Asynchronous Coordinate Update Methods

Ernest K. Ryu and Wotao Yin

Large-Scale Convex Optimization via Monotone Operators

Background

- Multi-core CPUs are common. To use CPU cores, we hope to parallelize coordinate updates
- Since (block) coordinates can depend on each other, their parallel updates need to exchange information
- Roughly speaking, if an update can compute without waiting for the latest information from the others, we call it asynchronous
- Asynchronism is important to the efficiency of parallel computing
- Without asynchronism, all updates would have to wait for the arrival of latest information, so the speed of parallel updates would be dictated by the slowest core, the most difficult update, and the longest communication delay
- This chapter introduces an asynchronous coordinate update method and analyzes its convergence
- Terminology: we call CPU core "computational agent" or "agent"

Coordinate partitioning

Let $\mathbb{T}\colon \mathbb{R}^n \to \mathbb{R}^n$ be θ-averaged with $\mathbb{T} = \mathbb{I} - \theta\mathbb{S}$.
Partition $x = (x_1, \ldots, x_m) \in \mathbb{R}^n$ and

$$\mathbb{T}(x) = \begin{bmatrix} (\mathbb{T}(x))_1 \\ \vdots \\ (\mathbb{T}(x))_i \\ \vdots \\ (\mathbb{T}(x))_m \end{bmatrix} \qquad \mathbb{T}_i(x) = \begin{bmatrix} x_1 \\ \vdots \\ x_{i-1} \\ (\mathbb{T}(x))_i \\ x_{i+1} \\ \vdots \\ x_m \end{bmatrix} \qquad \mathbb{S}_i(x) = \begin{bmatrix} 0 \\ \vdots \\ 0 \\ (\mathbb{S}(x))_i \\ 0 \\ \vdots \\ 0 \end{bmatrix}.$$

We begin with a standard synchronous implementation of

$$x^{k+1} = x^k - \eta\mathbb{S}x^k,$$

on multiple computational agents.

Synchronous parallel FPI

Multiple agents simultaneously run:

```
// p agents run while loop simultaneously
// x and s are vectors in shared memory
WHILE (not converged)
  1. WHILE (not all indices claimed)
       Claim index i not yet claimed
       Read x
       Write s[i] = eta*S[i](x)
  2. Synchronize: wait for all agents
  ---------------------------------------
  3. WHILE (not all indices claimed)
       Claim index i not yet claimed
       Write x[i] = x[i] - s[i]
  4. Synchronize: wait for all agents
  ---------------------------------------
```

Steps 2, 4 are *synchronization barriers*. Algorithm is *synchronous parallel*.

Synchronization

```
WHILE (not converged)
  1. WHILE (not all indices processed)
       Read x
       Write s[i] = eta*S[i](x)
  2. Synchronize: wait for all agents
  ----------------------------------------
  3. WHILE (not all indices processed)
       Write x[i] = x[i] - s[i]
  4. Synchronize: wait for all agents
  ----------------------------------------
```

At any time, the agents are either all in Steps 1–2 or all in Steps 3–4.

- Step 1 reads `x`, does not write to `x`. Step 3 reads `x`, writes to `x`.
- Step 2 prevents agents finished with Step 1 from proceeding to Step 3 and changing `x` while the other agents are still using `x` in Step 1.
- Step 4 prevents agents finished with Step 3 from proceeding to Step 1 and reading `x` while other agents are still changing `x`.

Cost of synchrony

As the number of computing agents grows, the cost of synchrony becomes significant.

Faster agents must wait idly for the slower ones.

The synchronization barrier is itself an algorithm with a cost.

Communication congestion is another cost that can occur when multiple agents simultaneously write data in Steps 1 and 3.

Asynchronous parallelism

An algorithm is *asynchronous parallel* if it avoids synchronization barriers.

Simply removing the synchronization barriers, however, does not work.

Naive barrier–free algorithm:

```
// p agents run the while loop asynchronously
// x and s are vectors in shared memory
WHILE (not converged)
  1. Select i from Uniform{1,2,...,m}
  2. Read x
  3. Compute s[i] = eta*S[i](x)
  4. Write x[i] = x[i] - s[i] //Incorrect!
```

Now, agents run completely uncoordinated, and the cost of synchronization is eliminated.

Now, the number of updates is determined by total computing power, rather than the slowest agent.

Stale information

```
// p agents run the while loop asynchronously
// x and s are vectors in shared memory
WHILE (not converged)
  1. Select i from Uniform{1,2,...,m}
  2. Read x
  3. Compute s[i] = eta*S[i](x)
  4. Write x[i] = x[i] - s[i] //Incorrect!
```

However, the algorithm does *not* work. It is neither the FPI $x^{k+1} = (\mathbb{I} - \eta\mathbb{S})x^k$ nor the RC-FPI $x^{k+1} = (\mathbb{I} - \eta\mathbb{S}_{i(k)})x^k$.

When an agent performs Step 4, other agents may have updated `x`, rendering `x` used to compute Step 3 outdated. In this case, we say `x` is *stale*. Step 4 does

$$x^{k+1} = x^k - \eta\mathbb{S}_{i(k)}\hat{x}^k$$

where $\hat{x}^k$ contains information *older than* x^k.

Asynchrony must be carefully considered and designed around.

Outline

Asynchronous coordinate-update fixed-point iteration

We can account for staleness if we enforce *exclusive access* in Step 4. An agent has exclusive access to `x` in shared memory if no other agent can read from or write to `x` simultaneously.

Asynchronous coordinate-update fixed-point iteration (AC-FPI) with an *operational definition*:

```
// p agents run the while loop asynchronously
// x is a vector stored in shared memory
WHILE (not converged)
  1. Select i from Uniform{1,2,...,m}
  2. Read x
  3. Compute s[i] = eta*S[i](x)
  4. Exclusively read x[i] and
                  write x[i] = x[i] - s[i]
```

While an agent is updating `x[i]` in Step 4, others cannot access `x[i]`.

If $(\mathbb{S}x)_i$ depends on only some blocks of x, an agent running Step 2 needs to read only those of `x`, unaffected by any exclusive access on the other blocks of `x` caused by other agents running Step 4 simultaneously.

Benefit of asynchrony

```
// p agents run the while loop asynchronously
// x is a vector stored in shared memory
WHILE (not converged)
  1. Select i from Uniform{1,2,...,m}
  2. Read x
  3. Compute s[i] = eta*S[i](x)
  4. Exclusively read x[i] and
                  write x[i] = x[i] - s[i]
```

AC-FPI removes explicit synchronization barriers, though it still requires exclusive access in writing to `x[i]`.

