Strobe: An Acceleration Meta-algorithm for Optimizing Robot Paths using Concurrent Interleaved Sub-Epoch Pods

Daniel Rakita[1], Bilge Mutlu, Michael Gleicher

***Abstract*— In this paper, we present a meta-algorithm intended to accelerate many existing path optimization algorithms. The central idea of our work is to strategically break up a waypoint path into consecutive groupings called "pods," then optimize over various pods concurrently using parallel processing. Each pod is assigned a color, either blue or red, and the path is divided in such a way that adjacent pods of the same color have an appropriate buffer of the opposite color between them, reducing the risk of interference between concurrent computations. We present a path splitting algorithm to create blue and red pod groupings and detail steps for a meta-algorithm that optimizes over these pods in parallel. We assessed how our method works on a testbed of simulated path optimization scenarios using various optimization tasks and characterize how it scales with additional threads. We also compared our meta-algorithm on these tasks to other parallelization schemes. Our results show that our method more effectively utilizes concurrency compared to the alternatives, both in terms of speed and optimization quality.**

I. Introduction

Many sub-problems in robotics require optimizing over *paths*, *i.e.,* ordered sequences of states within a configuration space [1]. For example, a hospital delivery robot may optimize its navigation trajectory to maximize distance from oncoming people in a hallway, or a home-care robot manipulator may optimize its joint-position path to move smoothly while maintaining an upright end-effector pose through a motion to avoid spilling a glass of water. A common strategy for optimizing over robot paths is to (1) discretize the path into a fixed set of waypoints connected by splines (often linear splines); and (2) use an optimization method to minimize some objective function relating to the set of waypoints subject to any constraints, starting from some initial condition. The objective function is often highly non-linear, especially in robotics problems, which adds to the computational complexity and typically precludes provably global optimal solutions.

While the path optimization strategy above is often successful for at least finding feasible, locally optimal solutions, especially when starting from a good initial condition, there are challenging trade-offs in practice. In particular, using a large number of waypoints is often desirable as it gives the optimizer more representational power to shape the path based on the specified objective function and constraints.

[1]Authors are with the Department of Computer Sciences, University of Wisconsin–Madison, Madison 53706, USA [rakita|bilge|gleicher]@cs.wisc.edu

This work was supported by a Microsoft Research PhD Fellowship, National Science Foundation award 1830242, and a NASA University Leadership Initiative (ULI) grant awarded to the UW-Madison and The Boeing Company (Cooperative Agreement # 80NSSC19M0124).

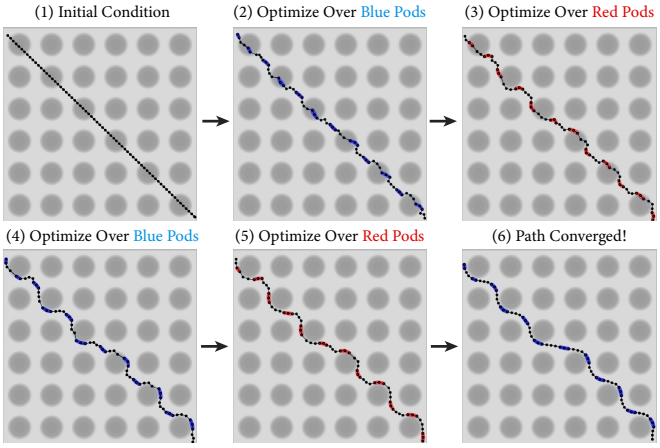

Fig. 1. We present a meta-algorithm, called *Strobe*, intended to accelerate path optimization algorithms. This figure illustrates our meta-algorithm optimizing a path to avoid areas of high cost (circles) in the image. Our method splits the path into blue and red consecutive subsets called "pods," and alternates optimizing over these groups until the path converges.

However, the more waypoints are used, the longer it takes the optimizer to converge, often taking minutes or hours to find a solution in the case of dense paths, especially if numerous initial conditions must be tried. These challenges significantly hinder path optimization methods from being used in important real-time or semi-online settings, such as shared control, teaching by demonstration, or inverse reinforcement learning, especially if many waypoints are needed to sufficiently parameterize the path.

In this paper, we present a meta-algorithm, called *Strobe*, intended to accelerate many path optimization algorithms. The central idea of our work is to strategically split the waypoint path into consecutive groupings called "pods," then optimize over various pods concurrently using parallel processing. Each pod is assigned a color, either blue or red, and the path is divided in such a way that adjacent pods of the same color have a buffer of the opposite color between them. Thus, each pod can reasonably assume that its surrounding neighborhood on the path is somewhat constant with respect to its own optimization, and each thread can focus on optimizing a small segment of the path with low risk of interfering with another thread's optimization. The method iteratively interleaves optimizing over blue and red pod groups until the process either converges or reaches a maximum number of epochs.

