Parallelization of Classical Numerical Optimization in Quantum Variational Algorithms

Marco Pistoia
JPMorgan Chase & Co.
marco.pistoia@jpmchase.com

Peng Liu
Google
liupen@google.com

Chun Fu (Richard) Chen
IBM Research
chenrich@us.ibm.com

Shaohan Hu
IBM Research
shaohan.hu@ibm.com

Stephen Wood
IBM Research
woodsp@us.ibm.com

***Abstract*—Numerical optimization has been extensively used in many real-world applications related to Scientific Computing, Artificial Intelligence and, more recently, Quantum Computing. However, existing optimizers conduct their internal computations sequentially, which affects their performance. We observed a general pattern that enabled us to parallelize such internal computations and achieve significant speedup. We designed a novel parallelization algorithm for optimizers, which consists of pattern detection, prediction, precomputation, and caching. Importantly, our design does not require any change to the optimizers. Instead, it simply modifies the function to be optimized, thereby leading to several engineering advantages, including simplicity, modularity and portability. We implemented this solution and included it in the Qiskit Aqua open-source project. In this paper, we present an evaluation on both standard benchmarks and real-world quantum-computing applications. The evaluation results confirm that our approach (1) incurs negligible overhead, (2) effectively speeds up optimization, and (3) does not affect the accuracy of the results or the convergence of the optimizers.**

***Index Terms*—Numerical optimization, Parallelization, Quantum Computing, Variational Algorithms**

I. Introduction

Software engineers from the Scientific Computing, Quantum Computing and Machine Learning areas heavily rely on numerical optimization to solve numerous domain-specific problems. Given a function $f(x_1, x_2, \ldots, x_n)$ that takes n parameters, *numerical optimization* finds the minimum or maximum function value and the corresponding parameter values, depending on whether the objective of the optimization is *minimization* or *maximization*, respectively. Without loss of generality, in this paper we will refer only to the minimization problem, since maximization can be easily reduced to minimization. Putting it more precisely, given $f : A \mapsto \mathbb{R}$, where $A \subseteq \mathbb{R}^n$, the objective of optimization is to compute the following:

$$\underset{x_1,\ldots,x_n}{\operatorname{argmin}} f(x_1, x_2, \ldots, x_n)$$

We use the vector notation $\vec{x}$ to compactly denote the parameters $[x_1, x_2, \ldots, x_n]$. We may also indicate a vector as a *point*. We refer to the function to be minimized as the *target function*.

Many optimizers have been designed, implemented and made available in popular libraries, such as SciPy [38]. These optimizers meet the following two *general applicability conditions*:

1) They do not assume that the target function f can be expressed in analytic form, which could be efficiently handled by specialized solvers [14], [19].
2) They do not assume that the derivatives or the gradient of f can be expressed in analytic form.

Note that the partial derivatives

$$\frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial x_2}, \ldots, \frac{\partial f}{\partial x_n}$$

are basically the projections of the gradient $\nabla f(\vec{x})$ along the different dimensions of $\mathbb{R}^n$. We refer to a function f such that neither f nor its derivatives can be expressed in analytic form as a *black-box function*.

A. Real-world Scenarios

Blackbox functions commonly exist in the real world and play an important role. In fact, most functions that involve feedback from the real-world environment are black-box functions. For example:

- *Reinforcement Learning* provides intelligent control to a wide range of applications, including robotic control and game AI. At the core of RL is the value function, which needs to be maximized by updating the control parameters. However, the value function is computationally intractable [40], and consequently does not have the analytic form, because it depends on the feedback from the environment.
- *Scientific Computing* includes many interesting problems, such as the ground-state energy of molecules,, which can be computed by minimizing the functions related to the dynamics of the molecules. These are described by the Schrödinger equations, most of which do not have analytic solutions [37]. Accordingly, the functions involved cannot be expressed in analytic forms.
- *Quantum Computing* is of particular interest when it comes to hybrid quantum/classical algorithms, such as the Variational Quantum Eigensolver [32] and the Quantum Approximate Optimization Algorithm [17], which interleave quantum computations and classical executions of numerical optimization routines. For such algorithms, the efficiency and of the classical optimization is crucial for maintaining the performance benefits of Quantum Computing.

978-1-7281-5778-8/20/$31.00 ©2020 IEEE
DOI 10.1109/ICST46399.2020.00039

- *Hyperparameter Optimization* is used for tuning software systems. For neural networks, hyperparameter optimization tunes the hyperparameters, such as the number of neurons in each layer, to obtain the best model accuracy. However, the relation between the accuracy and the number of neurons is rather complex and cannot be put into analytic form. Similarly, in the Spark system [39], hyperparameter optimization tunes the configurations to obtain the best performance, but the relation between the performance and the configurations cannot be expressed in analytic form.

B. Problem

Given a function $f(\vec{x})$, which may be a black-box function, the optimizer evaluates a sequence of points to search for the minimum value. During the search, the evaluation results of the previous points affect the choice of the point to be evaluated next. Such choice is optimizer-specific.

Regardless of the differences between the optimizers, we make a general observation: many optimizers rely on the gradient information. Since the analytic form of the gradient may be unavailable, optimizers approximate the gradient using numerical differentiation. Each entry in the gradient vector is the derivative along one dimension, as follows:

$$\nabla f(\vec{x}) = \left[\frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial x_2}, \ldots, \frac{\partial f}{\partial x_n}\right].$$

Let $\vec{u_1}, \vec{u_2}, \ldots, \vec{u_n}$ denote the unit vectors along the n dimensions. The derivative along each dimension is *approximated* using the numerical differentiation as:

$$\frac{\partial f}{\partial x_i} \approx \frac{f(\vec{x} + \alpha\vec{u_i}) - f(\vec{x})}{\alpha}$$

Clearly, the optimizer needs to evaluate $f(\vec{x} + \alpha\vec{u_i})$, in addition to $f(\vec{x})$. Overall, to approximate the gradient $\nabla f(\vec{x})$ at point $\vec{x}$, the optimizer needs to evaluate the n neighboring points around $\vec{x}$, which are $f(\vec{x} + \alpha\vec{u_i})$ for $i = 1, 2, \ldots n$.

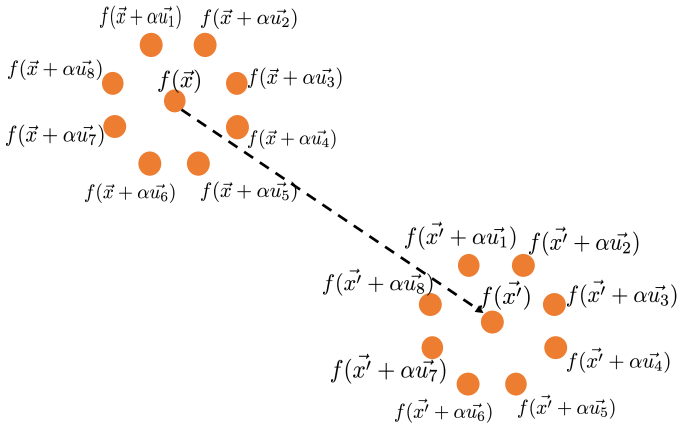


Fig. 1: Illustration of the Optimization Logic

Figure 1 illustrates this observation. At point $\vec{x}$, with dimension 8, the optimizer evaluates $\vec{x}$ and its 8 neighbors to approximate the gradient. Then, following some internal logic that determines the direction to move towards and the step size, the optimizer moves to the next point $\vec{x'}$. Again, it needs to evaluate the 8 neighbors of $\vec{x'}$ in addition to $\vec{x'}$ itself. These steps are performed iteratively by the optimizer.

C. Practical Challenges

Each evaluation may be *very slow*. Consider the hyperparameter optimization of a neural network; each evaluation takes a specific hyperparameter configuration and performs the full evaluation (including training and testing) to obtain the accuracy. Furthermore, the optimizer may need to evaluate *many* neighbors of point $\vec{x}$ if the dimension is high. This is especially true given when such functions relate to modern computing (e.g., scientific computing, AI and modern software systems), and so depend on many parameters. For example, the Spark system depends on hundreds of parameters [39].

Unfortunately, the optimizers evaluate the large number of neighbors *sequentially*, thereby imposing a severe performance bottleneck. A natural idea to speed up the optimization is to evaluate the neighbors in parallel, since they are independent of each other. One may wish to modify the optimizers so that they could perform their evaluations in parallel. However, several practical software-engineering challenges make this choice highly impractical. First, changing the code of an optimizer requires deep understanding of the optimization logic, or subtle bugs may be introduced. The optimization logic is very complex as heavily utilizes mathematics principles and notations (e.g., the Hessian matrix [11]), which are very likely beyond a software developer's skill set. Even worse, the code may be written in the old Fortran style, which further complicates any code change. For instance, the optimizers in SciPy [38], which are widely used [41], wrap legacy Fortran code in a Python wrapper. To the best of our knowledge, *no* existing publicly available optimizers [13], [28], [31], [38] support parallel evaluations of the neighbors. [1] Second, a change in one optimizer does not generally apply to other optimizers. When developers switch to a new optimizer, they need to start over to figure out the changes required.