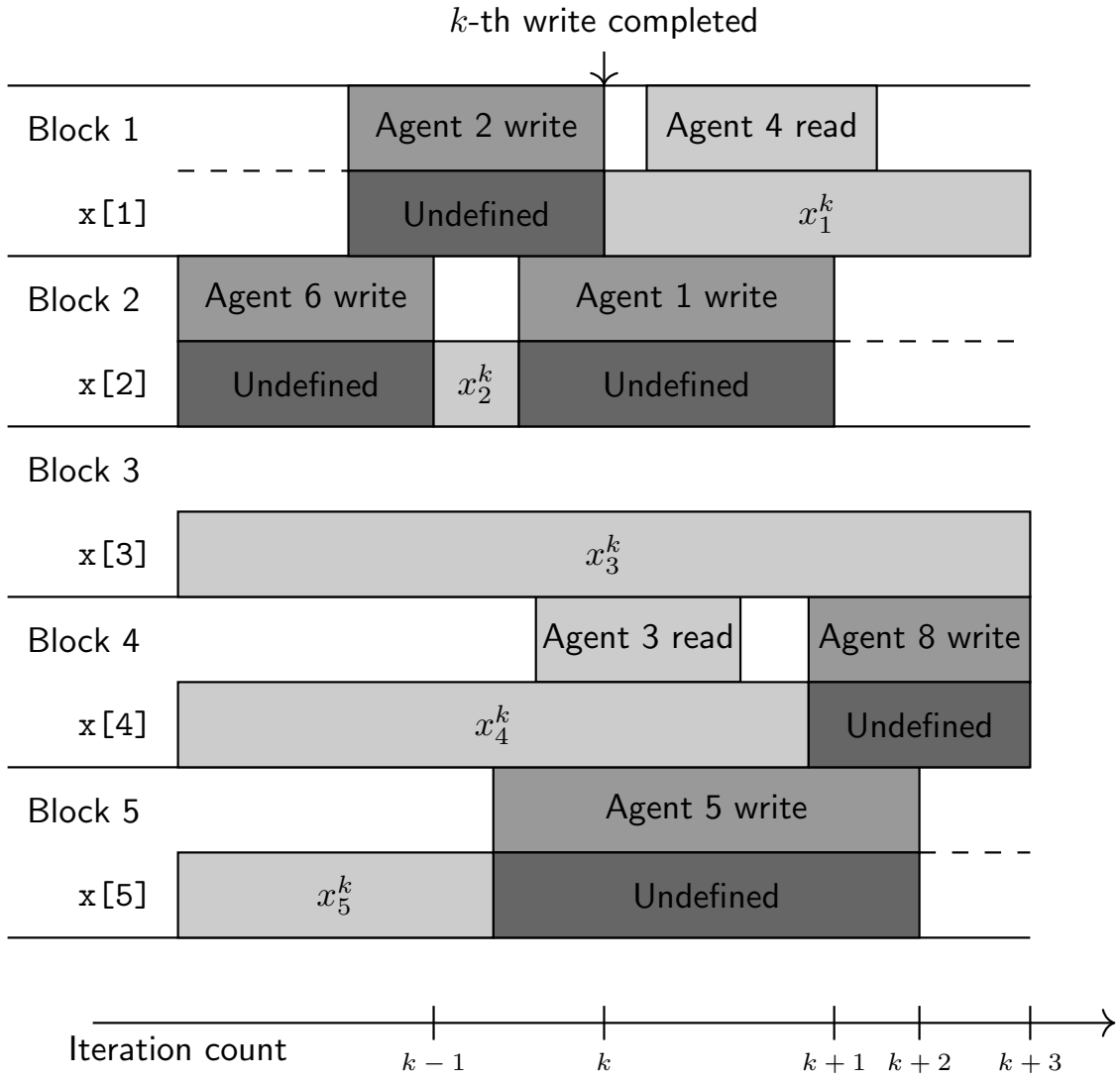
When there are much more blocks than agents, i.e., $p \ll m$, it is rare for an agent to wait for the release of a block's exclusive access; hence, most, albeit not all, idle time is eliminated.

Defining "iterates"

To analyze AC-FPI, we need a *mathematical definition*:

- x^0 is the state of `x` before the start of the algorithm.
- The kth iterate is $x^k = (x_1^k, \ldots, x_m^k)$.
- Iteration count increments by 1 when an agent completes an update of `x` in global memory, i.e., when an agent completes Step 4.
- When the iteration counter becomes k, if no agent is writing to `x[j]` then x_j^k is the state of `x[j]` at that time.
- When the iteration counter becomes k, if an agent is updating `x[j]` then x_j^k is what `x[j]` used to be right before the agent currently writing to the block started the update.
- If two or more agents finish updating different blocks at the same time, we break the tie arbitrarily.

(There are other valid approaches to defining the iterates. See exercises.)



In this illustration, x^k is defined when Agent 2 completes writing to Block 1. Since Blocks 3 and 4 are not being updated, x_3^k and x_4^k are the state of x[3] and x[4] at the time. Since Blocks 2 and 5 are being updated, x_2^k and x_5^k are the state of x[2] and x[5] before the writes had begun.

Delay notation of staleness

Write $i(k)$ for the index of the kth update. Again consider the IID random coordinate selection rule for simplicity.

The value of `x` that an agent has read in AC-FPI Step 2 may become stale by the time the agent performs Step 4. Write $\hat{x}^k$ for the stale value of `x` used for the update of x^k to x^{k+1}, i.e., $x^{k+1} = x^k - \eta \mathbf{S}_{i(k)} \hat{x}^k$.

It is possible that $\hat{x}^k \neq x^\ell$ for any $\ell = 0, \ldots, k$ since other agents can update blocks while $\hat{x}^k$ is being read block-by-block in Step 2. The exclusive access of Step 4 is on the block `x[i]`, not on the entire `x`.

Consider a coordinate-by-coordinate notion of staleness:

$$\hat{x}^k = \left(x_1^{k-d_1(k)}, \ldots, x_m^{k-d_m(k)}\right),$$

to denote that the ith block of $\hat{x}^k$ is outdated by $d_i(k) \geq 0$ iterations. $d_1(k), \ldots, d_m(k)$ are the *block delays*, and

$$\boldsymbol{d}(k) = (d_1(k), \ldots, d_m(k)) \in \mathbb{N}_+^m$$

is the *vector delay*. Write $\hat{x}^k = x^{k-\boldsymbol{d}(k)}$.

Mathematical definition of the AC-FPI

Mathematical definition of the AC-FPI:

$$x^{k+1} = x^k - \eta \mathbb{S}_{i(k)} x^{k-\boldsymbol{d}(k)}. \tag{1}$$

AC-FPI is a stochastic algorithm realized by the random variables $i(0), i(1), \ldots$ and $\boldsymbol{d}(0), \boldsymbol{d}(1), \ldots$.

Randomness of $i(0), i(1), \ldots$ is injected by design. Randomness of $\boldsymbol{d}(0), \boldsymbol{d}(1), \ldots$ comes from the randomness of $i(0), i(1), \ldots$ and the randomness of the agents' computation time.

ARock and convergence of the AC-FPI

The AC-FPI update $x^{k+1} = x^k - \eta \mathbb{S}_{i(k)} x^{k-\boldsymbol{d}(k)}$ models many asynchronous algorithms considered in the literature.

We analyze an instance of AC-FPI with the *ARock assumptions*:

- $i(0), i(1), \ldots$ are IID with uniform probability,
- $i(k)$ and $\boldsymbol{d}(\ell)$ are independent for $k = 0, 1, \ldots$ and $\ell \leq k$, and
- $\boldsymbol{d}(0), \boldsymbol{d}(1), \ldots$ is a stochastic process with nonincreasing $Q_0, Q_1, \ldots \in [0, 1]$ such that for every $k = 0, 1, \ldots$,

$$\text{Prob}\left[\max_{i=1,\ldots,m} d_i(k) \geq \ell \,\middle|\, \boldsymbol{d}(k-1), \ldots, \boldsymbol{d}(0), i(k-1), \ldots, i(0)\right] \leq Q_\ell,$$

$$\sum_{\ell=1}^{\infty} \ell (Q_\ell)^{1/2} < \infty. \tag{2}$$

This summability assumption is very mild. (C.f. Exercise 6.2.)

Convergence of AC-FPI

Theorem 3.

Assume $\mathbb{S}\colon \mathbb{R}^n \to \mathbb{R}^n$ *is* $(1/2)$*-cocoercive or, equivalently, that* $\mathbb{T} = \mathbb{I} - \theta\mathbb{S}$ *is* θ*-averaged with* $\theta \in (0,1)$*. Assume* $\operatorname{Fix}\mathbb{T} \neq \emptyset$*. Under the ARock assumptions, the AC-FPI* $x^{k+1} = x^k - \eta\mathbb{S}_{i(k)}x^{k-\boldsymbol{d}(k)}$ *with any starting point* $x^0 \in \mathbb{R}^n$ *and step size* η *obeying*

$$0 < \eta < \left(1 + \frac{2}{\sqrt{m}}\sum_{\ell=1}^{\infty} Q_\ell^{1/2}\right)^{-1}$$

converges to one fixed point with probability 1, i.e.,

$$x^k \to x^\star \quad \text{for some } x^\star \in \operatorname{Fix}\mathbb{T}$$

holds with probability 1. Furthermore, with probability 1,

$$\operatorname{dist}(x^k, \operatorname{Fix}\mathbb{T}) \to 0.$$

Discussion on stepsize and staleness

For the stepsize

$$0 < \eta < \left(1 + \frac{2}{\sqrt{m}} \sum_{\ell=1}^{\infty} Q_\ell^{1/2}\right)^{-1},$$

given a fixed distribution of delays, i.e., fixed values of $Q_0, Q_1, \ldots$, larger m is *more favorable*. That is, the staleness becomes less harmful as the number of blocks grows. If $m \to \infty$ with fixed $Q_0, Q_1, \ldots$, the stepsize requirement becomes the same as that of Theorem 2.

In practice, staleness may slow down AC-FPI. Therefore, RC-FPI and AC-FPI represent a trade-off between better and faster iterations.

Discussion of assumptions: Exclusive access

In the operational definition of the AC-FPI, Step 4 requires exclusive access. Otherwise, we would not be able to use the notation

$$\hat{x}^k = x^{k-\boldsymbol{d}(k)} = (x_1^{k-d_1(k)}, \ldots, x_m^{k-d_m(k)}),$$

as an agent can read a block while another agent is halfway through writing to the same block.