Our work offers three main contributions: (1) presenting a path splitting algorithm that attempts to maximize the separation between adjacent pods of the same color while minimizing the number of waypoints per pod given any num-

ber of available threads; (2) formulating a meta-algorithm that optimizes over these pods in parallel; and (3) providing open-source code for an implementation of our method.[1]

We assessed the efficacy of our method by running a testbed of experiments involving two-dimensional path optimization and higher dimensional robot path optimization tasks (§V). We assessed how our method works with various optimization algorithms and characterize how it scales when adding more threads. We also compared our meta-algorithm to other parallelization schemes on these tasks. Our results show that our method more effectively utilizes concurrency compared to the alternatives, both in terms of speed and optimization quality. Interestingly, our method often converges on higher quality paths than even its single-threaded counterparts. We conclude with an overall discussion about the implications of our work based on our results (§VI).

II. Related Works

Motion Planning—Robot motion planning is the process of finding collision-free, feasible paths from a start state to a goal state in configuration space [1]. Sampling-based motion planners often use random samples to bootstrap a search strategy and build a graph structure from start to goal [2]–[5]. Solutions found from these methods often have many extraneous and unnecessary motions, thus post-processing of the motions is common. We believe our method could help at this post-processing phase by quickly optimizing over an already feasible path, *e.g.*,, smooth velocities to shorten the path or adjust the motion such that a robot has a specified end-effector orientation throughout. Some sampling-based planners can accommodate limited objectives, such as achieving a shortest path in the limit [6], [7], though this additional objective comes at the cost of more computation time and cannot accommodate arbitrary objectives.

Trajectory Optimization—Path optimization has many similarities with the area of *trajectory optimization*, an approach used to optimize the motion of a body over time to match desired motion qualities (see Betts [8] for a review). In the context of robotics, trajectory optimization often involves optimizing a dynamical system that includes both state and control variables over time [9], [10]. For example, much work on quadruped and humanoid robots has been done on optimizing over the kinematic states, dynamics states, and joint torque control inputs of the robot to create robust and fluid locomotion trajectories [11], [12]. In this work, we only focus on optimizing over states along a path and not over control variables and time as well. However, we believe trajectory optimization methods, especially those that use direct collocation techniques, may benefit from the acceleration ideas presented in our work. We plan to explore these possibilities in future work.

Some trajectory optimization approaches in robotics are used primarily for kinematic, motion planning-based tasks [13]–[17]. Many of the gradient and non-gradient based optimization algorithms we test our method on in this work share similarities with the optimization procedures underlying these proposed approaches, so believe our method would be compatible with these approaches as well.

Optimization in Parallel—Our work draws on prior work that uses concurrency to accelerate optimization and planning procedures. Betts et al. proposed a scheme for performing trajectory optimization using a parallel shooting method [18]. Paine et al. presented a method that parallelized the ITOMP trajectory optimization algorithm where many initial conditions were tried at once [19]. We note that our method differs from a multiple-restart parallel algorithm, as our goal is to accelerate a single optimization path instance instead of trying multiple at once. Also, stochastic gradient descent (SGD) is a popular technique for optimizing high dimensional functions in parallel, particularly popularized by the training of neural networks [20]. Path optimization differs from the problems that SGD is commonly used on as it commonly considers derivatives and continutity between points rather than treating parts of the optimization vector as independent sub-problems. We compare our method to a generalized SGD strategy in our evaluation.

III. Technical Overview

In this section, we provide preliminaries about notation and the problem we are attempting to accelerate, and we overview the main technical premise of our method.

A. Path Optimization Problem Statement

A *path*, which we will denote as Γ, can be thought of as a function that maps some parameter, $u \in [0, 1]$, to some state, $\mathbf{q}$, within an n-dimensional configuration space, $\mathbf{q} \in \chi$. We will denote the space of all possible paths in χ as Ξ. From these definitions, the path optimization problem is as follows:

$$\begin{aligned} \Gamma^* &= \arg\min_{\Gamma \in \Xi} \mathbf{f}(\Gamma) \\ \mathbf{f}(\Gamma) &= \mathbf{g}(\Gamma(0), \Gamma(1)) + \int_0^1 \mathbf{h}(\Gamma(u), \Gamma'(u), \Gamma''(u), ...)\, du \\ \text{s.t.} \quad & \mathbf{c}_{\text{be}}(\Gamma(0), \Gamma(1)) = \mathbf{0}, \quad \mathbf{c}_{\text{bi}}(\Gamma(0), \Gamma(1)) < \mathbf{0} \\ & \mathbf{c}_{\text{e}}(\Gamma(u), \Gamma'(u), \Gamma''(u), ...) = \mathbf{0} \ \forall u \in [0, 1], \\ & \mathbf{c}_{\text{i}}(\Gamma(u), \Gamma'(u), \Gamma''(u), ...) < \mathbf{0} \ \forall u \in [0, 1] \end{aligned} \tag{1}$$

Here, $\mathbf{g}$ is a function that characterizes the quality of the path boundary points, $\mathbf{h}$ is a function that characterizes the quality of an interior point on the path and any of its derivatives at that point, $\mathbf{c}_{\text{be}}$ and $\mathbf{c}_{\text{bi}}$ are sets of equality and inequality constraints, respectively, that dictate whether particular boundary points are feasible, and $\mathbf{c}_{\text{e}}$ and $\mathbf{c}_{\text{i}}$ are sets of equality and inequality constraints, respectively, that dictate whether interior points along the path and any of their respective derivatives are feasible. Any of the functions in Equation 1 can be non-linear, which is often the case for robotics-related path optimization problems.

