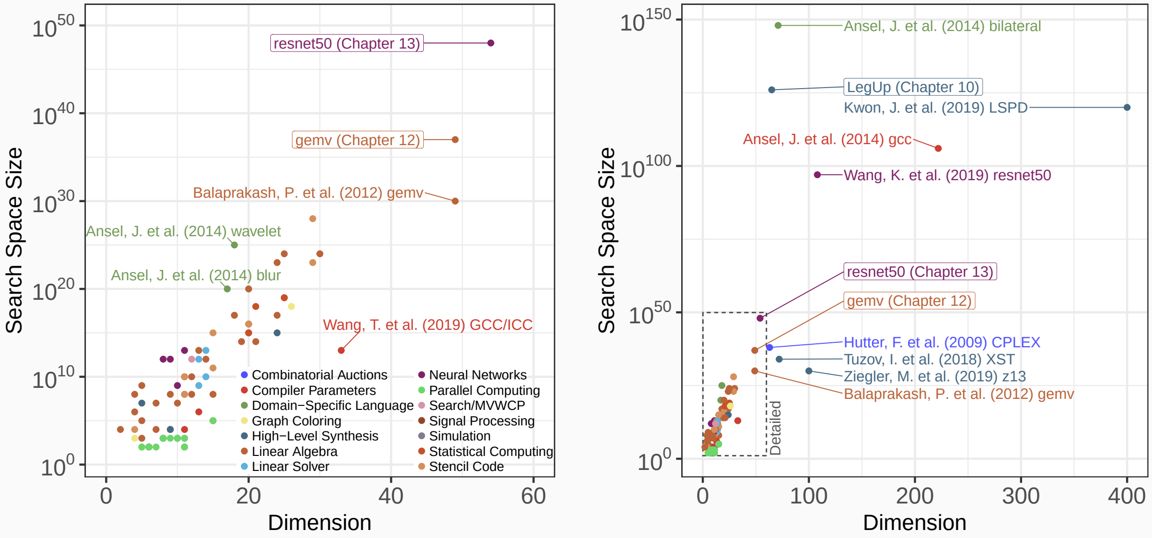



```c
int N = 256;
int B_size = 4;
float A[N][N], B[N][N], C[N][N];
int i, j, k; // int U_size = 16;
// Initialize A, B, C
for(i = 0; i < N; i += B_size){
  for(j = 0; j < N; j += B_size){
    // Load block (i, j) of C to fast memory
    for(k = 0; k < N; k++){
      // Load block (i, k) of A to fast memory
      // Load block (k, y) of B to fast memory
      C[i + 0][j + 0] += A[i + 0][k] * B[k][j + 0];
      C[i + 0][j + 1] += A[i + 0][k] * B[k][j + 1];
      // Unroll the other 13 iterations
      C[i + Bsize - 1][j + B_size - 1] += A[i +
       ↪  Bsize - 1][k] * B[k][j + B_size - 1];
    }
    // Write block (i, j) of C to main memory
  }
} // Two parameters: B_size and U_size
```

Autotuning Problems in Other Domains: Dimension Becomes an Issue



Autotuning Problems in Other Domains: Dimension Becomes an Issue



Autotuning as an *Optimization* or *Learning* Problem

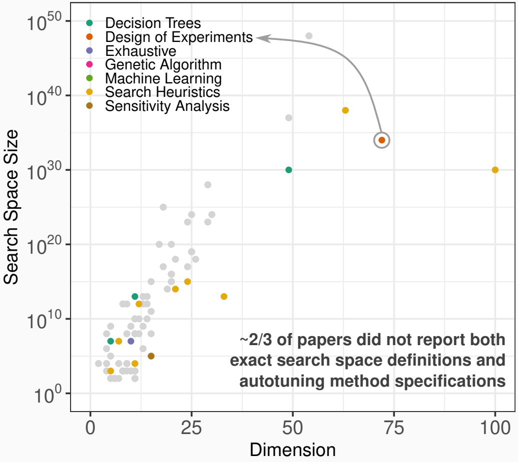
Performance as a Function

Performance: $f : \mathcal{X} \to \mathbb{R}$

- Parameters: $\mathbf{x} = [x_1 \ \ldots \ x_n]^\mathsf{T} \in \mathcal{X}$
- Performance metric: $y = f(\mathbf{x})$