Discussion of assumptions: Independence

- We do assume that the sequence $i(0), i(1), \ldots$ is IID. When index `i` is sampled in Step 1 of the AC-FPI, we do not yet know which iteration count k the index will be associated with. If the computational cost of each block is equal, then the IID sampling of Step 1 makes $i(0), i(1), \ldots$ an IID sequence.
  - If the blocks have non-uniform computational costs, the choice of index affects the iteration count the update is assigned to and the IID assumption is violated. For example, if the jth block takes longer to compute, then $i(0) = j$ will have a lower probability than, say, $i(1000) = j$.
- We do assume that $i(k)$ and $\boldsymbol{d}(k)$ are independent for $k = 0, 1, \ldots$. This is realistic if the computational costs of the blocks are uniform.
  - On the other hand, if, for example, the jth block is much more expensive to compute than others and $i(k) = j$, then it is likely that $\boldsymbol{d}(k)$ contains large delays.

Removing these assumption makes the analysis much more difficult and leads to weaker results.

Discussion of assumptions: dependence allowed

- We do not assume $\boldsymbol{d}(0), \boldsymbol{d}(1), \ldots$ is an independent sequence.
  - It is likely that $\hat{x}^k$ and $\hat{x}^{k+1}$ are read at close points in time, and this makes $\boldsymbol{d}(k)$ and $\boldsymbol{d}(k+1)$ highly correlated.
- We do not assume $i(k)$ and $\boldsymbol{d}(\ell)$, $\ell > k$, are independent.
  - For example, if $i(k) = j$, then $d_j(k+1) > 0$ is very likely.

Even when the computational costs of the blocks are uniform, it is unrealistic to make these independence assumptions.

Discussion of assumptions: Delays

A common assumption in the literature is that the delays are bounded:

$$\max\{d_1(k), \ldots, d_m(k)\} \leq D \quad \text{for } k = 0, 1, \ldots$$

for some $D < \infty$. We do not make this assumption.

Proof of Theorem 3

Write $\mathbb{E}$ for the total expectation.

Write $\mathbb{E}_{i(k)}$ for the expectation over $i(k)$ conditioned on $\boldsymbol{d}(k), \ldots, \boldsymbol{d}(0), i(k-1), \ldots, i(0)$.

Write $\mathbb{E}_{\boldsymbol{d}(k)}$ for the expectation over $\boldsymbol{d}(k)$ conditioned on $\boldsymbol{d}(k-1), \ldots, \boldsymbol{d}(0), i(k-1), \ldots, i(0)$.

Write $\mathbb{E}_{i(k),\boldsymbol{d}(k)}$ for the expectation over $i(k)$ and $\boldsymbol{d}(k)$ conditioned on $\boldsymbol{d}(k-1), \ldots, \boldsymbol{d}(0), i(k-1), \ldots, i(0)$.

Note that the random variables $\boldsymbol{d}(k-1), \ldots, \boldsymbol{d}(0), i(k-1), \ldots, i(0)$ completely determine $x^k, \ldots, x^1$.

The proof is divided into three stages.

Proof of Theorem 3: Stage 1

Define the Lyapunov function

$$V^k = \|x^k - x^\star\|^2 + \frac{1}{m}\sum_{d=1}^{\infty} c_d \|x^{k-d+1} - x^{k-d}\|^2,$$

where $x^\star \in \operatorname{Fix} \mathbb{T}$ and $x^0 = x^{-1} = x^{-2} = \cdots$. Coefficients $c_d \geq 0$ will be determined later. Clearly, $V^k \geq 0$. We have $V^k < \infty$ since the sum is finite for a fixed $k < \infty$.

We show the key inequality

$$\mathbb{E}_{i(k),\boldsymbol{d}(k)} V^{k+1} \leq V^k - \frac{\eta}{m}\left(1 - \eta\left(1 + \frac{2}{\sqrt{m}}\sum_{d=1}^{\infty} Q_d^{1/2}\right)\right) \mathbb{E}_{\boldsymbol{d}(k)} \|\mathbb{S} x^{k-\boldsymbol{d}(k)}\|^2. \tag{3}$$

(To skip the details of Stage 1, skip the next six slides.)

Mathematical definition of AC-FPI gives us

$$\begin{aligned}\|x^{k+1}-x^\star\|^2 &= \|x^k-\eta\mathbb{S}_{i(k)}x^{k-\boldsymbol{d}(k)}-x^\star\|^2\\ &= \|x^k-x^\star\|^2-2\eta\langle\mathbb{S}_{i(k)}x^{k-\boldsymbol{d}(k)},x^k-x^\star\rangle+\eta^2\|\mathbb{S}_{i(k)}x^{k-\boldsymbol{d}(k)}\|^2.\end{aligned}$$

Independence between $i(k)$ and $\boldsymbol{d}(k)$ gives us

$$\mathbb{E}_{i(k)}\mathbb{S}_{i(k)}x^{k-\boldsymbol{d}(k)}=\frac{1}{m}\mathbb{S}x^{k-\boldsymbol{d}(k)},\quad \mathbb{E}_{i(k)}\|\mathbb{S}_{i(k)}x^{k-\boldsymbol{d}(k)}\|^2=\frac{1}{m}\|\mathbb{S}x^{k-\boldsymbol{d}(k)}\|^2$$

and

$$\mathbb{E}_{i(k)}\|x^{k+1}-x^\star\|^2=\|x^k-x^\star\|^2-\frac{2\eta}{m}\langle\mathbb{S}x^{k-\boldsymbol{d}(k)},x^k-x^\star\rangle+\frac{\eta^2}{m}\|\mathbb{S}x^{k-\boldsymbol{d}(k)}\|^2. \tag{4}$$

Using $(1/2)$-cocoercivity of $\mathbb{S}$, bound the inner-product term as

$$\begin{aligned}&-2\langle\mathbb{S}x^{k-\boldsymbol{d}(k)},x^k-x^\star\rangle\\ &\quad=-2\langle\mathbb{S}x^{k-\boldsymbol{d}(k)},x^{k-\boldsymbol{d}(k)}-x^\star\rangle-2\langle\mathbb{S}x^{k-\boldsymbol{d}(k)},x^k-x^{k-\boldsymbol{d}(k)}\rangle\\ &\quad\le-\|\mathbb{S}x^{k-\boldsymbol{d}(k)}\|^2-2\langle\mathbb{S}x^{k-\boldsymbol{d}(k)},x^k-x^{k-\boldsymbol{d}(k)}\rangle.\end{aligned} \tag{5}$$

Since the blocks have different delays, we decompose the second term of (5) over the blocks as

$$-2\langle \mathbb{S}x^{k-\boldsymbol{d}(k)}, x^k - x^{k-\boldsymbol{d}(k)}\rangle = 2\sum_{i=1}^{m}\langle (-\mathbb{S}x^{k-\boldsymbol{d}(k)})_i, x_i^k - x_i^{k-d_i(k)}\rangle$$
$$= \sum_{i=1}^{m}\sum_{d=1}^{d_i(k)} 2\langle (-\mathbb{S}x^{k-\boldsymbol{d}(k)})_i, x_i^{k-d+1} - x_i^{k-d}\rangle.$$

For each term in the summation, we apply Young's inquality

$$-2\langle u, v\rangle \leq \frac{1}{\varepsilon}\|u\|^2 + \varepsilon\|v\|^2 \qquad \forall\, \varepsilon > 0$$

to get

$$2\langle (-\mathbb{S}x^{k-\boldsymbol{d}(k)})_i, x_i^{k-d+1} - x_i^{k-d}\rangle \leq \frac{\eta}{\varepsilon_d}\|(\mathbb{S}x^{k-\boldsymbol{d}(k)})_i\|^2 + \frac{\varepsilon_d}{\eta}\|x_i^{k-d+1} - x_i^{k-d}\|^2,$$

where we choose $\varepsilon_d > 0$ later.

