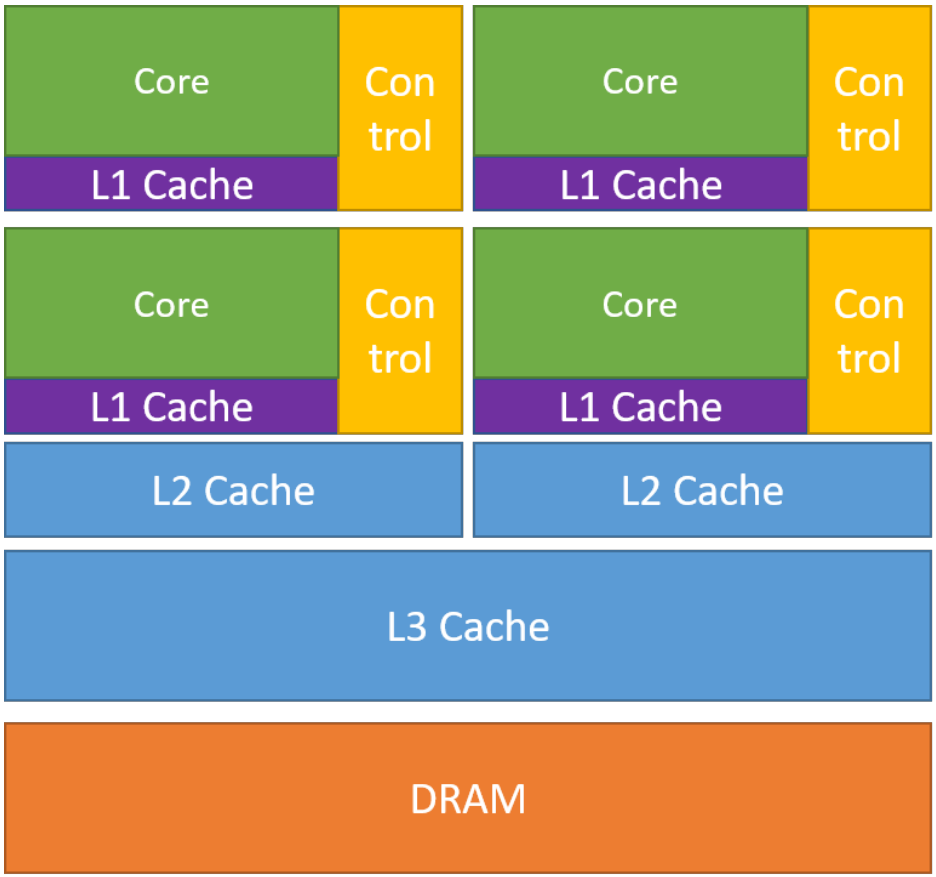
Serially Equivalent + Scalable Parallel Machine Learning

Dimitris Papailiopoulos

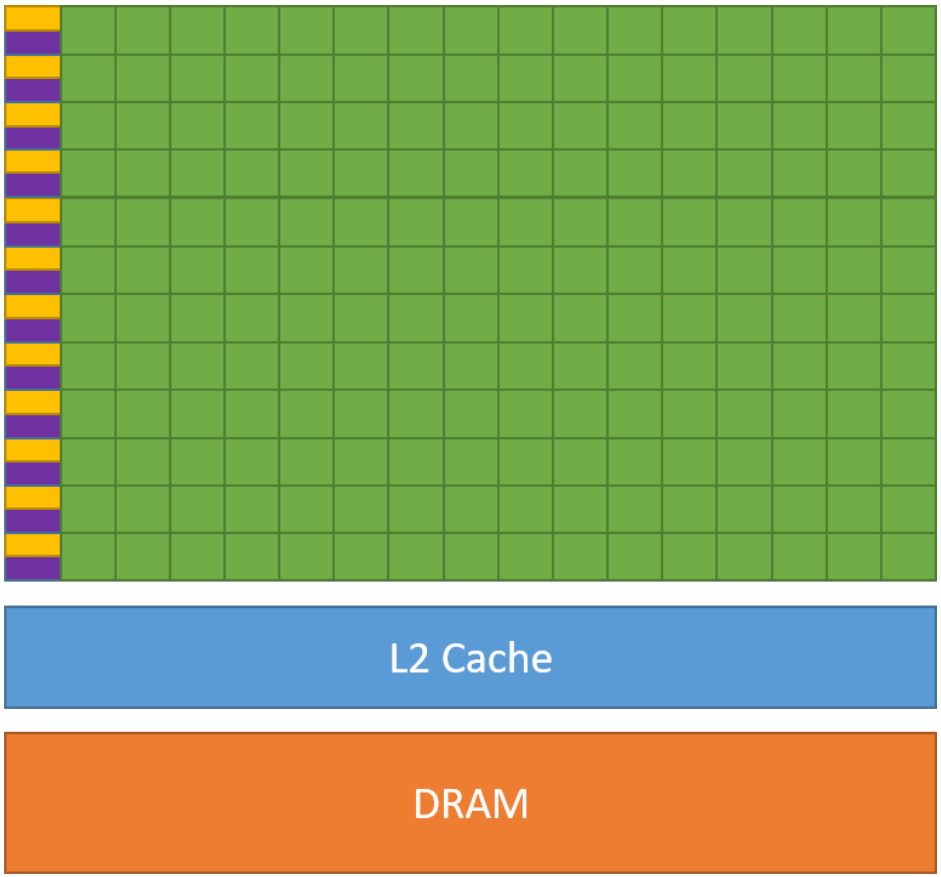
Today

- Serial Equivalence
- Beyond Hogwild
- How much asynchrony is possible?
- Open Problems