B. Path Optimization Discretization

While it may be possible to analytically solve some problems using Equation 1, *e.g.*, using calculus of variations [21], it is common to solve difficult, real-world problems by discretizing the path optimization problem and using

[1] https://github.com/uwgraphics/lynx

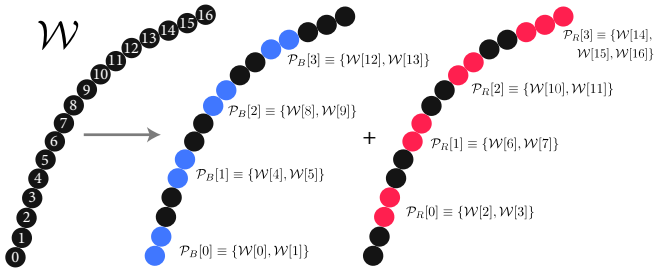

Fig. 2. Waypoints along path being divided into blue and red pods.

numerical methods over a finite set of variables. Formally, we consider a path Γ to be parameterized by a set of M ordered waypoint states, $\mathcal{W}$. We will use $\mathcal{W}[i]$ to refer to the i-th waypoint in the set $\mathcal{W}$. If all waypoint states are concatenated together into a single vector, we will denote this as $\mathbf{X}$ and call it the *full path vector*, *i.e.,* $\mathbf{X} \equiv [\mathcal{W}[0]^\top \ \mathcal{W}[1]^\top \cdots \ \mathcal{W}[\mathrm{M}]^\top]^\top \in \mathbb{R}^{\mathrm{M}*\mathrm{n}}$. The waypoints in $\mathcal{W}$ can always be converted to a continuous path Γ by treating the waypoints as knots along splines. For instance, it is standard to use linear splines, meaning the resulting continuous path would be linear interpolation between all waypoints from $\mathcal{W}[0]$ to $\mathcal{W}[\mathrm{M}]$.

Using the notation above, a discrete version of Equation 1 can be formulated as follows:

$$\begin{aligned} \mathbf{X}^* &= \underset{\mathbf{X}\in\mathbb{R}^{\mathrm{M}*\mathrm{n}}}{\arg\min}\, \mathbf{f}(\mathbf{X}) \\ \mathbf{f}(\mathbf{X}) &= \mathbf{g}(\mathcal{W}[0], \mathcal{W}[\mathrm{M}]) + \sum_{i=0}^{\mathrm{M}} \mathbf{h}(\mathcal{W}[i], \mathcal{W}'[i], \mathcal{W}''[i], ...) \\ \text{s.t. } & \mathbf{c}_{\mathrm{be}}(\mathcal{W}[0], \mathcal{W}[\mathrm{M}]) = \mathbf{0}, \quad \mathbf{c}_{\mathrm{bi}}(\mathcal{W}[0], \mathcal{W}[\mathrm{M}]) < \mathbf{0}, \\ & \mathbf{c}_{\mathrm{e}}(\mathcal{W}[i], \mathcal{W}'[i], \mathcal{W}''[i], ...) = \mathbf{0} \ \forall i \in \{0, 1, ..., \mathrm{M}\}, \\ & \mathbf{c}_{\mathrm{i}}(\mathcal{W}[i], \mathcal{W}'[i], \mathcal{W}''[i], ...) < \mathbf{0} \ \forall i \in \{0, 1, ..., \mathrm{M}\} \end{aligned} \tag{2}$$

Here, the derivatives at the waypoints $\mathcal{W}[\mathrm{i}]$ are typically approximated using the neighboring predecessors and successors, *e.g.,* using finite differencing.

Our work is focused on accelerating numerical optimization methods used to solve Equation 2. In the following section, we overview the structure of our meta-algorithm that achieves this acceleration.

C. Strobe Overview

The central idea of our method is to strategically split the set of waypoints $\mathcal{W}$ into smaller consecutive groupings called *pods*, then optimize over various pods concurrently using parallel processing (either multi-threaded CPU or GPU). A pod is a consecutive subset of waypoints in $\mathcal{W}$ that is assigned a color, either blue or red. If a pod is blue, it is considered part of the set $\mathcal{P}_\mathrm{B}$, and conversely, if a pod is red, it is considered part of the set $\mathcal{P}_\mathrm{R}$. Thus, $\mathcal{P}_\mathrm{B}$ and $\mathcal{P}_\mathrm{R}$ are sets of consecutive $\mathcal{W}$ subsets. We will refer to the i-th set in $\mathcal{P}_\mathrm{B}$ or $\mathcal{P}_\mathrm{R}$ as $\mathcal{P}_\mathrm{B}[i]$ and $\mathcal{P}_\mathrm{R}[i]$, respectively. An illustration of these sets can be seen in Figure 2. It is required that all waypoints in $\mathcal{W}$ belong to exactly one pod.