Define $\tau(k) = \max_{i=1,\ldots,m} d_i(k)$. Using $d_i(k) \le \tau(k)$ and swapping the orders of sums, we get

$$
\begin{aligned}
&-2\langle \mathbb{S}x^{k-\boldsymbol{d}(k)}, x^k - x^{k-\boldsymbol{d}(k)} \rangle \\
&\quad \le \sum_{i=1}^{m} \sum_{d=1}^{\tau(k)} \left(\frac{\eta}{\varepsilon_d} \|(\mathbb{S}x^{k-\boldsymbol{d}(k)})_i\|^2 + \frac{\varepsilon_d}{\eta} \|x_i^{k-d+1} - x_i^{k-d}\|^2 \right) \\
&\quad = \eta \left(\sum_{d=1}^{\tau(k)} \varepsilon_d^{-1} \right) \|\mathbb{S}x^{k-\boldsymbol{d}(k)}\|^2 + \frac{1}{\eta} \left(\sum_{d=1}^{\tau(k)} \varepsilon_d \|x^{k-d+1} - x^{k-d}\|^2 \right). \quad (6)
\end{aligned}
$$

Substituting (6) into (5) and substituting (5) into (4), we get

$$
\begin{aligned}
\mathbb{E}_{i(k)} \|x^{k+1} - x^\star\|^2 \le \|x^k - x^\star\|^2 - \frac{\eta}{m} \left(1 - \eta - \eta \sum_{d=1}^{\tau(k)} \varepsilon_d^{-1} \right) \|\mathbb{S}x^{k-\boldsymbol{d}(k)}\|^2 \\
+ \frac{1}{m} \sum_{d=1}^{\tau(k)} \varepsilon_d \|x^{k-d+1} - x^{k-d}\|^2. \quad (7)
\end{aligned}
$$

By the definition of V^k,

$$\mathbb{E}_{i(k)}V^{k+1} = \mathbb{E}_{i(k)}\|x^{k+1} - x^\star\|^2 + \frac{1}{m}\mathbb{E}_{i(k)}\sum_{d=1}^{\infty} c_d\|x^{k-d+2} - x^{k-d+1}\|^2$$

$$= \mathbb{E}_{i(k)}\|x^{k+1} - x^\star\|^2 + \frac{c_1}{m}\mathbb{E}_{i(k)}\|x^{k+1} - x^k\|^2 + \frac{1}{m}\sum_{d=2}^{\infty} c_d\|x^{k-d+2} - x^{k-d+1}\|^2.$$

We bound $\mathbb{E}_{i(k)}\|x^{k+1} - x^\star\|^2$ by (7), substitute

$$\mathbb{E}_{i(k)}\|x^{k+1} - x^k\|^2 = \mathbb{E}_{i(k)}\|\eta\mathbb{S}_{i(k)}x^{k-\boldsymbol{d}(k)}\|^2 = \frac{\eta^2}{m}\|\mathbb{S}x^{k-\boldsymbol{d}(k)}\|^2,$$

and decrement the summation index to get

$$\mathbb{E}_{i(k)}V^{k+1} \leq (\text{RHS of (7)}) + \frac{c_1\eta^2}{m^2}\|\mathbb{S}x^{k-\boldsymbol{d}(k)}\|^2 + \frac{1}{m}\sum_{d=1}^{\infty} c_{d+1}\|x^{k-d+1} - x^{k-d}\|^2$$

$$= \|x^k - x^\star\|^2 - \frac{\eta}{m}\left(1 - \eta - \frac{c_1\eta}{m} - \eta\sum_{d=1}^{\tau(k)}\varepsilon_d^{-1}\right)\|\mathbb{S}x^{k-\boldsymbol{d}(k)}\|^2$$

$$+ \frac{1}{m}\left(\sum_{d=1}^{\tau(k)}\varepsilon_d\|x^{k-d+1} - x^{k-d}\|^2 + \sum_{d=1}^{\infty} c_{d+1}\|x^{k-d+1} - x^{k-d}\|^2\right).$$

We now choose

$$\varepsilon_d = \frac{m^{1/2}}{Q_d^{1/2}} \quad \text{and} \quad c_d = \sum_{\ell=d}^{\infty} \varepsilon_\ell Q_\ell = m^{1/2} \sum_{\ell=d}^{\infty} {Q_\ell}^{1/2}, \quad d = 1, 2, \ldots.$$

By the assumption (2), $c_d < \infty$ for all d. Since

$$\begin{aligned}
\mathbb{E}_{\boldsymbol{d}(k)} \sum_{d=1}^{\tau(k)} \varepsilon_d \|x^{k-d+1} - x^{k-d}\|^2 &= \sum_{\ell=1}^{\infty} \mathrm{Prob}[\tau(k) = \ell] \sum_{d=1}^{\ell} \varepsilon_d \|x^{k-d+1} - x^{k-d}\|^2 \\
&= \sum_{d=1}^{\infty} \varepsilon_d \mathrm{Prob}[\tau(k) \geq d] \|x^{k-d+1} - x^{k-d}\|^2 \\
&\leq \sum_{d=1}^{\infty} \varepsilon_d Q_d \|x^{k-d+1} - x^{k-d}\|^2
\end{aligned}$$

and since $c_d = \varepsilon_d Q_d + c_{d+1}$,

we obtain

$$\begin{aligned}
\mathbb{E}_{i(k),\boldsymbol{d}(k)} V^{k+1} &\leq \|x^k - x^\star\|^2 - \frac{\eta}{m}\mathbb{E}_{\boldsymbol{d}(k)}\left[\left(1 - \eta - \frac{c_1\eta}{m} - \eta\sum_{d=1}^{\tau(k)} \varepsilon_d^{-1}\right)\|\mathbb{S}x^{k-\boldsymbol{d}(k)}\|^2\right] \\
&\quad + \frac{1}{m}\sum_{d=1}^{\infty} c_d \|x^{k-d+1} - x^{k-d}\|^2, \\
&\leq V^k - \frac{\eta}{m}\left(1 - \eta - \frac{c_1\eta}{m} - \eta\sum_{d=1}^{\infty} \varepsilon_d^{-1}\right)\mathbb{E}_{\boldsymbol{d}(k)}\|\mathbb{S}x^{k-\boldsymbol{d}(k)}\|^2, \\
&= V^k - \frac{\eta}{m}\underbrace{\left(1 - \eta\left(1 + \frac{2}{\sqrt{m}}\sum_{d=1}^{\infty} Q_d^{1/2}\right)\right)}_{>0 \text{ by assumption on } \eta \text{ in Theorem 3.}}\mathbb{E}_{\boldsymbol{d}(k)}\|\mathbb{S}x^{k-\boldsymbol{d}(k)}\|^2,
\end{aligned}$$

which is (3). As an aside, the coefficients c_d and ε_d are carefully chosen to construct the Lyapunov function, rather than being given by the the algorithm or the assumptions.

Proof of Theorem 3: Stage 2

Let $L > 0$ be large enough such that $1 - Q_L > 0$, which exists by assumption (2).

We show, for each integer $D > L$, there is a subsequence $k'_j \to \infty$ (as $j \to \infty$) such that

$$(x^{k'_j}, x^{k'_j+1}, \ldots, x^{k'_j+D-1}) \to (\bar{x}, \bar{x}, \ldots, \bar{x}) \in (\mathbb{R}^n)^D,$$

and $\mathbb{S}\bar{x} = 0$, i.e., $\bar{x} \in \operatorname{Fix} \mathbb{T}$.

(To skip the details of Stage 1, skip the next three slides.)

To make the dependence on $x^\star \in \operatorname{Fix} \mathbb{T}$ explicit, write

$$V^k(x^\star) = \|x^k - x^\star\|^2 + \frac{1}{m}\sum_{d=1}^{\infty} c_d \|x^{k-d+1} - x^{k-d}\|^2.$$

We apply the supermartingale convergence theorem to (3), apply the arguments of Proposition 1, and use $\|x^k - x^\star\|^2 \leq V^k$ to get

(i) $\mathbb{E}_{\boldsymbol{d}(k)} \|\mathbb{S} x^{k-\boldsymbol{d}(k)}\|^2 \to 0$,

(ii) $V^k(x^\star) \to V^\infty(x^\star)$ for all $x^\star \in \operatorname{Fix} \mathbb{T}$,

(iii) $\|x^k\| < B$ for all $k = 0, 1, \ldots$ for some $B < \infty$,

with probability 1.

