PASSCoDe: **P**arallel **AS**ynchronous **S**tochastic dual **Co**-ordinate **De**scent

Cho-Jui Hsieh

Department of Computer Science
University of Texas at Austin

Joint work with H.-F. Yu and I. S. Dhillon

Outline

- L2-regularized Empirical Risk Minimization
- Dual Coordinate Descent (Hsieh et al., 2008)
- Parallel Dual Coordinate Descent (on multi-core machines)
- Theoretical Analysis
- Experimental Results

L2-regularized ERM

$$\boldsymbol{w}^* = \arg\min_{\boldsymbol{w} \in R^d} P(\boldsymbol{w}) := \frac{1}{2}\|\boldsymbol{w}\|^2 + \sum_{i=1}^{n} \ell_i(\boldsymbol{w}^T \boldsymbol{x}_i)$$

- SVM with hinge loss: $\ell_i(z_i) = C \max(1 - z_i, 0)$
- SVM with squared hinge loss: $\ell_i(z_i) = C \max(1 - z_i, 0)^2$
- Logistic regression: $\ell_i(z_i) = C \log(1 + e^{-z_i})$

Primal and Dual Formulations

- Primal Problem

$$\boldsymbol{w}^* = \arg\min_{\boldsymbol{w} \in R^d} P(\boldsymbol{w}) := \frac{1}{2}\|\boldsymbol{w}\|^2 + \sum_{i=1}^{n} \ell_i(\boldsymbol{w}^T \boldsymbol{x}_i)$$

- Dual Problem

$$\boldsymbol{\alpha}^* = \arg\min_{\boldsymbol{\alpha} \in R^n} D(\boldsymbol{\alpha}) := \frac{1}{2}\left\|\sum_{i=1}^{n} \alpha_i \boldsymbol{x}_i\right\|^2 + \sum_{i=1}^{n} \ell_i^*(-\alpha_i),$$

  - $\ell_i^*(\cdot)$: the conjugate of $\ell_i(\cdot)$
- Primal-Dual Relationship between $\boldsymbol{w}^*$ and $\boldsymbol{\alpha}^*$

$$\boldsymbol{w}^* = \boldsymbol{w}(\boldsymbol{\alpha}^*) := \sum_{i=1}^{n} \alpha_i^* \boldsymbol{x}_i$$

Coordinate Descent on the Dual Problem

Randomly select an $i \in \{1, \ldots, n\}$ and update $\alpha_i \leftarrow \alpha_i + \delta^*$, where

$$\delta^* = \arg\min_{\delta} \; D(\boldsymbol{\alpha} + \delta \boldsymbol{e}_i)$$

Coordinate Descent on the Dual Problem

Randomly select an $i \in \{1, \ldots, n\}$ and update $\alpha_i \leftarrow \alpha_i + \delta^*$, where

$$\begin{aligned}
\delta^* &= \arg\min_{\delta} \; D(\boldsymbol{\alpha} + \delta \boldsymbol{e}_i) \\
&= \arg\min_{\delta} \; \frac{1}{2}\left(\delta + \frac{\left(\sum_{i=1}^{n} \alpha_i \boldsymbol{x}_i\right)^T \boldsymbol{x}_i}{\|\boldsymbol{x}_i\|^2}\right)^2 + \frac{1}{\|\boldsymbol{x}_i\|^2}\ell_i^*\left(-\left(\alpha_i + \delta\right)\right) \\
&= T_i\left(\left(\sum_{i=1}^{n} \alpha_i \boldsymbol{x}_i\right)^T \boldsymbol{x}_i, \alpha_i\right)
\end{aligned}$$

- Simple univariate problem, but $O(nnz)$ construction time

Coordinate Descent on the Dual Problem

Randomly select an $i \in \{1, \ldots, n\}$ and update $\alpha_i \leftarrow \alpha_i + \delta^*$, where

$$\begin{aligned}\delta^* &= \arg\min_{\delta} \; D(\boldsymbol{\alpha} + \delta \boldsymbol{e}_i) \\ &= \arg\min_{\delta} \; \frac{1}{2}\left(\delta + \frac{\left(\sum_{i=1}^n \alpha_i \boldsymbol{x}_i\right)^T \boldsymbol{x}_i}{\|\boldsymbol{x}_i\|^2}\right)^2 + \frac{1}{\|\boldsymbol{x}_i\|^2}\ell_i^*\left(-\left(\alpha_i + \delta\right)\right) \\ &= T_i\left(\left(\sum_{i=1}^n \alpha_i \boldsymbol{x}_i\right)^T \boldsymbol{x}_i, \alpha_i\right)\end{aligned}$$

- ~~Simple univariate problem, but $O(nnz)$ construction time~~ $\Rightarrow O(n_i)$

DCD: [Hsieh et al 2008]

- Maintain primal variable $\boldsymbol{w} = \sum_{i=1}^n \alpha_i \boldsymbol{x}_i$ and $\delta^* = T_i\left(\boldsymbol{w}^T \boldsymbol{x}_i, \alpha_i\right)$
- $O(n_i)$ construction time: $n_i = nnz$ of $\boldsymbol{x}_i$
- $O(n_i)$ maintenance cost: $\boldsymbol{w} \leftarrow \boldsymbol{w} + \delta^* \boldsymbol{x}_i$

Dual Coordinate Descent

Stochastic Dual Coordinate Descent

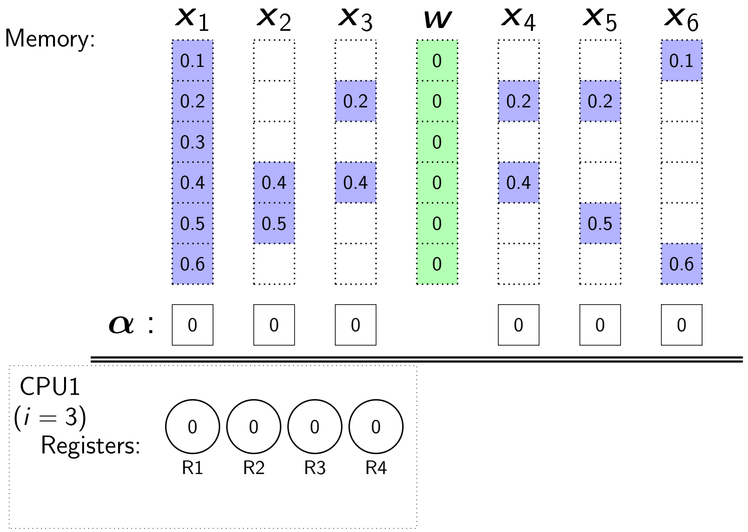
For $t = 1, 2, \ldots$

1. Randomly pick an index i
2. Compute $\boldsymbol{w}^T \boldsymbol{x}_i$
3. Update $\alpha_i \leftarrow \alpha_i + \delta^*$ where $\delta^* = T_i(\boldsymbol{w}^T \boldsymbol{x}_i, \alpha_i)$
4. Update $\boldsymbol{w} \leftarrow \boldsymbol{w} + \delta^* \boldsymbol{x}_i$.

- Implemented in LIBLINEAR:

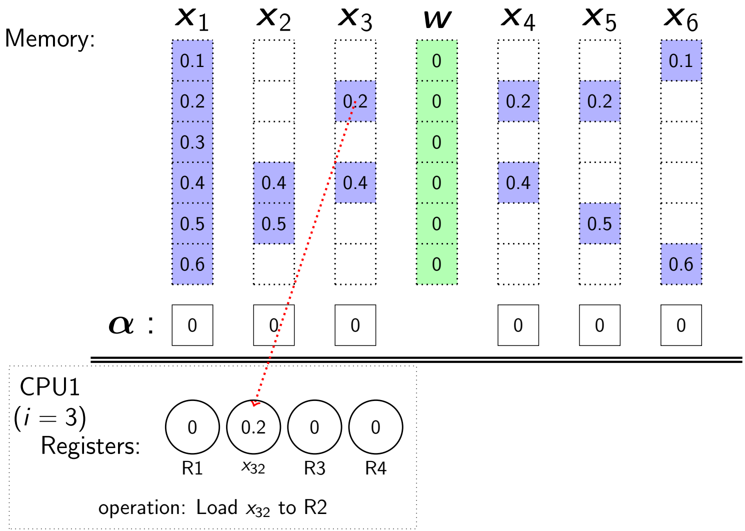
  Linear SVM (Hsieh et al., 2008), multi-class SVM (Keerthi et al., 2008), Logistic regression (Yu et al., 2011).
- Analysis: (Nesterov et al., 2012; Shalev-Shwartz et al., 2013)

Dual Coordinate Descent



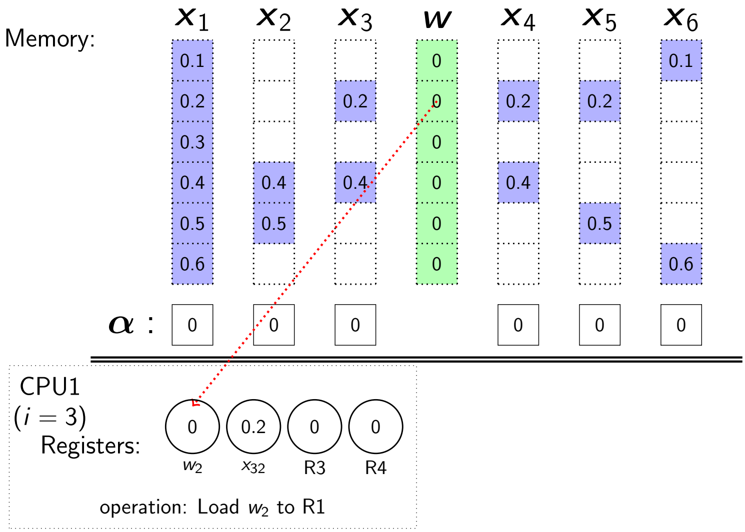
DCD step: compute $w^T x_i$

Dual Coordinate Descent



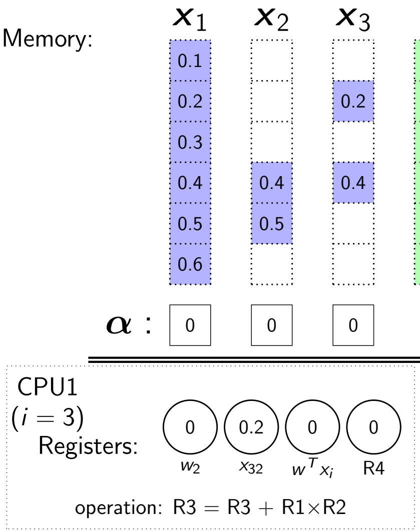
DCD step: compute $\boldsymbol{w}^T \boldsymbol{x}_i$

Dual Coordinate Descent



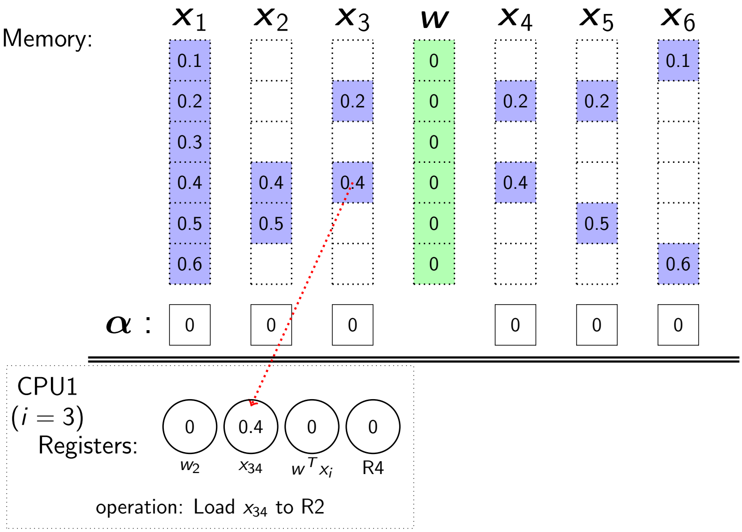
DCD step: compute $\boldsymbol{w}^T \boldsymbol{x}_i$