Using the definition of pods, the Strobe meta-algorithm follows three steps: (1) Optimize over all blue pods in parallel; (2) Optimize over all red pods in parallel; (3) Check if the whole path has converged subject to some tolerance. If it has converged, return the path, and if not, go back to step 1. Here, steps 1 and 2 can accommodate any base optimization algorithm that is compatible with the model's underlying objectives and constraints. Also, note that all waypoints will be optimized over after each pass through steps 1 and 2. We refer to one pass through both steps 1 and 2 as an *epoch* and each pass through either blue or red pods (step 1 or 2, respectively) as *sub-epochs*.

Algorithm 1: `split_path_into_pods`($\mathcal{W}$, *num_threads*, ℓ)

```
A ← ∅ // will store all pods before splitting them by color
num_pods ← num_threads * 2 // total number of pods. This may be an
    overestimate, but the true number will never exceed this.
wpp_max ← ℓ + 1 // initial maximum number of waypoints per pod
while not wpp_max * num_pods > |W| do
    wpp_max += 1 // add 1 to the maximum number of waypoints
wpp_min ← wpp_max − 1 // the minimum number of waypoints per pod will
    always be one less than the calculated maximum
n_min ← min(wpp_max * num_pods − |W|, num_pods) // number of pods
    that will have wpp_min nodes per pod
n_max ← num_pods − n_min // number of pods that will have wpp_max nodes
    per pod
c ← 0 // count variable
for i = 0...n_min do
    add_pod(W, A, c, wpp_min, ℓ)
for i = 0...n_max do
    add_pod(W, A, c, wpp_max, ℓ)
P_B ← ∅ // will store all blue pods
P_R ← ∅ // will store all red pods
curr_color ← blue
for i = 0...|A| do
    if curr_color ≡ blue then
        P_B ←+ A[i]
        curr_color ← red
    else
        P_R ←+ A[i]
        curr_color ← blue
return (P_B, P_R)
```

A challenge addressed in this work is how to effectively divide the waypoint path into separate pods. Our goal is to split the path in such a way that the whole path successfully converges, even with much work being done on separate threads that are not explicitly communicating with one another. We divide the path based on two heuristics: (1) The number of blue and red pods should be as high as possible, but no more than the number of available threads on the processor. This will help evenly distribute the work being done at once in steps 1 and 2 above; and (2) Pods should be no less than ℓ distance away (in terms of waypoint index distance) of the nearest pod of the same color. This helps ensure that parallel optimizations do not cross-pollinate and inadvertently form contradicting strategies toward convergence. It is good to choose ℓ based on the number of predecessors and successors used for approximating derivatives at a given waypoint in Equation 2. In Figure 2, $\ell = 2$, which is a reasonable value for many applications.

IV. Technical Details

In this section, we present our path splitting algorithm based on the heuristics above, and provide more details about the Strobe meta-algorithm.

A. Path Splitting Algorithm

Our path splitting algorithm can be seen in Algorithms 1 and 2. The algorithm progresses in four steps. In step 1,

Algorithm 2: `add_pod`($\mathcal{W}, \mathcal{A}, c, pod\_size, \ell$)

```
curr_pod ← ∅
for i=0...pod_size do
    curr_pod +← W[c]
    c += 1
    if c ≥ |W| then
        if |curr_pod| ≥ ℓ then
            A +← curr_pod
            return
        else
            for j = 0...|curr_pod| do
                A[|A| − 1] +← curr_pod[j] // absorb current pod into
                    previous pod
            return
A +← curr_pod
return
```

Algorithm 3: `strobe`(`optimize`, $\mathcal{W}$, *num_threads*, *num_epochs*, ℓ, *tol*)

```
1  (P_B, P_R) ← split_path_into_pods(W, num_threads, ℓ)
2  X ← stack(W) // stack waypoints into full state vector initial
       condition
3  f_prev ← ∞
4  for i=0...num_epochs do
5      for j=0...num_threads in parallel do
6          curr_pod ← P_B[j]
7          (a, b) ← get_bounding_indices_of_pod(curr_pod)
8          X[a : b]* ← optimize(X, a, b)
9          X[a : b] ← X[a : b]*
10     for j=0...num_threads in parallel do
11         curr_pod ← P_R[j]
12         (a, b) ← get_bounding_indices_of_pod(curr_pod)
13         X[a : b]* ← optimize(X, a, b)
14         X[a : b] ← X[a : b]*
15     f ← f(X)
16     Δf ← | f − f_prev |
17     if Δf < tol then
18         return X
```

the algorithm calculates a low number of waypoints per pod, *wpp_min*, and a high number of waypoints per pod, *wpp_max*, based on the number of threads available and ℓ. Note that the minimum number of waypoints per pod will never be less than ℓ based on the distance constraint, and *wpp_min* will always be *wpp_max* – 1. The values for *wpp_min* and *wpp_max* are selected such that they are the lowest values possible that will be guaranteed to cover all M waypoints given the number of threads available. This helps achieve heuristic 1 outlined in §III-C above. In step 2, the algorithm computes how many pods of length *wpp_min* and *wpp_max* it will take to reach the total number of waypoints, $|\mathcal{W}|$, denoted as *n_min* and *n_max*, respectively. These values will be exact unless the number of waypoints is too low to utilize all available threads.