Write $\mathcal{F}_k$ for the σ-algebra generated by $\boldsymbol{d}(k), \ldots, \boldsymbol{d}(0), i(k), \ldots, i(0)$. This implies, for all $k = 0, 1, \ldots,$

$$\text{Prob}\left[\max_{i=1,\ldots,m} d_i(k) < L \,\middle|\, \mathcal{F}_{k-1}\right] \geq 1 - Q_L > 0.$$

Let $\boldsymbol{b}(k)$ be a $\mathcal{F}_{k-1}$-measurable random variable defined as

$$\boldsymbol{b}(k) = \underset{\boldsymbol{b}<L}{\text{argmax}} \left\{\text{Prob}\left[\boldsymbol{d}(k) = \boldsymbol{b} \,\middle|\, \mathcal{F}_{k-1}\right]\right\},$$

where $\text{argmax}_{\boldsymbol{b}<L}$ is the maximizer over all $\boldsymbol{b} = (b_1, \ldots, b_m) \in \mathbb{N}_+^m$ satisfying $\max_{i=1,\ldots,m} b_i < L$. When argmax is not unique, break ties in a deterministic manner, say with the lexicographical ordering on $\mathbb{N}_+^m$. Then

$$\text{Prob}\left[\boldsymbol{d}(k) = \boldsymbol{b}(k) \,\middle|\, \mathcal{F}_{k-1}\right] \geq \frac{1 - Q_L}{L^m} > 0$$

since the event $\max_{i=1,\ldots,m} d_i(k) < L$ has probability at least $1 - Q_L$ with L^m possible realizations of $\boldsymbol{d}(k)$ and $\boldsymbol{b}(k)$ is defined as the most likely among them.

By (i), we have

$$\underbrace{\mathbb{E}_{\boldsymbol{d}(k)}\|\mathbb{S}x^{k-\boldsymbol{d}(k)}\|^2}_{\to 0} = \mathbb{E}\left[\|\mathbb{S}x^{k-\boldsymbol{d}(k)}\|^2 \,\middle|\, \mathcal{F}_{k-1}\right]$$
$$\geq \mathbb{E}\left[\mathbb{1}_{\{\boldsymbol{d}(k)=\boldsymbol{b}(k)\}}\|\mathbb{S}x^{k-\boldsymbol{b}(k)}\|^2 \,\middle|\, \mathcal{F}_{k-1}\right] \geq \frac{1-Q_L}{L^m}\|\mathbb{S}x^{k-\boldsymbol{b}(k)}\|^2 \to 0$$

as $k \to \infty$, so $\|\mathbb{S}x^{k-\boldsymbol{b}(k)}\|^2 \to 0$.

By the Borel–Cantelli lemma, there exists a subsequence $k_j \to \infty$ such that $\boldsymbol{d}(k_j+\ell) = \boldsymbol{b}(k_j+\ell)$ for $\ell = 0, 1, \ldots, D-1$.

Since x^{k_j} is bounded by (iii), there is a further subsequence $k'_j \to \infty$ such that $x^{k'_j} \to \bar{x}$. Since

$$\|x^{k'_j+\ell+1} - x^{k'_j+\ell}\|^2 = \eta^2\|\mathbb{S}_{i(k'_j+\ell)}x^{k'_j+\ell-\boldsymbol{b}(k'_j+\ell)}\|^2 \leq \eta^2\|\mathbb{S}x^{k'_j+\ell-\boldsymbol{b}(k'_j+\ell)}\|^2 \to 0$$

for $\ell = 0, 1, \ldots, D-1$, we have

$$(x^{k'_j}, x^{k'_j+1}, \ldots, x^{k'_j+D-1}) \to (\bar{x}, \bar{x}, \ldots, \bar{x}) \in (\mathbb{R}^n)^D.$$

If $D > L$, then $x^{k'_j+D-1-\boldsymbol{b}(k'_j+D-1)} \to \bar{x}$, and $\mathbb{S}x^{k'_j+D-1-\boldsymbol{b}(k'_j+D-1)} \to 0$ implies $\mathbb{S}\bar{x} = 0$ by continuity of $\mathbb{S}$.

Proof of Theorem 3: Stage 3

Given $D > L$, consider a subsequence $k_j \to \infty$ such that

$$(x^{k_j}, x^{k_j+1}, \ldots, x^{k_j+(D-1)}) \to (\bar{x}_D, \bar{x}_D, \ldots, \bar{x}_D) \in (\mathbb{R}^n)^D.$$

We write $\bar{x}_D$ to clarify that the limit may depend on D. Since

$$V^{k_j}(\bar{x}_D) \to V^\infty(\bar{x}_D)$$

by (ii), we have

$$V^\infty(\bar{x}_D) = \lim_{k_j \to \infty} \frac{1}{m} \sum_{d=D}^{\infty} c_d \|x^{k_j-d+D+1} - x^{k_j-d+D}\|^2 \leq \frac{2B^2}{m} \sum_{d=D}^{\infty} c_d.$$

Therefore

$$\limsup_{k \to \infty} \|x^k - \bar{x}_D\|^2 \leq \frac{2B^2}{m} \sum_{d=D}^{\infty} c_d. \tag{8}$$

By (2), we have

$$\sum_{d=1}^{\infty} c_d = m^{1/2} \sum_{d=1}^{\infty} \sum_{\ell=d}^{\infty} {Q_\ell}^{1/2} = m^{1/2} \sum_{\ell=1}^{\infty} \ell {Q_\ell}^{1/2} < \infty.$$

Therefore,

$$\sum_{d=D}^{\infty} c_d \to 0 \qquad \text{as } D \to \infty.$$

For any $D \in \mathbb{N}_+$,

$$\limsup_{k\to\infty} \|x^k - \bar{x}_D\|^2 \leq \lim_{k\to\infty} V^k(\bar{x}_D) \leq \frac{2B^2}{m} \sum_{d=D}^{\infty} c_d. \tag{8}$$

implies the accumulation points of x^k reside in the closed ball centered at $\bar{x}_D$ with a radius that goes to 0 as $D \to \infty$. The intersection of these balls contains a single accumulation point x^∞. (The intersection cannot be empty as the bounded sequence x^k must have at least one accumulation point.) □

Outline

Extended CF operators and exclusive memory access

Let $\mathbb{T}$ be extended coordinate friendly with auxiliary quantity $y(x)$. (E.g. $y(x) = Ax$.) Computing $x^{k+1} = \mathbb{T}x^k$ parallel synchronously:

```
// p agents run the while loop asynchronously
WHILE (not converged)
  1. WHILE (not all indices claimed)
       Claim index i not yet claimed
       Read x,y
       Compute s[i] = eta*S[i](x) using y
  2. Synchronize: wait for all agents to finish
  ---------------------------------------------
  3. WHILE (not all indices claimed)
     a. Select index i not yet claimed
     b. y -= A[:,i]*s[i] (Sequential, any order)
     c. x[i] = x[i] - s[i]
  4. Synchronize: wait for all agents to finish
  ---------------------------------------------
```