Dual Coordinate Descent



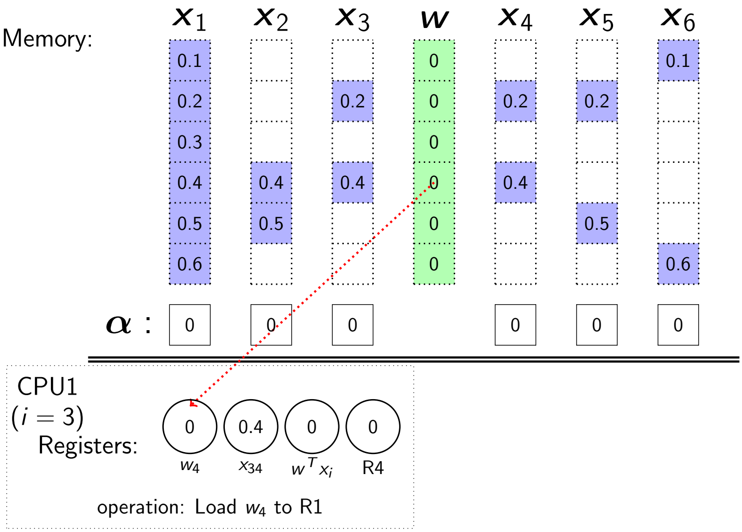
DCD step: compute $\boldsymbol{w}^T \boldsymbol{x}_i$

Dual Coordinate Descent



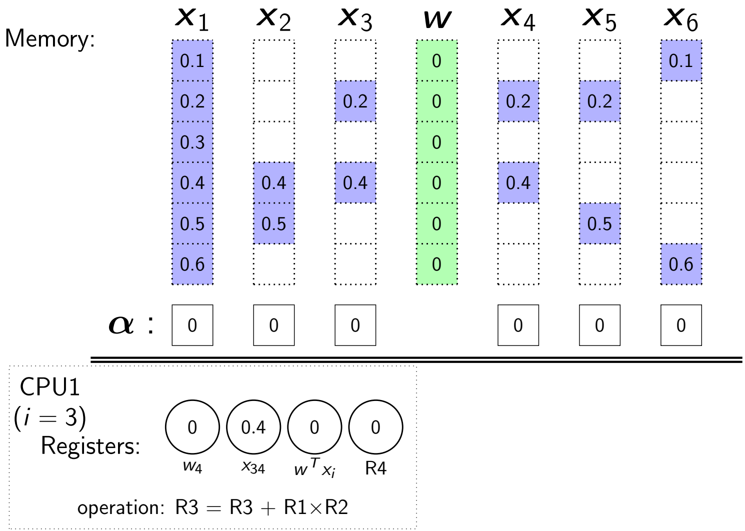
DCD step: compute $w^T x_i$

Dual Coordinate Descent



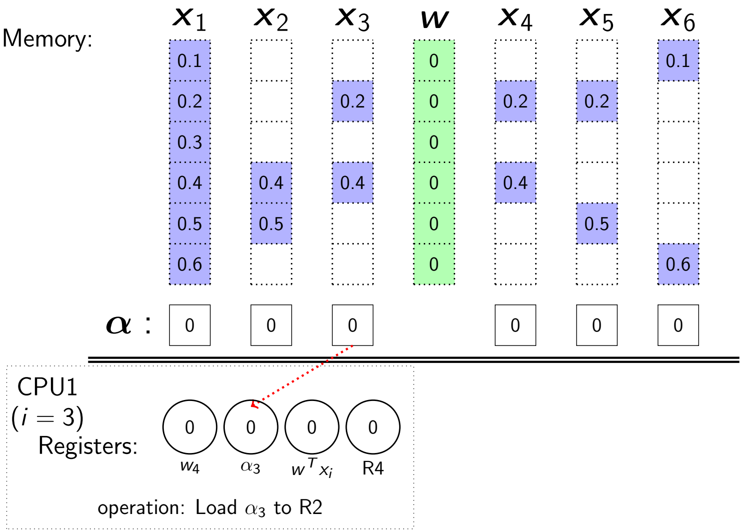
DCD step: compute $\boldsymbol{w}^T \boldsymbol{x}_i$

Dual Coordinate Descent



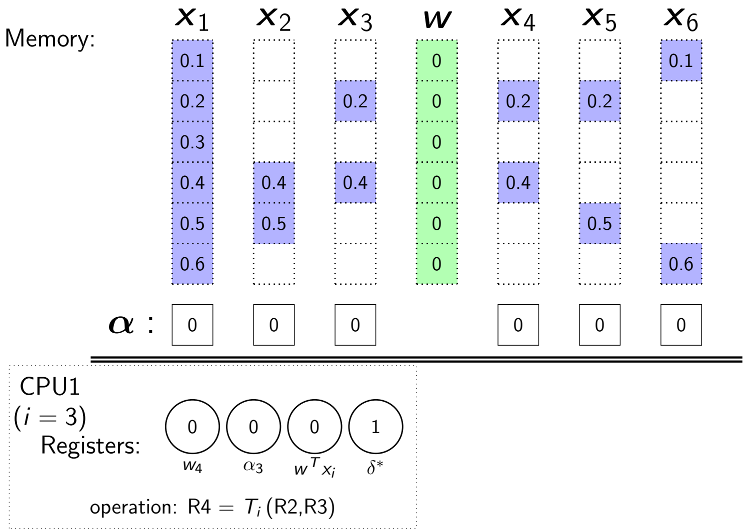
DCD step: compute $w^T x_i$

Dual Coordinate Descent



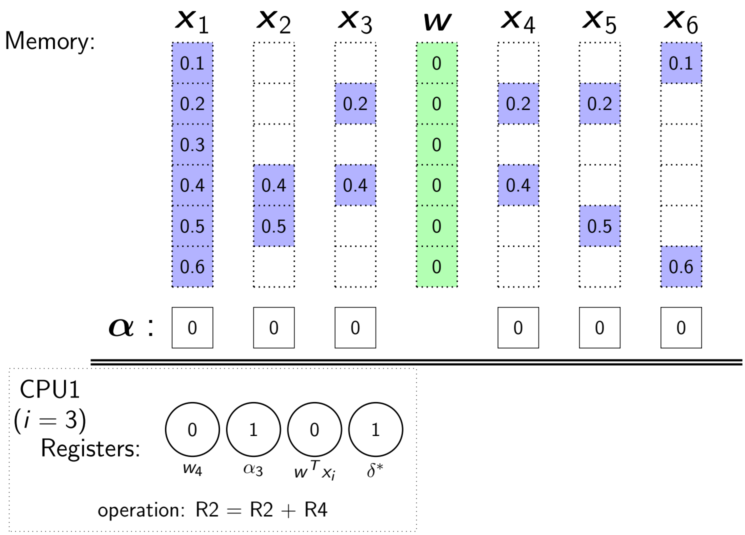
DCD step: compute $\delta^* = T_i\left(\mathbf{w}^T\mathbf{x}, \alpha_i\right)$

Dual Coordinate Descent



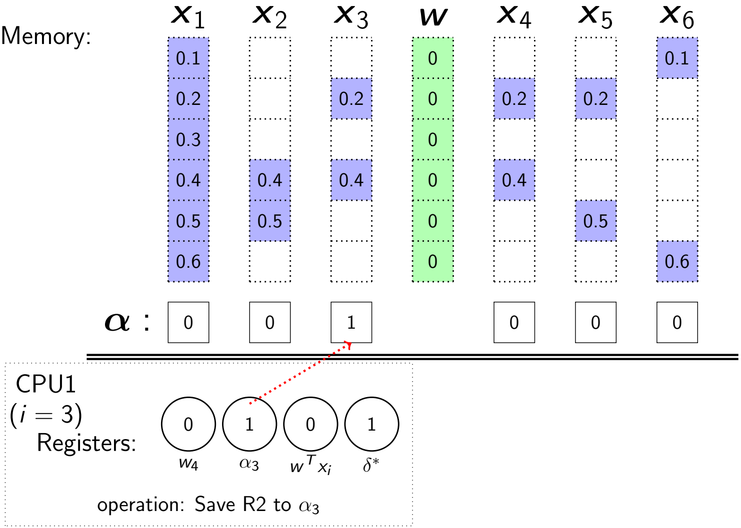
DCD step: compute $\delta^* = T_i\left(\mathbf{w}^T\mathbf{x}, \alpha_i\right)$

Dual Coordinate Descent



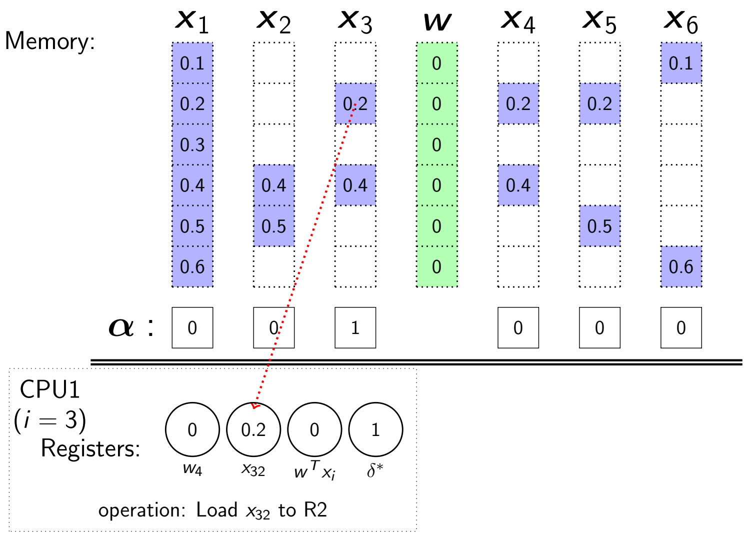
DCD step: update $\alpha_i = \alpha_i + \delta^*$

Dual Coordinate Descent



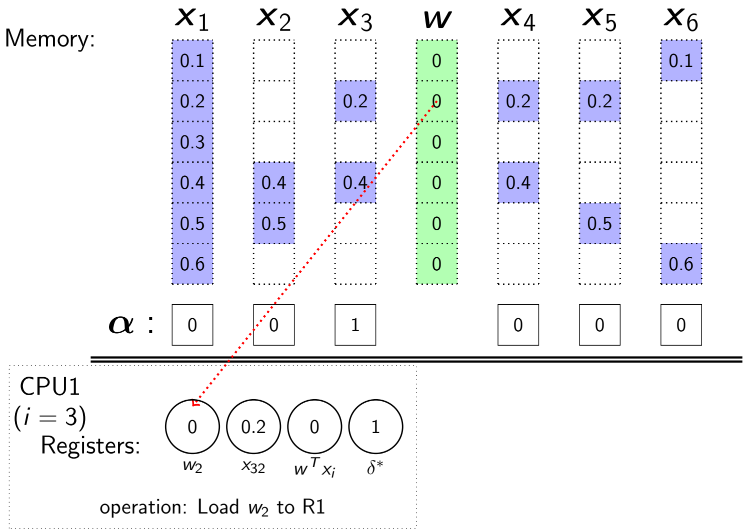
DCD step: update $\alpha_i = \alpha_i + \delta^*$

Dual Coordinate Descent



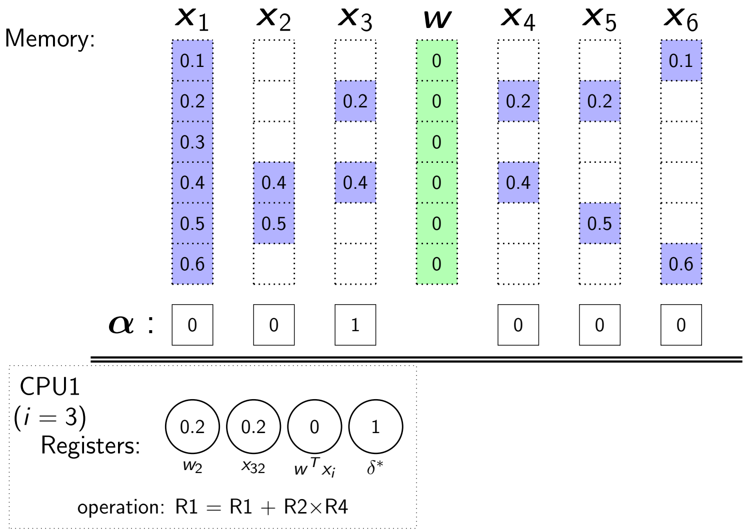
DCD step: update $\boldsymbol{w} = \boldsymbol{w} + \delta^* \boldsymbol{x}_i$