Single Machine, Multi-core



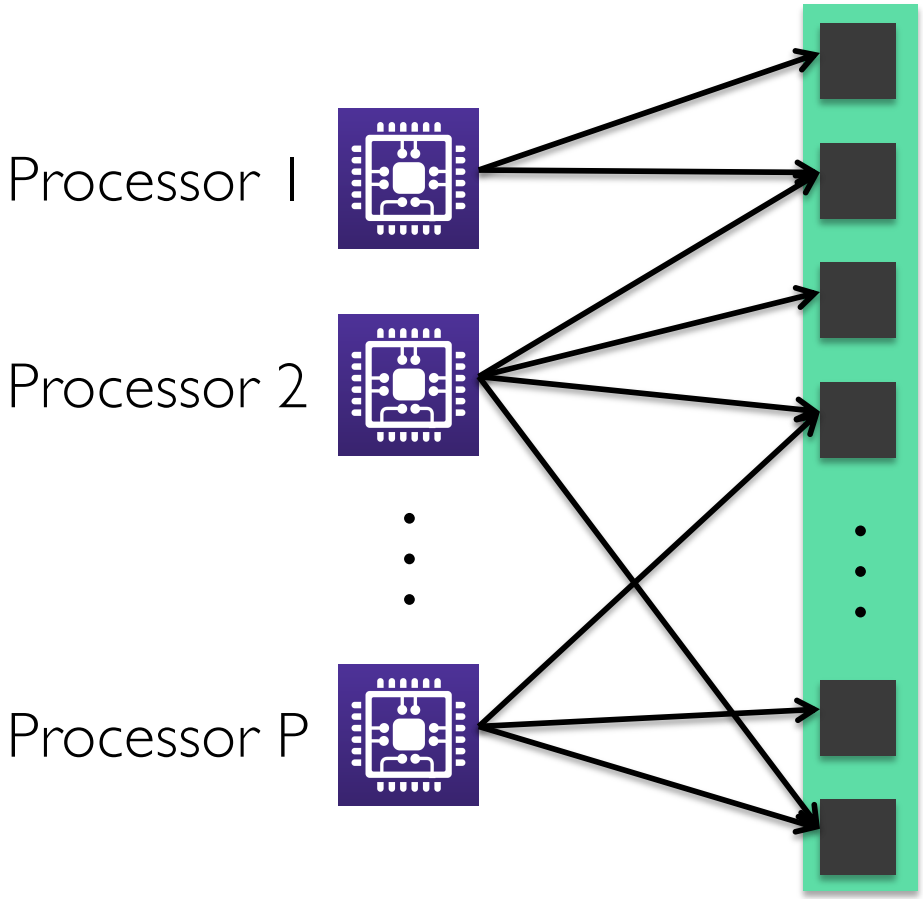
CPU



GPU

Today

(Maximally) Asynchronous



Today

Serial Equialence

$$A_{\text{serial}}(S, \pi) = A_{\text{parallel}}(S, \pi)$$

For all Data sets S

For all data order π (data points can be arbitrarily repeated)

Main advantage:

- we only need to "prove" speedups
- Convergence proofs inherited directly from serial

Main Issue:

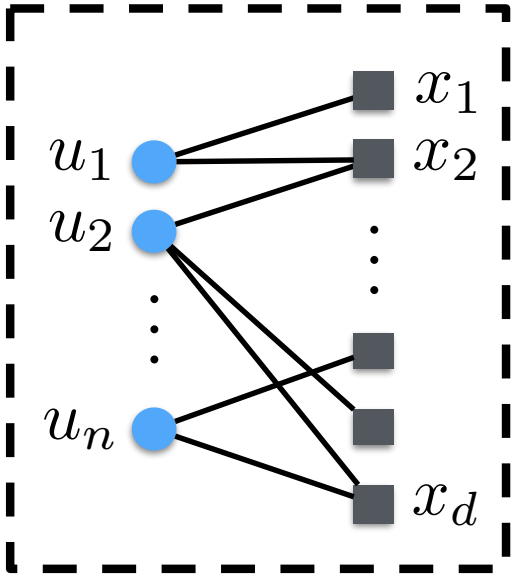
- Serial equivalence too strict
- Cannot guarantee any speedups in the general case

The Stochastic Updates Meta-algorithm

Stochastic Updates

Algorithm 1 Stochastic Updates pseudo-algorithm

1: Input: $\mathbf{x}$; $f_1, \ldots, f_n$; $u_1, \ldots, u_n$; $\mathcal{D}$; T.
2: **for** $t = 1 : T$ **do**
3: sample $i \sim \mathcal{D}$
4: $\mathbf{x}_{\mathcal{S}_i} = u_i(\mathbf{x}_{\mathcal{S}_i}, f_i)$ //update global model on $\mathcal{S}_i$
5: **Output:** $\mathbf{x}$



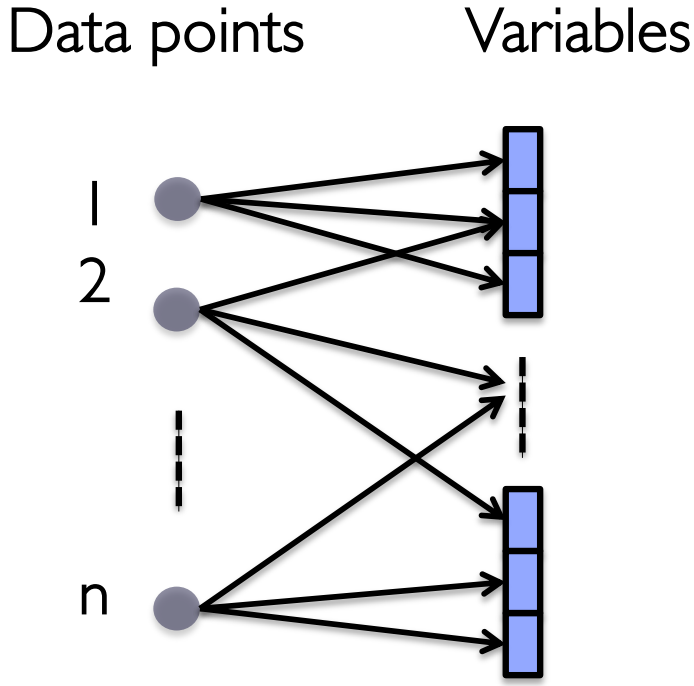
What does this solve?