D. Our Solution

We propose a parallelization approach based on the observation of a general pattern: *Each neighbor $\vec{x}+\alpha\vec{u_i}$ of $\vec{x}$ shares the same values with $\vec{x}$ at all entries, except for the ith entry*. Besides, the difference at the ith entry is α, which is fixed across all the neighbors as computed internally by the optimizer. This observation lays the ground for our parallelization approach: We detect the pattern when the first neighbor is about to be evaluated and predict all the neighbors, which we then evaluate in parallel. We perform the parallelization by wrapping the target function within our *run-time logic*. Importantly, *this design does not require any change to the optimizer*, which we shall elaborate soon.

[1] We have downloaded and studied five sets of optimizers, including the popular ones, such as SciPy [38], NLOpt [28], pyOpt [31] and RBFOpt [13]. Note that the parallel support in RBFOpt is not about the parallel evaluation of the neighbors.

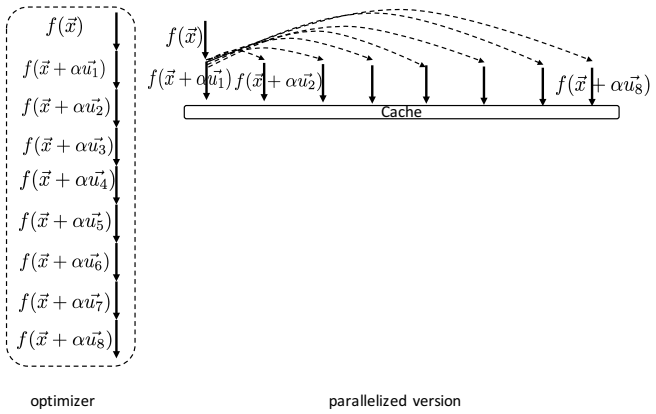


Fig. 2: Parallelization

1) Add-on Run-time Logic: Figure 2 illustrates our parallelization technique. On the left, it shows the original logic, which evaluates $\vec{x}$ and its neighbors sequentially. On the right, it shows our parallelized version. Note that the optimizer makes the same sequence of calls[2] in both versions because our parallelization does not impose any change to the optimizer. After the optimizer calls $f(\vec{x}+\alpha\vec{u_1})$, our run-time logic compares $\vec{x}+\alpha\vec{u_1}$ against the parameter of the last call, i.e., $\vec{x}$, which is maintained by our run-time logic. Our logic finds that the two differ only at one entry, which suggests that $\vec{x}+\alpha\vec{u_1}$ is a neighbor of $\vec{x}$. Meanwhile, our run-time logic automatically extracts the α value by computing the difference. Afterwards, predicts the remaining 7 neighbors, and spawns 8 processes to evaluate the 8 neighbors in parallel. After the evaluation is done, the results are associated with the points and stored in a cache. Later on, when the optimizer calls $f(\vec{x}+\alpha\vec{u_2})$, ..., $f(\vec{x}+\alpha\vec{u_8})$, the result can be immediately retrieved from the cache using the point as the key.

We carefully designed the run-time logic to limit its overhead incurred. For instance, it clears the cache when the optimizer moves from $\vec{x}$ to $\vec{x}'$ in Figure 1 because the results of the neighbors of $\vec{x}$ are no longer needed. At any time, the cache needs to store only the results of the n neighbors around the point of interest. In a typical situation with $n \leq 10000$, the cache lookup is efficient.

2) No Change to the Optimizer: Our approach does not impose any change to the optimizer, which leads to numerous engineering advantages, including *simplicity*, *modularity* and *portability*. Our approach first creates a function f_{wrap}, which wraps f with the add-on run-time logic, and then uses f_{wrap} as a drop-in replacement of f. In essence, we ask the optimizer to treat f_{wrap} as the new target function. In particular, our solution offers a generic function f_{wrap} that takes any function f as its input and wraps it. Therefore, the users can enjoy the speedup brought by our parallelization without writing any extra code. More importantly, our parallelization solution applies to multiple optimizers given that it is not specialized for, or limited to, any single optimizer. Thus, there are no *portability* issues.

3) Evaluation: We implemented our approach and included it in the Qiskit Aqua open-source project for Quantum Computing [1]. We also evaluated it on both standard benchmarks and real-world scientific computing applications.

The evaluation results lead to the following conclusions:

1) The overhead incurred by our run-time logic is negligible ($\leq 3\%$).
2) With our approach, the running time decreases gradually when we increase the number of parallel worker processes, which confirms the effectiveness of our solution. In particular, with 8 parallel workers, we observed 4X speedup at most, 1.5X speedup at least and 2.5X speedup on average.
3) By inspecting the final result and logging the trace, we found that our optimized version produced the same result as the original version and followed the same trace, which indicates that our approach does not affect the accuracy of the final results or the convergence of the optimizers.
4) We studied the applicability of our approach to the SciPy [38] optimizer suite, which is widely used [41] and is representative of existing optimizer suites [13], [28], [31], [38]. We found that our approach applies to four popular optimizers, including those with the best performance among all. We believe that the applicability to them is important since users simply use the optimizers with the best performance in most cases.
5) Although we conducted the evaluation on the SciPy optimizer suite only, our approach is not specific to SciPy and also applies to other optimizer suites. For example, we have tested it on the DIRECT-L optimizer in NLOpt [28] and observed similar speedup.

E. Contributions of This Work

Through this work, we make the following contributions:

1) We identify a general pattern that enables the parallelization of the internal computations of numerical optimizers, leading to numeric-optimization speedup.
2) We propose a parallelization algorithm that consists of the pattern detection, prediction, precomputation and caching. It imposes no code changes to the optimizer, thereby leading to software-engineering advantages such as simplicity, modularity and portability.
3) We conduct an extensive evaluation on both standard benchmarks and real-world scientific-computing applications. The results show that our design incurs negligible overhead, effectively speeds up the execution, and does not affect correctness.
4) The solution presented in this paper has been implemented and deployed in the Qiskit Aqua open-source project for Quantum Computing [1].

We present our parallelization algorithm in Section II, and an extensive evaluation in Section III.

[2]The function call is different from the function execution, where the former makes the request for the evaluation and the latter performs the actual evaluation.

II. Parallelization Algorithm

In this section, we first formalize the pattern we observed, then present our parallelization algorithm, and lastly demonstrate the algorithm by applying it to a concrete example during which we discuss more details.

As mentioned above, optimizers often approximate the gradient $\nabla f(\vec{x})$ through numerical differentiation, which requires evaluating the neighbors of $\vec{x}$ in addition to $\vec{x}$ itself. We observe that the neighbors satisfy a general pattern:

Pattern 1: If the dimension of $\vec{x}$ is n, then $\vec{x}$ has n neighbors, each of the form $\vec{x} + \alpha\vec{u_i}$, where $i = 1, 2, \ldots, n$. Each neighbor $\vec{x}+\alpha\vec{u_i}$ of $\vec{x}$ shares the same values with $\vec{x}$ at all entries except for the ith one. Besides, $\vec{x}$ is at the same distance α from all its neighbors, where α is computed internally by the optimizer.

A. Concrete Example

Suppose $\vec{x} = [0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8]$ and $\alpha = 0.002$. Then, the neighbors of $\vec{x}$ are: as follows:

$$\vec{x}_1 = [0.102, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8]$$
$$\vec{x}_2 = [0.1, 0.202, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8]$$
$$\vec{x}_3 = [0.1, 0.2, 0.302, 0.4, 0.5, 0.6, 0.7, 0.8]$$
$$\vec{x}_4 = [0.1, 0.2, 0.3, 0.402, 0.5, 0.6, 0.7, 0.8]$$
$$\vec{x}_5 = [0.1, 0.2, 0.3, 0.4, 0.502, 0.6, 0.7, 0.8]$$
$$\vec{x}_6 = [0.1, 0.2, 0.3, 0.4, 0.5, 0.602, 0.7, 0.8]$$
$$\vec{x}_7 = [0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.702, 0.8]$$
$$\vec{x}_8 = [0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.802]$$

B. Algorithm Description

As mentioned in Section I, our parallelization technique relies on the run-time logic. We need to first determine where the run-time logic should be embedded. In favor of feasibility and general applicability, the run-time logic should be applied in a non-intrusive way with respect to the optimizer, i.e., it should not interfere with the original workflow of the optimizer. For instance, since the optimizer makes the function calls sequentially (as illustrated on the left of Figure 2), our parallelization should also assume that the function calls are made sequentially. Under this constraint, we find that the target function is the ideal place where to plug the run-time logic because the function is specified by the users and can be freely modified.