Dual Coordinate Descent



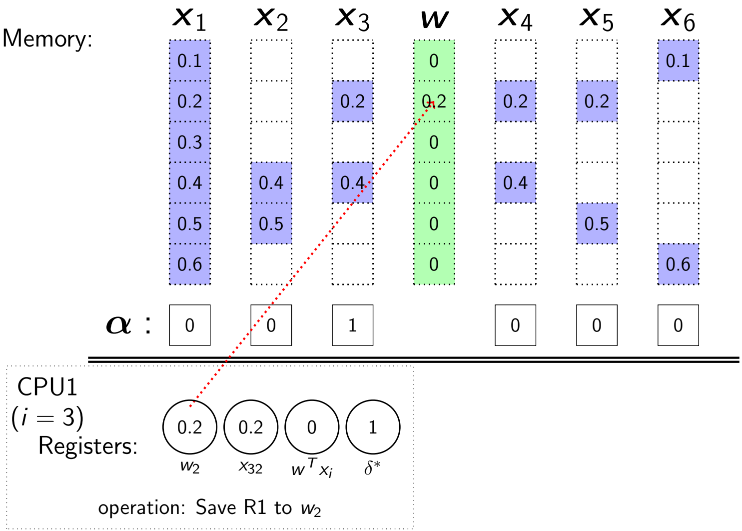
DCD step: update $\boldsymbol{w} = \boldsymbol{w} + \delta^* \boldsymbol{x}_i$

Dual Coordinate Descent



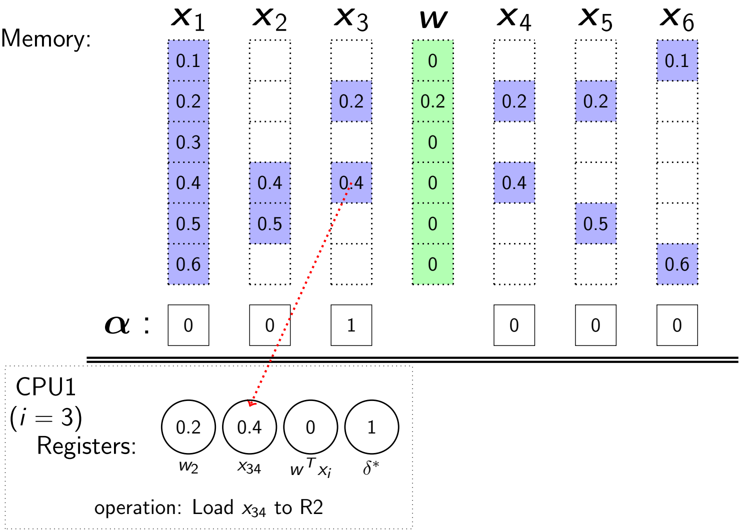
DCD step: update $w = w + \delta^* x_i$

Dual Coordinate Descent



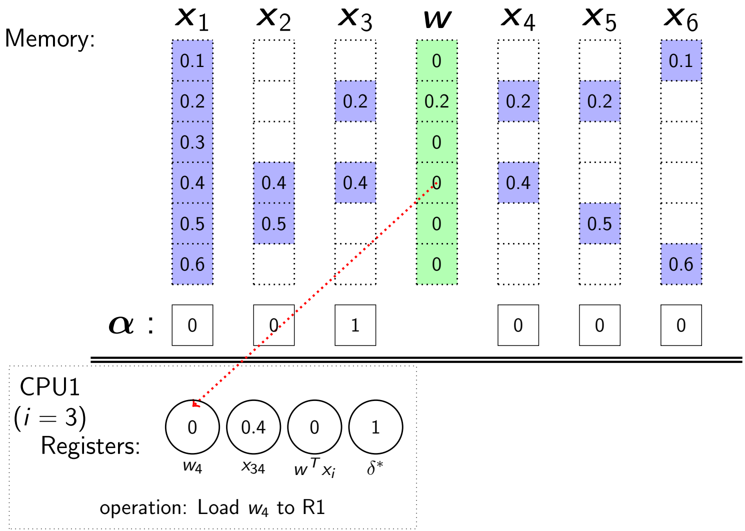
DCD step: update $\boldsymbol{w} = \boldsymbol{w} + \delta^* \boldsymbol{x}_i$

Dual Coordinate Descent



DCD step: update $\boldsymbol{w} = \boldsymbol{w} + \delta^* \boldsymbol{x}_i$

Dual Coordinate Descent



DCD step: update $\boldsymbol{w} = \boldsymbol{w} + \delta^* \boldsymbol{x}_i$

Dual Coordinate Descent

Memory:

	$\boldsymbol{x}_1$	$\boldsymbol{x}_2$	$\boldsymbol{x}_3$	$\boldsymbol{w}$	$\boldsymbol{x}_4$	$\boldsymbol{x}_5$	$\boldsymbol{x}_6$
	0.1			0			0.1
	0.2		0.2	0.2	0.2	0.2	
	0.3			0			
	0.4	0.4	0.4	0	0.4		
	0.5	0.5		0		0.5	
	0.6			0			0.6
$\boldsymbol{\alpha}$:	0	0	1		0	0	0

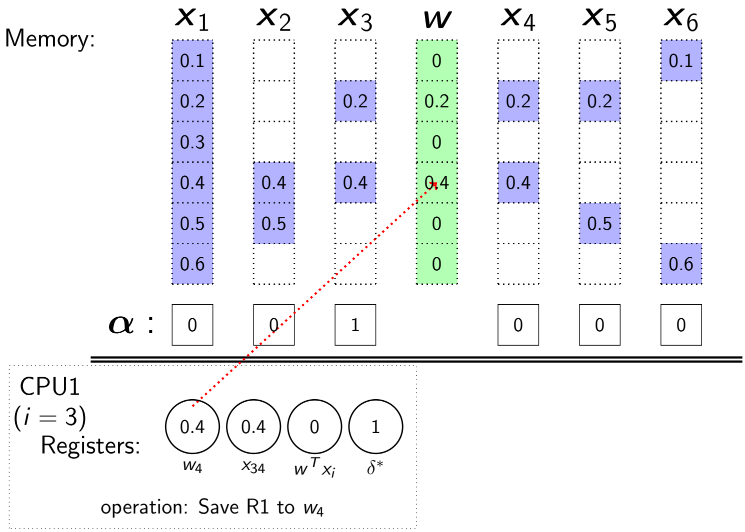
CPU1
($i = 3$)
Registers:

0.4	0.4	0	1
w_4	x_{34}	$w^T x_i$	δ^*

operation: R1 = R1 + R2×R4

DCD step: update $\boldsymbol{w} = \boldsymbol{w} + \delta^* \boldsymbol{x}_i$

Dual Coordinate Descent



DCD step: update $\boldsymbol{w} = \boldsymbol{w} + \delta^* \boldsymbol{x}_i$

Parallel DCD in Shared-memory Multi-core System

- Serial DCD updates:

For $t = 1, 2, \ldots$

1. Randomly pick an index i
2. Compute $\boldsymbol{w}^T \boldsymbol{x}_i$
3. Update $\alpha_i \leftarrow \alpha_i + \delta^*$ where $\delta^* = T_i(\boldsymbol{w}^T \boldsymbol{x}_i, \alpha_i)$
4. Update $\boldsymbol{w} \leftarrow \boldsymbol{w} + \delta^* \boldsymbol{x}_i$.

Parallel DCD in Shared-memory Multi-core System

- Parallel DCD updates:

Each thread repeatedly performs the following updates.
For $t = 1, 2, \ldots$

1. Randomly pick an index i
2. Compute $\boldsymbol{w}^T \boldsymbol{x}_i$
3. Update $\alpha_i \leftarrow \alpha_i + \delta^*$ where $\delta^* = T_i(\boldsymbol{w}^T \boldsymbol{x}_i, \alpha_i)$
4. Update $\boldsymbol{w} \leftarrow \boldsymbol{w} + \delta^* \boldsymbol{x}_i$.

Parallel DCD in Shared-memory Multi-core System

- Parallel DCD updates:

Each thread repeatedly performs the following updates.
For $t = 1, 2, \ldots$

1. Randomly pick an index i
2. Compute $\boldsymbol{w}^T \boldsymbol{x}_i$
3. Update $\alpha_i \leftarrow \alpha_i + \delta^*$ where $\delta^* = T_i(\boldsymbol{w}^T \boldsymbol{x}_i, \alpha_i)$
4. Update $\boldsymbol{w} \leftarrow \boldsymbol{w} + \delta^* \boldsymbol{x}_i$.

- Easy to implement using OpenMP.
- Variables $\boldsymbol{\alpha}$ and $\boldsymbol{w}$ stored in shared memory.

Parallel DCD in Shared-memory Multi-core System

- Parallel DCD updates:

Each thread repeatedly performs the following updates.
For $t = 1, 2, \ldots$

1. Randomly pick an index i
2. Compute $\boldsymbol{w}^T \boldsymbol{x}_i$
3. Update $\alpha_i \leftarrow \alpha_i + \delta^*$ where $\delta^* = T_i(\boldsymbol{w}^T \boldsymbol{x}_i, \alpha_i)$
4. Update $\boldsymbol{w} \leftarrow \boldsymbol{w} + \delta^* \boldsymbol{x}_i$.

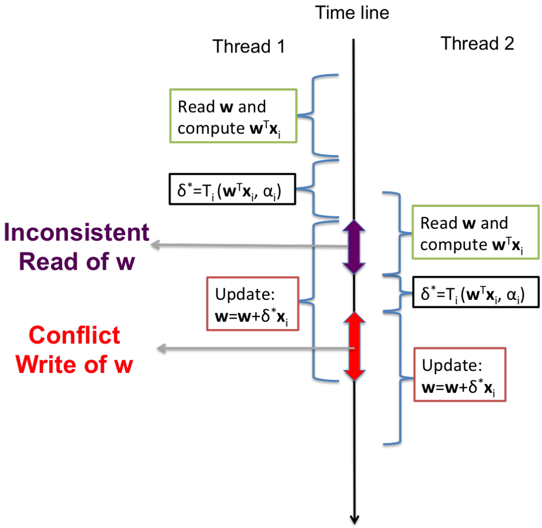
- Easy to implement using OpenMP.
- Variables $\boldsymbol{\alpha}$ and $\boldsymbol{w}$ stored in shared memory.
- Distributed Dual Coordinate Descent:

  Each machine has local copy of $\boldsymbol{\alpha}, \boldsymbol{w}$

  (Yang, 2013; Jaggi et al, 2014; Lee and Roth, 2015; Ma et al., 2015).

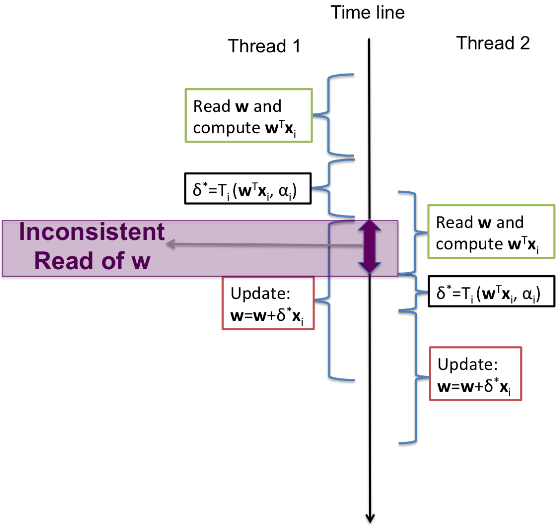
Parallel Dual Coordinate Descent: Two Issues for Correctness

- Inconsistent Read of $\boldsymbol{w}$
- Conflict Write of $\boldsymbol{w}$



Inconsistant Read

- Thread 2 reads $\boldsymbol{w}$ and thread 1 writes to $\boldsymbol{w}$ simultaneously.
- There may not exist any α such that $\boldsymbol{w} = \sum_i \alpha_i \boldsymbol{x}_i$.
- The "bounded delay" analysis in (Liu and Wright, 2014) cannot be directly applied.