Stochastic Updates: A family of ML Algorithms

Many algorithms with sparse access patterns:

- SGD
- SVRG / SAGA
- Matrix Factorization
- word2vec
- K-means
- Stochastic PCA
- Graph Clustering

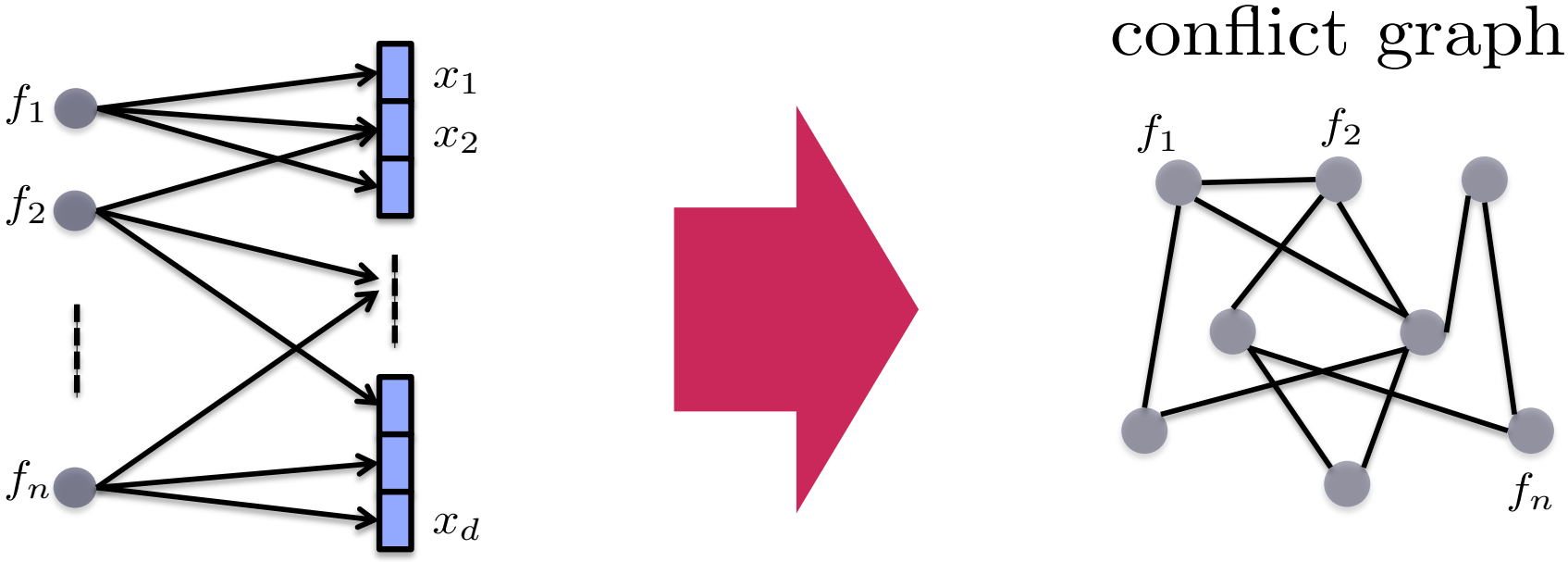
…



Can we parallelize under Serial Equivalence?

A graph view of Conflicts in Parallel Updates

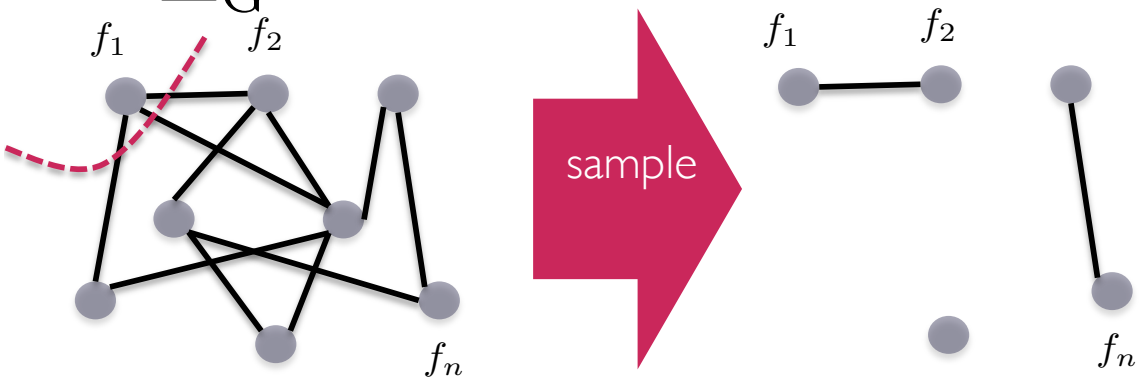
The Update Conflict Graph



An edge between 2 updates if they overlap

The Theorem
[Krivelevich'14]