Given the user-specified target function f, we create a wrapper function f_{wrap} that performs our run-time logic, calls f, and sets f_{wrap} as the new target function in place of f. Accordingly, the optimizer will sequentially call f_{wrap}. Algorithm 1 details the design of f_{wrap}. At the interface level, f_{wrap} takes the same input as f, i.e., the point $\vec{x_c}$ to be evaluated, and produces the same result. Note that variables $cache$, $\vec{x}_{last}$ and N_{worker} are *global*.

First, we look up the cache (line 1); we return the evaluation result associated with $\vec{x_c}$, if any (line 4), or else we follow the

ALGORITHM 1: The Wrapper Function f_{wrap}

```
   Input: the current point x_c
1  v_c = lookup(cache, x_c)
2  if v_c ≠ null then
3      x_last = x_c
4      return v_c
5  else
6      if x_last ≠ null then
7          x_Δ = x_c - x_last
8          if nonzeros(x_Δ) == 1 then
9              α = |x_Δ|
10             n = dimension(x_c)
11             foreach i = 1 ... n do
12                 x_i = x_last.copy()
13                 x_i[i] += α
14                 todo.append(x_i)
15             end
16             parallel_run(f, todo, N_worker, cache)
17             x_last = x_c
18             return lookup(cache, x_c)
19         end
20     end
21     cache.clear()
22     x_last = x_c
23     return f(x_c)
24 end
```

false branch (line 5). At line 6, we check whether the point $\vec{x}_{last}$ involved by the last call exists. Since $\vec{x}_{last}$ is updated prior to the return of every call (lines 3, 17 and 22), $\vec{x}_{last}$ exists unless the current point is the first point. Suppose $\vec{x}_{last}$ exists for now. We further check if $\vec{x_c}$ differs from $\vec{x}_{last}$ only at a single entry (lines 7-8), i.e., whether $\vec{x_c}$ and $\vec{x}_{last}$ satisfy Pattern 1. If so, we extract α (line 9), prepare all the neighbors (lines 11-15), and evaluate them in parallel (line 16). Note the helper function $parallel\_run$ evaluates f against the items in the *todo* list in parallel and stores the results into *cache*. N_{worker} is user-specified and controls the number of worker processes spawned. Ideally, to achieve the maximal speedup, it should be equal to the number of available cores. Lastly, in case $\vec{x}_{last}$ does not exist (line 6) or the pattern is not satisfied (line 8), we can do nothing but faithfully evaluate $f(\vec{x_c})$ and return the value (line 23).

Inside the parallel_run function, we separate the items in the *todo* list evenly into N_{worker} parts and assign each part to a worker. If the size N_{size} of the *todo* list is not a multiple of N_{worker}, then some workers need to work on $\lceil N_{size}/N_{worker} \rceil$ items and some on $\lfloor N_{size}/N_{worker} \rfloor$ items. These workers are joined at the end of parallel_run, i.e., they wait for each other.

C. Applying the Algorithm

Let us apply Algorithm 1 to the above concrete example to discuss more details. The optimizer first calls f_{wrap} on $\vec{x}$. The

execution takes the false branches at lines 2 (due to the cache miss) and 6 (since $\vec{x}_{last}$ has not yet been set). The execution then evaluates $f(\vec{x}_c)$ and sets $\vec{x}_{last}$ (lines 22-23).

Next, the optimizer calls f_{wrap} upon $\vec{x}_1$. The execution reaches line 7 and computes $\vec{x}_\Delta = \vec{x}_1 - \vec{x} = [0.002, 0, 0, 0, 0, 0, 0, 0]$. $\vec{x}_\Delta$ has only one non-zero entry (line 8) and the value at such entry, which is equal to the length $|\vec{x}_\Delta|$ of $\vec{x}_\Delta$, is extracted as α (line 9). The loop at line 11 prepares the 8 neighbors by first making a copy of $\vec{x}$ and then increasing the ith entry by $\alpha = 0.002$. For example, $\vec{x}_8$ becomes $[0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8]$ after the copying (line 12) and $[0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.802]$ after the increase (line 13). The neighbors are put into the list *todo*. The execution then spawns N_{worker} processes to evaluate f against the 8 neighbors in parallel, and stores the results in *cache* (line 16). The evaluation result of the current point, i.e., $\vec{x}_1$, must be in the cache and is returned at line 18.

Then, the optimizer calls f_{wrap} upon $\vec{x}_2$. The execution immediately returns the result (line 4), precomputed and stored into *cache* by the call on $\vec{x}_1$. Similarly, after the optimizer calls f_{wrap} on other neighbors, the execution immediately returns the relevant results.

D. Cache

The cache is cleared at line 21, which happens only if the conditions at lines 6 or 8 do not hold. If the condition at line 6 does not hold, $\vec{x}_c$ is the first point evaluated, then the cache is empty and cache.clear() is trivial. If the condition at line 8 does not hold, $\vec{x}_c$ represents the new center whose neighbors need to be evaluated, e.g., $\vec{x'}$ in Figure 1. In this case, the cache holds the results of the neighbors of the old center $\vec{x}$ in Figure 1, which are no longer needed and hence can be removed. At any time, the cache is either empty or holds the results of n neighbors. Given that n is relatively small (e.g., ≤ 10000), the cache lookup is very efficient. In particular, the lookup is implemented by sequentially scanning the points in the cache and checking for equality. For equality checking, we use the `array_equal` API provided by NumPy. In future work, we will study how to speedup the cache lookup operations further (e.g., with hashing).

E. Discussion on Correctness

Algorithm 1 does not change the sequence of points that the optimizer evaluates. Instead, it only parallelizes the evaluation of a subsequence of points to achieve the desired speedup. Therefore, Algorithm 1 does not affect the accuracy of the final result or the convergence of the optimizer.

III. Evaluation

In this section, we present our experimental results.

A. Subject Optimizers

In our evaluation, we focus on the SciPy V1.1.0 optimizers [38]. The algorithms underlying the optimizers were invented independently before SciPy was created. They were selected and included in SciPy, which suggests that their effectiveness is well accepted by the community. Furthermore, these optimizers are widely used by existing software, such as the deep-learning [41] and quantum-application frameworks [1]. Although we conducted the evaluation only on the SciPy optimizer suite, our approach is not SciPy=specific. For example, we have successfully tested it on the NLOpt DIRECT-L optimizer [28]. In total, we studied the following optimizers: CG [30], COBYLA [34], L-BFGS-B [10], Nelder-Mead [27], Powell [33], SLSQP [25], TNC [20], trust_constr [29], trust_ncg [29], dogleg [29], trust_krylov [29] and trust_exact [29]. We refer the reader to the references for the details on each of these optimizers.

B. Research Questions

We are interested in answering the following four research questions:

1) *Applicability*: What optimizers benefit from our approach?
2) *Performance Study*: How much speedup does our approach introduce, and how much overhead does our runtime logic incur?
3) *Correctness*: Does our algorithm affect the convergence of the optimizers or the accuracy of their results?
4) *Real-world Applications*: Does our approach apply to the real-world uses of these optimizers?

To answer the first three questions, we applied the optimizers to a set of functions well studied and commonly used for benchmarking the optimizers [22]. The definition and visualization of the functions can be found online [4] [21]. From this collection, we select the representative functions such that:

- They can be generalized to the high dimension (because our approach is most effective for high-dimension problems), and
- They have a single minimum value (because otherwise the optimizers may easily get stuck at local optima, which is an open research challenge).

In particular, we selected the following functions: Rosenbrock [6] [21], Giunta [21], Trid [8], Sum of Different Powers [7], Power Sum [5] and Zakharov [9]. We refer to these as the *benchmark* functions. In our experiments, each of such functions takes 8 parameters, i.e., the dimension of $\vec{x}$ given as input is 8.

To answer the fourth and last question, we conducted the study on the real-world scientific-computing application that computes the ground-state energy of a molecule via minimization on a quantum computer using the Variational Quantum Eigensolver (VQE) algorithm [32].

C. Settings

To simulate different execution times, we add a parameterized loop to each benchmark function. The loop has a *deterministic* computation logic, i.e., no randomness is introduced, and it does not affect the function result. Specifically, the loop repeatedly computes the inverse of a matrix and multiplies it

by the matrix itself. The loop count is a controllable parameter. To simulate the time-consuming function mentioned in Section I, we set the loop count as a relatively large number (1,000 by default). Accordingly, the execution of the loop takes roughly 70 milliseconds.

