# Convex Optimization

(EE227A: UC Berkeley)

## Lecture 22
## (Parallel, Distributed Optimization)

11 Apr, 2013

Suvrit Sra

# Simple parallelization / distribution

$$\min f(x) = \sum_{i=1}^m f_i(x)$$

$$x_{k+1} = x_k - \alpha_k \sum_{i=1}^m g_i(x_k),$$

where $g_i \in \partial f_i(x_k)$ — so that $\sum_i g_i \in \partial f(x_k)$

# Simple parallelization / distribution

$$\min f(x) = \sum_{i=1}^{m} f_i(x)$$

$$x_{k+1} = x_k - \alpha_k \sum_{i=1}^{m} g_i(x_k),$$

where $g_i \in \partial f_i(x_k)$ — so that $\sum_i g_i \in \partial f(x_k)$

♣ The sum has $m$ components

# Simple parallelization / distribution

$$\min f(x) = \sum_{i=1}^{m} f_i(x)$$

$$x_{k+1} = x_k - \alpha_k \sum_{i=1}^{m} g_i(x_k),$$

where $g_i \in \partial f_i(x_k)$ — so that $\sum_i g_i \in \partial f(x_k)$

- ♣ The sum has $m$ components
- ♣ Easy parallelization: compute each $g_i(x_k)$ on diff. processor

# Simple parallelization / distribution

$$\min f(x) = \sum_{i=1}^{m} f_i(x)$$

$$x_{k+1} = x_k - \alpha_k \sum_{i=1}^{m} g_i(x_k),$$

where $g_i \in \partial f_i(x_k)$ — so that $\sum_i g_i \in \partial f(x_k)$

- ♣ The sum has $m$ components
- ♣ Easy parallelization: compute each $g_i(x_k)$ on diff. processor
- ♣ Then collect the answers on a master node

# Simple parallelization / distribution

$$\min f(x) = \sum_{i=1}^{m} f_i(x)$$

$$x_{k+1} = x_k - \alpha_k \sum_{i=1}^{m} g_i(x_k),$$

where $g_i \in \partial f_i(x_k)$ — so that $\sum_i g_i \in \partial f(x_k)$

- ♣ The sum has $m$ components
- ♣ Easy parallelization: compute each $g_i(x_k)$ on diff. processor
- ♣ Then collect the answers on a master node
- ♣ Update $\alpha_k$ and $x_{k+1}$ in serial

# Simple parallelization / distribution

$$\min f(x) = \sum_{i=1}^{m} f_i(x)$$

$$x_{k+1} = x_k - \alpha_k \sum_{i=1}^{m} g_i(x_k),$$

where $g_i \in \partial f_i(x_k)$ — so that $\sum_i g_i \in \partial f(x_k)$

- ♣ The sum has $m$ components
- ♣ Easy parallelization: compute each $g_i(x_k)$ on diff. processor
- ♣ Then collect the answers on a master node
- ♣ Update $\alpha_k$ and $x_{k+1}$ in serial
- ♣ Share / Broadcast $x_{k+1}$ and repeat

# Simple parallelization / distribution

$$\min f(x) = \sum_{i=1}^{m} f_i(x)$$

$$x_{k+1} = x_k - \alpha_k \sum_{i=1}^{m} g_i(x_k),$$

where $g_i \in \partial f_i(x_k)$ — so that $\sum_i g_i \in \partial f(x_k)$

- ♣ The sum has $m$ components
- ♣ Easy parallelization: compute each $g_i(x_k)$ on diff. processor
- ♣ Then collect the answers on a master node
- ♣ Update $\alpha_k$ and $x_{k+1}$ in serial
- ♣ Share / Broadcast $x_{k+1}$ and repeat
- ♣ Highly **synchronized** computation

## Simple parallelization / distribution

$$\min f(x) = \sum_{i=1}^{m} f_i(x)$$

$$x_{k+1} = x_k - \alpha_k \sum_{i=1}^{m} g_i(x_k),$$

where $g_i \in \partial f_i(x_k)$ — so that $\sum_i g_i \in \partial f(x_k)$

- ♣ The sum has $m$ components
- ♣ Easy parallelization: compute each $g_i(x_k)$ on diff. processor
- ♣ Then collect the answers on a master node
- ♣ Update $\alpha_k$ and $x_{k+1}$ in serial
- ♣ Share / Broadcast $x_{k+1}$ and repeat
- ♣ Highly **synchronized** computation
- ♣ Makes sense if computing a single $g_i$ is much slower than the involved costs of *synchronization*

# More realistic methods

If even one of the processors is slow in computing its subgradient $g_i(x_k)$, the whole update gets blocked due to synchronization

# More realistic methods

If even one of the processors is slow in computing its subgradient $g_i(x_k)$, the whole update gets blocked due to synchronization

**Asynchronous updates**

$$x_{k+1} = x_k - \alpha_k \sum_{i=1}^{m} g_i(k - \delta_i)$$

where $g_i(k - \delta_i)$ is a *delayed subgradient*.

