ParaOpt: Automated Application Parameterization and Optimization for the Cloud

Chaofeng Wu*, Ted Summer*, Zhuozhao Li*, Anna Woodard*, Ryan Chard†,
Matt Baughman*, Yadu Babuji*, Kyle Chard*, Jason Pitt‡, Ian Foster*†
*Department of Computer Science, University of Chicago, Chicago, IL, USA
†Data Science and Learning Division, Argonne National Laboratory, Argonne, IL, USA
‡Cancer Science Institute of Singapore, National University of Singapore, Singapore

***Abstract*—The variety of instance types available on cloud platforms offers enormous flexibility to match the requirements of applications with available resources. However, selecting the most suitable instance type and configuring an application to optimally execute on that instance type can be complicated and time-consuming. For example, application parallelism flags must match available cores and problem sizes must be tuned to match available memory. As the search space of application configurations can be enormous, we propose an automated approach, called ParaOpt, to automatically explore and tune application configurations on arbitrary cloud instances. ParaOpt supports arbitrary applications, enables use of custom optimization methods, and can be configured with different optimization targets such as runtime and cost. We evaluate ParaOpt by optimizing genomics, molecular dynamics, and machine learning applications with four types of optimizers. We show with as few as 15 parameterized executions of an application, representing between 1.2%-26.7% of the search space, that ParaOpt is able to identify the optimal configuration in 32.7% of experiments and a near-optimal configuration in 83.2% of cases. As a result of using near-optimal configurations, ParaOpt reduces overall execution time by up to 85.8% when compared with using the default configuration.**

I. Introduction

Traditionally users have meticulously optimized software to work with particular hardware. However, increasing hardware heterogeneity and availability of cloud infrastructure now allow users to customize computing environments for any problem. While this flexibility can improve efficiency and decrease costs, it also creates new challenges for users, not least of which is the enormous search space of possible software and hardware configurations. Utilizing default application configurations or selecting suboptimal configurations may result in increased execution time [1] and costs [2].

This configuration optimization problem is particularly challenging when considering the myriad choices available to cloud users. For example, at the time of writing Amazon Web Services (AWS) offers over 200 distinct virtual server configurations, referred to as instance types, ranging in cost from 0.5 cents per hour to more than $30 per hour and with resources ranging from one to hundreds of CPUs and from gigabytes to terabytes of memory. Application performance may vary significantly depending on the configuration of the application and on which instance type it is executed. Applications may have hundreds of parameters that directly affect performance—far too many for a user to manually explore on even a single instance type.

In this paper we present ParaOpt, a system that automates the optimization of application configurations for cloud instances subject to user-specified optimization targets, such as time or cost. ParaOpt is implemented as a service which manages the execution of *experiments*. Each experiment consists of multiple *trials*, each of which aims to explore a unique combination of application parameters on a selected instance type. ParaOpt applies a user-specified optimization strategy to determine the parameter combination for each trial. ParaOpt offers an extensible interface via which arbitrary optimization strategies can be applied.

We evaluate ParaOpt by optimizing genomics, molecular dynamics, and machine learning applications on various cloud instance types using four optimization strategies. Using ParaOpt's Bayesian optimizer strategy, we find the optimal configurations for 55.8% of experiments with two parameters and near-optimal, within 10% of optimal, for 77.5% of experiments with three and four parameters. By using ParaOpt-identified configurations we reduce application execution time by up to 85.8% over the default configuration.

The remainder of this paper is organized as follows. In §II we formulate the optimization problem. In §III we outline example applications. In §IV we present the ParaOpt system architecture. In §VI we evaluate the performance of ParaOpt. Finally, in §VII we discuss related work and in §VIII we summarize our contributions.

II. Requirements and Problem Formulation

In this section we describe the key requirements for optimizing application configurations and formulate the problem.

A. Requirements

We identify the following requirements as necessary to establish a flexible and robust optimization platform.

Application agnostic. Applications may have a wide range of behaviors, dependencies, runtimes, and configurations.

Scale to large search spaces. Applications may offer many configurable parameters, each of which can affect runtime and accuracy. For example, some bioinformatics applications have more than 50 parameters, creating a search space with millions of potential configurations.

2330-2186/19/$31.00 ©2019 IEEE
DOI 10.1109/CloudCom.2019.00045

Support custom optimization objectives. Users may have various priorities and constraints which define the optimal application configuration. For example, they may have a limited budget to perform the computation, making cost the priority, or they may have a strict deadline, making run time more important.

Support custom optimizers. Users may want to select specific optimization strategies when optimizing configuration for a particular application.

Resource agnostic. Users may wish to use different cloud provider based on cost, performance, funding support, and/or collaboration needs. Cloud providers offer different interfaces and instance types.

Extensible to other information. The optimal configuration for an application is often dependent on other features, such as the data being processed. For example, optimally configuring an application to analyze a small dataset may result in a small buffer size. Using the same configuration to analyze a large dataset could result in thrashing that would incur execution overheads. Therefore, ParaOpt must be extensible to consider external factors, such as data size, when optimizing configurations.

B. Problem formulation

We define a configuration $\vec{x}$ as a multi-dimensional vector that represents the parameters one may want to optimize. A user would like to find an optimal configuration for an application that yields some user-specific constraints (i.e., runtime, cost, or quality of results).