To minimize f, we can adapt proven methods from other domains:

- **Function minimization**, **Learning**: not necessarily parsimonious and transparent
- **Design of Experiments**: can help, but not widely used for autotuning



Toward Transparent and Parsimonious Autotuning

Contributions of this Thesis

- Developing transparent and parsimonious autotuning methods based on the Design of Experiments
- Evaluating different autotuning methods in different HPC domains

Transparent

- Use statistics to justify code optimization choices
- Learn about the search space

Parsimonious

- Carefully choose which experiments to run
- Minimize f using as few measurements as possible

Domain	Method
CUDA compiler parameters	F, D
FPGA compiler parameters	F
OpenCL Laplacian Kernel	F, L, D
SPAPT Kernels	L, D
CNN Quantization	L, D

F: Function Minimization, L: Learning,
D: Design of Experiments

Toward Transparent and Parsimonious Autotuning

Contributions of this Thesis

- Developing transparent and parsimonious autotuning methods based on the Design of Experiments
- Evaluating different autotuning methods in different HPC domains

Transparent

- Use statistics to justify code optimization choices
- Learn about the search space

Parsimonious

- Carefully choose which experiments to run
- Minimize f using as few measurements as possible

Domain	Method
CUDA compiler parameters	F, D
FPGA compiler parameters	F
OpenCL Laplacian Kernel	F, L, D
SPAPT Kernels	L, D
CNN Quantization	L, D

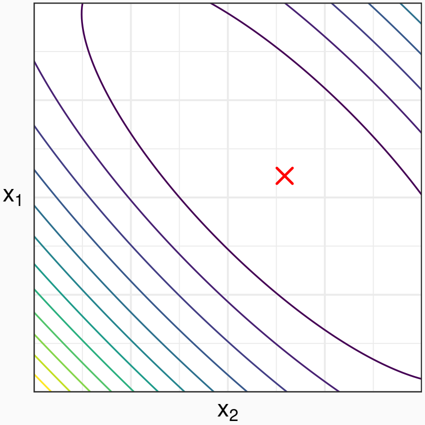
F: Function Minimization, L: Learning,
D: Design of Experiments
: In this presentation

Autotuning with Generic Methods for Function Minimization

Minimizing Functions using Derivatives and Other Information

We know or can compute information about f

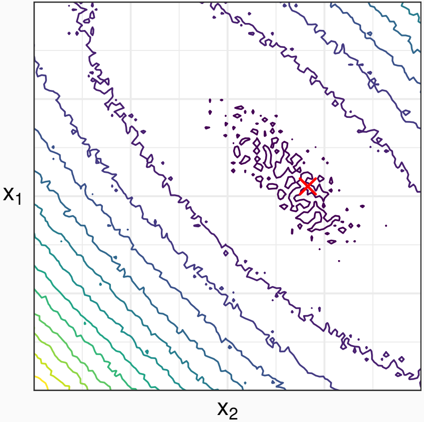
- Directly measure new $\mathbf{x}_1, \cdots, \mathbf{x}_k, \cdots, \mathbf{x}_n$
- Search for the global optimum, try to escape local optima



Minimizing Functions using Derivatives and Other Information

We know or can compute information about f

- Directly measure new $\mathbf{x}_1, \cdots, \mathbf{x}_k, \cdots, \mathbf{x}_n$
- Search for the global optimum, try to escape local optima



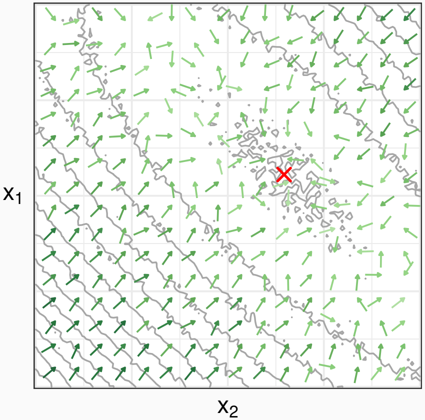
Minimizing Functions using Derivatives and Other Information