We measure the performance of the optimizers following the fixed-target strategy [3]. That is, we specify a target accuracy of the result and measure the running time required by each optimizer to reach the target. Assuming the minimum function value is -2.5, we specify the target as -2.5+ϵ, where ϵ stands for an acceptable error (0.001 by default). In the sequential settings, we may alternatively use the number of function evaluations, instead of the running time, to measure the performance given that each function evaluation is slow and the function evaluations dominate the running time.

All the experiments were run on a MacBook Pro 6.1 equipped with a 12-Core Intel Xeon E5 processor and 64GB RAM. Every experiment was repeated 20 times. Here, we report the average over the 20 runs.

D. Answering the Research Questions

In this section, we discuss how we answered the four research questions formulated in Section III-B.

1) Applicability: According to Section II, our approach applies as long as the optimizer uses the gradient to guide the search. Specifically, the optimizer approximates the gradient with the numerical differentiation by evaluating the neighboring points. Following the algorithmic descriptions of each optimizer, we determined that our approach applies to four optimizers: CG [30], L-BFGS-B [10], SLSQP [25] and TNC [20]. We also logged the points evaluated by the optimizers and manually inspected the logs. The inspection confirmed that the four optimizers follow Pattern 1.

Importantly, although only four optimizers benefit from our improvement (Section III-D2), we observed they have the best performance among all. We argue that it is crucial to improve the optimizers with the best performance rather than a random set of optimizers because the users commonly pick the optimizers with the best performance regardless of their internal logic.

Specifically, we applied the original optimizers (without our improvement) to the benchmark functions and measured how many function evaluations each optimizer needs to reach the minimum. Each function evaluation is slow and the evaluations dominate the overall running time. The results are shown in Figure 3, where the results are grouped by the benchmark functions (X-axis); the Y-axis stands for the running time. According to the results, trust_constr, trust_ncg, dogleg, trust_krylov and trust_exact are clearly the slowest. For this reason, we will not consider them hereafter. Note that we did not show the result of the dogleg optimizer because it failed to find the minimum values for Giunta, Sum of Different Powers and Power Sum.

We also observe that SLSQP and L-BFGS-B consistently have the best performance. Next to them, CG performs comparably to COBYLA. Lastly, TNC performs comparably to POWELL and NELDER_MEAD.

2) Performance Study: From now on, we consider only the four optimizers that our approach applies to. In this section, we are interested in the speedup that our parallelization brings, and the overhead that our run-time logic incurs. We compare the baseline version, i.e., the original benchmark function, with our parallelized version, i.e., the wrapper function that executes both the run-time logic and the benchmark function. We evaluated the parallelized version with different concurrency settings ranging from 1 to 8 processes. We refer to them as p1, p2, ..., p8, respectively. We may refer to the baseline version as *base*.

The evaluation results for the benchmark functions are shown in Figure 5, where the X-axis stands for the different versions and the Y-axis shows the running time in seconds. For clarity, we show in Figure 4 the normalized results, where the Y-axis stands for the ratio of the running time between a particular version and the baseline version, labelled on the X-axis. We use T_{p1} to refer to the running time of the p1 version and use $R_{p1/base} = T_{p1}/T_{base}$ to refer to the ratio between the p1 version and the baseline version.

To approximate the overhead incurred by our run-time logic, we focus on the baseline version and the p1 version. The p1 version shares the same workload and the same degree of concurrency as the baseline version (i.e., both with a single worker process). Meanwhile, the p1 version additionally includes the run-time logic, e.g., the prediction of the neighbors, the caching operations and the process spawning/joining. Therefore, we compute the overhead as $R_{p1/base} - 1$, i.e., $(T_{p1} - T_{base})/T_{base}$. We observe that the overhead is $\leq 3\%$ in all the cases except when L-BFGS-B is applied to Zakharov (6% overhead) and when CG is applied to Giunta (9% overhead). This observation shows that our run-time logic incurs very little overhead.

To measure the speedup, we compare the running time of the parallelized versions (i.e., from p2 to p8) against the baseline version. From Figure 4, we observed an interesting general trend shared by all the curves: $T_{p1} > T_{p2} > T_{p3} > T_{p4} \approx T_{p5} \approx T_{p6} \approx T_{p7} > T_{p8}$. Without loss of generality, let us take the application of SLSQP to Rosenbrock as an example. If the baseline takes time t, p2 takes $\sim 65\%t$, p3 $\sim 55\%t$, p4, p5, p6 and p7 $\sim 45\%t$, and p8 $\sim 37\%t$. Most importantly, the parallelization takes less time with higher degree of concurrency, which confirms the effectiveness of our algorithm.

We also observe that p4, p5, p6 and p7 takes roughly the same running time. This is explained as follows. As mentioned in the experiment settings, each function takes 8 parameters, which implies that the optimizer needs to evaluate 8 neighbors at each iteration (Figure 1). Let us assume that the evaluation of each neighbor takes roughly the same time, t. With four parallel worker processes, each worker needs to evaluate 8/4=2 neighbors, which takes roughly $2t$ time. With five parallel workers, at least one worker needs to evaluate $\lceil 8/5 \rceil = 2$ neighbors which requires $2t$, and the other workers that

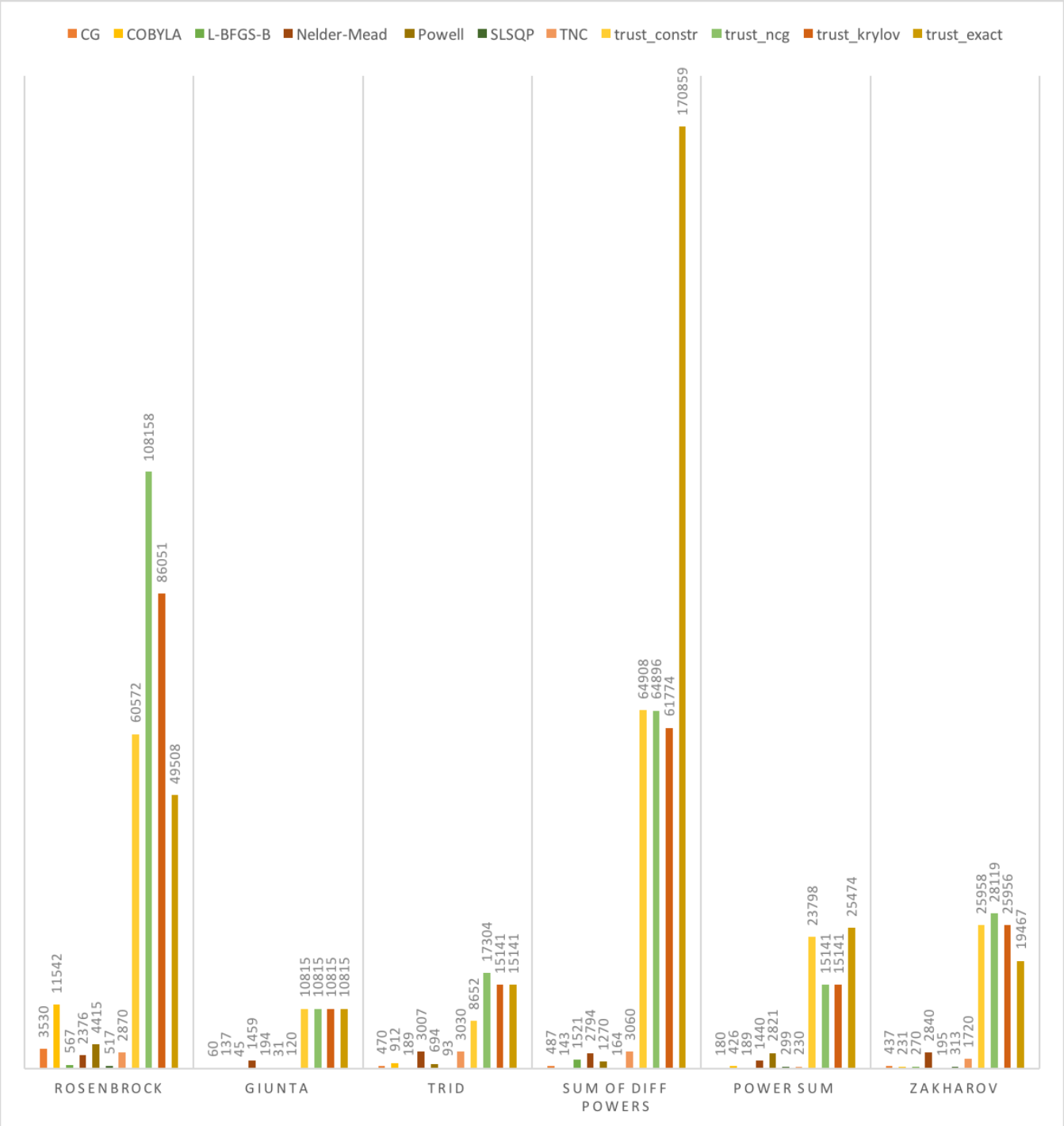
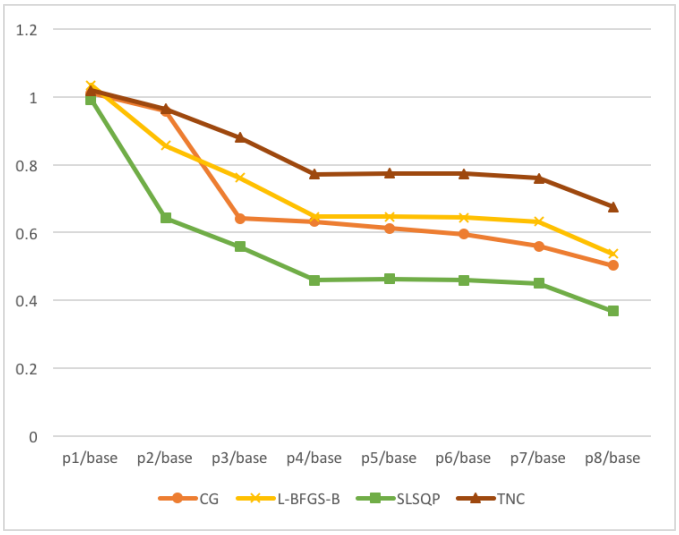