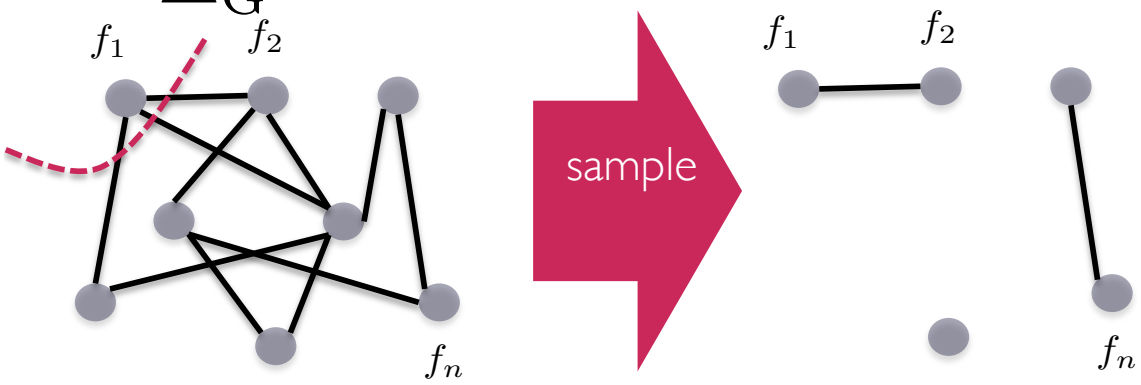
conflict graph Δ_G



Lemma:
Sample less than $P \leq (1-\epsilon)\frac{n}{\Delta_G}$ vertices (with/without replacement)

The Theorem
[Krivelevich'14]

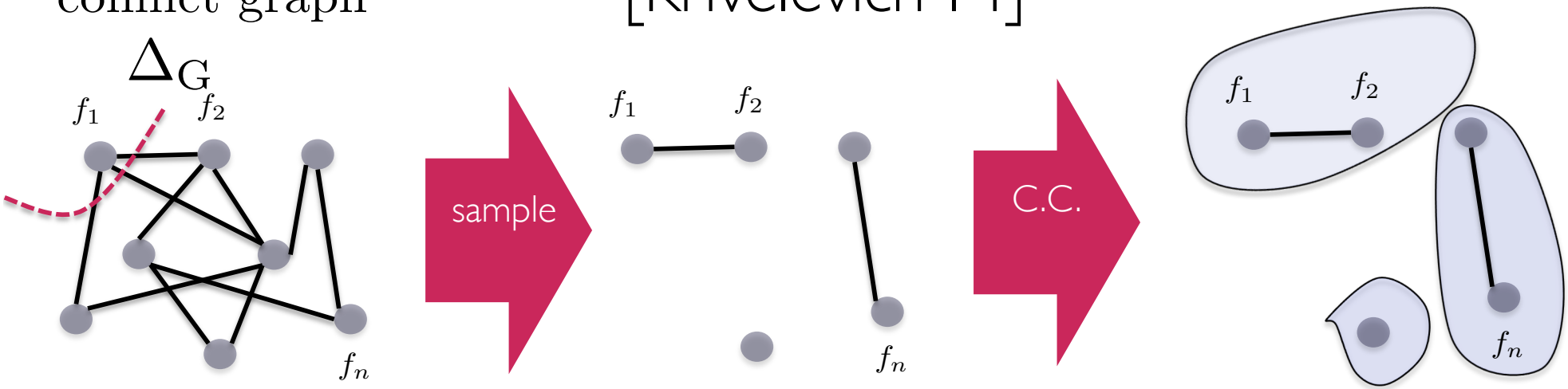
conflict graph Δ_G



Lemma:
Sample less than $P \leq (1-\epsilon)\frac{n}{\Delta_G}$ vertices (with/without replacement)

Then, the induced sub-graph shatters

The Theorem
[Krivelevich'14]



Lemma:
Sample less than $P \leq (1-\epsilon)\frac{n}{\Delta_G}$ vertices (with/without replacement)

Then, the induced sub-graph shatters,
The largest connected component has size

$$O\left(\frac{\log n}{\epsilon^2}\right)$$

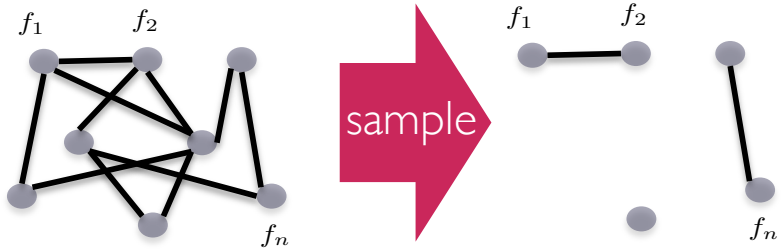
Even if the Graph was a Single Huge Conflict Component!