We know or can compute information about f

- Directly measure new $\mathbf{x}_1, \cdots, \mathbf{x}_k, \cdots, \mathbf{x}_n$
- Search for the global optimum, try to escape local optima

Strong Hypothesis: we can Compute Derivatives

- $\mathbf{x}_k = \mathbf{x}_{k-1} - \mathbf{H}f(\mathbf{x}_{k-1})\nabla f(\mathbf{x}_{k-1})$
- Locally and globally



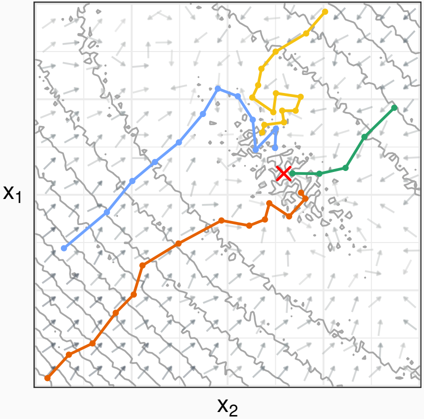
Minimizing Functions using Derivatives and Other Information

We know or can compute information about f

- Directly measure new $\mathbf{x}_1, \cdots, \mathbf{x}_k, \cdots, \mathbf{x}_n$
- Search for the global optimum, try to escape local optima

Strong Hypothesis: we can Compute Derivatives

- $\mathbf{x}_k = \mathbf{x}_{k-1} - \mathbf{H}f(\mathbf{x}_{k-1})\nabla f(\mathbf{x}_{k-1})$
- Locally and globally



Minimizing Functions using Derivatives and Other Information

We know or can compute information about f

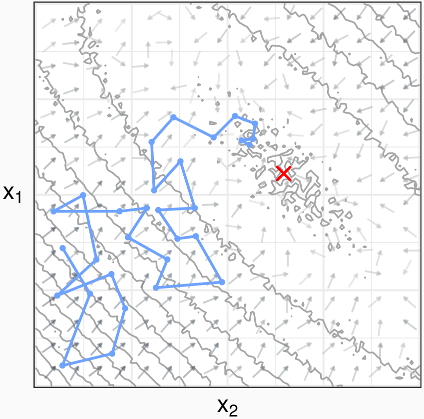
- Directly measure new $\mathbf{x}_1, \cdots, \mathbf{x}_k, \cdots, \mathbf{x}_n$
- Search for the global optimum, try to escape local optima

Strong Hypothesis: we can Compute Derivatives

- $\mathbf{x}_k = \mathbf{x}_{k-1} - \mathbf{H}f(\mathbf{x}_{k-1})\nabla f(\mathbf{x}_{k-1})$
- Locally and globally

Hard to State Hypotheses: Search Heuristics

- Random Walk, Simulated Annealing, Genetic Algorithms, Nelder-Mead, and many others



Search Heuristics with Multi-Armed Bandit

- OpenTuner
- Ensemble of search heuristics
  - Coordinated by a MAB algorithm (online learning)

Application: High-Level Synthesis for FPGAs

Search Space and Performance Metrics

Results

Discussion

- Function minimization methods are not parsimonious
- Curse of dimensionality
- It is often unclear:
  - if there is something to find, and when to stop exploring
- It is impossible to:
  - interpret optimizations

Applying Sequential Design of Experiments

Linear Models

- Learning: Building surrogates, used for optimization
- Introduce notation:
    - $\hat{f}_\theta : \mathcal{X} \to \mathbb{R}$
    - Model of f: $f(\mathbf{x}) = \mathbf{x}^\mathsf{T}\theta + \varepsilon$
    - Surrogate $\hat{f}_\theta(\mathbf{x}) = \mathbf{x}^\mathsf{T}\hat{\theta}$.
- Best Linear Unbiased Estimator
- Learning methods assume the design $\mathbf{X}$ is given

Design of Experiments

- Statistical methods to choose the design $\mathbf{X}$ to minimize surrogate model variance
- Notation:
  - Factors, levels, design
- Simple linear model example for 2-factor designs
- Factorial designs , screening