In step 3, The algorithm adds *n_min* pods of length *wpp_min* and *n_max* pods of length *wpp_max* to a holding set $\mathcal{A}$. If at any point during these pod additions the total number of waypoints is reached (meaning the numbers in step 2 were overestimates), the remainder of the waypoints are either absorbed by the previously added pod or are appended as their own pod of truncated size. This decision is based on whether the excess nodes are greater than or less than ℓ as the method will not allow a pod of size less than ℓ. Lastly, step 4 involves the algorithm assigning every other pod in the set $\mathcal{A}$ to be either blue or red, such that they alternate from start to finish. Alternating colors between pods that are each guaranteed to be no less than length ℓ will ensure that heuristic 2 outlined in §III-C will also be achieved.

B. Strobe Details

A pseudocode view of the Strobe meta-algorithm can be seen in Algorithm 3. At a high level, the algorithm alternates between optimizing over blue and red pods in parallel over some maximum number of epochs or until the optimization converges subject to some specified tolerance. The `optimize` input can be any optimization algorithm that is compatible with the underlying objectives and constraints.

Because the optimizations per thread are only happening over a subset of the full path on a single pod, each must optimize a modified version of Equation 2:

$$\begin{aligned}
\mathbf{X}[a:b]^* &= \underset{\mathbf{X}[a:b]\in\mathbb{R}^{(b-a+1)*n}}{\arg\min} \mathbf{f}_s(\mathbf{X}, a, b) \\
\mathbf{f}_s(\mathbf{X}, a, b) &= \mathbf{g}(\mathcal{W}[0], \mathcal{W}[\mathrm{M}]) + \sum_{i=a}^{b} \mathbf{h}(\mathcal{W}[i], \mathcal{W}'[i], \mathcal{W}''[i], ...) \\
\text{s.t. } \mathbf{c}_{be}(\mathcal{W}[0], \mathcal{W}[\mathrm{M}]) &= \mathbf{0}, \quad \mathbf{c}_{bi}(\mathcal{W}[0], \mathcal{W}[\mathrm{M}]) < \mathbf{0}, \\
\mathbf{c}_e(\mathcal{W}[i], \mathcal{W}'[i], \mathcal{W}''[i], ...) &= \mathbf{0} \ \forall i \in \{0, 1, ..., \mathrm{M}\}, \\
\mathbf{c}_i(\mathcal{W}[i], \mathcal{W}'[i], \mathcal{W}''[i], ...) &< \mathbf{0} \ \forall i \in \{0, 1, ..., \mathrm{M}\}
\end{aligned} \tag{3}$$

Here, a and b are the lower and upper waypoint index bounds of the path being optimized over, respectively. Note that if $a > 0$ and $b < \mathrm{M}$, the $\mathbf{g}$ objective and $\mathbf{c}_{be}$ and $\mathbf{c}_{bi}$ constraints do not have to be considered in the particular optimization as these terms will be constant with respect to the given path segment.

The only parts of our meta-algorithm that involve the separate threads writing to a shared data structure occurs at Algorithm 3 lines 9 and 14 when the full state vector is updated based on each thread's individual optimization. This writing process is quite fast and we find that this requires little synchronization overhead in practice.

V. Evaluation

In this section, we overview the experiments designed to assess the efficacy of our method.

A. Implementation Details

A prototype implementation of our method was implemented in the Rust programming language. Parallel operations were run using the Rayon multithreading library[2]. All evaluations were run on a Lenovo Legion laptop with an i7-9750H processor with six physical cores (twelve threads) and 32GB RAM.

B. Evaluation Benchmark

We developed three benchmark scenarios to evaluate our method. The first scenario is the 2-dimensional *Circle Grid* planning example seen in Figure 1. Here, the dark circles represent areas of high cost, and the primary objective is to steer the path to lighter areas of lower cost. The path also has additional objectives to maintain smooth derivatives. The second scenario, *Upright EE* involves a 7-DOF Sawyer robot optimizing its motion to exhibit an upright end-effector orientation throughout the motion. The third

[2]https://docs.rs/rayon/1.5.0/rayon/

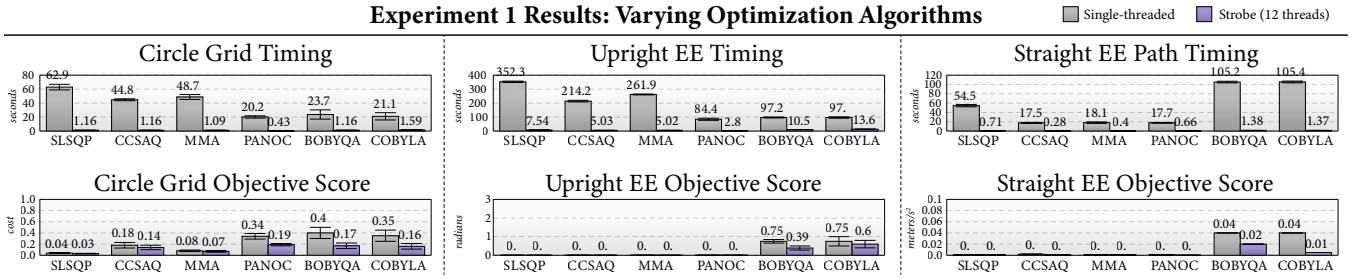

Fig. 3. Results from Experiment 1. Range values denote standard error. Gray bars on the left show results for the optimization algorithm run on a single thread, while the purple bars on the right show the results for the optimization algorithm run using the Strobe meta-algorithm.