We denote the runtime, the monetary cost, and the quality of results as $T(\vec{x})$, $M(\vec{x})$, and $Q(\vec{x})$, $\vec{x} \in \mathbf{X}$, respectively, where $\mathbf{X}$ is the full configuration space. To evaluate the configuration $\vec{x}$, we define the utility function $U(T(\vec{x}), M(\vec{x}), Q(\vec{x}))$ as a function of both runtime and monetary cost. The optimization objective is to maximize the utility, while ensuring that the runtime and monetary cost are no more than the maximum acceptable runtime T_{max} and monetary cost M_{max}, and the quality of results is higher than the minimum acceptable quality Q_{min}. Thus, the problem can be formulated as follows:

$$\begin{aligned} \max_{\vec{x} \in \mathbf{X}} \quad & U(T(\vec{x}), M(\vec{x}), Q(\vec{x})), \\ \text{s.t.} \quad & T(\vec{x}) \le T_{max}, M(\vec{x}) \le M_{max}, Q(\vec{x}) \ge Q_{min}. \end{aligned} \tag{1}$$

III. Application Use Cases

To explore the generalizability of ParaOpt we focus on three distinct application domains.

A. Variant Calling

A variant caller is a bioinformatics application used to detect mutations in DNA sequencing data—an important step in identifying genetic causes of disease. Researchers have developed a large number of variant callers, each using distinct algorithms with different accuracy and performance characteristics [3]. Here we study three common variant callers: Platypus [4], Strelka2 [3], and GATK3 [5].

Typically, variant callers offer a number of configurable parameters that influence result accuracy and algorithm performance. For example, GATK3 offers more than 50 parameters. To better understand the performance of these variant callers we conducted preliminary profiling experiments on a c5.2xlarge instance from AWS. Using aligned DNA sequencing data, we varied some selected parameters (e.g., the number of CPU cores, buffer size, and memory size) of these callers. We found that the optimal configuration decreased execution time by 86% and 47% when compared with the default configuration for Platypus and GATK3, respectively.

B. Molecular Dynamics

Large-scale Atomic/Molecular Massively Parallel Simulator [6] (LAMMPS) is a well-known application for molecular dynamics simulations. It can model ensembles of particles in a liquid, solid, or gaseous state using a variety of interatomic potentials and boundary conditions. LAMMPS is designed to be efficiently executed on single processor computers and on parallel computers using MPI [7]. LAMMPS also uses the Kokkos [8] library to enable execution on accelerator architectures.

To ensure efficient parallel execution of LAMMPS, the MPI and Kokkos-related parameters must be set appropriately. We conducted a preliminary experiment by using LAMMPS to carry out a stochastic rotation dynamics simulation of a 2D rigid box of particles. We ran LAMMPS on a c5.2xlarge instance and varied the number of MPI processes and the number of Kokkos threads. Our results indicate that a near-optimal configuration can decrease the runtime by 60× when compared with the worst configuration.

C. Machine Learning

TensorFlow [9] and Keras [10] are two of the most commonly used machine learning libraries. Both libraries offer a huge number of configurable parameters such as the number of threads, GPU memory per process, and batch size.

In a preliminary experiment we explored the training of two convolutional neural networks (CNNs) using CIFAR10 [11] and MNIST [12] datasets, with TensorFlow and Keras, respectively. We ran the training on a c5.2xlarge instance and varied the number of CPU processes, number of CPU threads, blocking time of threads, and batch size. By tuning two to four parameters we could decrease runtime with TensorFlow by 79% and 25% over the worst and default configurations, and with Keras by 99% and 66% over the worst and default configurations.

IV. Using ParaOpt

Users interact with ParaOpt by defining an *experiment*. The experiment includes the application name, the parameters that can be modified, the environment in which to perform the experiment, and a template command line invocation to run the application. An example experiment definition is given in Listing 1. Parameter definitions must specify the name of the parameter and its type (e.g., int). Users may optionally specify the bounds of continuous or discrete values. The compute environment section of the experiment definition determines how the experiments will be executed. This includes the instance type to explore as well as the Amazon Machine Image (AMI) to be used. The AMI provides a base image which must contain the application and any dependencies.

Listing 1: An example experiment configuration.

```
{
  "tool_name": "strelka_c5.2x_grid",
  "parameters": [
    {
      "maximum": "8",
      "minimum": "1",
      "name": "nCPUs",
      "type": "int"
    },
    {
      "maximum": "16",
      "minimum": "2",
      "name": "totalMem",
      "type": "int"
    },
  ],
  "compute": {
    "ami": "ami-048fd10984c33048b",
    "instance_model": "c5.2xlarge",
    "type": "ec2"
  },
  "command_template_string": "
    bam=\"data/C440.TCGA-BR.4_gdc_realn.bam\"
    ref=\"data/GRCh38.d1.vd1.fa\"
    strelka/bin/configureStrelkaGermlineWorkflow.py --ref
         ${ref} --bam ${bam} --runDir res
    res/runWorkflow.py -m local -j ${nCPUs} -g ${totalMem}"
}
```

When defining an experiment users may also specify the optimization strategy to be applied. The experiment includes the type of optimizer (e.g., Bayesian, Grid, Random, Coordinate, or user-defined) and any optimizer-specific arguments. The optimizer configuration also states how many configurations should be explored for each parameter. The example shown in Listing 2 defines an experiment that uses Grid search to search eight nCPUs values and fifteen totalMem values.