**Notation:** We write $g_i(k) \equiv g_i(x_k)$

# More realistic methods

- ♣ If no delay, then $\delta_i = 0$ – synchronized case

# More realistic methods

- ♣ If no delay, then $\delta_i = 0$ – synchronized case
- ♣ Each processor can have its own delay $\delta_i$

# More realistic methods

- ♣ If no delay, then $\delta_i = 0$ – synchronized case
- ♣ Each processor can have its own delay $\delta_i$
- ♣ If $g_i(k)$ **not available** from node $i$, **don't** block the update

# More realistic methods

- If no delay, then $\delta_i = 0$ – synchronized case
- Each processor can have its own delay $\delta_i$
- If $g_i(k)$ **not available** from node $i$, **don't** block the update
- instead we go ahead and use the **most recently available** subgradient $g_i(k - \delta_i)$ from processor $i$

# More realistic methods

- If no delay, then $\delta_i = 0$ – synchronized case
- Each processor can have its own delay $\delta_i$
- If $g_i(k)$ **not available** from node $i$, **don't** block the update
- instead we go ahead and use the **most recently available** subgradient $g_i(k - \delta_i)$ from processor $i$
- The delays can be random / arbitrary but **bounded**

# More realistic methods

- If no delay, then $\delta_i = 0$ – synchronized case
- Each processor can have its own delay $\delta_i$
- If $g_i(k)$ **not available** from node $i$, **don't** block the update
- instead we go ahead and use the **most recently available** subgradient $g_i(k - \delta_i)$ from processor $i$
- The delays can be random / arbitrary but **bounded**
- Key idea to analyze: view asynchronous method as an iterative gradient-method with deterministic or stochastic errors.

# More realistic methods

- If no delay, then $\delta_i = 0$ – synchronized case
- Each processor can have its own delay $\delta_i$
- If $g_i(k)$ **not available** from node $i$, **don't** block the update
- instead we go ahead and use the **most recently available** subgradient $g_i(k - \delta_i)$ from processor $i$
- The delays can be random / arbitrary but **bounded**
- Key idea to analyze: view asynchronous method as an iterative gradient-method with deterministic or stochastic errors.

| Delays impact speed of convergence |
|---|

# More realistic methods

- If no delay, then $\delta_i = 0$ – synchronized case
- Each processor can have its own delay $\delta_i$
- If $g_i(k)$ **not available** from node $i$, **don't** block the update
- instead we go ahead and use the **most recently available** subgradient $g_i(k - \delta_i)$ from processor $i$
- The delays can be random / arbitrary but **bounded**
- Key idea to analyze: view asynchronous method as an iterative gradient-method with deterministic or stochastic errors.

| Delays impact speed of convergence |
|---|

| Delay $\delta$, leads to convergence rate: $O(\sqrt{\delta/T})$. |
|---|

# More realistic methods

**Algorithm**

$$x_{k+1} = \underset{x}{\operatorname{argmin}} \quad \left\{ \langle g_i(k - \delta_i),\, x \rangle + \frac{1}{\alpha_k} \|x - x_k\|_2^2 \right\}$$

# More realistic methods

**Algorithm**

$$x_{k+1} = \underset{x}{\operatorname{argmin}} \quad \left\{ \langle g_i(k-\delta_i),\, x\rangle + \frac{1}{\alpha_k}\|x - x_k\|_2^2 \right\}$$

**Algorithm 2: Mirror descent version**

$$x_{k+1} = \underset{x}{\operatorname{argmin}} \quad \left\{ \langle g_i(k-\delta_i),\, x\rangle + \frac{1}{\alpha_k} D_\phi(x, x_k) \right\}$$

$D_\phi(x, y)$ is some strongly convex Bregman divergence

# More realistic methods

**Algorithm**

$$x_{k+1} = \underset{x}{\operatorname{argmin}} \quad \left\{ \langle g_i(k-\delta_i),\, x \rangle + \frac{1}{\alpha_k} \|x - x_k\|_2^2 \right\}$$

**Algorithm 2: Mirror descent version**

$$x_{k+1} = \underset{x}{\operatorname{argmin}} \quad \left\{ \langle g_i(k-\delta_i),\, x \rangle + \frac{1}{\alpha_k} D_\phi(x, x_k) \right\}$$

$D_\phi(x, y)$ is some strongly convex Bregman divergence

The above methods work for *stochastic optimization*

# More realistic methods

**Algorithm**

$$x_{k+1} = \underset{x}{\operatorname{argmin}} \quad \left\{ \langle g_i(k-\delta_i),\, x \rangle + \frac{1}{\alpha_k} \|x - x_k\|_2^2 \right\}$$

**Algorithm 2: Mirror descent version**

$$x_{k+1} = \underset{x}{\operatorname{argmin}} \quad \left\{ \langle g_i(k-\delta_i),\, x \rangle + \frac{1}{\alpha_k} D_\phi(x, x_k) \right\}$$

$D_\phi(x, y)$ is some strongly convex Bregman divergence

The above methods work for *stochastic optimization*

Rates depend on: *network topology* and *delay process*