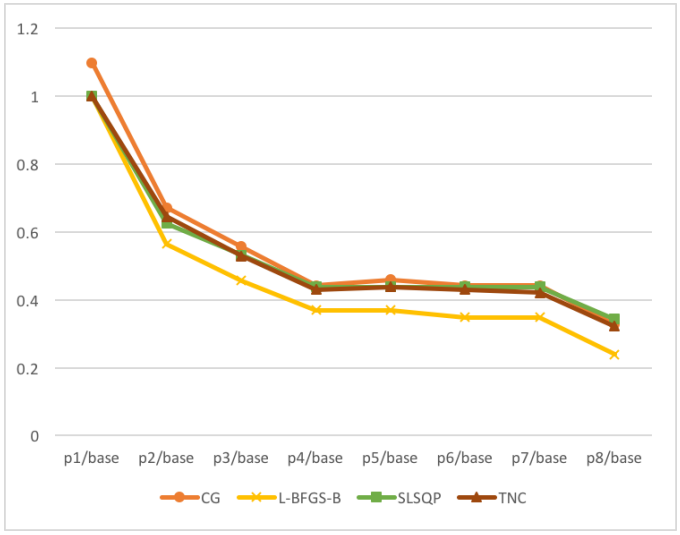
Fig. 3: Performance Comparison of All Optimizers in SciPy

complete sooner need to wait for it (line 16 in Algorithm 1). Similarly, with 6 or 7 parallel workers, at least one worker evaluates $\lceil 8/6 \rceil = \lceil 8/7 \rceil = 2$ neighbors, which takes $2t$. By generalizing this analysis to the versions from p1 to p8, we find that the number of neighbors evaluated by the slowest worker in each iteration is $\lceil 8/1 \rceil > \lceil 8/2 \rceil > \lceil 8/3 \rceil > \lceil 8/4 \rceil = \lceil 8/5 \rceil = \lceil 8/6 \rceil = \lceil 8/7 \rceil > \lceil 8/8 \rceil$, consistent with the trend observed above.

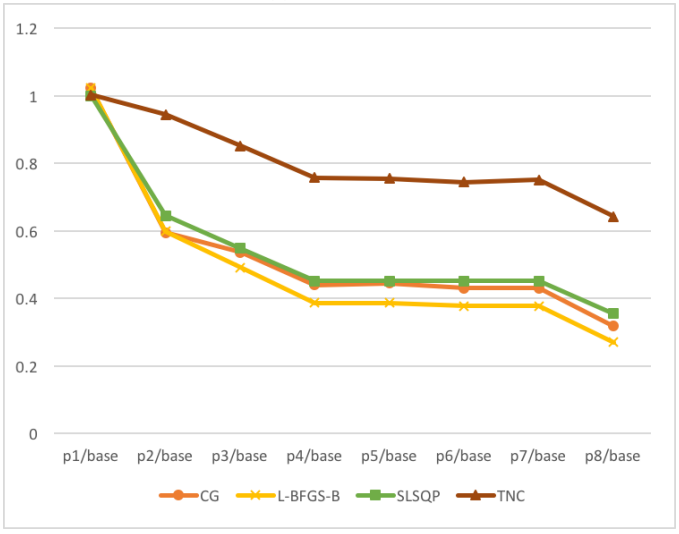
We also conducted a quantitative analysis of the results. By computing the average over all the optimizers and all benchmark functions, we found that our algorithm reduces the running time t to $69.4\%t$ with 2 processes, $58.8\%t$ with 3 processes, $49.2\%t$ with four processes, and $38.7\%t$ with eight processes.