Listing 2: An example optimizer configuration.

```
{
  "optimizer":
  {
    num_configs_per_param: [8, 15]
    type: "grid"
  }
}
```

ParaOpt creates a set of *trials* for each experiment. A trial is a fully parameterized invocation of the application. ParaOpt automatically selects appropriate values for each configurable parameter, deploys the trial and monitors its execution. Once the trial execution completes, ParaOpt records the application runtime, selected parameters, and information about the resource used for execution. The results of an experiment are stored in the ParaOpt database to be returned to the user and consumed by ParaOpt to inform subsequent trials. These results are then made available to the user through the Web interface. Results are used to bootstrap new experiments and for guiding experiments on different resource types. ParaOpt can also be used for online profiling. In this mode users can execute production workloads using ParaOpt to select appropriate configuration parameters.

ParaOpt defines a modular objective function interface that allows users to implement a Python function that wraps the trial execution and returns a value for that trial (e.g., time, cost, or accuracy). Users can extend this model by registering their own objective functions when initializing ParaOpt.

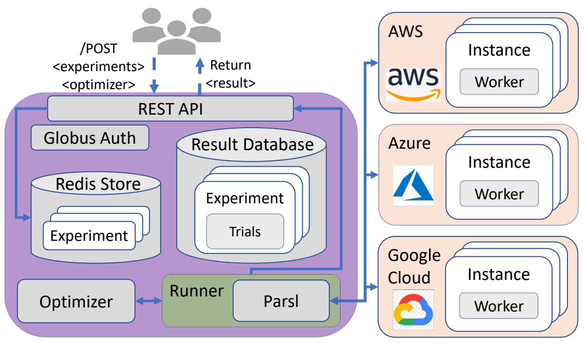


Fig. 1: System architecture

V. Architecture and Implementation

Fig. 1 depicts the high-level architecture of ParaOpt. We implement ParaOpt as a set of modular microservices. The main components of ParaOpt are the *Redis Store*, *Result Database*, *Frontend Service*, *Optimizer*, *Runner*, and *Worker*. ParaOpt uses Docker-compose to dockerize all system components (except the Worker). This enables reliable and scalable deployment of the dynamic components of the system and also simplifies deployment by others.

A. Interfaces

The Frontend Service is the user-facing interface to ParaOpt. It provides a RESTful API for users to submit experiments and monitor the status of experiments. The Frontend Service parses the experiment description and creates a unique record in the Redis Store. This record is updated as the experiment progresses and users may at any time inspect the status of the experiment via the Frontend Service. We provide a Python Software Development Kit (SDK) and Command Line Interface (CLI) to enable programmatic interaction with the ParaOpt service. Listing 3 shows an example of using the CLI to submit a new experiment to ParaOpt.

Listing 3: Example experiment submission via the CLI.

```
$$ paraopt --experiment target_experiment_file \
         --optimizer target_optimizer_file
```

The Frontend Service is secured using Globus Auth [13]. Users can authenticate with ParaOpt using one of hundreds of supported identity providers, such as their institutional identities or Google accounts. The Frontend Service implements an OAuth 2 authentication flow in which users are redirected to authenticate with their selected identity provider. After successful authentication, the Frontend Service retrieves an access token that is used to obtain information about the user.

B. Redis Store and Result Database

ParaOpt uses a Redis database to store information about experiments and trials. Specifically, when an experiment is submitted, ParaOpt creates a record for the experiment including the application name, parameters to be explored, range of values to explore for each parameter, resource information such as instance type, command line template for parametrized invocation of the application, and information about the optimizer. As ParaOpt divides up the experiment into trials, it stores a record for each trial in the Redis Store thereby ensuring that trials are not repeated and that the status of the experiment can be tracked and retrieved by users.

ParaOpt also maintains a separate Result Database using AWS RDS. The database records the results of each trial, including the application name, instance type, parameter configuration, and runtime. ParaOpt uses this database to select trials and to bootstrap exploration of the search space. Users may also retrieve information from the database to infer the optimal parameter configurations from prior experiments.

C. Runner and Worker

ParaOpt uses a Runner to manage the execution of trials in an experiment. The Runner is responsible for acquiring the appropriate instance type, submitting the trial for execution on that instance, and recording the result of the trial. We build the Runner on Parsl [14], a parallel scripting library for Python. Parsl's provider interface abstracts the complexity of using a wide range of clouds, clusters, and supercomputers, and therefore enables ParaOpt to dynamically provision resources from almost any computing system. Currently, Parsl supports three major cloud providers: Amazon AWS, Microsoft Azure, and Google Cloud.

Once the Runner is notified of an experiment, it first determines the instance type to be provisioned from the experiment description. Using Parsl it provisions an instance of that type from the specific cloud provider and deploys a Parsl Worker on that instance to manage the execution of the trials. ParaOpt decomposes the experiment into a set of trials, each a parameterized execution of the application. When the Worker is idle it requests a single trial from the Runner. The Worker then executes the trial and returns the result (or the exception on failure) to the Runner. The worker also monitors and records resource usage throughout the trial. The experiment concludes when the stopping condition is met. The results of each trial are stored in the RDS database.

D. Optimizer

The Optimizer evaluates prior executions and selects candidate values for each parameter for the next trial. We have designed the Optimizer module to be flexible and extensible such that different optimization strategies can be selected and new strategies can be added. ParaOpt currently supports four optimization strategies: Random search [15], Grid search, Coordinate search [16], and Bayesian optimization (BO) [17],

Random search randomly selects a value for each parameter. Although Random search is a simple strategy, it can be efficient in some cases such as hyper-parameter tuning in machine learning [15].

