Parallelizing Randomized Convex Optimization

Michael Kamp[1], Mario Boley[2,3,1], and Thomas Gärtner[4]

[1]Fraunhofer IAIS, michael.kamp@iais.fhg.de
[2]Fritz Haber Institute of the Max Planck Society, Berlin, boley@fhi-berlin.mpg.de
[3]Max Planck Institute for Informatics, Saarbrücken
[4]University of Nottingham, thomas.gaertner@nottingham.ac.uk

Abstract

We define a general parallelization scheme for randomized convex optimization algorithms which optimize not directly observable functions. The scheme is consistent in the sense that it is able to maintain probabilistic performance guarantees of the underlying algorithm. At the same time—due to the parallelization—it achieves a speedup over the optimization algorithm of $\Theta(c^{1/\lg d})$, where c is the number of employed processors and d is the dimension of the solutions. A particular consequence is that all convex optimization problems that can be solved in polynomial time with probabilistic performance guarantees can also be solved efficiently in parallel, i.e., in polylogarithmic time on polynomially many processors. To achieve that, the scheme runs the algorithm in parallel to produce intermediate solutions of a weak guarantee and improves them by iteratively replacing solution sets by their Radon point.

1 Introduction

We consider the problem of parallelizing randomized optimization algorithms in settings where the quality of the solution cannot be directly observed but instead a probabilistic guarantee on the solution is provided. This subsumes several machine learning and optimization settings such as (regularized) empirical risk minimization for convex losses [14], stochastic convex optimization [11] and convex programming [3]. Assume an algorithm $\mathcal{A}$ that accepts a numerical parameter $N \in \mathbb{N}$ (e.g., in machine learning it denotes the number of training examples) and finds approximate solutions to an optimization problem $w^* = \arg\min_{w \in \mathcal{W}} f(w)$, where the solution space $\mathcal{W} \subseteq \mathbb{R}^d$ is a convex set and $f : \mathcal{W} \to \mathbb{R}$ is a quasi-convex function that cannot be directly observed. We call $\mathcal{A}$ consistent, if it provides a probabilistic guarantee of the following form. Its solutions are distributed according to a distribution $\mathcal{D}_N : \mathcal{W} \to [0, 1]$ which satisfies that for all $\epsilon > 0$ and $\Delta \in (0, 1)$ there exists an $N_0 \in \mathbb{N}$ such that for all $N \geq N_0$ it holds that

$$P_{w \sim \mathcal{D}_N} \left(|f(w) - f(w^*)| > \epsilon \right) < \Delta \ . \tag{1}$$

Typically, for more strict (ϵ, Δ)-guarantees a larger N is required but that increases the runtime cost $T(N)$. Furthermore, this runtime is usually at least polynomial in N so that for large-scale problems, in order to achieve a desired guarantee, substantial scalability issues arise. Thus, an efficient and consistent parallelization is desirable, i.e., one that improves the order of the runtime with the number of employed processors, while retaining (Δ, ϵ)-guarantees. Several such parallelization schemes already exist for concrete algorithms in specific settings but none of them is general and retains consistency at the same time [5, 10, 12, 15].

In this paper we provide a scheme that is consistent, efficient and generally applicable to all randomized convex optimization algorithms. Our main idea is to run the serial optimization algorithm $\mathcal{A}$ in parallel on c processors, each with a small parameter n, resulting in a short runtime but also in a

Algorithm 1: *Radon parallelization*

input : algorithm $\mathcal{A}$, parameter $N \in \mathbb{N}$, number of processors $c \in \mathbb{N}$
output: solution $w \in \mathcal{W} \subseteq \mathbb{R}^d$

1: $n \leftarrow {}^{N}/_{c}$
2: **run** $\mathcal{A}(n)$ on each of the c processors in parallel
3: **obtain** intermediate solutions $w_1, \dots, w_c$
4: $r \leftarrow d + 2$
5: $S = \{w_1, \dots, w_c\}$
6: **for** $i = 1, \dots, \lfloor \log_r c \rfloor$ **do**
7: **partition** S into subsets $S_1, \dots, S_{|S|/r}$ of size r
8: **calculate** $\mathfrak{r}(S_1), \dots, \mathfrak{r}(S_{|S|/r})$ in parallel
9: $S \leftarrow \{\mathfrak{r}(S_1), \dots, \mathfrak{r}(S_{|S|/r})\}$
10: **end for**
11: **return** a random $w \in S$

weak solution. We then combine these weak solutions using the iterated Radon point algorithm [4] to boost the confidence to the desired value $1 - \Delta$. A Radon point is defined as follows.