Optimal Design

- Distributing points according to initial modeling hypotheses decreases model matrix determinant (associated with variance)
- Good for exploiting known search space structure, or verifying existing hypotheses

Space-filling Designs

- Curse of dimensionality for sampling:
    - Most sampled points will be on the "shell"
- LHS: Partition and then sample, need to optimize later
- Low-discrepancy: deterministic space-filling sequences

Interpreting Significance

- ANOVA for Linear Models
    - Isolate "significant" factors
- Sobol indices
    - expensive computation

A Transparent and Parsimonious Approach to Autotuning

- Explain paper diagram

Application: GPU Laplacian

Search Space and Performance Metric

Results

- Comparison with multiple methods
- Leave GPR for later

Interpreting the Optimization

Application: SPAPT kernels

- Pick one?

Search Spaces and Performance Metric

Results

- Is there anything to find?
- Leave GPR for later

Interpreting the Optimization

Discussion

- Sequential and incremental
  - Definitive restrictions
  - Improvements by batch
  - Low model flexibility (rigid models)
- Motivating results in the Laplacian kernel
- It is possible to interpret results, guide optimization
  - sometimes simpler models give better results
- For SPAPT kernels, it is still unclear:
  - if there is something to find, and when to stop exploring
  - is there a global optimum, is it "hidden"?
    - how to find it, if so? (can learning do it?)
- Random Sampling has good performance
  - Abundance of local optima?
- What is the most effective level of abstraction for optimizing a program?
  - Compiler, kernel, machine, model, dependencies?

Active Learning with Gaussian Processes

More Flexibility with Gaussian Process Regression

- Introduce notation:
  - Model of f: $f(\mathbf{x}) \sim \mathcal{N}(\boldsymbol{\mu}, \boldsymbol{\Sigma})$
  - Surrogate $\hat{f}_\theta(\mathbf{x}) \sim f(\mathbf{x}) \mid \mathbf{X}, \mathbf{y}$

Expected Improvement: Balancing Exploitation and Exploration

- How to decide where to measure next?

Application: GPU Laplacian and SPAPT

- GPR was applied to these problems too

Search Spaces and Performance Metrics

Results: GPU Laplacian

- GPR is good too, but the simpler model is more consistent

Results: SPAPT

- GPR still can’t find better configurations

Application: Quantization for Convolutional Neural Networks

Search Space, Constraints, and Performance Metrics

- Comparing with a Reinforcement Learning approach in the original paper
- ImageNet

Results

Interpreting the Optimization

- Sobol indices, inconclusive

Discussion

- Low-discrepancy sampling in high dimension
- Constraints complicate exploration
- Multi-objective optimization
- A more complex method usually produces less interpretable results, but not always achieves better optimizations

Conclusion

Contributions of this Thesis

- Striving to develop and apply transparent and parsimonious autotuning methods
- Applications in this thesis:

Domain	Method
CUDA compiler parameters	Function minimization methods with Online Learning
FPGA compiler parameters	Function minimization methods with Online Learning
OpenCL Laplacian Kernel	Function minimization methods, Linear Models, Gaussian Process Regression
SPAPT Kernels	Linear Models, Gaussian Process Regression
CNN Mixed-Precision Quantization	Gaussian Process Regression

Reproducibility of Performance Tuning Experiments

- Redoing all the work for different problems
- Complementary approaches:
    - Completely evaluate small sets of a search space
    - Collaborative optimizing for different architectures, problems

Key Discussions

Curse of Dimensionality for Autotuning Problems

- Implications for Sampling and Learning
- Space-filling helps, but does not solve
- Constraints

Which method to use?

- Design of Experiments for transparency and parsimony when building and interpreting statistical models
- Linear models for simpler spaces and problems
- Gaussian Process Surrogates for more complex situations

It is often unclear if there is something to find

- Abundance of local optima
- Is there a global optimum, is it "hidden"?
    - How to find it, if so? (can learning do it?)
- What is the most effective level of abstraction for optimizing a program?
    - Compiler, kernel, machine, model, dependencies?
- When to stop?

Conclusion

Toward Transparent and Parsimonious Methods for Automatic Performance Tuning

Pedro Bruel
phrb@ime.usp.br

July 9 2021