Grid search exhaustively searches the full configuration space. The user selects the number of steps for each parameter and the optimizer will search the Cartesian product of these parameters. This strategy is useful for those applications that have a relatively small search space.

Coordinate search optimizes each parameter independently. It iteratively selects each coordinate (i.e., parameter) in turn and optimizes that coordinate with the remaining parameters kept fixed.

Bayesian optimization (BO) is a technique used to find the global optima of a function whose evaluation is expensive. Generally, the objective function (e.g., utility) does not have a mathematical expression. BO creates a surrogate model (e.g., Gaussian Process) for the objective function. For each trial, a sample point of the objective function is evaluated and the surrogate model is updated according to the sampled point. Based on the surrogate model, BO can predict the value of any given point, as well as the uncertainty. BO defines an acquisition function, which is a function of the prediction value and uncertainty. High prediction value and high uncertainty both correspond to high acquisition function values. At each step, the point which maximizes the acquisition function is chosen as the next point to sample. We build BO based on an existing package [18], leveraging Gaussian Process as the surrogate model and upper confidence bound (UCB) [19] as the acquisition function. We have also extended this package to support discrete integer values. ParaOpt can support other acquisition functions.

The optimizer will terminate upon satisfaction of the stopping condition. Currently the stopping condition can be defined as follows: the experiment reaches a timeout, maximum number of trials, maximum budget, or the increased performance does not exceed a threshold.

VI. Evaluation

We evaluate the ability of ParaOpt to identify optimal configurations for real applications on different cloud instances.

A. Experimental Setup

Applications. Due to limited budget, we select only a few parameters for each application to show the effectiveness of ParaOpt. Specifically, we perform experiments using Platypus with two and three parameters (**P-2D** and **P-3D**), Strelka2 with two parameters (**S-2D**), GATK3 with two parameters (**G-2D**), LAMMPS with Kokkos with two parameters (**L-2D**), TensorFlow with two and three parameters (**TF-2D** and **TF-3D**), Keras with two, three, and four parameters (**K-2D**, **K-3D**, and **K-4D**). Table I summarizes the the parameters explored and the size of the resulting search space for each experiment.

For the variant callers (P-2D, P-3D, S-2D, and G-2D), we used DNA sequencing data for a Stomach Adenocarcinoma sample [20] from The Cancer Genome Atlas as the input dataset and with GRCh38.d1.vd1 as the reference genome

[21]. For LAMMPS (L-2D), we carry out a stochastic rotational dynamics simulation of a 2D rigid box of aspherical particles in coarse-grained solvent, including self-diffusion and viscosity [22]. For TensorFlow (TF-2D and TF-3D), we modify the convolutional neural network from the TensorFlow tutorial [23] to classify the CIFAR10 [11] dataset. For Keras (K-2D, K-3D, and K-4D), we modify the CNN model from the Keras example [24] to classify the MNIST dataset [12].

TABLE I: Experiment summary showing the parameters explored and the size of the search space.

Experiment	Parameter range [min, max, total steps]	Search space
P-2D	*nCPU* [1, 8, 8], *bufferSize* [10000, 100000, 10]	80
P-3D	P-2D, *minFlank* [1, 40, 10]	800
S-2D	*JOBS* [1, 8, 8], *MEMGB* [2, 16, 15]	120
G-2D	*MEM* [512, 16384, 7], *nct* [1, 8, 8]	56
L-2D	*NP* [1, 8, 8], *KT* [1, 8, 8]	64
TF-2D	*inter_threads* [0, 8, 9], *intra_threads* [0, 40, 12]	108
TF-3D	TF-2D, *kmp_bt* [0, 100, 11]	1188
K-2D	*intra_threads* [0, 40, 7], *batch_size* [16, 2048, 8]	56
K-3D	K-2D, *inter_threads* [0, 8, 4]	224
K-4D	K-3D, *kmp_bt* [0, 60, 5]	1120

ParaOpt deployment. We deploy the Frontend Service, Runner, and Optimizer using docker on an AWS t2.micro instance, which has 1 CPU and 1 GB memory. Workers are deployed on instances from the C5 family (optimized for compute-intensive workloads). We use c5.large, c5.xlarge, and c5.2xlarge instance types, with 2, 4, and 8 cores as well as 4, 8, and 16 GB RAM per instance, respectively.

Optimizers. We apply the Random search, Grid search, Coordinate search, and Bayesian optimization optimizers to each experiment. We use Grid search to identify the optimal configuration and Random search and Coordinate search are used as baselines for comparison. For Bayesian optimization, we set the number of initial trials to 3, and use upper confidence bound (UCB) [19] as the acquisition function. Note that our aim is not to compare the performance of various strategies, but to demonstrate ParaOpt's ability to apply different optimizers.