Definition 1 (Radon [9])**.** *For $S \subset \mathbb{R}^d$ a Radon point $\mathfrak{r}(S)$ is any point $\mathfrak{r} \in \langle A \rangle \cap \langle B \rangle$ for any $A, B \subset S$ with $A \cup B = S$, $A \cap B = \emptyset$ and $\langle A \rangle \cap \langle B \rangle \neq \emptyset$. Here, $\langle A \rangle$ denotes the convex hull of A. The Radon number $r \in \mathbb{N}$ is the smallest number such that for all sets S with $|S| \geq r$ a Radon point exists. For $\mathbb{R}^d$ it holds that $r = d + 2$ and a Radon point can be constructed in time $r^3 = (d + 2)^3$.*

The proposed scheme, denoted *Radon parallelization* $\mathcal{R}$ and given in Alg. 1, gets as input the parameters $N, c \in \mathbb{N}$. It runs c instances of the algorithm $\mathcal{A}$ with parameter $n = {}^{N}/_{c}$ in parallel on the given processors (line 1-2). Then the set of solutions (line 5) is iteratively partitioned into subsets of size r (line 7). For each subset, its Radon point is then calculated in parallel (line 8). The final step of each iteration is to replace the set of solutions by the set of their Radon points (line 9).

2 Consistency and Runtime

In the following, we show that *Radon parallelization* $\mathcal{R}$ is consistent and analyze its runtime for a desired (Δ, ϵ)-guarantee with respect to the runtime $T_{\mathcal{A}}(N)$ of $\mathcal{A}$ that is to be parallelized. For that, we consider the properties of $\mathcal{A}$ with respect to its input parameter $N \in \mathbb{N}$; other possible parameters are assumed fixed for the purpose of parallelization. Furthermore, for the analysis we fix $\epsilon > 0$ and analyze the dependency of Δ on N. Note that an improvement in Δ for a fixed ϵ is similar to an improvement in ϵ for a fixed Δ.

For a Radon point $\mathfrak{r}(W)$ it holds for all measures P over $\mathcal{W} \subseteq \mathbb{R}^d$, a quasi-convex function f, all $w \in \mathcal{W}$ and all $\epsilon \in \mathbb{R}_+$ that $P\left(|f(\mathfrak{r}(W)) - f(w^*)| > \epsilon\right) < \left(rP\left(|f(w) - f(w^*)| > \epsilon\right)\right)^2$ [8]. In particular, the result of the iterated Radon point algorithm improves Δ exponentially with $h = \log_r |\mathcal{W}|$, i.e., $P\left(|f(\mathfrak{r}) - f(w^*)| > \epsilon\right) < \left(rP\left(|f(w) - f(w^*)| > \epsilon\right)\right)^{2^h}$. For r^h solutions produced by $\mathcal{A}$ with confidence $1-\delta$, combining these solutions with the iterated Radon point algorithm results in a solution with a confidence of $1-(r\delta)^{2^h}$, i.e., $\Delta = (r\delta)^{2^h}$. In order to ensure that $\Delta < \delta$, the algorithm has to achieve $\delta < r^{-2^h/(2^h - 1)}$ when run with parameter n. If $\mathcal{A}$ is consistent, there always exists a minimum parameter $n_0 \in \mathbb{N}$ such that $\mathcal{A}$ reaches this confidence for all $n \geq n_0$. That is, the parameters N, c have to be set so that ${}^{N}/_{c} \geq n_0$. It follows that if $\mathcal{A}$ is consistent then $\mathcal{R}$ is consistent as well.