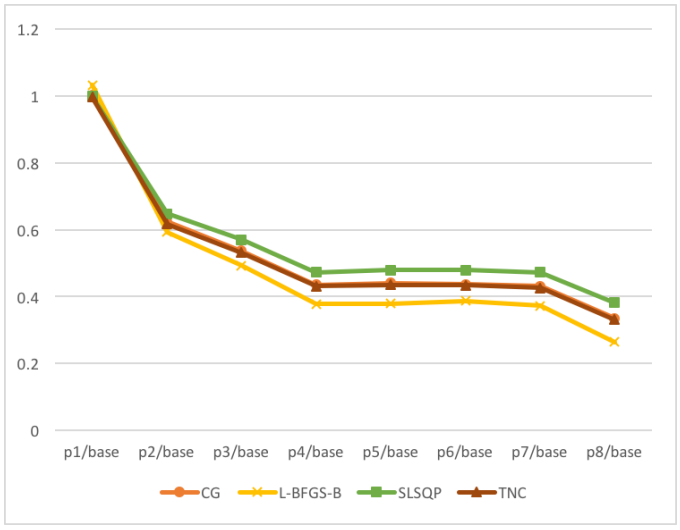
(a) Rosenbrock



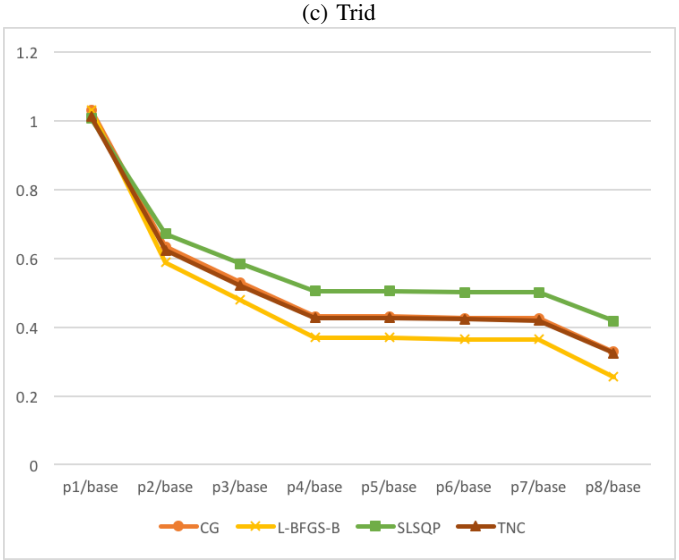
(b) Giunta



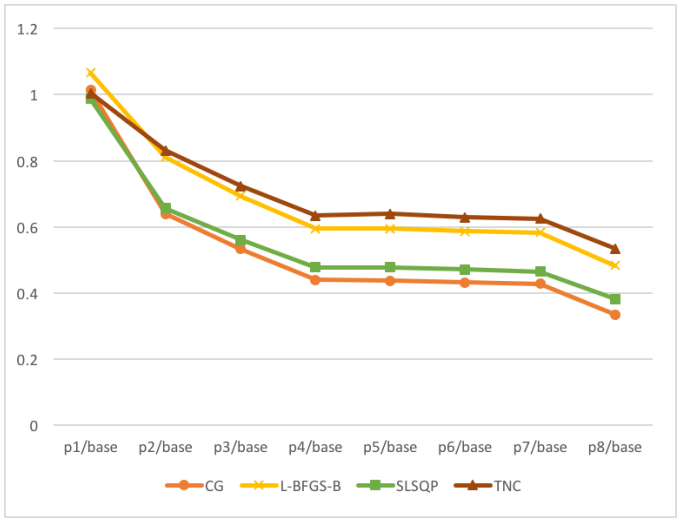
(c) Trid



(d) Sum of Different Powers

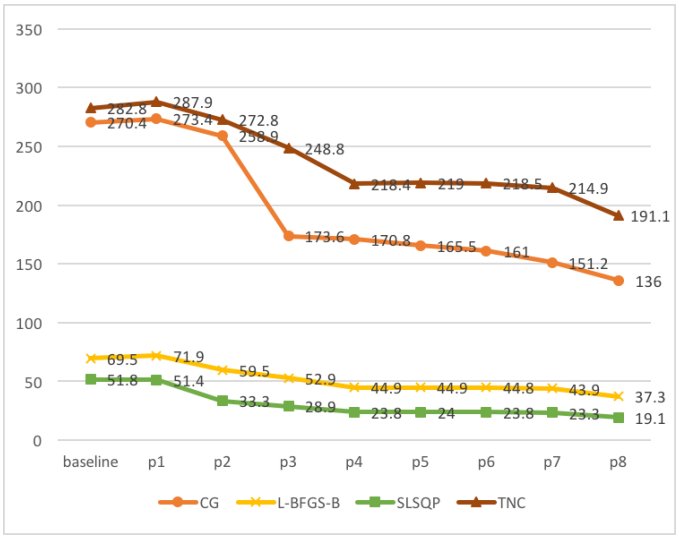


(e) Power Sum

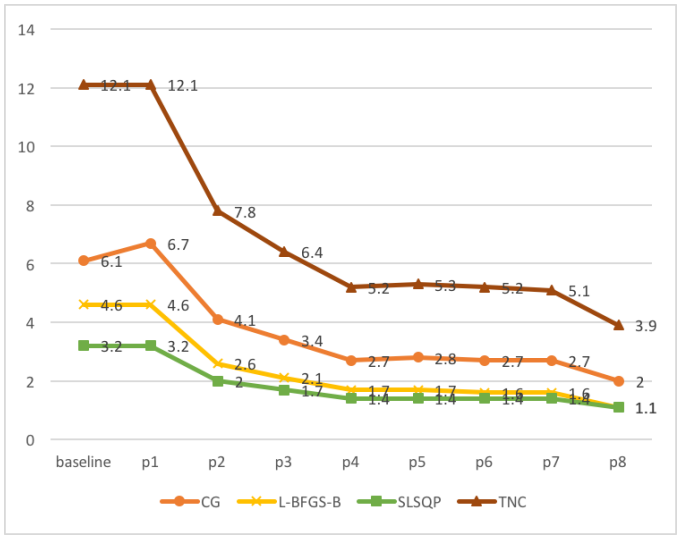


(f) Zakharov

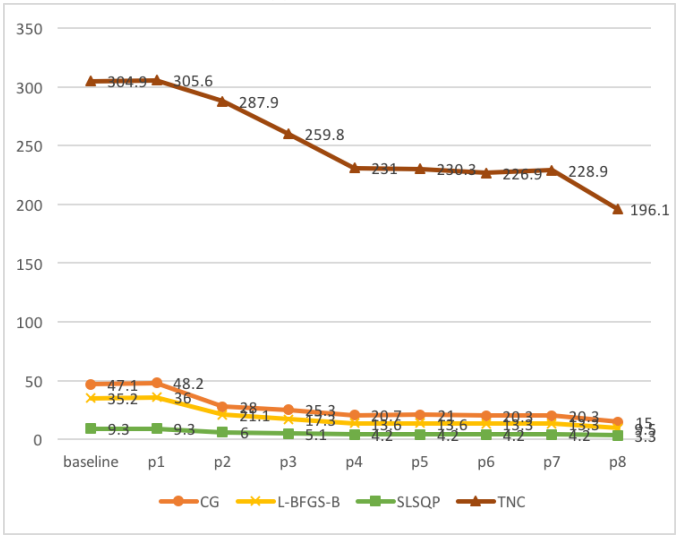
Fig. 4: Performance Study



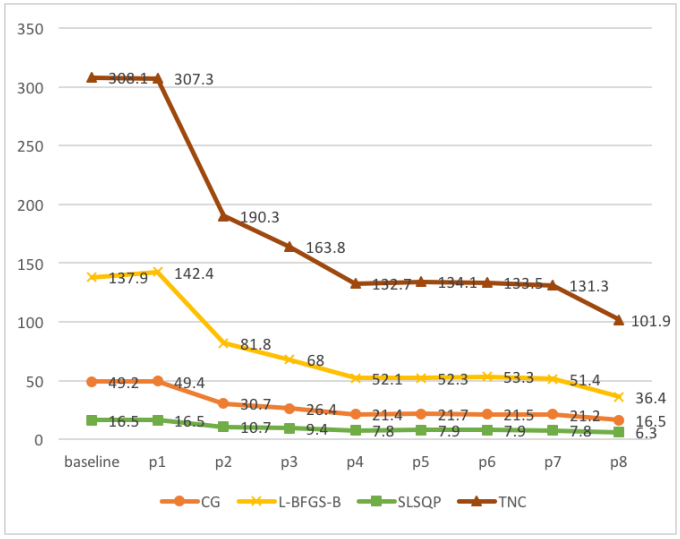
(a) Rosenbrock



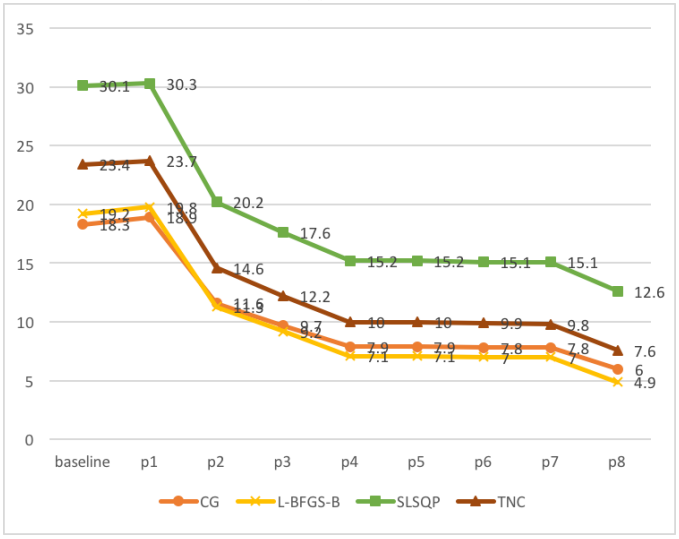
(b) Giunta



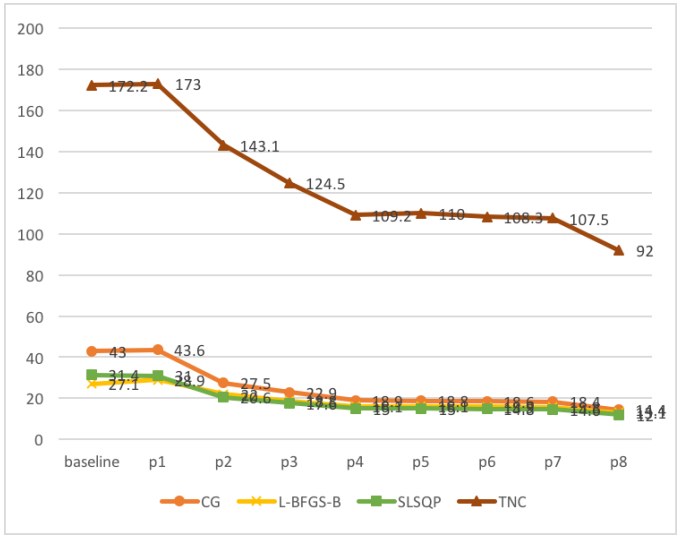
(c) Trid



(d) Sum of Different Powers



(e) Power Sum



(f) Zakharov

Fig. 5: Performance Study

Additionally, with 8 parallel processes, our algorithm reduces the running time to $23.9\%t$ (when L-BFGS-B is applied to Giunta) at most, and to $67.6\%t$ (when TNC is applied to Rosenbrock) at least, which corresponds to a 4X speedup and 1.5X speedup, respectively, where the *speedup* is defined as T_{base}/T_{p8}.

3) Correctness: We also manually inspected the results of the parallelized versions from p1 to p8, and confirmed that the results were the same as the ones in the non-parallelized version. Empirically, this indicates that our approach does not affect the convergence of the optimizers or the accuracy of the final results.