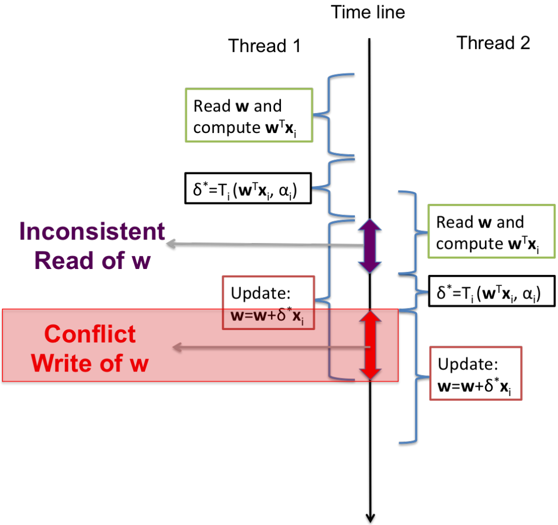


Conflict Write

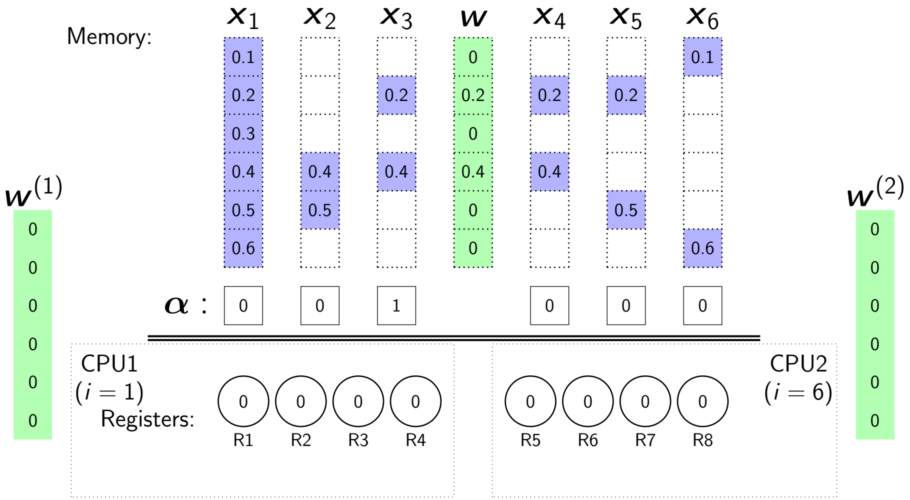
- Thread 1 and 2 write to $\boldsymbol{w}$ simultaneously.
- Updates to $\boldsymbol{w}$ can be overwritten, so the converged solution $\hat{\boldsymbol{w}}$ and $\hat{\alpha}$ may be inconsistent:

$$\hat{\boldsymbol{w}} \neq \sum_i \hat{\alpha}_i \boldsymbol{x}_i.$$

	CPU1: $w = w + 0.2$ OP	R1	w	CPU2: $w = w + 0.5$ OP	R2
0		0.0	1.0		0.0
1	load w	1.0	1.0	load w	1.0
2	add 0.2	1.2	1.0	add 0.5	1.5
3	save w	1.2	1.2		1.5
4		1.2	1.5	save w	1.5

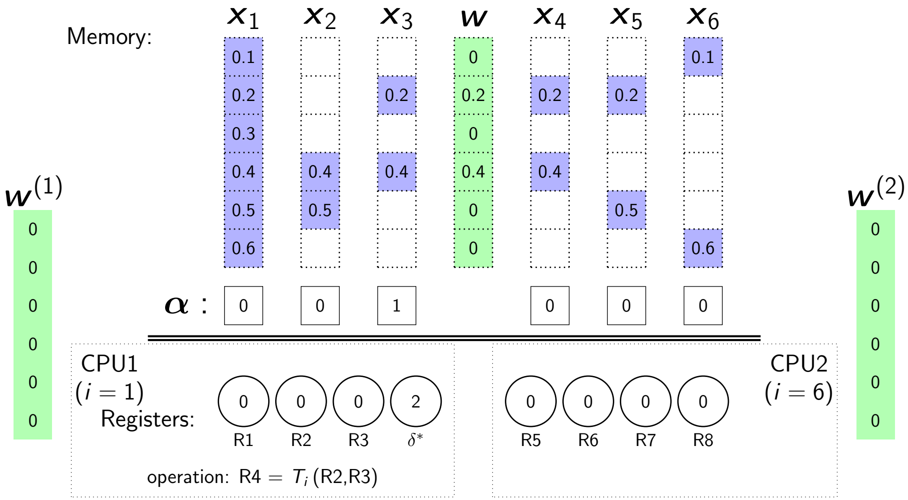


Dual Coordinate Descent in Parallel



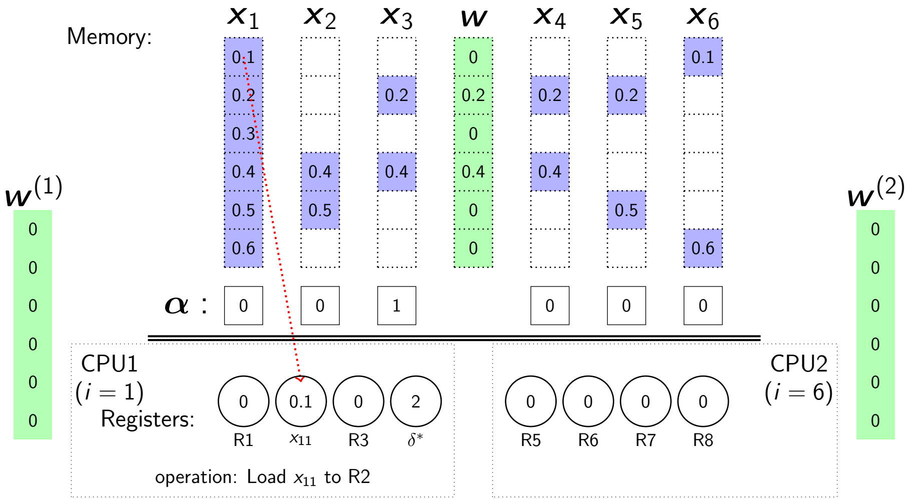
DCD step: compute $\delta^* = T_i\left(\boldsymbol{w}^T\boldsymbol{x}, \alpha_i\right)$

Dual Coordinate Descent in Parallel



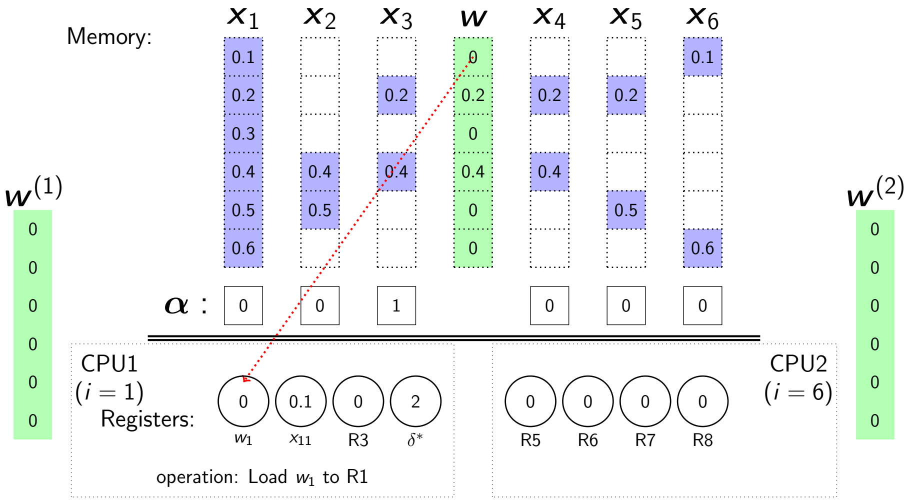
DCD step: compute $\delta^* = T_i\left(\boldsymbol{w}^T\boldsymbol{x}, \alpha_i\right)$