We now analyze the runtime $T_{\mathcal{R}}(N, c)$ of $\mathcal{R}$ using c processors. It can be decomposed into the runtime $T_{\mathcal{A}}(n)$ of $\mathcal{A}$ executed on each processor in parallel with $n = {}^{N}/_{c}$ and the runtime of the iterated Radon point algorithm. The Radon points in each level can be calculated in parallel with a runtime of r^3 for each Radon point [4]. The $h = \log_r c$ levels can be calculated consecutively resulting in a runtime of $r^3 \log_r c$ for the calculation of the final Radon point. With this, the runtime of $\mathcal{R}$ is $T_{\mathcal{R}}(N, c) = T_{\mathcal{A}}({}^{N}/_{c}) + r^3 \log_r c$. Using this result, we can follow that any polynomial time randomized convex optimization algorithm can be efficiently computed in parallel, i.e., with

polylogarithmic time on polynomially many processors. This is not trivial, since it requires that the overhead for combining the parallel solutions is at most logarithmic in the number of processors.

Proposition 2. *Let $\mathcal{A}$ be a consistent randomized convex optimization algorithm with polynomial runtime, i.e., $T_{\mathcal{A}}(N) \in \mathcal{O}(N^k)$. Then, the parallelization $\mathcal{R}$ of this algorithm has runtime $\mathcal{O}\left((\log_2 N)^k\right)$ on $\mathcal{O}(N)$ processors for $N > 2^{n_0} \in \mathbb{N}$.*

Proof. Let $n_0 \in \mathbb{N}$ be the minimum parameter such that $\mathcal{A}$ achieves a confidence of $1/r^2$. Set the number of processors to $c = N/\log_2 N$, i.e., $N/c \geq n_0$. Then the runtime of $\mathcal{R}$ is

$$T_{\mathcal{R}}(N, c) = T_{\mathcal{A}}\left(\frac{N}{c}\right) + r^3 \log_r c \in \mathcal{O}\left((\log_2 N)^k + \log_r\left(\frac{N}{\log_2 N}\right)\right) \in \mathcal{O}\left((\log_2 N)^k\right)$$

□

This runtime corresponds to the runtime of decision problems in the randomized variant of Nick's Class $\mathcal{RNC}$ [1]. Thus, with the proposed parallelization scheme we show that any problem in the optimization variant of the complexity class P, denoted $optP$ [7], is in the optimization variant of $\mathcal{RNC}$.

3 Speedup for Polynomial-Time Algorithms

The actual runtime of $\mathcal{R}$ is determined by the runtime of $\mathcal{A}$ in N and the dependency between N and the desired (ϵ, Δ)-guarantee. In this section we consider scenarios in which the parameter N required to achieve a given Δ for fixed ϵ depends polylogarithmically on $1/\Delta$, i.e., $N(\Delta) = \left(\alpha_\epsilon + \beta_\epsilon \log_2 \frac{1}{\Delta}\right)^k$, for some constants $\alpha_\epsilon \beta_\epsilon \in \mathbb{R}_+, k \in \mathbb{N}$. Moreover, assume that $\mathcal{A}$ has a polynomial runtime in N.

To motivate this class of algorithms we show that machine learning algorithms, such as support vector machines or linear regression, that perform (regularized) empirical risk minimization with respect to a convex loss function belong to this class. In empirical risk minimization, an algorithm $\mathcal{A}$ aims to find an element of a hypothesis class $\mathcal{W} \subseteq \mathbb{R}^d$ that minimizes the risk functional, i.e., the expected loss, $f(w) = \int l(w, x, y) dP(x, y)$ for a convex loss function $l \colon \mathcal{W} \times \mathcal{X} \times \mathcal{Y} \to \mathbb{R}_+$. Here, $\mathcal{X}$ denotes the input space and $\mathcal{Y}$ the target space, distributed according to an underlying data distribution $\mathcal{Q} \colon \mathcal{X} \times \mathcal{Y} \to [0, 1]$. Since $f(w)$ is not directly observable, the algorithm solves $\min_{w \in \mathcal{W}} \frac{1}{n} \sum_{(x,y) \in E} l(w, x, y)$ for a set E of N training examples drawn iid according to $\mathcal{Q}$ (implying a distribution $\mathcal{D}_N$ over its solutions). If the hypothesis class $\mathcal{W}$ has finite Vapnik-Chervonenkis dimension [13] or finite Rademacher complexity [2], then this approach is consistent [14] and we can express the sample size as $N(\Delta) = \left(\alpha_\epsilon + \beta_\epsilon \log_2 \frac{1}{\Delta}\right)^k$ [1]. For regularization this holds if the regularizer is down-weighted by a factor that decreases for increasing N [14]. The runtime is typically polynomial, e.g., $T(N) \in \mathcal{O}(N^3)$ for support vector machines.