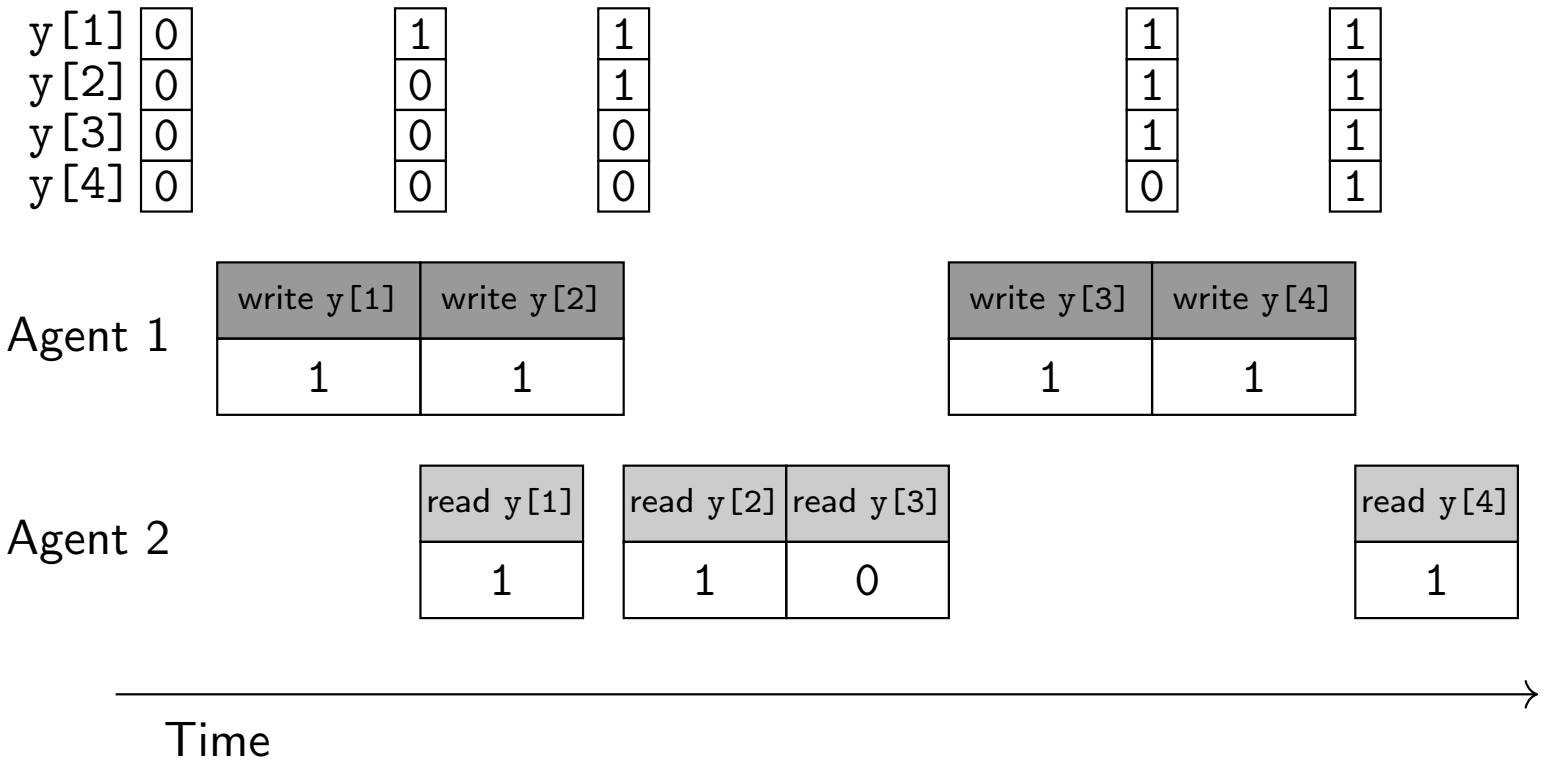
Race conditions

Consider removing the synchronization barrier:

```
// p agents run the while loop asynchronously
WHILE (not converged)
  Select i from Uniform{1,2,...,m}
  Read x,y
  Compute s[i] = eta*S[i](x) using y
  Read y and write y = y - A[:,i]*s[i] //Wrong!
  Exclusively read x[i] and
               write x[i] = x[i] - s[i]
```

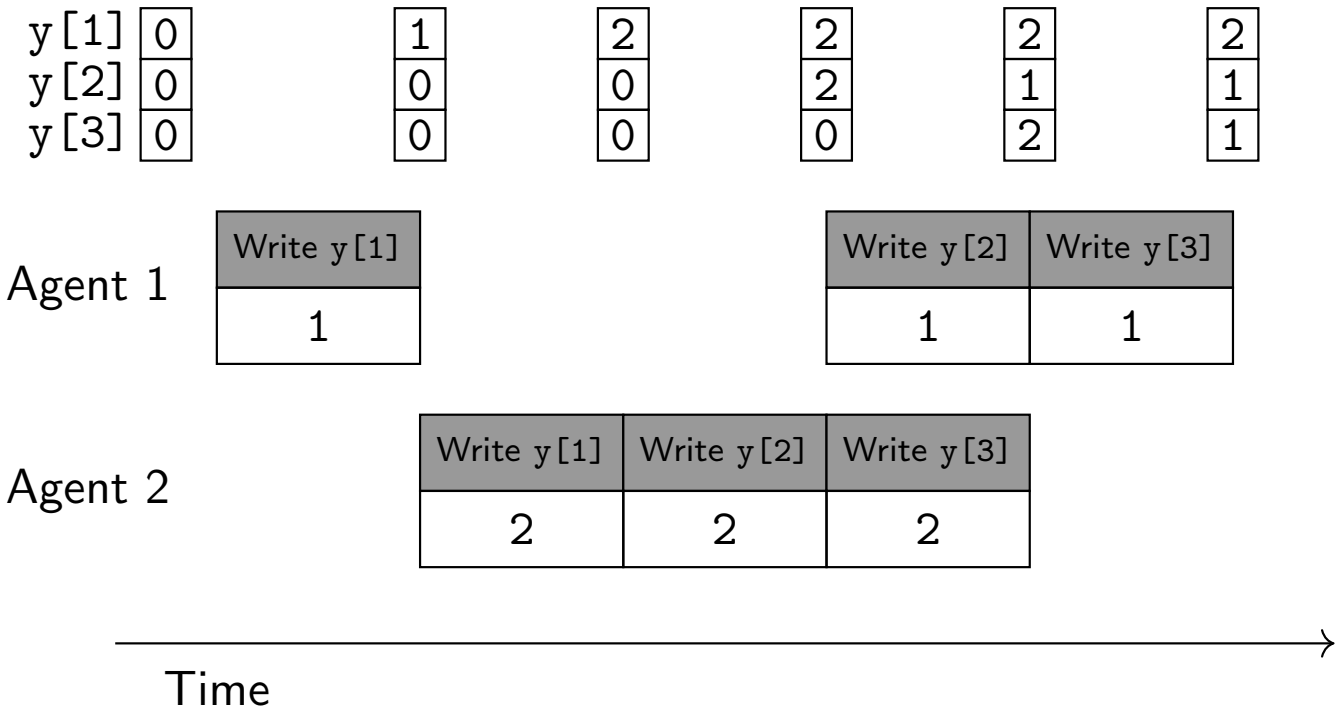
This is not an instance of ARock, due *race conditions*, negative behaviors of a parallel method whose result depends on the order in which the agents complete their tasks.

Race condition: inconsistent read



Example of an inconsistent read. Agent 2 reads `y=[1,1,0,1]`, which was never an actual state of `y` in memory. AC-FPI requires consistent reads within a single block but does allow inconsistent reads on `x` across different blocks. (The delay within a single block must be the same, but different blocks may have different delays.)

Race condition: inconsistent write



Example of an inconsistent write. The two writes of Agents 1 and 2 partially overwrite each other, and `y=[2,1,1]` is the resulting state. An inconsistent write can occur when multiple agents attempt to concurrently write to the same block.

Race condition: inconsistency between x and y

For this method to be an instance of ARock, the x and y that an agent reads must be related through the relationship y=A*x. This may fail to be hold if x and y are updated separately. We can prevent this by enforcing exclusive access for the whole (x,y) pair:

```
WHILE (not converged)
  1. Select i from Uniform{1,2,...,m}
  2. Read x,y
  3. Compute s[i] = eta*S[i](x) using y
  4. dy[i] = A[:,i]*s[i]
  5. Acquire exclusive access to (x,y)
  6. Read y and write y = y - dy[i]
     Read x[i] and write x[i] = x[i] - s[i]
  7. Release exclusive access to (x,y)
```

In some setups, exclusive access of Steps 5–7 can be a bottleneck.

Outline

Parameter server framework

The discussion so far was based on a *shared memory system*, where multiple agents freely access variables stored in shared memory. A single computer with a multi-core CPU is modeled well by a shared memory system.

In the *parameter server framework* or *server–worker framework*, a *server*, or *parameter server*, is a dedicated agent that collects, updates, and distributes variables over a network connected to *workers*, the computational agents working in parallel.

The parameter server can also perform minimal computation, so long as the server can keep up with the total throughput of the workers.

A cluster of multiple computers connected to a central server node over a network is modeled well by the parameter server framework.

Server–worker synchronous FPI

One server and m workers to compute $x^{k+1} = x^k - \eta \mathbb{S} x^k$ synchronously:

Server runs

```
WHILE (not converged)
  Broadcast x to workers
  WHILE (not all indices processed)
    Pick any arrived, unprocessed s_i
    x[i] = x[i] - s_i
```

Each worker runs (`i = agent number`)

```
WHILE (not converged)
  x << receive from server (wait until receive)
  s_i = eta*S[i](x)
  s_i >> server
```

This server–worker synchronous parallel algorithm has several potential sources of inefficiencies.

- Between a broadcast and the first arrival of `s_i`, the server is idle.
- Workers upload the `s_i`'s around the same time, due to synchrony, and can cause a computational and communication bottleneck.
- A single *straggler*, a worker taking significantly longer to process its work, can slow down the entire algorithm.