Dual Coordinate Descent in Parallel



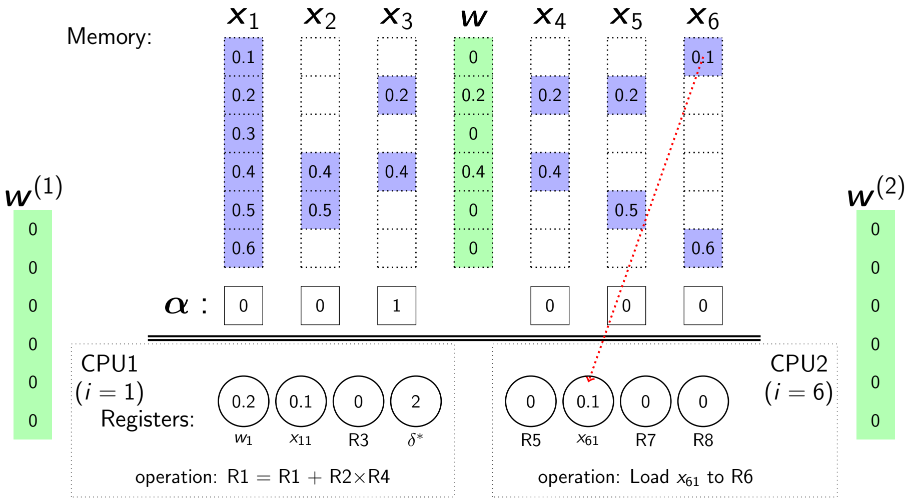
DCD step: update $w = w + \delta^* x_i$

Dual Coordinate Descent in Parallel



DCD step: update $w = w + \delta^* x_i$

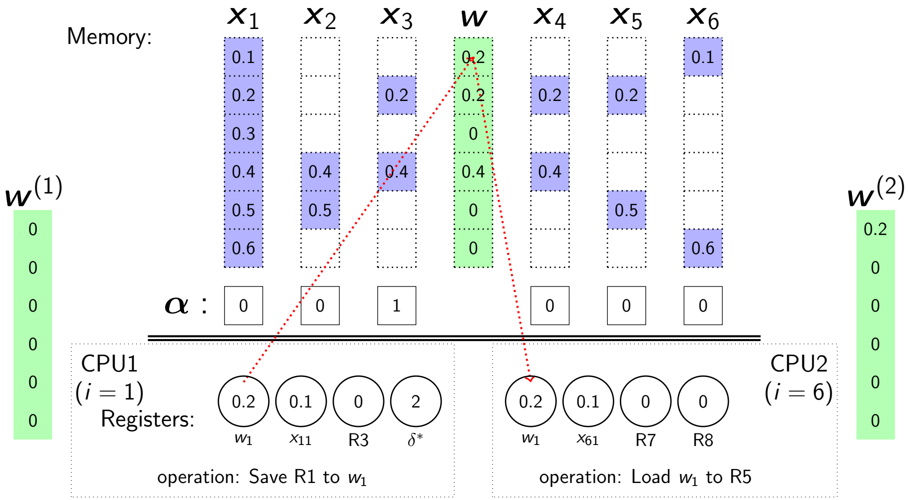
Dual Coordinate Descent in Parallel



DCD step: update $w = w + \delta^* x_i$

DCD step: compute $w^T x_i$

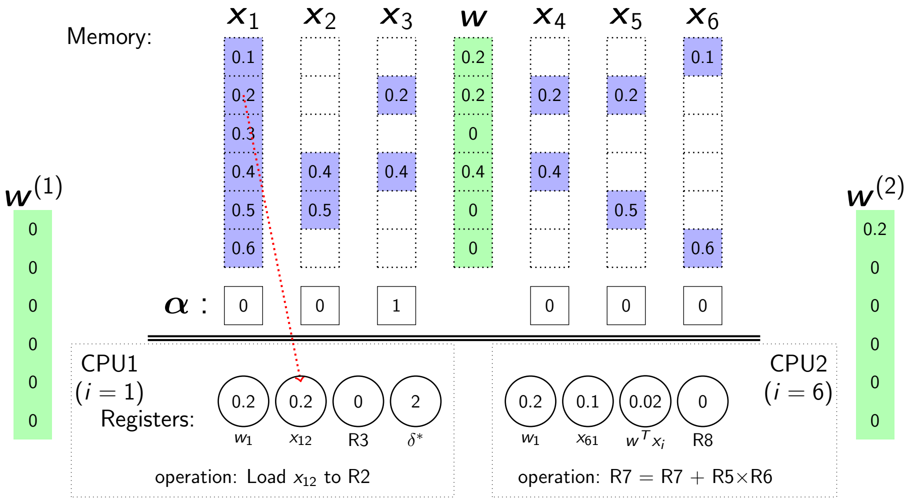
Dual Coordinate Descent in Parallel



DCD step: update $\boldsymbol{w} = \boldsymbol{w} + \delta^* \boldsymbol{x}_i$

DCD step: compute $\boldsymbol{w}^T \boldsymbol{x}_i$

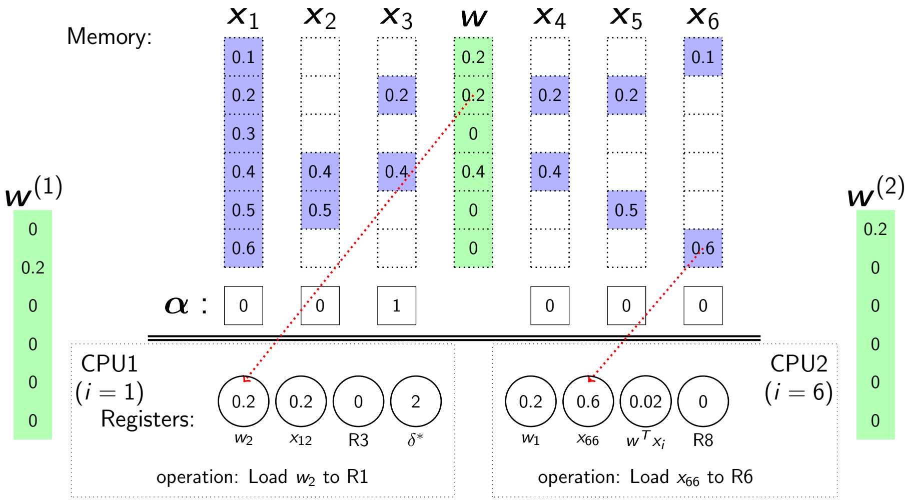
Dual Coordinate Descent in Parallel



DCD step: update $\boldsymbol{w} = \boldsymbol{w} + \delta^* \boldsymbol{x}_i$

DCD step: compute $\boldsymbol{w}^T \boldsymbol{x}_i$

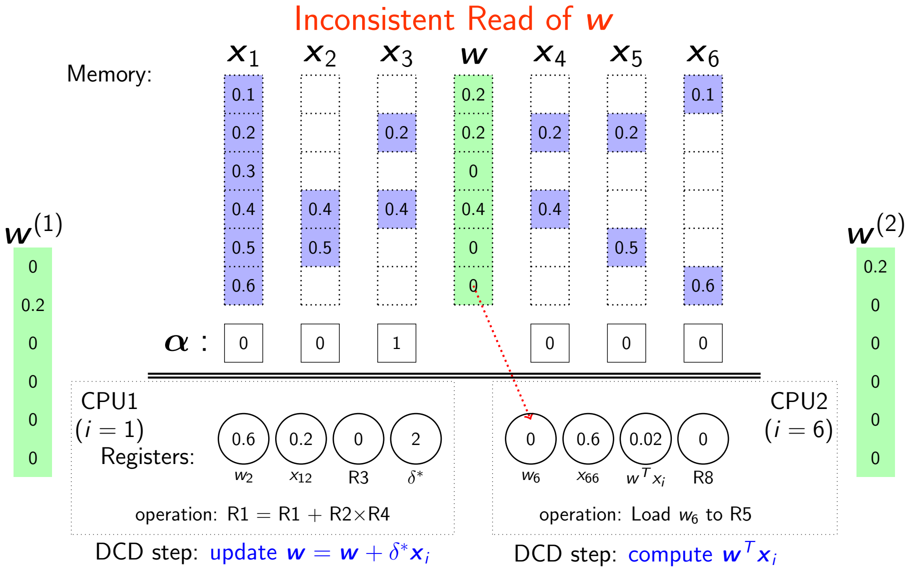
Dual Coordinate Descent in Parallel



DCD step: update $\boldsymbol{w} = \boldsymbol{w} + \delta^* \boldsymbol{x}_i$

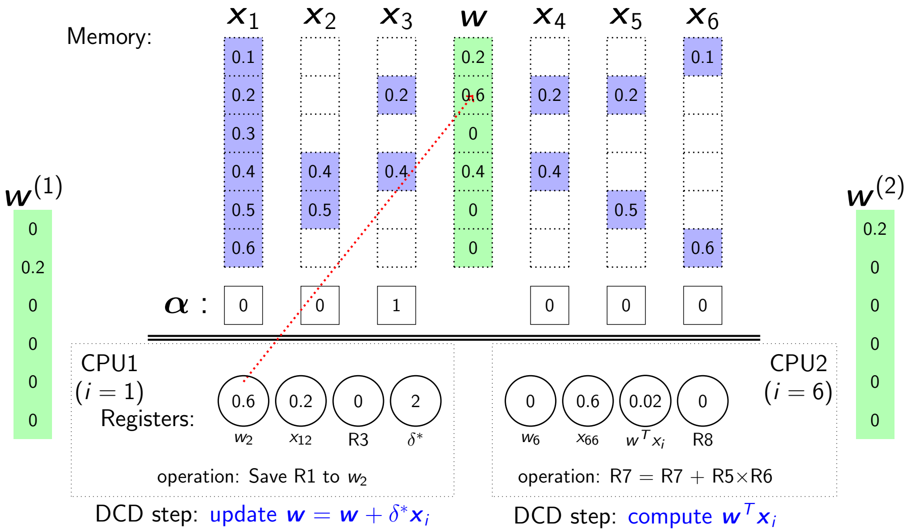
DCD step: compute $\boldsymbol{w}^T \boldsymbol{x}_i$

Dual Coordinate Descent in Parallel

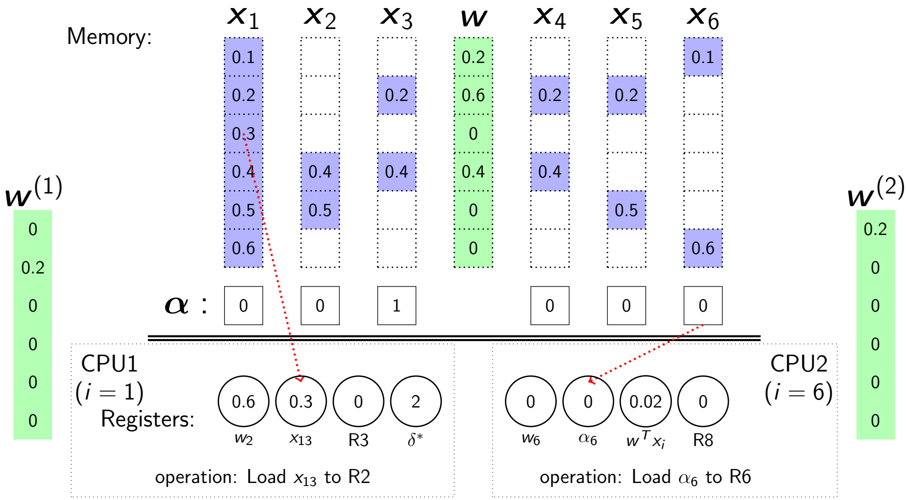


Dual Coordinate Descent in Parallel

Inconsistent Read of w

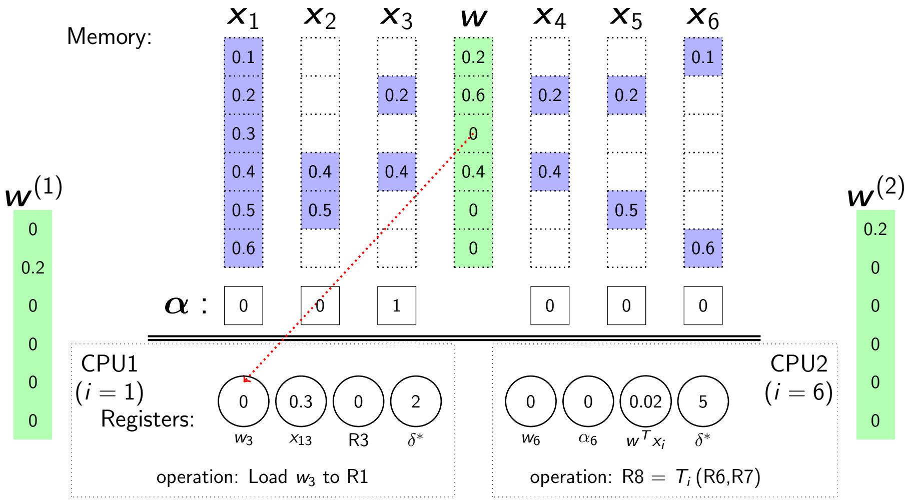


Dual Coordinate Descent in Parallel



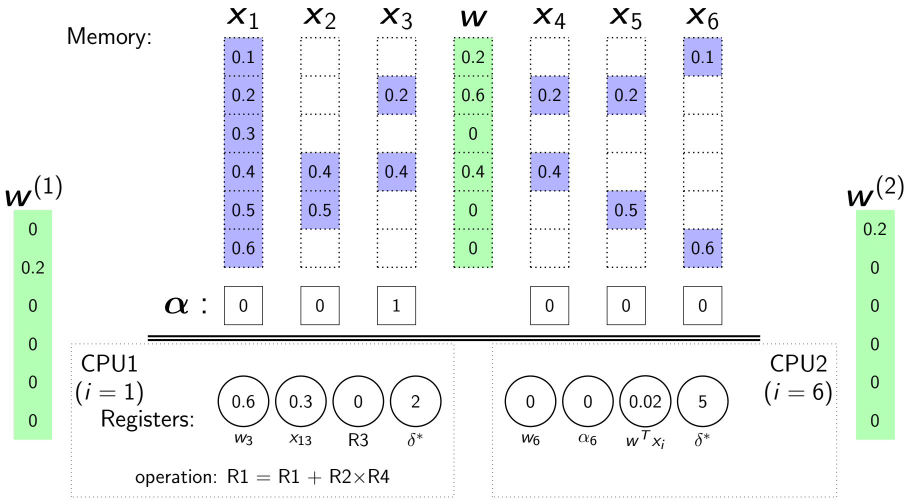
DCD step: update $w = w + \delta^* x_i$ DCD step: compute $\delta^* = T_i\left(w^T x, \alpha_i\right)$

Dual Coordinate Descent in Parallel



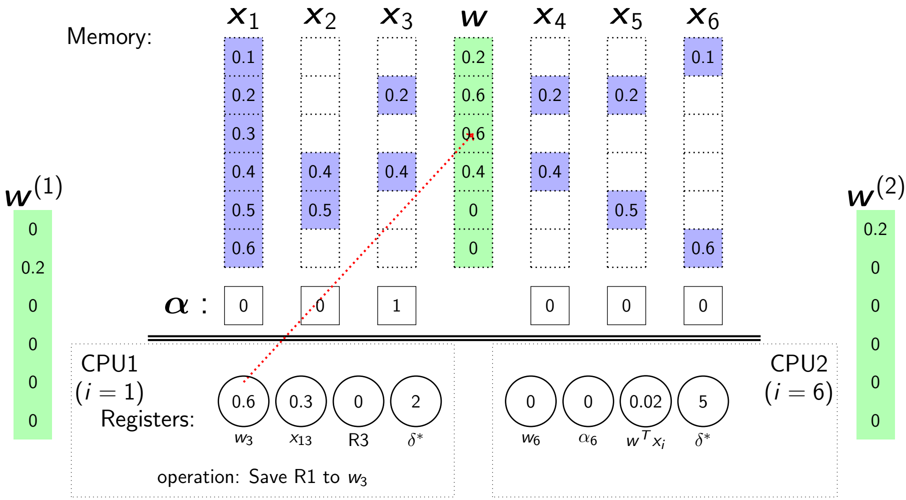
DCD step: update $\mathbf{w} = \mathbf{w} + \delta^* \mathbf{x}_i$ DCD step: compute $\delta^* = T_i\left(\mathbf{w}^T \mathbf{x}, \alpha_i\right)$