We now analyze the speedup *Radon parallelization* achieves. For that, we derive the runtime of $\mathcal{R}$ under the above assumptions.

Proposition 3. *Given a randomized convex optimization algorithms $\mathcal{A}$ that achieves a confidence of $1 - \Delta$ when run with parameter $N_{\mathcal{A}}(\Delta) = (\alpha_\epsilon + \beta_\epsilon \log_2 1/\Delta)^k$. Then,* Radon parallelization $\mathcal{R}$ *applied to $\mathcal{A}$ on c processors achieves the same confidence $1 - \Delta$ when run with parameter*

$$N_{\mathcal{R}}(\Delta, c) = \left(\alpha_\epsilon c^{\frac{1}{k}} + \beta_\epsilon c^{\frac{1}{k}} \left(c^{-\frac{1}{\log_2 r}} \log_2 \frac{1}{\Delta} + \log_2 r\right)\right)^k .$$

Proof. Recall that $\mathcal{R}$ achieves $\Delta = (r\delta)^{2^h}$, thus $\delta = r^{-1} \Delta^{2^{-h}}$. To achieve δ at each processor, we have to set $n = N_{\mathcal{A}}(\delta)$, i.e.,

$$n = \left(\alpha_\epsilon + \beta_\epsilon \left(\frac{1}{2^{\log_r c}} \log_2 \frac{1}{\Delta} + \log_2 r\right)\right)^k .$$

[1] For finite Vapnik-Chervonenkis dimension the constants are $\alpha_\epsilon = 4 \ln 2 1/\epsilon^2, \beta_\epsilon = 4/\epsilon^2 \log_2 e$ and $k = 2$. For finite Rademacher Complexity they are $\alpha_\epsilon = 4 \ln 2 1/\epsilon^2, \beta_\epsilon = 4/\epsilon^2 \log_2 e$ and $k = 2$.

Since $N_{\mathcal{R}} = cn$, we get that $N_{\mathcal{R}}(\Delta, c) = c\left(\alpha_\epsilon + \beta_\epsilon \left({}^{1}\!/_{(2^{\log_r c})} \log_2({}^{1}\!/_{\Delta}) + \log_2 r\right)\right)^k$ and from ${}^{1}\!/_{(2^{\log_r c})} = c^{-{}^{1}\!/_{\log_2 r}}$ follows the result. □

The speedup of $\mathcal{R}$ is its runtime $T_{\mathcal{R}}(N_{\mathcal{R}}(\Delta), c)$ using c processor required to achieve a given (ϵ, Δ)-guarantee in contrast to the corresponding runtime $T_{\mathcal{A}}(N_{\mathcal{A}}(\Delta))$ of $\mathcal{A}$ on one processor, i.e., $\sigma_h = {}^{T_{\mathcal{A}}(N_{\mathcal{A}}(\Delta))}\!/_{T_{\mathcal{R}}(N_{\mathcal{R}}(\Delta), c)}$. For convenience, we abbreviate $T(N(\Delta))$ as $T(\Delta)$. Inserting $T_{\mathcal{R}}(\Delta, c) = T_{\mathcal{A}}(\delta) + (\log_r c) r^3$ with $\delta = r^{-1}\Delta^{2^{-h}}$ yields

$$\sigma_h(\Delta) = \frac{T_{\mathcal{A}}(\Delta)}{T_{\mathcal{A}}\left(r^{-1}\Delta^{1/2^{\log_r c}}\right) + (\log_r c) r^3} \quad . \tag{2}$$

Since the runtime of $\mathcal{A}$ is polynomial in N, we can express it w.l.o.g. by $T_{\mathcal{A}}(\Delta) = \left(\gamma_\epsilon + \rho_\epsilon \log_2 {}^{1}\!/_{\Delta}\right)^\kappa$ for some constants $\gamma_\epsilon, \rho_\epsilon \in \mathbb{R}_+$ and $\kappa \geq 1$. For algorithms with this runtime we get the following result for the speedup.