Metrics. We use two metrics, utility and cost normalized to the optimal (CNO), to evaluate performance. We define utility as:

$$U(\vec{x}) = \frac{1}{C(\vec{x})}, \tag{2}$$

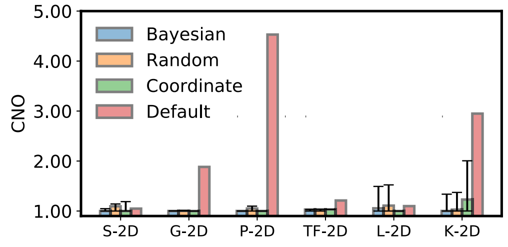
where $C(\vec{x})$ is the cost (e.g., runtime or monetary cost) of a given configuration $\vec{x}$. We define CNO as:

$$CNO = \frac{C(\vec{x})}{C(\vec{x}^{*})}, \tag{3}$$

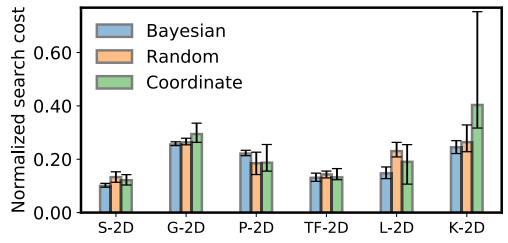
where $\vec{x}$ is the best configuration found by ParaOpt and $\vec{x}^{*}$ is the optimal configuration of the entire search space (attained from the Grid search). We configure the optimizer to maximize the utility (thus minimize the cost). Generally, the smaller the CNO, the better the configuration. The best CNO is 1, which means the current best configuration is optimal. We also report the *search cost*, which is the aggregate runtime of an experiment using different parameter configurations.

B. Optimization Strategy

We first evaluate the impact of the optimization strategy on ParaOpt performance. We select runtime minimization as the optimization objective. We limit the number of trials to 15. The Bayesian optimizer starts with 3 initial random trials, and then performs 12 consecutive trials on different parameter configurations (in total 15 trials). We repeat each experiment at least 20 times and collect all results. All experiments were conducted on c5.2xlarge instances.



(a) CNO.



(b) Normalized search cost.

Fig. 2: Applications with various optimizers on ParaOpt. Error bars show the 10th and 90th percentiles of CNO.

Fig. 2a shows the median CNO of each application for the default configuration and when using the Random, Coordinate, and Bayesian optimizer. We observe the best performance from the Bayesian optimizer for all applications except L-2D. To understand this result, we investigated the grid search on L-2D and found that the number of Kokkos threads *KT* is monotonic in runtime regardless of the number of MPI processes *NP*. This effectively reduces the search space to one dimension for the Coordinate optimizer. Because almost 70% of the configurations exceeded the timeout, a large part of the search space gives a low value in the surrogate model. The low values near the optimal will decrease both the predicted value and the uncertainty for the optimal point. As a result, the Bayesian optimizer is less likely to select the optimal point. For S-2D and G-2D, the Random, Coordinate, and Bayesian optimizers all achieve a similar CNO. This is because for both applications, the majority of configurations result in near-optimal performance (within 20% of optimal), while the optimal performance is achieved with only a few configurations.

In summary, after 15 trials, ParaOpt is able to identify the optimal configuration in 55.8% of 2D experiments and a near-optimal (within 10% of optimal) configuration in 88.3% of 2D experiments. On average, ParaOpt reduces runtime by 33.2% compared with using the default configuration.

Fig. 2b shows the median search cost (normalized to the search cost of performing a grid search) of each application. Using the default configuration has no search cost. Since the Bayesian optimizer tends to search towards the space that has the highest confidence to be the optimal configuration (hence a higher confidence to have the lowest runtime), it is more likely (not guaranteed) that the Bayesian optimizer has the lowest search cost. L-2D shows this clearly, since poor configurations will take much more time than good configurations. The Bayesian optimizer outperforms both Random and Coordinate. The smaller error for the Bayesian optimizer implies better stability than other strategies.

C. Search Space Size

Fig. 3 compares the median CNO and the median search cost over various search space sizes for Platypus, TensorFlow, and Keras. We include the CNO of the default configuration for comparison, where the default configuration has no search cost. We observe that the CNOs for the Random and Coordinate optimizers worsen as search space dimensions are added, while the CNO of the Bayesian optimizer is relatively stable regardless of the number of dimensions. This is because it is difficult for the Random and Coordinate optimizers to explore the search space sufficiently as the search space size increases. For example, for the K-2D experiment, in 15 trials, the Random optimizer has almost 30% probability of finding the optimal and the Coordinate optimizer can explore both parameters sufficiently. However, for K-4D, within 15 trials, the Random optimizer has only a 2% probability of finding the optimal and the Coordinate optimizer can only explore 2 or 3 parameters due to the limited number of trials.

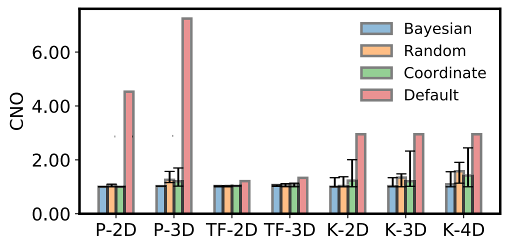
Increasing dimensionality and search space size significantly increases the difficulty of finding the optimal configuration. For example, after 15 trials, ParaOpt has only a 20% chance to find the optimal configuration. After searching 1.2%-6.6% of the search space, the likelihood of finding a near-optimal configuration increases to 77.5% in 3D and 4D experiments. On average, ParaOpt reduces runtime by 55.4% in 3D and 4D experiments when compared with using the default configuration.

D. Number of Trials

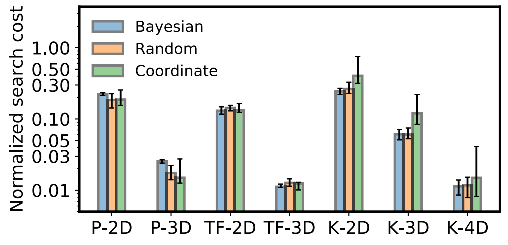
Next, we investigate how the number of trials impacts optimizer performance. Here we use the same experiment settings as described in §VI-C and vary the number of trials from 5 to 25 (including three initial configurations for the Bayesian optimizer).