Dual Coordinate Descent in Parallel



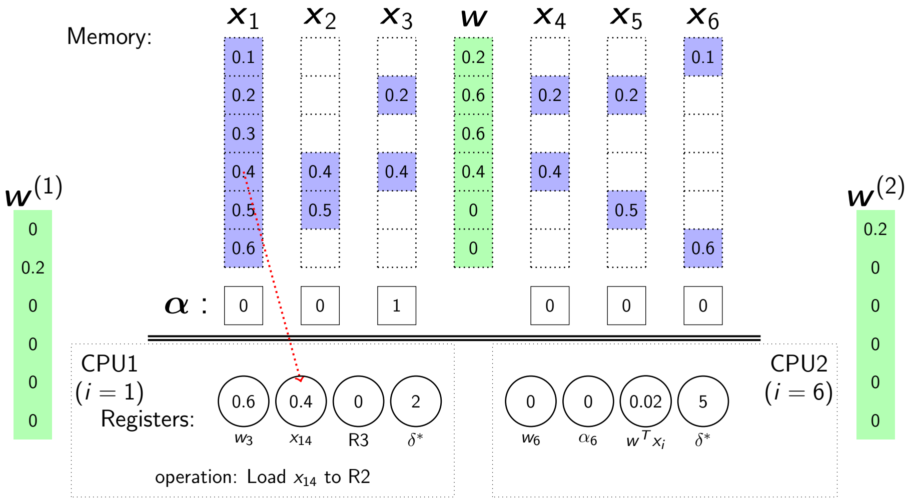
DCD step: update $\boldsymbol{w} = \boldsymbol{w} + \delta^* \boldsymbol{x}_i$ DCD step: compute $\delta^* = T_i\left(\boldsymbol{w}^T \boldsymbol{x}, \alpha_i\right)$

Dual Coordinate Descent in Parallel



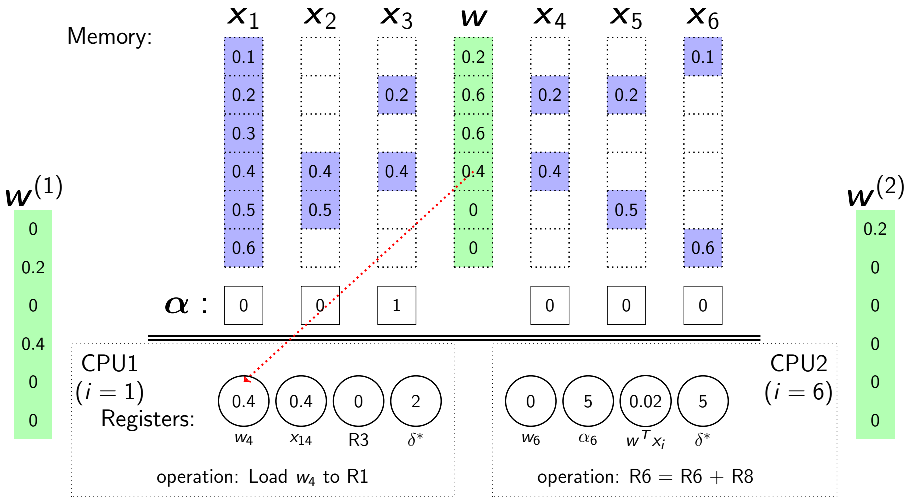
DCD step: update $\boldsymbol{w} = \boldsymbol{w} + \delta^* \boldsymbol{x}_i$ DCD step: compute $\delta^* = T_i \left(\boldsymbol{w}^T \boldsymbol{x}, \alpha_i \right)$

Dual Coordinate Descent in Parallel



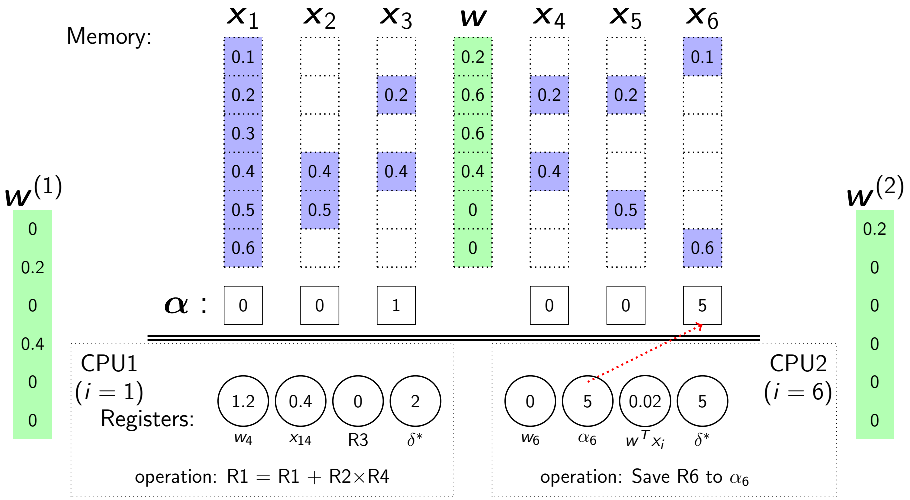
DCD step: update $\boldsymbol{w} = \boldsymbol{w} + \delta^* \boldsymbol{x}_i$ DCD step: compute $\delta^* = T_i\left(\boldsymbol{w}^T\boldsymbol{x}, \alpha_i\right)$

Dual Coordinate Descent in Parallel



DCD step: update $\boldsymbol{w} = \boldsymbol{w} + \delta^* \boldsymbol{x}_i$ DCD step: update $\alpha_i = \alpha_i + \delta^*$

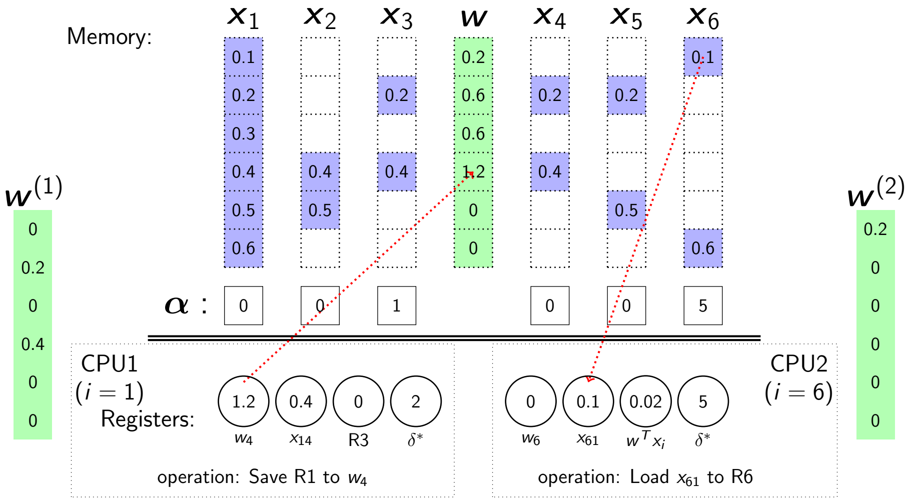
Dual Coordinate Descent in Parallel



DCD step: update $\boldsymbol{w} = \boldsymbol{w} + \delta^* \boldsymbol{x}_i$

DCD step: update $\alpha_i = \alpha_i + \delta^*$

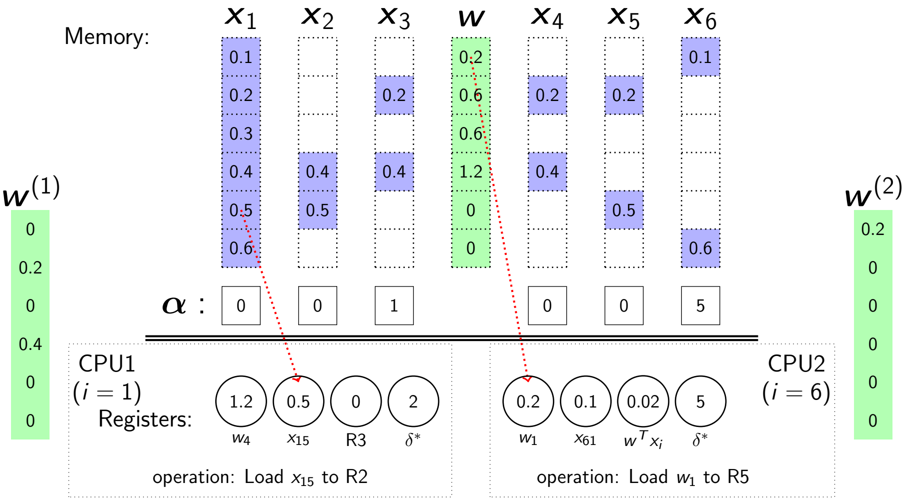
Dual Coordinate Descent in Parallel



DCD step: update $\boldsymbol{w} = \boldsymbol{w} + \delta^* \boldsymbol{x}_i$

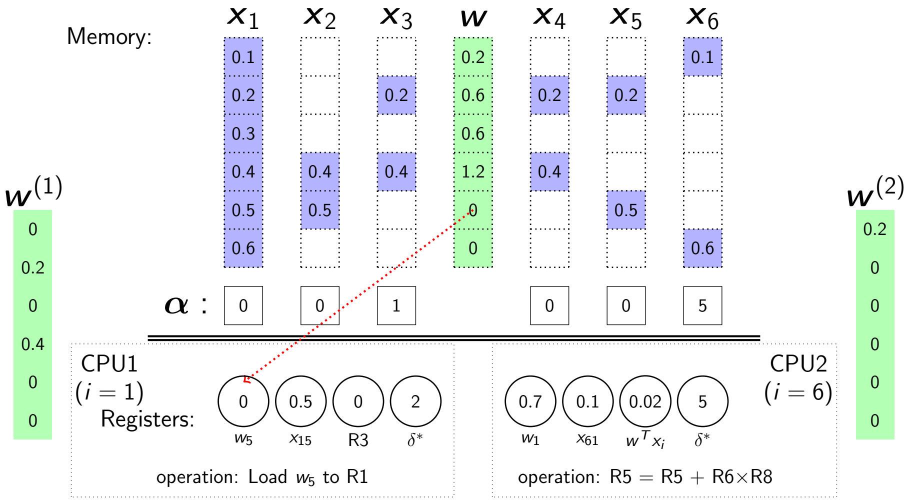
DCD step: update $\boldsymbol{w} = \boldsymbol{w} + \delta^* \boldsymbol{x}_i$

Dual Coordinate Descent in Parallel



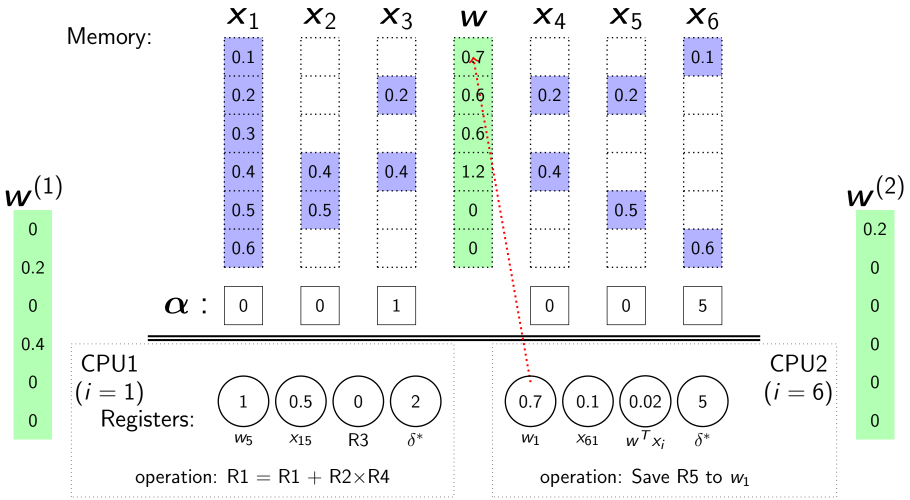
DCD step: update $w = w + \delta^* x_i$ DCD step: update $w = w + \delta^* x_i$