Proposition 4. *For a consistent randomized convex optimization algorithm $\mathcal{A}$ with runtime $T_{\mathcal{A}}(\Delta) = \left(\gamma_\epsilon + \rho_\epsilon \log_2 {}^{1}\!/_{\Delta}\right)^\kappa$ with $\gamma_\epsilon > 0, \kappa \geq 1$ and $\rho_\epsilon \geq {}^{1}\!/_{\log_2 r}$, the speedup through parallelization with $\mathcal{R}$ on c processors is*

$$\sigma_h(\Delta) \geq \left(\frac{1}{2^{\log_r c}}(\gamma_\epsilon + 1) + \frac{1}{\log_2 \frac{1}{\Delta}}\left((\gamma_\epsilon + 1)\log_2 r + \frac{r^3 \log_r c}{\rho_\epsilon}\right)\right)^{-\kappa} \quad .$$

Proof. Using $\delta = r^{-1}\Delta^{\frac{1}{2^{\log_r c}}}$ in $T_{\mathcal{R}}(\Delta, c) = T_{\mathcal{A}}(\delta) + (\log_r c) r^3$ and inserting this in Eq. 2 with $T_{\mathcal{A}}(\Delta) = \left(\gamma_\epsilon + \rho_\epsilon \log_2 {}^{1}\!/_{\Delta}\right)^\kappa \geq \left(\rho_\epsilon \log_2 {}^{1}\!/_{\Delta}\right)^\kappa$ yields

$$\sigma_h(\Delta) \geq \frac{\left(\rho_\epsilon \log_2 {}^{1}\!/_{\Delta}\right)^\kappa}{\left(\gamma_\epsilon + \rho_\epsilon\left(\frac{1}{2^{\log_r c}} \log_2 \frac{1}{\Delta} + \log_2 r\right)\right)^\kappa + r^3 \log_r c}$$

Since $\rho_\epsilon \geq {}^{1}\!/_{\log_2 r}$ it holds that $\rho_\epsilon\left(\frac{1}{2^{\log_r c}} \log_2 \frac{1}{\Delta} + \log_2 r\right) > 1$. Since for real numbers $y, a, k \in \mathbb{R}^+$ with $y \geq 1$ it holds that $(y + a)^k \leq y^k (a + 1)^k$, we get

$$\sigma_h(\Delta) \underbrace{\geq}_{\log_r c, r, \kappa \geq 1} \left(\frac{1}{2^{\log_r c}}(\gamma_\epsilon + 1) + \frac{1}{\log_2 \frac{1}{\Delta}}\left((\gamma_\epsilon + 1)\log_2 r + \frac{r^3 \log_r c}{\rho_\epsilon}\right)\right)^{-\kappa}$$

□

Using this result we determine the asymptotic speedup for $\Delta \to 0$.

Corollary 5. *For $\mathcal{A}$ as in Prop. 4, the asymptotic speedup is in $\Theta\left(c^{\kappa/\log_2(d)}\right)$, i.e., for $\Delta \to 0$ it holds that $\lim_{\Delta \to 0} \sigma_h(\Delta) = c^{\kappa/\log_2 r}(\gamma_\epsilon + 1)^{-\kappa}$.*

Proof. Using Prop. 4 and observing that for $\Delta \to 0$ the factor $\frac{1}{\log_2 \frac{1}{\Delta}} \to 0$, we get that

$$\lim_{\Delta \to 0} \sigma_h(\Delta) = \left(\frac{1}{2^{\log_r c}}(\gamma_\epsilon + 1)\right)^{-\kappa} = \left(\frac{2^{\log_r c}}{\gamma_\epsilon + 1}\right)^\kappa \underbrace{=}_{2^{\log_r c} = c^{1/\log_2 r}} \left(\frac{c^{1/\log_2 r}}{\gamma_\epsilon + 1}\right)^\kappa \in \Theta\left(c^{\kappa/\log_2 r}\right)$$

□

Since $r = d + 2$ and for polynomial time algorithms $\kappa \geq 1$, the speedup of $\Theta\left(c^{\kappa/\log_2(r)}\right)$ is larger than $\Theta\left(c^{1/\log_2(d)}\right)$.