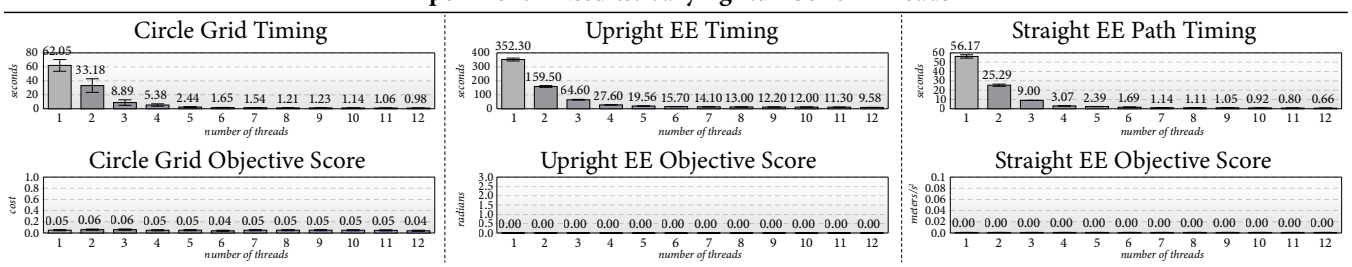

Fig. 4. Results from Experiment 2. Range values denote standard error.

scenario, *Straight EE Path* involves the robot optimizing its motion to minimize acceleration in end-effector space, thus tracing a straight line with its end-effector translation. In both of the Sawyer scenarios, the robot has additional objectives to maintain smooth joint velocities, acceleration, and jerks, and also avoid self-collisions.

Every condition tested in our experiments performed these scenarios 100 times with different initial conditions each time. All initial conditions involved taking a straight line between random points of a fixed distance within the bounds of the planning space, populating a desired number of waypoints along this line, and adding uniformly random noise to each waypoint. All conditions used identical initial conditions in our experiments to ensure fair comparisons. We set a maximum time limit of 20 minutes for each task.

C. Metrics

We report on two metrics per task: the computation time it took to converge, and some information per task that gives a sense of optimization quality. For the *Circle Grid* task, our optimization quality metric was the average "image cost" of the waypoints along the path, where dark areas of the image have cost 1, and light areas of the image have cost 0. For the *Upright EE* task, our optimization quality metric was the average rotation error away from the given goal orientation (in radians). For the *Straight EE Path* task, our optimization quality metric was the average acceleration in end-effector space along the path, where 0 designates an exact line. These accelerations were approximated by finite differencing over the waypoints along the path.

D. Experiment 1: Varying Optimization Algorithms

In Experiment 1, we assessed the performance of our method on six optimization algorithms: COBYLA (constrained optimization by linear approximation) [22], BOBYQA (bound optimization by quadratic approximation) [23], MMA (method of moving asymptotes) [24], CCSAQ (conservative convex separable approximation) [24], SLSQP (sequential least-squares quadratic programming) [25], and PANOC [26]. The COBYLA and BOBYQA solvers are non-derivative-based, while MMA, CCSAQ, and SLSQP, and PANOC solvers are derivative-based. All gradients for the derivative-based solvers were calculated using finite differencing. We compared the performance single-threaded versions of these algorithms to versions that use the Strobe acceleration meta-algorithm on 12 threads. All paths in this experiment contained 100 waypoints.

Our results, summarized in Figure 3, show that the Strobe algorithm greatly accelerates the path optimization process, often by over an order of magnitude. Also, the objective metrics indicate that this performance gain did not come at the expense of optimization quality. In fact, Strobe often achieved greater optimization quality, which we speculate may be because it is somewhat rare for all of the pods to fall into local minima at the same time.

E. Experiment 2: Varying Number of Threads

In Experiment 2, we vary the number of threads to assess how Strobe scales with more parallelization. We only used the SLSQP optimization algorithm in Experiment 2, and paths in this experiment contained 100 waypoints.

Our results for Experiment 2 can be seen in Figure 4. We see that Strobe scales successfully to more threads, again without sacrificing optimization quality.

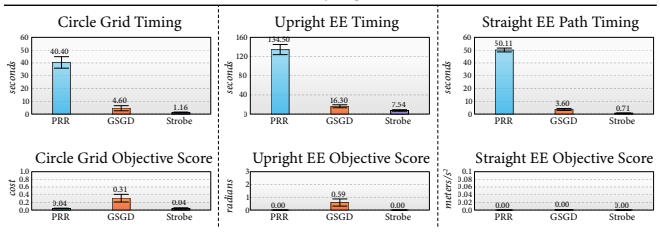

Fig. 5. Results from Experiment 3. Range values denote standard error.