Fig. 4a and Fig. 4b show the median CNO and the median search cost for the K-4D experiment, respectively.

When restricted to only five trials, the Bayesian optimizer generates a slightly worse CNO than the Random optimizer.

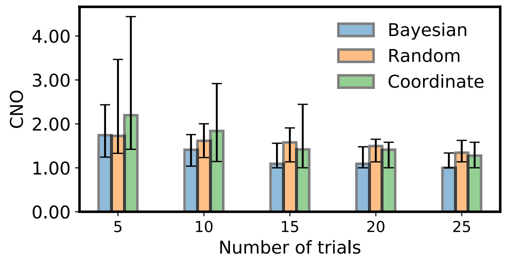


(a) CNO.

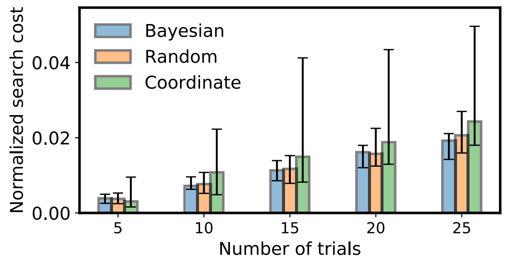


(b) Normalized search cost.

Fig. 3: Comparison of P-2D, P-3D, TF-2D, TF-3D, K-2D, K-3D, and K-4D. Error bars show the 10th and 90th percentiles.



(a) CNO.



(b) Normalized search cost.

Fig. 4: Results of K-4D experiments with varying the number of trials. Error bars correspond to the 10th and 90th percentiles.

This is because there is insufficient data for the Bayesian optimizer to reduce the confidence interval. However, as the number of trials increases, the Bayesian optimizer starts to outperform the other optimizers.

E. Optimization Objectives

To explore ParaOpt's ability to support different objective functions we now optimize Strelka2 (S-2D) for cost on three instance types. To illustrate the problem we first perform a grid search of S-2D on c5.large, c5.xlarge, and c5.2xlarge instance types, with 2, 4, 8 cores, and 4, 8, 16GB RAM. Fig. 5 shows the *MEMGB* (amount of memory in GB) plotted against the *JOBS* (number of CPUs), where the color indicates the total cost. The cost is computed as the runtime multiplied by the unit price of the instance.

The corresponding value of (*MEMGB*, *JOBS*) reported in the figure shows the monetary cost to run S-2D on the smallest instance type that can accommodate that configuration. For example, the value of (7, 4) corresponds to the cost to run on c5.xlarge, as that is the smallest instance type that can accommodate 7 GB of RAM and 4 CPUs.

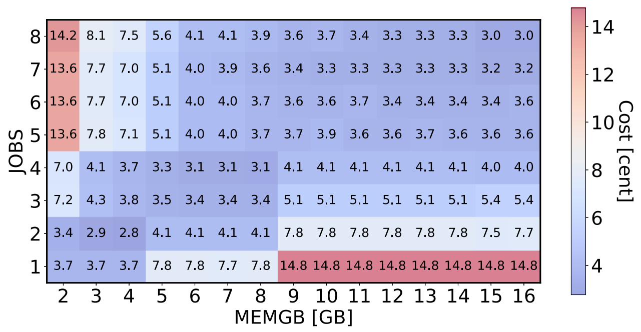


Fig. 5: Monetary cost map of S-2D.

We find that the c5.large instance type is the cheapest ($0.028), followed by c5.2xlarge ($0.030) and c5.xlarge ($0.031). The runtimes for S-2D on the c5.large, c5.xlarge, and c5.2xlarge are 19.6, 10.8, and 5.3 minutes, respectively. This leads to a trade-off between runtime and monetary cost. Next, we select monetary cost as the optimization objective and compare the performance of the Random, Coordinate, and Bayesian optimizers using the same experimental settings as described in §VI-B. The results are shown in Fig. 6. For all experiments, ParaOpt finds a configuration within 10-15% of the optimal cost within 15 trials.

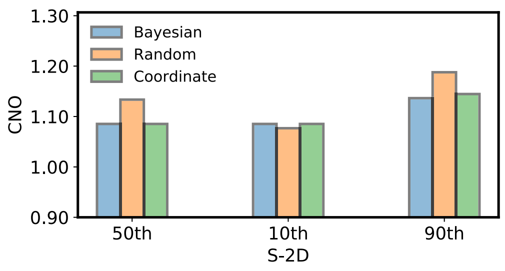


Fig. 6: Parameter optimization for S-2D with ParaOpt.

VII. Related Work

Practices in cloud configurations. Google Cloud offers a service [25] which monitors the average resource usage and provides recommendations to reduce cost. Amazon offers a rightsizing recommendation service [26], which suggests when to downsize or terminate instances based on analysis of resources usage. Unlike ParaOpt, these services are only capable of optimizing for cost and do not support arbitrary application parameterizations.