4 Conclusion

The proposed scheme is - to the best of our knowledge - the first to provide solid error guarantees for parallel optimization of unobservable functions, which is not provided for, e.g., the widely used averaging of solutions. Since *Radon parallelization* achieves a substantial speed up for a large number of processors, our approach targets massively distributed systems, like clouds or applications on mobile phones, sensors or other ubiquitous intelligent systems. For future work it is interesting to apply the scheme to large-scale streaming scenarios, where models are learned online and combined periodically [5], inducing communication costs (which can be reduced by using dynamic synchronization protocols [6]).

References

[1] Sanjeev Arora and Boaz Barak. *Computational complexity: a modern approach*. Cambridge University Press, 2009.

[2] Peter L Bartlett and Shahar Mendelson. Rademacher and gaussian complexities: Risk bounds and structural results. *The Journal of Machine Learning Research*, 3:463–482, 2003.

[3] Dimitris Bertsimas and Santosh Vempala. Solving convex programs by random walks. *Journal of the ACM (JACM)*, 51(4):540–556, 2004.

[4] Kenneth L Clarkson, David Eppstein, Gary L Miller, Carl Sturtivant, and Shang-Hua Teng. Approximating center points with iterative radon points. *International Journal of Computational Geometry & Applications*, 6(03):357–377, 1996.

[5] Ofer Dekel, Ran Gilad-Bachrach, Ohad Shamir, and Lin Xiao. Optimal distributed online prediction using mini-batches. *The Journal of Machine Learning Research*, 13(1):165–202, 2012.

[6] Michael Kamp, Mario Boley, Daniel Keren, Assaf Schuster, and Izchak Sharfman. Communication-efficient distributed online prediction by dynamic model synchronization. In *Machine Learning and Knowledge Discovery in Databases*, pages 623–639. Springer, 2014.

[7] Mark W Krentel. The complexity of optimization problems. In *Proceedings of the eighteenth annual ACM symposium on Theory of computing*, pages 69–76. ACM, 1986.

[8] Olana Missura and Thomas Gärtner. Online optimisation in convexity spaces. In *Proceedings of the Workshop on Discrete and Combinatorial Problems in Machine Learning (DISML) at NIPS 2014*, 2014.

[9] Johann Radon. Mengen konvexer körper, die einen gemeinsamen punkt enthalten. *Mathematische Annalen*, 83(1):113–115, 1921.

[10] Rocco A Servedio. Perceptron, winnow, and pac learning. *SIAM Journal on Computing*, 31 (5):1358–1369, 2002.

[11] Shai Shalev-Shwartz, Ohad Shamir, Karthik Sridharan, and Nathan Srebro. Stochastic convex optimization. In *Proceedings of the 22nd Annual Conference on Learning Theory (COLT)*, 2009.

[12] Ohad Shamir and Nathan Srebro. On distributed stochastic optimization and learning. In *Proceedings of the 52nd Annual Allerton Conference on Communication, Control, and Computing*, 2014.

[13] Vladimir Naumovich Vapnik and Alexey Yakovlevich Chervonenkis. On the uniform convergence of relative frequencies of events to their probabilities. *Theory of Probability & Its Applications*, 16(2):264–280, 1971.

[14] Ulrike Von Luxburg and Bernhard Schölkopf. Statistical learning theory: Models, concepts, and results. In *Handbook for the History of Logic*, volume 10, pages 751–706. Cambridge University Press, 2009.

[15] Martin Zinkevich, Markus Weimer, Alexander J. Smola, and Lihong Li. Parallelized stochastic gradient descent. In *Advances in Neural Information Processing Systems (NIPS)*, pages 2595–2603, 2010.