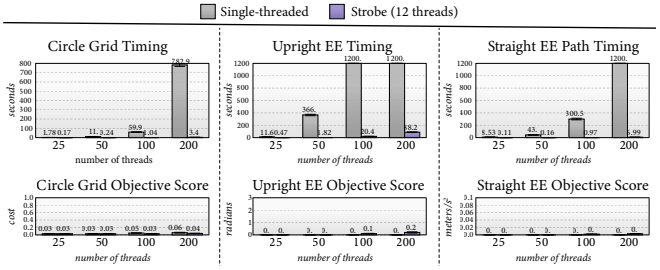

Fig. 6. Results from Experiment 4. Range values denote standard error. Gray bars on the left show results for the optimization algorithm run on a single thread, while the purple bars on the right show the results for the optimization algorithm run using the Strobe meta-algorithm.

F. Experiment 3: Varying Parallelization Scheme

In Experiment 3, we compare Strobe to other possible parallelization schemes. Specifically, we compare against *parallel random restart* (PRR) that uses all available threads to try optimizing from different initial conditions. The algorithm stops as soon as one thread successfully converges with respect to a given tolerance. We also compare against *generalized stochastic gradient descent* (GSGD) that is meant to resemble the common SGD strategy, but applied to other optimization algorithms other than just gradient descent. This condition involved each thread randomly selecting a subset of the waypoint path and optimizing over this subset. We again only used the SLSQP optimization algorithm in Experiment 3, and paths in this experiment contained 100 waypoints.

Our results from Experiment 3 can be seen in Figure 5. We see that Strobe outperforms the alternative parallelization schemes in terms of both run-time and optimization quality. GSGD did not find optimal solutions for all tasks. We believe this condition had a difficult time successfully converging because parallel optimizations sometimes overlapped and inadvertently disrupted and contradicted the optimization strategies of threads nearby. This appears to be a particular problem when optimizing over paths due to the structured and ordered nature of the problem. In contrast, Strobe ensures that none of its individual optimizations are affecting any other optimizations nearby, and we find that it successfully converges on our evaluation tasks.

G. Experiment 4: Varying Number of Waypoints

In Experiment 4, we vary the number of waypoints along the path being optimized. We compare the performance of single-threaded SLSQP with SLSQP using Strobe on waypoint paths of length 25, 50, 100, and 200.

Our results from Experiment 4 can be seen in Figure 6. We see that Strobe scales significantly better on denser paths than its single-threaded counterpart. In fact, SLSQP on a single thread did not find a solution within the maximum time (20 minutes) on the *Upright EE* or *Straight EE Path Timing* when using 200 waypoints. In contrast, these computations remain tractable using Strobe. On the sparser paths, we see that Strobe also greatly accelerated the computations. Often, the path successfully converged in tens of milliseconds.

VI. Discussion

In this work, we presented an acceleration meta-algorithm for robot path optimization based on the premise of splitting the input path into separate pods, and optimizing over various pods concurrently. In this section, we discuss the limitations and implications of our work.

A. Limitations

We note a number of limitations of our work that suggest future extensions. First, despite the connections between our work and trajectory optimizers, we have not shown our method to work on full trajectories that optimize over state and control variables over time. We plan to explore this exciting direction in future work. Second, while our pod splitting strategy appears to be successful across many scenarios, even compared to other parallelization schemes, we cannot claim that this is an optimal way to split and optimize over path subsets. Extensions to this work could characterize the speed and quality of convergence based on particular path splitting strategies. We also note that a theoretical analysis of our meta-algorithm was beyond the scope of our current work. In particular, the convergence behavior at the boundaries between blue and red pods may be of particular interest for future theoretical investigations. Further, we cannot guarantee that our method will enable a solver to converge to the same solution that it would have if the solver considered the whole path. Lastly, while showing more optimization scenarios and more optimization algorithms was beyond the scope of this work, we believe characterizing our method across more domains will be important in extensions to this work.

B. Implications

At a high level, we expect our method to have three levels of practical benefit: (1) Relatively sparse paths could be optimized in tens of milliseconds, meaning optimization at this level would be amenable for many real-time, performance critical control loops; (2) Moderately dense paths could be optimized in hundreds of milliseconds, meaning optimization over reasonably rich paths would be usable in many motion refinement loops in real-time settings; and (3) dense paths could be optimized in a few seconds, meaning optimization over complex and highly parameterized paths would be practical in semi-online, interactive applications. We believe these benefits could impact many areas of robotics, such as teleoperation, shared-control, or model predictive control.

REFERENCES

[1] S. M. LaValle, *Planning algorithms*. Cambridge University Press, 2006.

[2] ——, "Rapidly-exploring random trees: A new tool for path planning," 1998.

[3] L. E. Kavraki, P. Svestka, J.-C. Latombe, and M. H. Overmars, "Probabilistic roadmaps for path planning in high-dimensional configuration spaces," *IEEE Transactions on Robotics and Automation*, vol. 12, no. 4, pp. 566–580, 1996.

[4] D. Hsu, J.-C. Latombe, and R. Motwani, "Path planning in expansive configuration spaces," in *Proceedings of International Conference on Robotics and Automation*, vol. 3. IEEE, 1997, pp. 2719–2726.