Selecting cloud configurations for specific applications. Several systems have been developed to aid selection of the "best" cloud configuration for a given application. CherryPick [27] leverages Bayesian optimization to build performance models for applications. It aims to find the optimal instance types and cluster size to minimize cost under a time budget. Similarly, Ernest [28] aims to select the number and type of instances for running jobs in AWS EC2. To do so, it creates efficient performance predictors by applying machine learning to data gathered from sample runs. We have also previously explored the use of automated experiment design to efficiently evaluate different cloud instance types for executing computational workflows [29]. While ParaOpt can be used to identify optimal instance types it is not limited to instance selection but also to identify optimal application configuration in a high-dimensional space. Lynceus [30] automatically provisions and tunes the parameters for analytics applications on clouds. It implements a budget-aware optimization method to identify the best configuration. This work is orthogonal to ours and could be integrated as an optimization strategy in ParaOpt.

Tuning parameters for specific applications. Similar approaches have been used to automate workflow parameter tuning. For example, many studies focus on turning the parameters of Hadoop MapReduce applications for performance [31]–[33]. However, these methods have limited adaptability for other types of applications and require many experiments for each new predictive task. Aken *et al.* [34] and Mahgoub *et al.* [35] both leverage Machine learning models for auto-tuning parameters for database systems [34], [35] and distributed stream processing [36]. ParaOpt shares similar approaches with these systems; however it focuses on enabling a framework to support different optimization strategies, application types, and objectives.

Optimization methods. Bayesian optimization is a well-established approach to deal with complex optimization problems with large search spaces [17]. Bayesian optimization has been widely used to aid selection of cloud configurations [27], [37], optimizing parameters of deep neural network [17], and system configurations [38].

Performance prediction. Wang *et al.* [39] use machine learning models to predict the performance of storage systems as a function of input workloads, requiring no knowledge of the storage system. However, building such models requires a large amount of training data. Performance prediction is also addressed by systems such as PARIS [40], which apply a data-driven approach with a hybrid offline and online data collection and modeling framework to provide accurate performance estimations. However, PARIS focuses predominantly on video encoding and latency. ARIA [32] uses historical traces and dynamically adjust resource allocation.

VIII. Conclusion

ParaOpt allows users to automatically configure arbitrary applications on heterogeneous cloud instance types using custom optimizers and objectives. Our experiments on AWS using six different applications show that ParaOpt can identify near-optimal configurations within 15 trials in 83.2% of cases. The ParaOpt-identified configurations decrease execution time by up to 85.8% when compared with using default configurations. ParaOpt is available on Github: https://github.com/globus-labs/ParaOpt.

Acknowledgments

This work was supported in part by the NSF (CNS-1816611) and DOE (DE-AC02-06CH11357). Experiments were conducted on Amazon Web Services, Jetstream [41], and at the National Supercomputing Center, Singapore.

References

[1] R. Chard, K. Chard, B. Ng *et al.*, "An automated tool profiling service for the cloud," in *16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*. IEEE, 2016, pp. 223–232.

[2] R. Chard, K. Chard, K. Bubendorfer *et al.*, "Cost-aware cloud provisioning," in *11th International Conference on e-Science (e-Science)*. IEEE, 2015, pp. 136–144.

[3] S. Kim, K. Scheffler, A. L. Halpern *et al.*, "Strelka2: fast and accurate calling of germline and somatic variants," *Nature methods*, vol. 15, no. 8, p. 591, 2018.

[4] A. Rimmer, H. Phan, I. Mathieson *et al.*, "Integrating mapping-, assembly-and haplotype-based approaches for calling variants in clinical sequencing applications," *Nature genetics*, vol. 46, no. 8, p. 912, 2014.

[5] A. McKenna, M. Hanna, E. Banks *et al.*, "The genome analysis toolkit: a mapreduce framework for analyzing next-generation dna sequencing data," *Genome research*, vol. 20, no. 9, pp. 1297–1303, 2010.

[6] S. Plimpton, "Fast parallel algorithms for short-range molecular dynamics," *Journal of computational physics*, vol. 117, no. 1, pp. 1–19, 1995.

[7] "LAMMPS Molecular Dynamics Simulator," https://lammps.sandia.gov/doc/Manual.html, Accessed Oct 25, 2019.

[8] H. C. Edwards, C. R. Trott, and D. Sunderland, "Kokkos: Enabling manycore performance portability through polymorphic memory access patterns," *Journal of Parallel and Distributed Computing*, vol. 74, no. 12, pp. 3202 – 3216, 2014.

[9] M. Abadi, P. Barham, J. Chen *et al.*, "TensorFlow: A system for large-scale machine learning." in *OSDI*, vol. 16, 2016, pp. 265–283.

[10] F. Chollet, "Keras," https://github.com/fchollet/keras, 2015.

[11] A. Krizhevsky, G. Hinton *et al.*, "Learning multiple layers of features from tiny images," Citeseer, Tech. Rep., 2009.

[12] Y. LeCun, L. Bottou, Y. Bengio *et al.*, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[13] S. Tuecke, R. Ananthakrishnan, K. Chard *et al.*, "Globus Auth: A research identity and access management platform," in *12th International Conference on e-Science*. IEEE, 2016, pp. 203–212.

[14] Y. Babuji, A. Woodard, Z. Li *et al.*, "Parsl: Pervasive parallel programming in python," in *28th ACM International Symposium on High-Performance Parallel and Distributed Computing (HPDC)*, 2019.

[15] J. Bergstra and Y. Bengio, "Random search for hyper-parameter optimization," *Journal of Machine Learning Research*, vol. 13, no. Feb, pp. 281–305, 2012.