Server–worker asynchronous coordinate-update FPI

An asynchronous implementation in the server–worker framework can avoid the inefficiencies of synchronization:

Server runs

```
// Queue holds s_i's. Queue is first-in-first-out
WHILE (not converged)
  WHILE (before next broadcast schedule)
    s_i =  Queue.pop() (wait until nonempty)
    x[i] = x[i] - s_i
  Broadcast(x)
```

Each worker runs (`i = agent number`)

```
// Buffer holds only most recent x received
// from server
WHILE (not converged)
  x << Buffer.read()
  s_i = eta*S[i](x)
  s_i >> server's queue
```

Server:

- The `Queue` stores the `s_i` from workers.
- It broadcasts the latest `x` at a certain interval.
- Between the broadcasts, the server processes the updates in the `Queue` on a first-in-first-out basis.
- It must process the received `s_i` sufficiently fast; otherwise, it cannot keep up with the workers and the `Queue` will overflow.

Worker:

- Each worker has its `Buffer` that holds the most recent copy of `x` received from the server.
- Each worker uses its copy of `x` to compute `s_i`.
- Server broadcast should be sufficiently frequent so that the workers do not process the same `x` too often.

Inconsistent reads and writes do not arise in this setup, so there is no need for exclusive memory access.

Asynchronous server–worker AC-FPI

We can still model this asynchronous algorithm with AC-FPI (1):

- Let k increment whenever the server updates a block.
- Let x^k be the copy of `x` in the server memory after kth update.
- If the arrival times of `s_1`,...,`s_m` are m independent and identical Poisson processes, then $i(0), i(1), \ldots$ is an IID random sequence.

In this case, this algorithm is an instance of ARock and we can apply Theorem 3.

When $i(0), i(1), \ldots$ is not an IID random sequence, Theorem 3 does not apply.

Outline

Methods

We now present instances of AC-FPI on shared memory systems and on the parameter server framework.

The ARock assumptions are approximately, but not fully, satisfied by these algorithms.

Asynchronous coordinate gradient descent

Consider the problem

$$\underset{x\in\mathbb{R}^n}{\text{minimize}} \quad f\left(\sum_{i=1}^m A_{:,i}x_i - b\right) + \sum_{i=1}^m g_i(x_i),$$

where

$$A = \begin{bmatrix} A_{:,1} & A_{:,2} & \cdots & A_{:,m} \end{bmatrix} \in \mathbb{R}^{r\times n}.$$

Assume for simplicity that we have $p = m$ agents.

Assume the ith agent has access to x_i^k, $\mathrm{Prox}_{\alpha g_i}$, $A_{:,i}$, and ∇f, for $i = 1, \ldots, m$.

The RC-FPI with the FBS operator is

$$
\begin{aligned}
x_{i(k)}^{k+1} &= \mathrm{Prox}_{\alpha g_{i(k)}}\left(x_{i(k)}^{k} - \alpha A_{:,i(k)}^{\mathsf{T}} \nabla f(y^k)\right) \\
y^{k+1} &= y^k + A_{:,i(k)}(x_{i(k)}^{k+1} - x_{i(k)}^{k}),
\end{aligned}
$$

where we initialize $y^0 = Ax^0 - b$. The corresponding AC-FPI is

$$
\begin{aligned}
s_{i(k)}^{k} &= \eta\left(\hat{x}_{i(k)}^{k} - \mathrm{Prox}_{\alpha g_{i(k)}}\left(\hat{x}_{i(k)}^{k} - \alpha A_{:,i(k)}^{\mathsf{T}} \nabla f(\hat{y}^k)\right)\right) \\
x_{i(k)}^{k+1} &= x_{i(k)}^{k} - s_{i(k)}^{k} \\
y^{k+1} &= y^k - A_{:,i(k)} s_{i(k)}^{k},
\end{aligned}
$$

where

$$
\hat{x}_{i(k)}^{k} = x_{i(k)}^{k-d_{i(k)}(k)}, \qquad \hat{y}^k = A_{:,1} x_1^{k-d_1(k)} + \cdots + A_{:,m} x_m^{k-d_m(k)}.
$$

In a shared memory system, we can implement AC-FPI with

```
// Shared memory code
// Initialize x=0, y=-b
// Pr_i = prox_{alpha*g_i}, G_f = gradient of f
WHILE (not converged) {
  //i = agent number
  Read y
  s[i] = eta*(x[i]-Pr_i(x[i]-alpha*A[:,i]'*G_f(y)))
  del[i] = -A[:,i]*s[i]
  Acquire exclusive access to y
  y = y + del[i]
  Release exclusive access to y
  x[i] = x[i] - s[i]
}
```

The momentary inconsistency between `y` and `x[i]` (after `y` is updated but before `x[i]` is updated) makes no difference to other agents since each `x[i]` is read and updated by agent i only. Effectively, `y` and `x[i]` are effectively updated simultaneously. But, staleness may still exist.

In a parameter server framework, we can implement AC-FPI with the server running

```
// Server code
// Initialize y=-b
WHILE (not converged) {
  WHILE (before next broadcast schedule)
    IF Queue is empty, THEN wait until nonempty
    y = y + Queue.pop()
  Broadcast(y)
}
```

and the m workers running (next slide)

```
// Worker code
// Initialize x(1)=...=x(m)=0
// Pr_i = Prox_{alpha*g_i}, G_f = gradient of f
WHILE (not converged) {
  // i = agent number
  y << last received from server
  s_i = eta*(x_i-Pr_i(x_i-alpha*A[:,i]'*G_f(y)))
  (-A[:,i]*s_i) >> server's Queue
  x_i = x_i - s_i
}
```

The value of y received from the parameter server may be inconsistent with x if the parameter server has not yet processed the worker's previous upload. For a means to ensure consistency, see Exercise 6.3.

If each agent always updates the same block, the independence assumption does not hold even if all agents are equally powerful and the computational costs of all blocks are identical. See Exercise 6.1.

Asynchronous ADMM

Consider the optimization problem

$$\underset{x\in\mathbb{R}^n}{\text{minimize}} \quad \frac{1}{m}\sum_{i=1}^{m} f_i(x) + g(x).$$

We recast this problem into

$$\begin{aligned} &\underset{x_1,\ldots,x_m,y\in\mathbb{R}^n}{\text{minimize}} && \tfrac{1}{m}\textstyle\sum_{i=1}^m f_i(x_i) + g(y) \\ &\text{subject to} && \underbrace{\begin{bmatrix} I & 0 & \ldots & 0 \\ 0 & I & \ldots & 0 \\ & & \ddots & \\ 0 & 0 & \ldots & I \end{bmatrix}}_{=A} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_m \end{bmatrix} - \underbrace{\begin{bmatrix} I \\ I \\ \vdots \\ I \end{bmatrix}}_{=B} y = 0. \end{aligned} \tag{9}$$

Define $f(x_1,\ldots,x_m) = (1/m)(f_1(x_1)+\cdots+f_m(x_m))$. As we did in §3.2, consider the dual problem

$$\underset{\nu}{\text{minimize}} \quad \underbrace{f^*(-A^\mathsf{T}\nu)}_{\tilde{f}(\nu)} + \underbrace{g^*(-B^\mathsf{T}\nu)}_{\tilde{g}(\nu)}.$$

The PRS operator (2.14) applied to the dual problem is

$$w^{k+1} = (2\mathrm{Prox}_{\alpha\tilde{f}} - \mathbb{I})(2\mathrm{Prox}_{\alpha\tilde{g}} - \mathbb{I})w^k.$$

The FPI with the PRS operator averaged by $\eta \in (0,1)$ is

$$y^{k+1} = \operatorname*{argmin}_{y\in\mathbb{R}^n} \left\{ g(y) - y^\intercal \sum_{j=1}^{m} w_j^k + \frac{\alpha m}{2}\|y\|^2 \right\}$$

$$x_i^{k+1} = \operatorname*{argmin}_{x\in\mathbb{R}^n} \left\{ \frac{1}{m} f_i(x) + x^\intercal \left(w_i^k - 2\alpha y^{k+1}\right) + \frac{\alpha}{2}\|x\|^2 \right\} \quad \text{for } i = 1,\dots,m$$

$$w_i^{k+1} = w_i^k + 2\eta\alpha(x_i^{k+1} - y^{k+1}) \quad \text{for } i = 1,\dots,m.$$

With the change of variables $w^k = \alpha u^k$ and $\rho = 1/(\alpha m)$, we get

$$
\begin{aligned}
y^{k+1} &= \mathrm{Prox}_{\rho g}\left(\frac{1}{m}\sum_{j=1}^{m} u_j^k\right) \\
x_i^{k+1} &= \mathrm{Prox}_{\rho f_i}\left(2y^{k+1} - u_i^k\right) \quad \text{for } i = 1, \ldots, m \\
u_i^{k+1} &= u_i^k + 2\eta(x_i^{k+1} - y^{k+1}) \quad \text{for } i = 1, \ldots, m.
\end{aligned}
$$

The corresponding AC-FPI is

$$
\begin{aligned}
y^{k+1} &= \mathrm{Prox}_{\rho g}\left((1/m)\hat{u}_{\mathrm{sum}}^k\right) \\
x_{i(k)}^{k+1} &= \mathrm{Prox}_{\rho f_{i(k)}}\left(2y^{k+1} - \hat{u}_{i(k)}^k\right) \\
u_{i(k)}^{k+1} &= u_{i(k)}^k + 2\eta(x_{i(k)}^{k+1} - y^{k+1})
\end{aligned}
$$

where

$$
\hat{u}_{i(k)}^k = u_{i(k)}^{k-d_{i(k)}(k)}, \qquad \hat{u}_{\mathrm{sum}}^k = u_1^{k-d_1(k)} + \cdots + u_m^{k-d_m(k)}.
$$

Asynchronous ADMM server–worker implementation

Assume

- there are $p = m$ workers,
- each ith worker has access to u_i^k and $\mathrm{Prox}_{f_i/\alpha}$, and
- the server has access to $\mathrm{Prox}_{g/(\alpha m)}$.