[5] J. J. Kuffner Jr and S. M. LaValle, "RRT-connect: An efficient approach to single-query path planning," in *2000 IEEE International Conference on Robotics and Automation*, vol. 2, 2000.

[6] S. Karaman and E. Frazzoli, "Sampling-based algorithms for optimal motion planning," *The International Journal of Robotics Research*, vol. 30, no. 7, pp. 846–894, 2011.

[7] J. D. Gammell, S. S. Srinivasa, and T. D. Barfoot, "Informed RRT*: Optimal sampling-based path planning focused via direct sampling of an admissible ellipsoidal heuristic," in *2014 IEEE/RSJ International Conference on Intelligent Robots and Systems*. IEEE, 2014, pp. 2997–3004.

[8] J. T. Betts, "Survey of numerical methods for trajectory optimization," *Journal of guidance, control, and dynamics*, vol. 21, no. 2, pp. 193–207, 1998.

[9] M. Toussaint, "Robot trajectory optimization using approximate inference," in *Proceedings of the 26th annual international conference on machine learning*, 2009, pp. 1049–1056.

[10] M. Posa, C. Cantu, and R. Tedrake, "A direct method for trajectory optimization of rigid bodies through contact," *The International Journal of Robotics Research*, vol. 33, no. 1, pp. 69–81, 2014.

[11] S. Kuindersma, R. Deits, M. Fallon, A. Valenzuela, H. Dai, F. Permenter, T. Koolen, P. Marion, and R. Tedrake, "Optimization-based locomotion planning, estimation, and control design for the atlas humanoid robot," *Autonomous robots*, vol. 40, no. 3, pp. 429–455, 2016.

[12] H. Dai, A. Valenzuela, and R. Tedrake, "Whole-body motion planning with centroidal dynamics and full kinematics," in *2014 IEEE-RAS International Conference on Humanoid Robots*. IEEE, 2014, pp. 295–302.

[13] N. Ratliff, M. Zucker, J. A. Bagnell, and S. Srinivasa, "Chomp: Gradient optimization techniques for efficient motion planning," in *2009 IEEE International Conference on Robotics and Automation*. IEEE, 2009, pp. 489–494.

[14] M. Kalakrishnan, S. Chitta, E. Theodorou, P. Pastor, and S. Schaal, "STOMP: Stochastic trajectory optimization for motion planning," in *2011 IEEE International Conference on Robotics and Automation*. IEEE, 2011, pp. 4569–4574.

[15] C. Park, J. Pan, and D. Manocha, "Itomp: Incremental trajectory optimization for real-time replanning in dynamic environments," in *Twenty-Second International Conference on Automated Planning and Scheduling*, 2012.

[16] J. Schulman, J. Ho, A. X. Lee, I. Awwal, H. Bradlow, and P. Abbeel, "Finding locally optimal, collision-free trajectories with sequential convex optimization." in *Robotics: science and systems*, vol. 9, no. 1. Citeseer, 2013, pp. 1–10.

[17] Z. Marinho, A. Dragan, A. Byravan, B. Boots, S. Srinivasa, and G. Gordon, "Functional gradient motion planning in reproducing kernel hilbert spaces," *arXiv preprint arXiv:1601.03648*, 2016.

[18] J. T. Betts and W. P. Huffman, "Trajectory optimization on a parallel processor," *Journal of Guidance, Control, and Dynamics*, vol. 14, no. 2, pp. 431–439, 1991.

[19] C. Park, J. Pan, and D. Manocha, "Real-time optimization-based planning in dynamic environments using gpus," in *2013 IEEE International Conference on Robotics and Automation*. IEEE, 2013, pp. 4090–4097.

[20] T. Paine, H. Jin, J. Yang, Z. Lin, and T. Huang, "Gpu asynchronous stochastic gradient descent to speed up neural network training," *arXiv preprint arXiv:1312.6186*, 2013.

[21] I. M. Gelfand and R. A. Silverman, *Calculus of variations*. Courier Corporation, 2000.

[22] M. J. Powell, "A direct search optimization method that models the objective and constraint functions by linear interpolation," in *Advances in optimization and numerical analysis*. Springer, 1994, pp. 51–67.

[23] ——, "The bobyqa algorithm for bound constrained optimization without derivatives," *Cambridge NA Report NA2009/06, University of Cambridge, Cambridge*, pp. 26–46, 2009.

[24] K. Svanberg, "A class of globally convergent optimization methods based on conservative convex separable approximations," *SIAM journal on optimization*, vol. 12, no. 2, pp. 555–573, 2002.

[25] D. Kraft, "A software package for sequential quadratic programming," *Forschungsbericht- Deutsche Forschungs- und Versuchsanstalt fur Luft- und Raumfahrt*, 1988.

[26] L. Stella, A. Themelis, P. Sopasakis, and P. Patrinos, "A simple and efficient algorithm for nonlinear model predictive control," in *2017 IEEE 56th Annual Conference on Decision and Control (CDC)*. IEEE, 2017, pp. 1939–1944.