We also logged the points evaluated during optimization and found that the parallelized versions evaluated the same sequence of points as the non-parallelized ones. This indicates that our algorithm does not change the optimization logic, and explains why the parallelised versions produce the same results as the original ones.

4) Real-world Applications: In this section, we present a real-world application of our approach in the scientific-computing field. We worked with a group of computational chemists to compute the *ground-state energy* E_g of a set of molecules—an important open research question [24]. E_g is closely related to the eigenvalue (and eigenvector) of the *Hamiltonian* H of the molecule—a matrix that summarizes the dynamics of the molecule. Applying a classical eigensolver to compute the eigenvalue is computationally intractable when the dynamics of the molecules are large and complex. Researchers have proposed the VQE algorithm [24], [32], [43] to address this challenge. We will briefly explain it.

VQE is based on the variational method, which in turn is based on the *variational principle* [32], according to which the eigenvector is the vector $\vec{x}$ that minimizes the function $f(\vec{x})$ in the *Rayleigh quotient form* [32], [43]. Simply speaking, function $f(\vec{x})$ conducts the computation related to the Hamiltonian H and the input vector $\vec{x}$. Such computation is very expensive and involves numerous tensor-product operations [44]. Furthermore, the variational method creates another function $g(\vec{\theta})$ to generate the input vector $\vec{x}$ of f. Therefore, the goal is to find the $\vec{\theta}$ values that minimize the $f(g(\vec{\theta}))$ or $F(\theta)$, where F is the composition of f and g. Under the hood, to solve this problem, the variational method relies on a classical optimizer, which simply treats F as a black-box function.

We focused on the LiH molecule [42], for which the black-box function F depends on 8 parameters. Our implementation is based on the Qiskit Aqua open-source project [1], which implements the variational method mentioned above.

The evaluation results are shown in Figure 6. In Figure 6 (a), we show the running time (Y-axis) taken by each version (X-axis), similar to Figure 5. In Figure 6 (b), we show the normalized result, i.e., the ratio (Y-axis) between each version (X-axis) and the baseline, similar to Figure 4. Here we omit the p5, p6 and p7 versions because their results are very similar to the p4 version, as explained in Section III-D2.

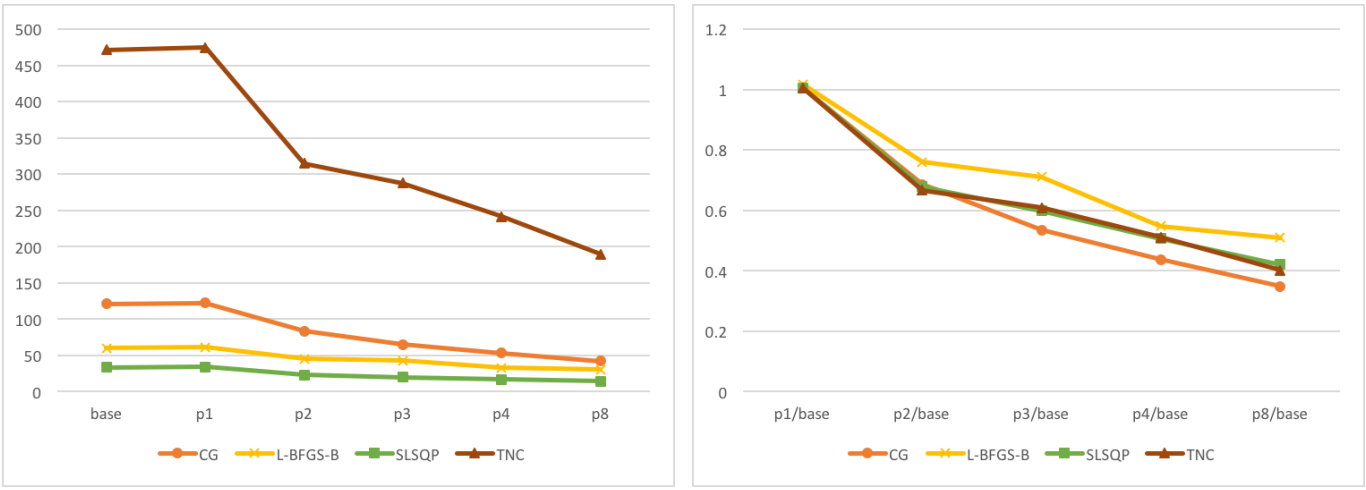
From the results, we draw conclusions consistent with the studies on the benchmark functions above. First, based on the $R_{p1/base}$ ratio, we found that the p1 version takes almost the same time as the baseline version. More precisely, the overhead incurred by the extra run-time logic in the p1 version is just $\leq 2\%$—a negligible overhead. Second, while we increase the number of parallel worker processes, the running time decreases. The p2, p3, p4 and p8 versions take on average 69.8%, 61.4%, 50.1% and 42.1% of the running time of the baseline version, respectively. In particular, the p4 and the p8 versions achieved a 2X and 2.4X speedup, respectively. This confirms that our algorithm effectively speeds up the numerical optimization. Third, we inspected the final results produced by different versions and found them to be identical. This confirms that our optimization does not affect the accuracy of the results or the convergence of the optimizers.

IV. Related Work

Numerical optimization [29] has been extensively used in many areas, including, but not limited to. AI, finance, scientific computing and operations research. It can be classified based on different criteria. Specifically, it can be classified as constrainted optimization or unconstrainted optimization depending on whether the search space is constrained or not. Our parallelization is applicable to both kinds. It can also be classified into local and global optimization, where local optimization has the advantage of high speed and global optimization has the advantage of not getting stuck at local optima. For demonstration purposes, our evaluation is based on local optimizers. However, our approach applies to any gradient-based local and global optimizer. For instance, we have successfully tested our approach on the global optimizer DIRECT-L [28].

Stochastic Gradient Descent (SGD) [2] is a numerical optimization algorithm popular in deep learning. Theoretically, L-BFGS-B and CG (both evaluated in this work) converge faster than SGD because they use both the first-order and second-order derivatives to guide the search, following the quasi-newton method [10]. This is more advanced than the gradient descent, which uses only the first-order derivatives. Researchers also conducted a comparison in the context of deep learning, and concluded that L-BFGS-B and CG outperform SGD practically in some cases [26]. In fact, L-BFGS-B is also included in popular deep-learning frameworks, such as Tensorflow [41] and PyTorch [35].

Additional work [12], [18] has been proposed to parallelize optimizers. For example, Fei et al. [18] parallelize L-BFGS-B using many GPU cores. Their work holds a different assumption on the characteristic of each function evaluation. In particular, it assumes that each function evaluation can be conducted fast by a GPU core, which is different from our assumption, according to which each function evaluation takes a long time. More importantly, existing parallelization work is specific to one particular optimizer (e.g., L-BFGS-B) and requires non-trivial engineering efforts to modify the code of the optimizer, thereby defying the extension to



(a) Running Time

(b) Ratio

Fig. 6: Real-world Application

other optimizers. In contrast, our approach is independent of the optimizer and hence is generally applicable to multiple optimizers. Our work shares some conceptual similarity with speculative execution [23], [36], which predicts the upcoming workload, precomputes and caches the results. There are many other improvements of the numerical optimization [45], which mostly focus on revising the algorithms from a mathematical perspective. Differently, our work tackles the optimization problem from a compiler optimization perspective. Besides, researchers [15], [16] have made interesting observations (e.g., triangle inequality) that help achieve significant speedup of machine learning applications.

V. Conclusion

We proposed a parallelization algorithm based on a general pattern to speed up numerical optimization. During numerical optimization, our algorithm detects the computational pattern, predicts the neighboring points, precomputes the results for them, and caches the results so that future requests for evaluating the neighboring points can be immediately completed. Besides, our design imposes no changes to the optimizer. Instead, it modifies the callback target function, which leads to several software-engineering advantages, including simplicity, modularity and portability. The evaluation results on both standard benchmarks and a real-world application shows that our approach possesses the following unique characteristics:

1) It incurs negligible overhead,
2) It effectively speeds up optimization, and
3) It does not affect the accuracy of the results or the convergence of the optimizers.

Our algorithm applies to gradient-based optimizers, which turn out to have the best performance among all optimizers, and is particularly useful when used to parallelize classical numerical optimization in hybrid quantum/classical variational algorithms, such as VQE and QAOA. Given that users often pick the optimizers with the best performance, we believe parallelizing such optimizers is highly useful.