Dual Coordinate Descent in Parallel



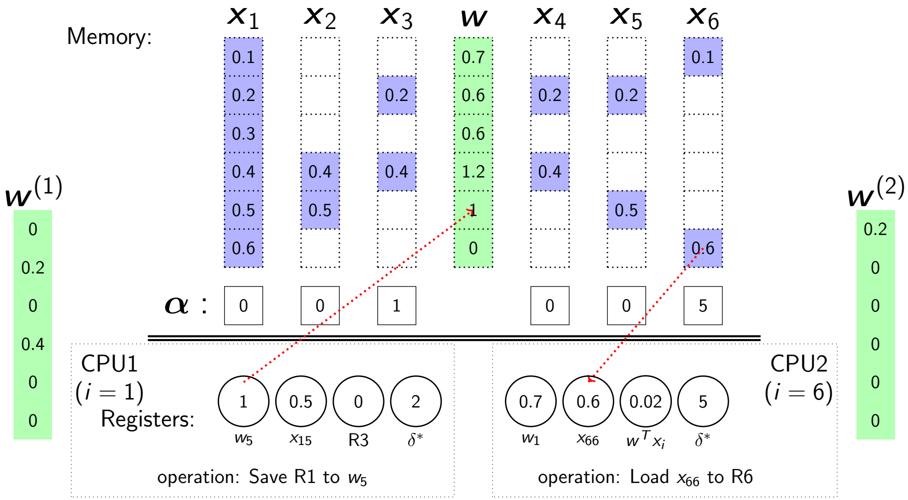
DCD step: update $\boldsymbol{w} = \boldsymbol{w} + \delta^* \boldsymbol{x}_i$ DCD step: update $\boldsymbol{w} = \boldsymbol{w} + \delta^* \boldsymbol{x}_i$

Dual Coordinate Descent in Parallel



DCD step: update $\boldsymbol{w} = \boldsymbol{w} + \delta^* \boldsymbol{x}_i$ DCD step: update $\boldsymbol{w} = \boldsymbol{w} + \delta^* \boldsymbol{x}_i$

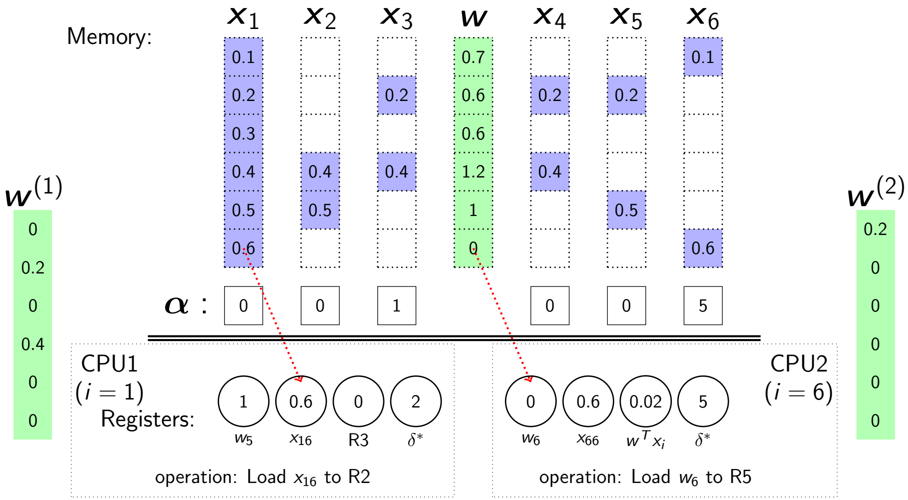
Dual Coordinate Descent in Parallel



DCD step: update $w = w + \delta^* x_i$ DCD step: update $w = w + \delta^* x_i$

Dual Coordinate Descent in Parallel

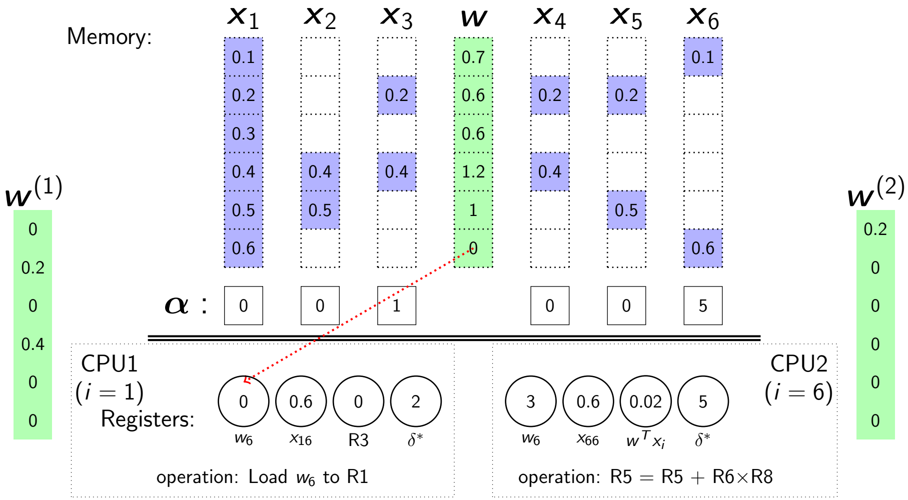
Conflict Write of w



DCD step: update $w = w + \delta^* x_i$ DCD step: update $w = w + \delta^* x_i$

Dual Coordinate Descent in Parallel

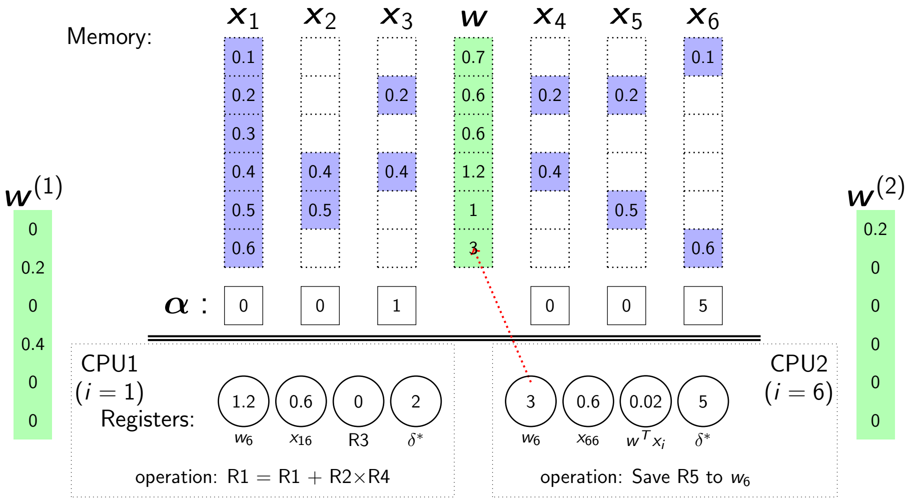
Conflict Write of $\boldsymbol{w}$



DCD step: update $\boldsymbol{w} = \boldsymbol{w} + \delta^* \boldsymbol{x}_i$ DCD step: update $\boldsymbol{w} = \boldsymbol{w} + \delta^* \boldsymbol{x}_i$

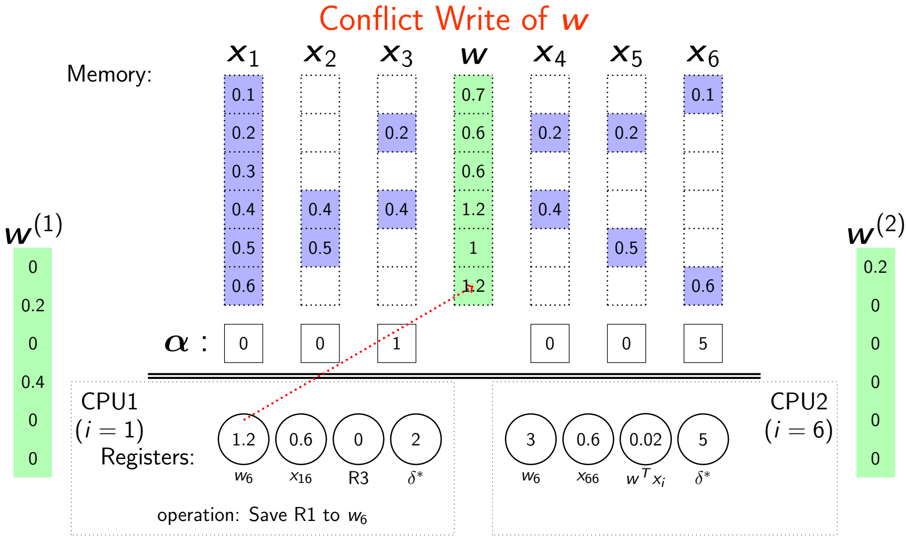
Dual Coordinate Descent in Parallel

Conflict Write of $\boldsymbol{w}$



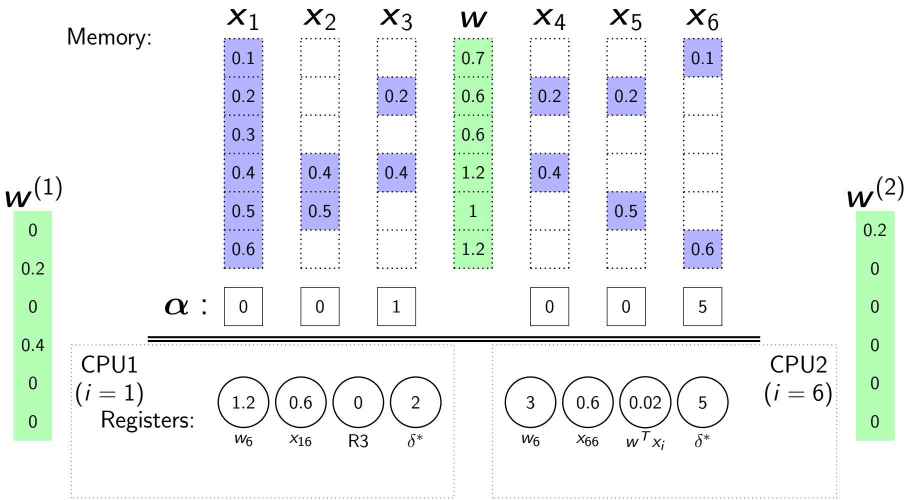
DCD step: update $\boldsymbol{w} = \boldsymbol{w} + \delta^* \boldsymbol{x}_i$ DCD step: update $\boldsymbol{w} = \boldsymbol{w} + \delta^* \boldsymbol{x}_i$

Dual Coordinate Descent in Parallel



DCD step: update $\boldsymbol{w} = \boldsymbol{w} + \delta^* \boldsymbol{x}_i$

Dual Coordinate Descent in Parallel



PASSCoDe-Lock

Each thread repeatedly performs the following updates.

For $t = 1, 2, \ldots$

1. Randomly pick an index i
2. **Lock** $\{w_j \mid (\boldsymbol{x}_i)_j \neq 0\}$
3. Compute $\boldsymbol{w}^T \boldsymbol{x}_i$
4. Update $\alpha_i \leftarrow \alpha_i + \delta^*$ where $\delta^* = T_i(\boldsymbol{w}^T \boldsymbol{x}_i, \alpha_i)$
5. Update $\boldsymbol{w} \leftarrow \boldsymbol{w} + \delta^* \boldsymbol{x}_i$.
6. **Unlock the variables.**

Three *PASSCoDe* approaches:

- **lock**: acquire locks for all necessary w_j before the update

	inconsistent read	conflict write
PASSCoDe-Lock	resolved	resolved

Scaling (on rcv1 with 100 epochs):

# threads	**Lock**
2	98.03s / 0.27x
4	106.11s / 0.25x
10	114.43s / 0.23x

PASSCoDe-Atomic

Each thread repeatedly performs the following updates.