Building a Parallelization Framework out of a Single Theorem

Phase I

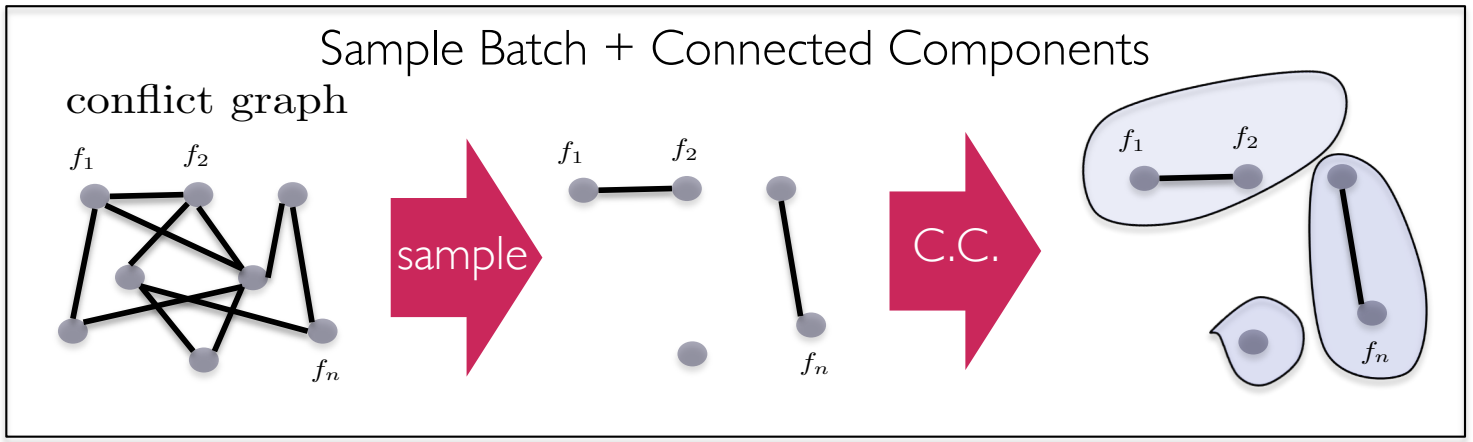
Sample Batch + Connected Components

conflict graph



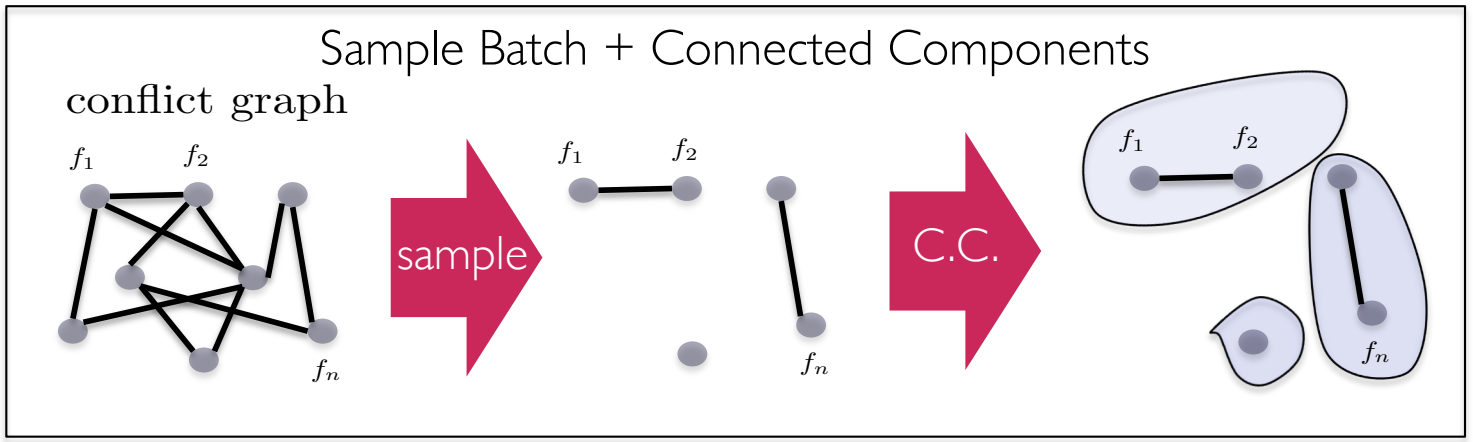
Sample $B = (1-\epsilon) \cdot \frac{n}{\Delta}$ vertices

Phase I



Sample $B = (1-\epsilon) \cdot \frac{n}{\Delta}$ vertices

Phase I



Sample $B = (1-\epsilon) \cdot \frac{n}{\Delta}$ vertices

Compute Conn. Components

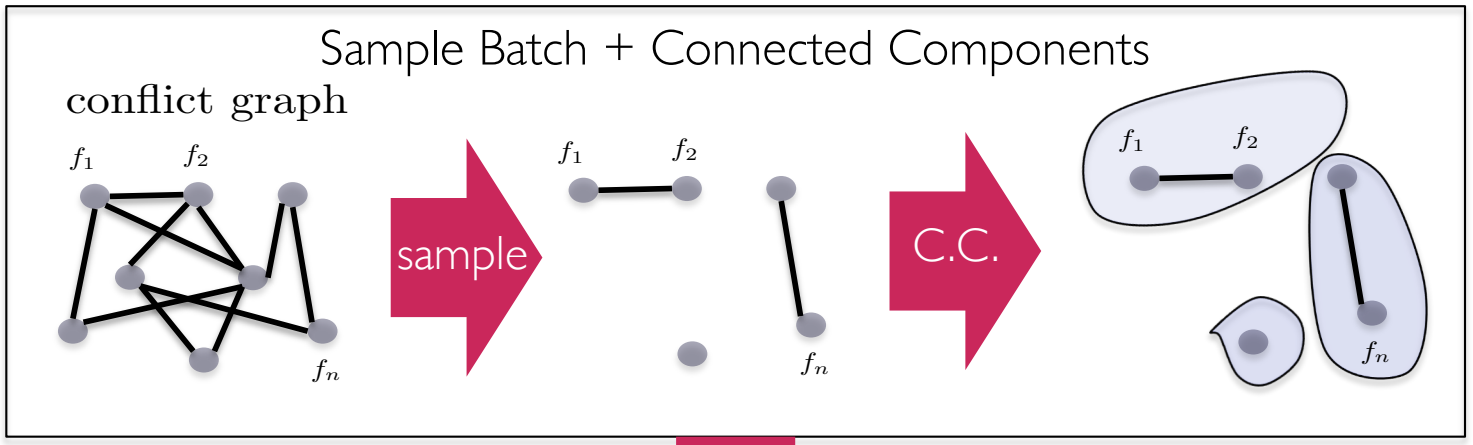
NOTE: No conflicts **across** groups!