VI. Disclaimer

This paper was prepared for information purposes by the Future Lab for Applied Research and Engineering (FLARE) Group of JPMorgan Chase & Co. and its affiliates, and is not a product of the Research Department of JPMorgan Chase & Co. JPMorgan Chase & Co. makes no explicit or implied representation and warranty, and accepts no liability, for the completeness, accuracy or reliability of information, or the legal, compliance, tax or accounting effects of matters contained herein. This document is not intended as investment research or investment advice, or a recommendation, offer or solicitation for the purchase or sale of any security, financial instrument, financial product or service, or to be used in any way for evaluating the merits of participating in any transaction.

References

[1] Qiskit Aqua, 2018. URL: https://github.com/Qiskit/qiskit-aqua.

[2] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. Tensorflow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 265–283, Savannah, GA, 2016. USENIX Association.

[3] Vahid Beiranvand, Warren Hare, and Yves Lucet. Best practices for comparing optimization algorithms. *Optimization and Engineering*, 18(4):815–848, Dec 2017.

[4] Derek Bingham. Optimization test problems, 2018. URL: https://www.sfu.ca/~ssurjano/optimization.html.

[5] Derek Bingham. Power sum function, 2018. URL: https://www.sfu.ca/~ssurjano/powersum.html.

[6] Derek Bingham. Rosenbrock function, 2018. URL: https://www.sfu.ca/~ssurjano/rosen.html.

[7] Derek Bingham. Sum of different powers function, 2018. URL: \\s://www.sfu.ca/~ssurjano/sumpow.html.

[8] Derek Bingham. Trid function, 2018. URL: https://www.sfu.ca/~ssurjano/trid.html.

[9] Derek Bingham. Zakharov function, 2018. URL: https://www.sfu.ca/~ssurjano/zakharov.html.

[10] Richard H. Byrd, Peihuang Lu, Jorge Nocedal, and Ciyou Zhu. A limited memory algorithm for bound constrained optimization. *SIAM J. Sci. Comput.*, 16(5):1190–1208, September 1995.

[11] Pei Chen. Hessian matrix vs. gauss-newton hessian matrix. *SIAM J. Numer. Anal.*, 49(4):1417–1435, July 2011.

[12] Weizhu Chen, Zhenghao Wang, and Jingren Zhou. Large-scale l-bfgs using mapreduce. In *Proceedings of the 27th International Conference on Neural Information Processing Systems - Volume 1*, NIPS'14, pages 1332–1340, Cambridge, MA, USA, 2014. MIT Press.

[13] A. Costa and G. Nannicini. Rbfopt, 2018. URL: https://github.com/coin-or/rbfopt.

[14] Joachim Dahl and Lieven Vandenberghe. CVXOPT: A python package for convex optimization, 2008. URL: http://www.abel.ee.ucla.edu/cvxopt.

[15] Yufei Ding, Lin Ning, Hui Guan, and Xipeng Shen. Generalizations of the theory and deployment of triangular inequality for compiler-based strength reduction. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2017, pages 33–48, New York, NY, USA, 2017. ACM.

[16] Yufei Ding, Yue Zhao, Xipeng Shen, Madanlal Musuvathi, and Todd Mytkowicz. Yinyang k-means: A drop-in replacement of the classic k-means with consistent speedup. In Francis Bach and David Blei, editors, *Proceedings of the 32nd International Conference on Machine Learning*, volume 37 of *Proceedings of Machine Learning Research*, pages 579–587, Lille, France, 07–09 Jul 2015. PMLR.

[17] Edward Farhi, Jeffrey Goldstone, and Sam Gutmann. A quantum approximate optimization algorithm. *arXiv:1411.4028*, 11 2014.

[18] Yun Fei, Guodong Rong, Bin Wang, and Wenping Wang. Technical section: Parallel l-bfgs-b algorithm on gpu. *Comput. Graph.*, 40:1–9, May 2014.

[19] H.J. Ferreau, C. Kirches, A. Potschka, H.G. Bock, and M. Diehl. qpOASES: A parametric active-set algorithm for quadratic programming. *Mathematical Programming Computation*, 6(4):327–363, 2014.

[20] Stephen G. Nash. Newton-type minimization via the lanczos method. 21:770–788, 08 1984.

[21] Irene. Benchmark optimization problems, 2018. URL: https://irene.readthedocs.io/en/latest/benchmarks.html.

[22] Momin Jamil and Xin-She Yang. A literature survey of benchmark functions for global optimization problems. *CoRR*, abs/1308.4008, 2013.

[23] Mark C. Jeffrey, Suvinay Subramanian, Maleen Abeydeera, Joel Emer, and Daniel Sanchez. Data-centric execution of speculative parallel programs. In *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-49, pages 5:1–5:13, Piscataway, NJ, USA, 2016. IEEE Press.

[24] Abhinav Kandala, Antonio Mezzacapo, Kristan Temme, Maika Takita, Markus Brink, Jerry M. Chow, and Jay M. Gambetta. Hardware-efficient variational quantum eigensolver for small molecules and quantum magnets. *Nature*, 549(7671):242–246, 2017.

[25] D. Kraft. *A Software Package for Sequential Quadratic Programming*. Deutsche Forschungs- und Versuchsanstalt für Luft- und Raumfahrt Köln: Forschungsbericht. Wiss. Berichtswesen d. DFVLR, 1988.

[26] Quoc V. Le, Jiquan Ngiam, Adam Coates, Abhik Lahiri, Bobby Prochnow, and Andrew Y. Ng. On optimization methods for deep learning. In *Proceedings of the 28th International Conference on International Conference on Machine Learning*, ICML'11, pages 265–272, USA, 2011. Omnipress.

[27] J.A. Nelder and RA Mead. A simplex method for function minimization comput. 7, 01 1965.

[28] NLopt. Nlopt, 2018. URL: https://nlopt.readthedocs.io/en/latest/NLopt_Algorithms/.

[29] Jorge Nocedal and Stephen J. Wright. *Numerical Optimization*. Springer, New York, NY, USA, second edition, 2006.

[30] Yvan Notay. Flexible conjugate gradients. *SIAM J. Sci. Comput*, 22:1444–1460, 2000.

[31] Ruben E. Perez, Peter W. Jansen, and Joaquim R. R. A. Martins. pyOpt: A Python-based object-oriented framework for nonlinear constrained optimization. *Structures and Multidisciplinary Optimization*, 45(1):101–118, 2012.
O'Brien. A variational eigenvalue solver on a photonic quantum processor, Jul 2014.

[33] M. J. D. Powell. An efficient method for finding the minimum of a function of several variables without calculating derivatives. *The Computer Journal*, 7(2):155–162, 1964.

[34] M. J. D. Powell. A Direct Search Optimization Method That Models the Objective and Constraint Functions by Linear Interpolation. In Susana Gomez and Jean-Pierre Hennart, editors, *Advances in Optimization and Numerical Analysis, Proceedings of the 6th Workshop on Optimization and Numerical Analysis, Oaxaca, Mexico*, volume 275, pages 51–67, Dordrecht, 1994. Kluwer Academic Publishers.

[35] PyTorch. Pytorch l-bfgs, 2018. URL: https://pytorch.org/docs/stable/_modules/torch/optim/lbfgs.html.

[36] Arun Raman, Greta Yorsh, Martin Vechev, and Eran Yahav. Sprint: Speculative prefetching of remote data. In *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '11, pages 259–274, New York, NY, USA, 2011. ACM.

[37] Anders W. Sandvik. Numerical solutions of the schrodinger equation, 2018. URL: http://physics.bu.edu/~py502/lectures4/schrod.pdf.

[38] Scipy. Scipy optimizers, 2018. URL: https://docs.scipy.org/doc/scipy/reference/optimize.html.

[39] Spark. Spark configuration, 2018. URL: https://spark.apache.org/docs/latest/configuration.html.

[40] Richard S. Sutton, David McAllester, Satinder Singh, and Yishay Mansour. Policy gradient methods for reinforcement learning with function approximation. In *Proceedings of the 12th International Conference on Neural Information Processing Systems*, NIPS'99, pages 1057–1063, Cambridge, MA, USA, 1999. MIT Press.

[41] Tensorflow. Tensorflow scipy optimizer interface, 2018. URL: https://www.tensorflow.org/api_docs/python/tf/contrib/opt/ScipyOptimizerInterface.

[42] Wikipedia. Lih, 2018. URL: https://en.wikipedia.org/wiki/Lithium_hydride.

[43] Wikipedia. Rayleigh quotient, 2018. URL: https://en.wikipedia.org/wiki/Rayleigh_quotient.

[44] Wikipedia. Tensor product, 2018. URL: https://en.wikipedia.org/wiki/Kronecker_product.

[45] R. Zhao, W. B. Haskell, and V. Y. F. Tan. Stochastic l-bfgs: Improved convergence rates and practical acceleration strategies. *IEEE Transactions on Signal Processing*, 66(5):1155–1169, March 2018.