[16] J. Friedman, T. Hastie, and R. Tibshirani, "Regularization paths for generalized linear models via coordinate descent," *Journal of statistical software*, vol. 33, no. 1, p. 1, 2010.

[17] J. Snoek, H. Larochelle, and R. P. Adams, "Practical bayesian optimization of machine learning algorithms," in *Advances in Neural Information Processing Systems 25*, F. Pereira, C. J. C. Burges, L. Bottou *et al.*, Eds. Curran Associates, Inc., 2012, pp. 2951–2959.

[18] "Pure Python implementation of bayesian global optimization with gaussian processes," https://github.com/fmfn/BayesianOptimization, Accessed October 25, 2019.

[19] N. Srinivas, A. Krause, S. M. Kakade *et al.*, "Gaussian process optimization in the bandit setting: No regret and experimental design," *arXiv preprint arXiv:0912.3995*, 2009.

[20] "National Cancer Institute GDC Data Portal, Genomic data," https://portal.gdc.cancer.gov/files/3f70141c-2868-410b-8228-ca680931f1c8, Accessed October 25, 2019.

[21] "National Cancer Institute GDC Data Portal, Genomic data," https://gdc.cancer.gov/about-data/data-harmonization-and-generation/gdc-reference-files, Accessed October 25, 2019.

[22] "Readme of Example in LAMMPS," https://github.com/lammps/lammps/tree/master/examples/ASPHERE, Accessed October 25, 2019.

[23] "CIFAR-10 Benchmark in TensorFlow." https://github.com/tensorflow/models/tree/master/tutorials/image/cifar10, Accessed October 25, 2019.

[24] "MNIST example with Keras." https://keras.io/examples/mnist_cnn/, Accessed October 25, 2019.

[25] "Apply Sizing Recommendations for Instances," https://cloud.google.com/compute/docs/instances/apply-sizing-recommendations-for-instances, Accessed October 25, 2019.

[26] "Optimizing Your Cost with Rightsizing Recommendations," https://docs.aws.amazon.com/awsaccountbilling/latest/aboutv2/ce-rightsizing.html, Accessed October 25, 2019.

[27] O. Alipourfard, H. H. Liu, J. Chen *et al.*, "CherryPick: Adaptively unearthing the best cloud configurations for big data analytics," in *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2017, pp. 469–482.

[28] S. Venkataraman, Z. Yang, M. J. Franklin *et al.*, "Ernest: Efficient performance prediction for large-scale advanced analytics," in *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2016, pp. 363–378.

[29] M. Baughman, R. Chard, L. Ward *et al.*, "Profiling and predicting application performance on the cloud," in *11th IEEE/ACM International Conference on Utility and Cloud Computing*, 2018.

[30] M. Casimiro, D. Didona, P. Romano *et al.*, "Lynceus: Tuning and provisioning data analytic jobs on a budget," *CoRR*, vol. abs/1905.02119, 2019.

[31] H. Herodotou, F. Dong, and S. Babu, "No one (cluster) size fits all: automatic cluster sizing for data-intensive analytics," in *2nd ACM Symposium on Cloud Computing*, Oct 2011.

[32] A. Verma, L. Cherkasova, and R. H. Campbell, "ARIA: automatic resource inference and allocation for mapreduce environments," in *8th ACM international conference on Autonomic computing*, Jun 2011.

[33] Z. Zhang, L. Cherkasova, A. Verma *et al.*, "Automated profiling and resource management of pig programs for meeting service level objectives," in *9th international conference on Autonomic computing*, Sept 2012.

[34] D. V. Aken, A. Pavlo, G. J. Gordon *et al.*, "Automatic database management system tuning through large-scale machine learning," in *2017 ACM International Conference on Management of Data (SIGMOD 2017)*, 2017, pp. 1009–1024.

[35] A. Mahgoub, P. Wood, S. Ganesh *et al.*, "Rafiki: A middleware for parameter tuning of nosql datastores for dynamic metagenomics workloads," in *Proceedings of the 18th ACM/IFIP/USENIX Middleware Conference*. ACM, 2017, pp. 28–40.

[36] L. M. Vaquero and F. Cuadrado, "Auto-tuning distributed stream processing systems using reinforcement learning," *arXiv preprint arXiv:1809.05495*, 2018.

[37] C. Hsu, V. Nair, V. W. Freeh *et al.*, "Arrow: Low-level augmented bayesian optimization for finding the best cloud vm," in *38th IEEE International Conference on Distributed Computing Systems (ICDCS)*, Jul 2018, pp. 660–670.

[38] V. Dalibard, "A framework to build bespoke auto-tuners with structured Bayesian optimisation," University of Cambridge, Computer Laboratory, Tech. Rep. UCAM-CL-TR-900, Feb. 2017.

[39] M. Wang, K. Au, A. Ailamaki *et al.*, "Storage device performance prediction with cart models," in *12th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems (MASCOTS)*, 2004, pp. 588–595.

[40] N. J. Yadwadkar, B. Hariharan, J. E. Gonzalez *et al.*, "Selecting the best VM across multiple public clouds: A data-driven performance modeling approach," in *Symposium on Cloud Computing*, 2017, pp. 452–465.

[41] C. A. Stewart, T. M. Cockerill, I. Foster *et al.*, "Jetstream: A self-provisioned, scalable science and engineering cloud environment," in *XSEDE Conference: Scientific Advancements Enabled by Enhanced Cyberinfrastructure*, 2015, pp. 29:1–29:8.