Max Conn. Comp = logn => n/(Δlogn) tiny components

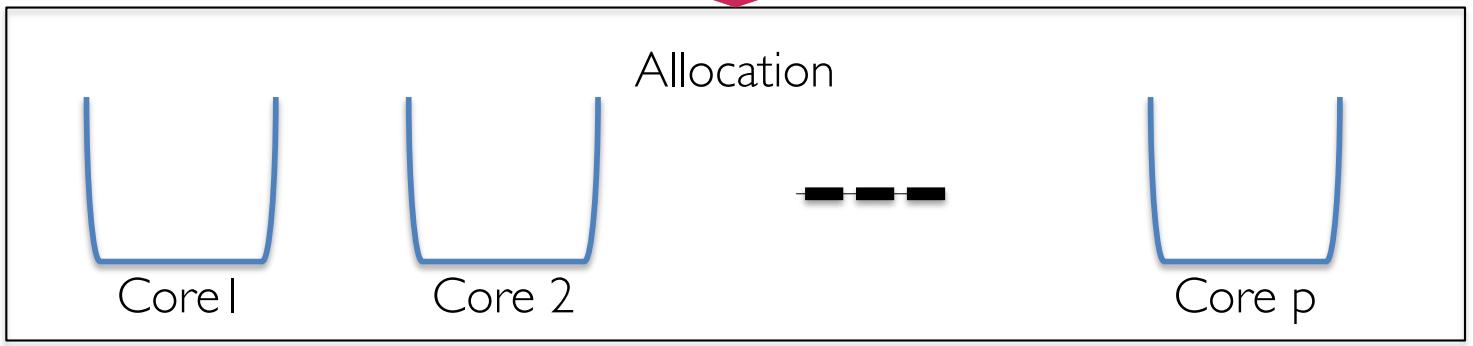
Yay! Good for parallelization

No conflicts across groups = we can run Stochastic Updates on each of them in parallel!

Phase I

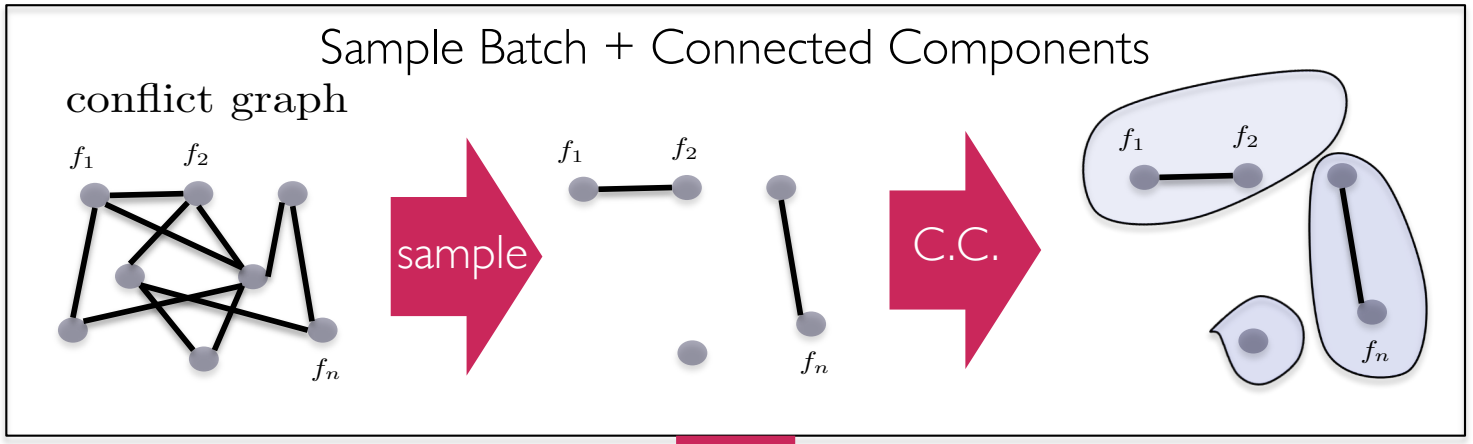


Phase II

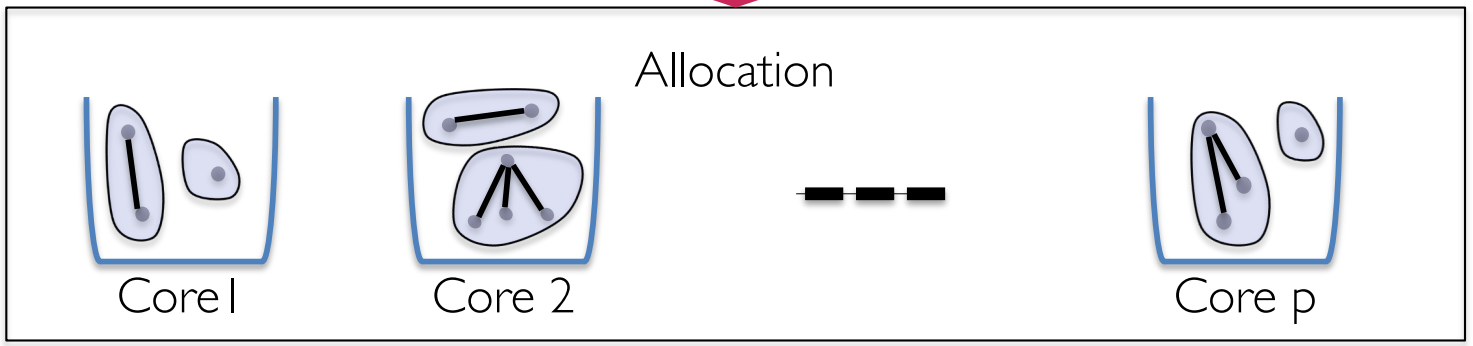


Cores < Batch size / logn = n/(Δlogn)