For $t = 1, 2, \ldots$

1. Randomly pick an index i
2. Compute $\boldsymbol{w}^T \boldsymbol{x}_i$
3. Update $\alpha_i \leftarrow \alpha_i + \delta^*$ where $\delta^* = T_i(\boldsymbol{w}^T \boldsymbol{x}_i, \alpha_i)$
4. For each $j \in N(i)$
5. Update $w_j \leftarrow w_j + \delta^*(\boldsymbol{x}_i)_j$ **atomically**

How to Resolve the Issues

Three *PASSCoDe* approaches:

- **lock**: acquire locks for all necessary w_j before the update
- **atomic**: apply atomic operation for $w_j = w_j + \delta^* x_{ij}$

	inconsistent read	conflict write
PASSCoDe-Lock	resolved	resolved
PASSCoDe-Atomic	remained	resolved

Scaling (on rcv1 with 100 epochs):

# threads	**Lock**	**Atomic**
2	98.03s / 0.27x	15.28s / 1.75x
4	106.11s / 0.25x	8.35s / 3.20x
10	114.43s / 0.23x	3.86s / 6.91x

Analysis for **PASSCoDe-Atomic**

- Atomic operations guarantee:
  - all updates to $\boldsymbol{w}$ will be performed eventually
  - $\hat{\boldsymbol{w}} = \sum_{i=1}^{n} \hat{\alpha}_i \boldsymbol{x}_i$ holds for the outputted $(\hat{\boldsymbol{w}}, \hat{\boldsymbol{\alpha}})$
- Bounded delay assumption: to handle inconsistent read of $\boldsymbol{w}$
  - all updates of $\boldsymbol{w}$ before τ iterations must be performed

Theorem

Under certain conditions on τ, **PASSCoDe – Atomic** has global linear convergence rate in expectation:

$$E\left[D(\boldsymbol{\alpha}^{j+1}) - D(\boldsymbol{\alpha}^*)\right] \leq \eta E\left[D(\boldsymbol{\alpha}^{j}) - D(\boldsymbol{\alpha}^*)\right]$$

Our analysis covers logistic regression and SVM with hinge loss (where the dual problem is not strictly convex).

PASSCoDe-Wild

Each thread repeatedly performs the following updates.

For $t = 1, 2, \ldots$

1. Randomly pick an index i
2. Compute $\boldsymbol{w}^T \boldsymbol{x}_i$
3. Update $\alpha_i \leftarrow \alpha_i + \delta^*$ where $\delta^* = T_i(\boldsymbol{w}^T \boldsymbol{x}_i, \alpha_i)$
4. Update $\boldsymbol{w} \leftarrow \boldsymbol{w} + \delta^* \boldsymbol{x}_i$

How to Resolve the Issues

Three *PASSCoDe* approaches:

- **lock**: acquire locks for all necessary w_j before the update
- **atomic**: apply atomic operation for $w_j = w_j + \delta^* x_{ij}$
- **wild**: do nothing to resolve either issue

	inconsistent read	conflict write
PASSCoDe-Lock	resolved	resolved
PASSCoDe-Atomic	remained	resolved
PASSCoDe-Wild	remained	remained

Scaling (on rcv1 with 100 epochs):

# threads	**Lock**	**Atomic**	**Wild**
2	98.03s / 0.27x	15.28s / 1.75x	**14.08s / 1.90x**
4	106.11s / 0.25x	8.35s / 3.20x	**7.61s / 3.50x**
10	114.43s / 0.23x	3.86s / 6.91x	**3.59s / 7.43x**

Analysis for **PASSCoDe-Wild**

- Some updates are missing due to memory conflicts
  - for the final $(\hat{\boldsymbol{w}}, \hat{\boldsymbol{\alpha}})$:

$$\hat{\boldsymbol{w}} \neq \sum_{i=1}^{n} \hat{\alpha}_i \boldsymbol{x}_i$$

  - construct $\bar{\boldsymbol{w}}$ from the final $\hat{\boldsymbol{\alpha}}$:

$$\bar{\boldsymbol{w}} = \sum_{i=1}^{n} \hat{\alpha}_i \boldsymbol{x}_i$$

  - Which one for prediction, $\hat{\boldsymbol{w}}$ or $\bar{\boldsymbol{w}}$?

	# threads	Prediction Accuracy (%) by $\hat{\boldsymbol{w}}$	$\bar{\boldsymbol{w}}$	LIBLINEAR
news20	4	97.1	96.1	97.1
	8	97.2	93.3	
covtype	4	67.8	38.0	66.3
	8	67.6	38.0	
rcv1	4	97.7	97.5	97.7
	8	97.7	97.4	
webspam	4	99.1	93.1	99.1
	8	99.1	88.4	
kddb	4	88.8	79.7	88.8
	8	88.8	87.7	

Question: why $\hat{\boldsymbol{w}}$ is better than $\bar{\boldsymbol{w}}$?

Backward Analysis for **PASSCoDe-Wild**

Recall the primal problem

$$\boldsymbol{w}^* = \arg\min_{\boldsymbol{w}} \; P(\boldsymbol{w}) := \frac{1}{2}\|\boldsymbol{w}\|^2 + \sum_{i=1}^{n} \ell_i\left(\boldsymbol{w}^T \boldsymbol{x}_i\right)$$

Theorem

Let ϵ be the error caused by the memory conflicts.

$$\hat{\boldsymbol{w}} = \arg\min_{\boldsymbol{w}} \; \hat{P}(\boldsymbol{w}) := \frac{1}{2}\|\boldsymbol{w} + \epsilon\|^2 + \sum_{i=1}^{n} \ell_i(\boldsymbol{w}^T \boldsymbol{x}_i)$$

$$\bar{\boldsymbol{w}} = \arg\min_{\boldsymbol{w}} \; \bar{P}(\boldsymbol{w}) := \frac{1}{2}\|\boldsymbol{w}\|^2 + \sum_{i=1}^{n} \ell_i\left((\boldsymbol{w} - \epsilon)^T \boldsymbol{x}_i\right)$$

- $\hat{P}(\boldsymbol{w})$ is the problem with the perturbation on the regularization term
- $\bar{P}(\boldsymbol{w})$ is the problem with the perturbation on the prediction term

Datasets and Experimental Settings

Datasets.

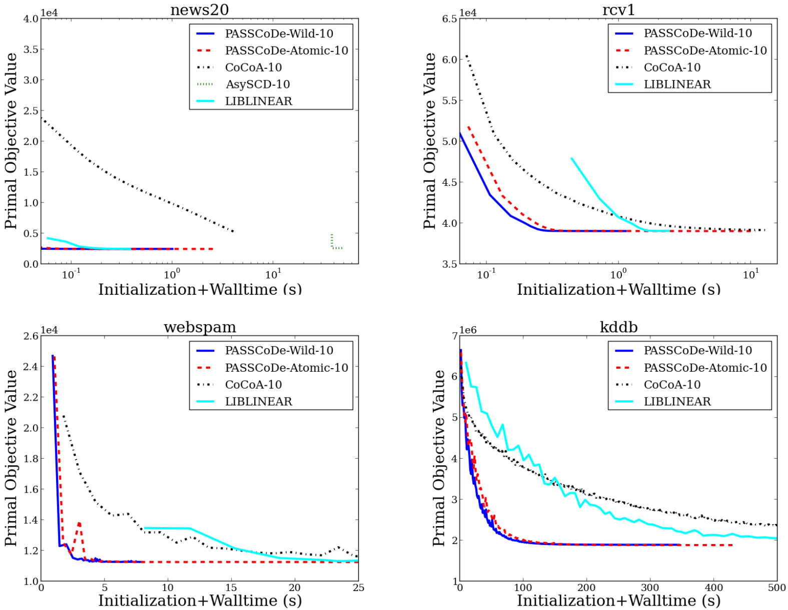
	n	$\tilde{n}$	d	$\bar{d}$	C
news20	16,000	3,996	1,355,191	455.5	2
rcv1	677,399	20,242	47,236	73.2	1
webspam	280,000	70,000	16,609,143	3727.7	1
kddb	19,264,097	748,401	29,890,095	29.4	1

Compared Implementation.

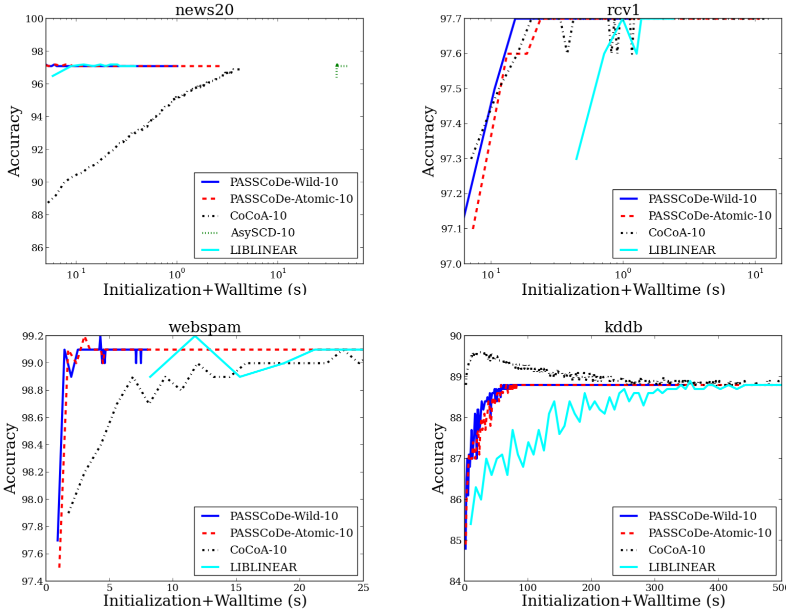
- LIBLINEAR: serial baseline
- *PASSCoDe-Wild* and *PASSCoDe-Atomic*: our methods
- *CoCoA*: a multi-core version of [Jaggi et al, 2014]
- *AsySCD*: [Liu & Wright, 2014]

Machine: Intel Multi-core machine with 256 GB Memory

Convergence in terms of Walltime

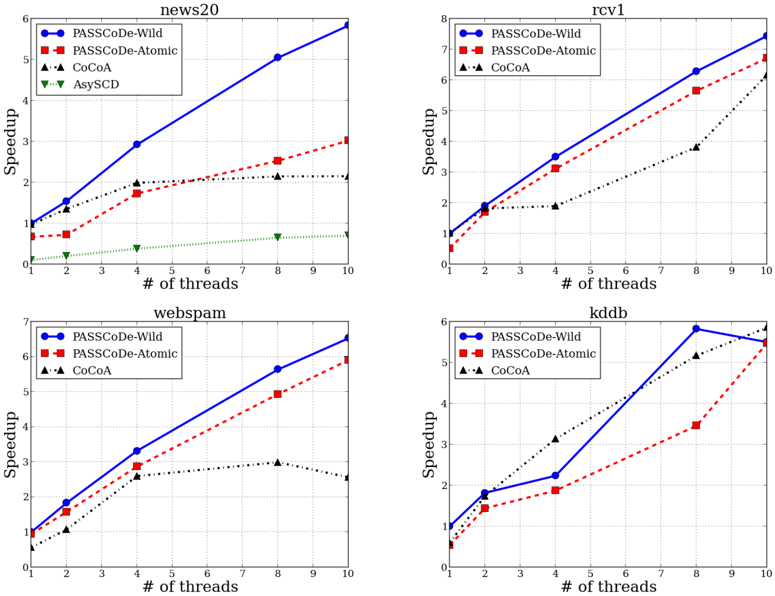


Accuracy



news20
rcv1
webspam
kddb
Accuracy
Initialization+Walltime (s)
PASSCoDe-Wild-10
PASSCoDe-Atomic-10
CoCoA-10
AsySCD-10
LIBLINEAR

Speedup


news20
PASSCoDe-Wild
PASSCoDe-Atomic
CoCoA
AsySCD
Speedup
# of threads
rcv1
PASSCoDe-Wild
PASSCoDe-Atomic
CoCoA
Speedup
# of threads
webspam
PASSCoDe-Wild
PASSCoDe-Atomic
CoCoA
Speedup
# of threads
kddb
PASSCoDe-Wild
PASSCoDe-Atomic
CoCoA
Speedup
# of threads

Conclusions

- *PASSCoDe*: an simple but effective asynchronous dual coordinate descent
- Analysis three variants
  - **PASSCoDe-Lock**
  - **PASSCoDe-Atomic**: established global linear convergence
  - **PASSCoDe-Wild**: backward analysis
- Future work: extend the analysis to L1-regularized problems
  - LASSO
  - L1-regularized Logistic Regression