We can implement ARock with the server running

```
// Parameter server code
// Initialize u_sum=0
WHILE (not converged) {
  WHILE (before next broadcast schedule)
    IF Queue is empty, THEN wait until nonempty
    s = Queue.pop()
    u_sum = u_sum + s
  y = Prox_{rho*g}(u_sum/m)
  Broadcast(y)
}
```

and each of the m workers running

```
// Worker code
// Initialize u[1]=...=u[m]=0
WHILE (not converged) {
  //i = agent number
  y << last received from server
  x_i = Prox_{rho*f_i}(2*y-u_i)
  2*eta*(x_i-y) >> server's Queue
  u_i = u_i + 2*eta*(x_i - y)
}
```

The discussions on consistency and the independence assumption for the previous example still apply to this example.

Outline

Exclusive memory access

We now discuss how to implement exclusive memory access using atomic operations and mutual exclusion locks. We limit the discussion at a superficial level, just enough to provide clarity on the behavior and the implementation of AC-FPI.

For a more thorough discussion on the lower-level considerations of concurrent programming, we refer to readers to standard resources.

Atomic operations

An operation of a computational agent is *atomic* if the whole operation is guaranteed to complete without interruption (or never start) in the presence of contention from other agents; if an atomic operation consists of multiple steps, other agents will not observe intermediate results.

In most modern systems, reading and writing a single number (represented as a 32- or 64-bit floating-point number) is an atomic operation.

Consider the case where all blocks are single coordinates, i.e., $m = n$ and $n_1 = \cdots = n_m = 1$. Then we can implement the AC-FPI as follows.

```
// p agents run the while loop asynchronously
WHILE (not converged) {
  1. Select i from Uniform{1,2,...,m}
  2. for j = 1,...,m
       read x[j]
  3. Compute s[i] = -eta*S[i](x)
  4. x[i] += s[i] (atomic with compare-and-swap)
}
```

In Step 2, each coordinate `x[j]` is read consistently though different coordinates may have different delays.

Step 4 uses the *increment* operator +=. a+=b implies a=a+b.

The increment operator reads from a and b, computes the sum, and writes to a. Despite several erroneous claims in the asynchronous optimization literature, += is often *not* atomic in many CPUs and GPUs; when multiple agents simultaneously increment the same variable, one agent can overwrite another's increment.

Atomic increment by compare-and-swap

The *atomic increment* can be implemented via compare-and-swap:

```
// atomic a += b
do {
  old <- a
} while ( !compare_and_swap(a, old, old+b) )
```

The compare-and-swap instruction

Most modern CPUs and GPUs support the *compare-and-swap* instruction as an atomic operation. It corresponds to:

```
// atomic execution
// input num is passed by reference and is
// modifiable
function compare_and_swap(num, old, new) {
  if num != old
    return false
  num <- new
  return true
}
```

If `num` is equal to `old`, then write `new` to `num` and return `true`; otherwise do nothing and return `false`.

AC-FPI when block has multiple coordinates

This algorithm is no longer valid:

```
// p agents run the while loop asynchronously
WHILE (not converged) {
  1. Select i from Uniform{1,2,...,m}
  2. for j = 1,...,m
       read block x[j]
  3. Compute s[i] = -eta*S[i](x)
  4. x[i] += s[i] (atomic with compare-and-swap)
}
```

The reads of Step 2 are no longer guaranteed to be consistent as it is possible for `x[j]` to be updated by another agents while it is being read.

Moreover, the atomic increment via compare-and-swap is no longer possible as compare-and-swap is usually only supported for data types of size 64-bits or smaller.

Mutual exclusion lock

A *mutual exclusion lock* or *mutex* comes with a *lock* and an *unlock* methods and is *acquired* by at most one agent at any given time.

An agent acquires a mutex with the `lock()` method. An agent *releases* a mutex with the `unlock()` method.

If the mutex is available, `lock()` returns immediately and acquires the mutex. Otherwise (if another agent has locked the mutex and has not yet unlocked it) `lock()` waits until the mutex becomes available and then acquires the mutex.

The `unlock()` method returns immediately. If other agents are waiting to acquire the mutex, then one of the waiting agents acquire the mutex upon the unlock.

Mutual exclusion lock

An inefficient way to implement exclusive access in AC-FPI is:

```
// AC-FPI with one mutex. Inefficient!
WHILE (not converged)
  1. Select i from Uniform{1,2,...,m}
  2. mutex.lock()
     Read x
     mutex.unlock()
  3. Compute s[i] = eta*S[i](x)
  4. mutex.lock()
     x[i] = x[i] - s[i]
     mutex.unlock()
```

Mutual exclusion lock

Rather it is more efficient to use separate mutexes for all blocks:

```
// AC-FPI with mutex for each block
WHILE (not converged)
  1. Select i from Uniform{1,2,...,m}
  2. for j = 1,...,m
       mutex[j].lock()
       read x[j]
       mutex[j].unlock()
  3. Compute s[i] = eta*S[i](x)
  4. mutex[i].lock()
     x[i] = x[i] - s[i]
     mutex[i].unlock()
```

This mechanism ensures the reads and writes of all blocks are consistent while allowing multiple agents to concurrently operate on separate blocks. However, Step 2 is still inefficient as it prevents multiple agents from concurrently reading the same block.

Readers-writers lock

A *readers-writers lock* allows concurrent access for reads while enforcing exclusive access for writes.

```
class rw_lock:
  int b; mutex MUTEX_b; mutex MUTEX_W
  rw_lock(): b = 0 //constructor
  function read_lock():
    MUTEX_b.lock()
    b++
    If b==1, MUTEX_W.lock()
    MUTEX_b.unlock()
  function read_unlock():
    MUTEX_b.lock()
    b--
    If b==0, MUTEX_W.unlock()
    MUTEX_b.unlock()
  function write_lock():
    MUTEX_W.lock()
  function write_unlock():
    MUTEX_W.unlock()
```

Variable `b` keeps track of the number of concurrent reads. Each call to `read_lock()` increases `b` by 1, and each call to `read_unlock()` decreases `b` by 1. `MUTEX_b` ensures only one reader can modify `b` at any time.

When an agent calls `read_lock()`, if it increases `b` from 0 to 1, it will also try to acquire `MUTEX_W`.

When `b` ≥ 1, `MUTEX_W` is locked and there are one or more readers. A writer must wait until `b` returns 0, that is, there are no more readers.

When `MUTEX_W` is locked by a writer, all readers must wait.

AC-FPI with readers–writers locks

```
// AC-FPI with readers-writers locks
WHILE (not converged)
  1. Select i from Uniform{1,2,...,m}
  2. for j = 1,...,m
        rw_lock[j].read_lock()
        read x[j]
        rw_lock[j].read_unlock()
  3. Compute s[i] = eta*S[i](x)
  4. rw_lock[i].write_lock()
     x[i] = x[i] - s[i]
     rw_lock[i].write_unlock()
```

This mechanism ensures the reads and writes of all blocks are consistent, allows multiple agents to concurrently operate on separate blocks, and allows multiple agents to concurrently read from the same block.

This implementation of the readers–writers locks prioritizes readers: if there are many readers, a writer must wait until there are no more readers. If there are many writers, a reader may acquire the lock while some other writers are waiting.

To reduce the staleness as much as possible, one could use a readers-writers lock prioritizing writers. However, such locks are more complex and allow for less concurrency.