Phase I



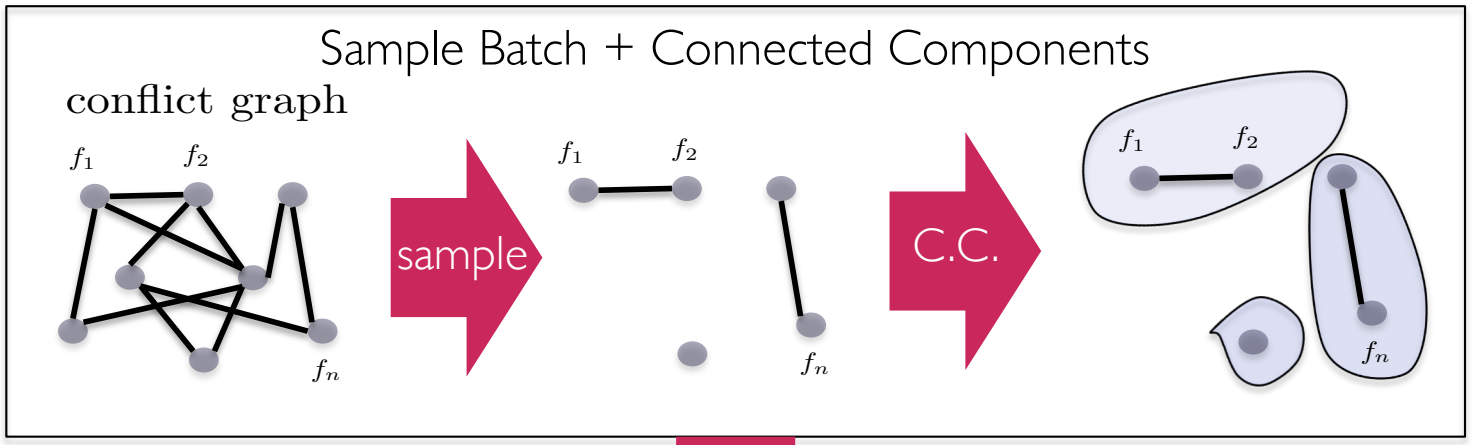
Phase II



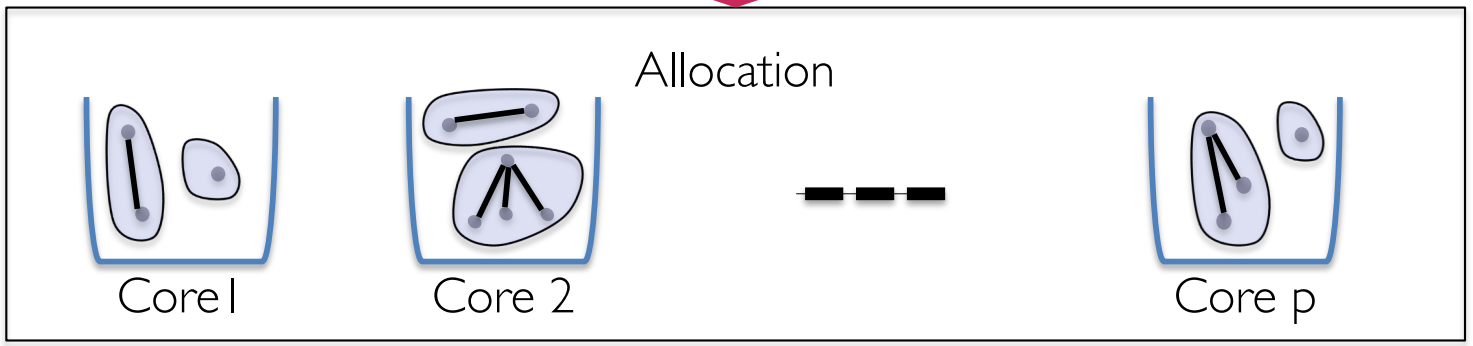
Cores < Batch size / logn = n/(Δlogn)

A Single Rule: Run the updates serially inside each connected component
Automatically Satisfied since we give each **conflict group** to a single core.

Phase I



Phase II

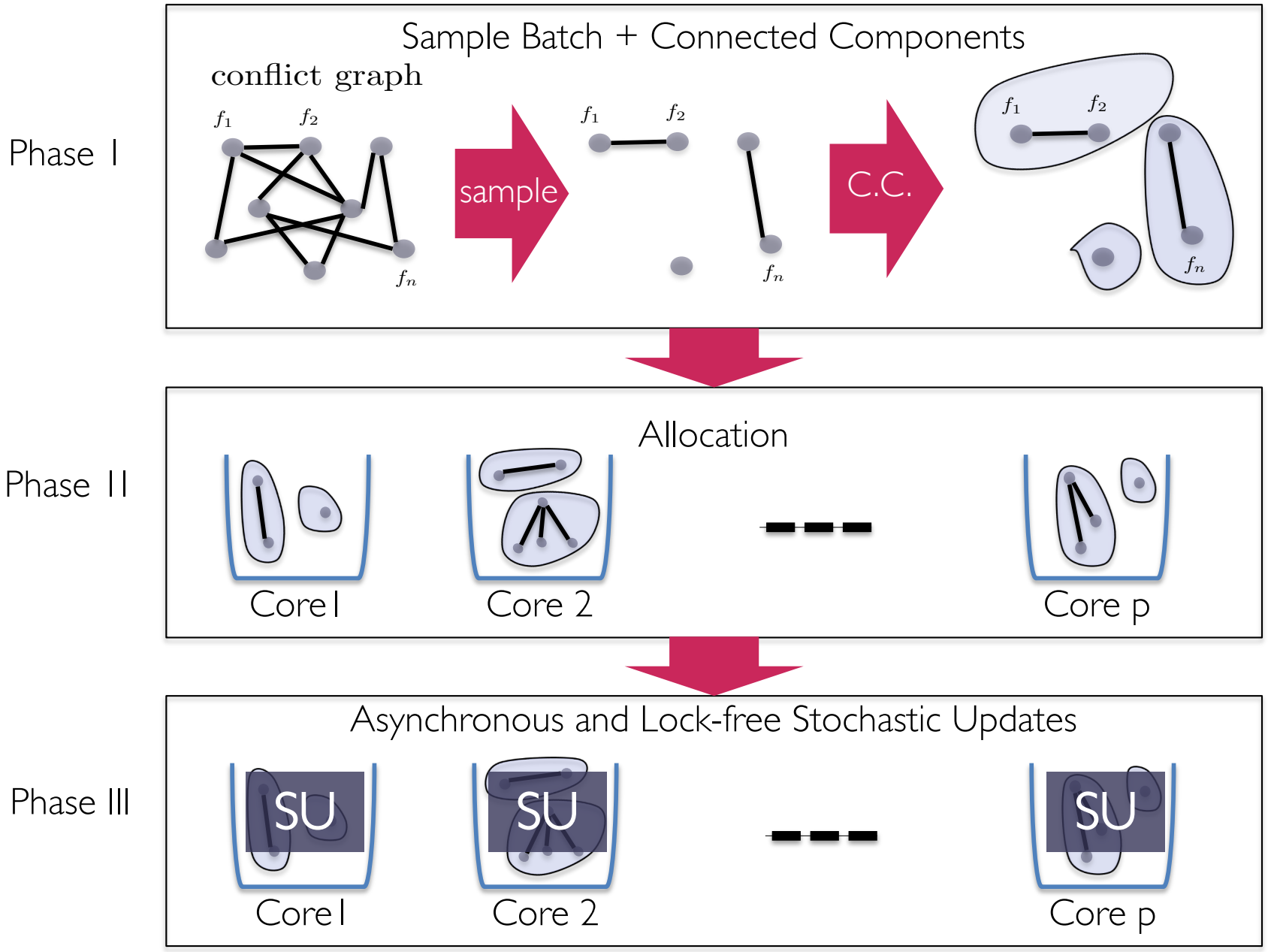


Policy I: Random allocation, good when cores << Batch size

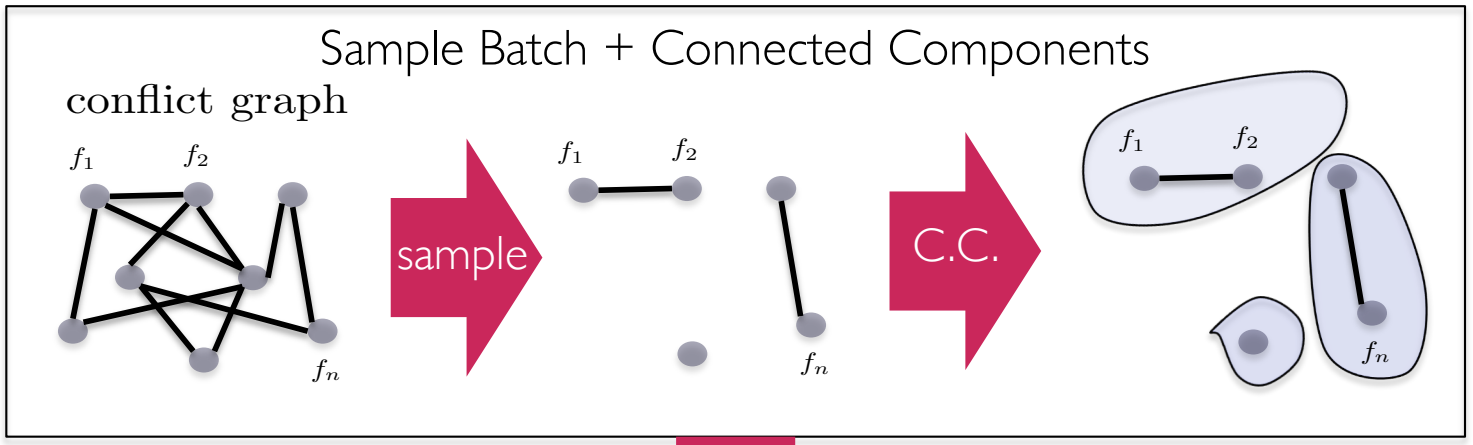
Policy II: Greedy min-weight allocation

(80% as good as optimal (which is NP-hard))

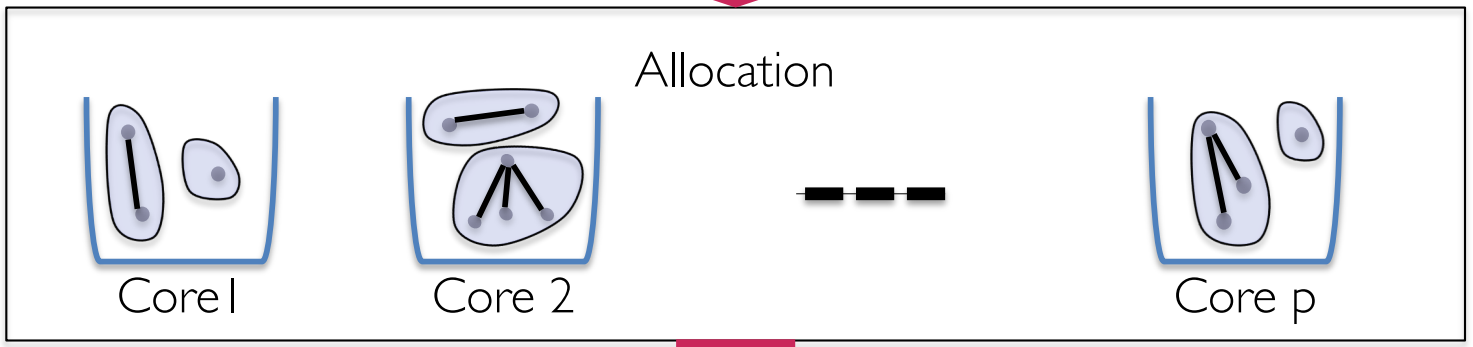
A Single Rule: Run the updates serially inside each connected component
Automatically Satisfied since we give each **conflict group** to a single core.


Phase I
Sample Batch + Connected Components
conflict graph
f_1
f_2
f_n
sample
C.C.
Phase II
Allocation
Core1
Core 2
Core p
Phase III
Asynchronous and Lock-free Stochastic Updates
SU
SU
SU
Core1
Core 2
Core p

Phase I



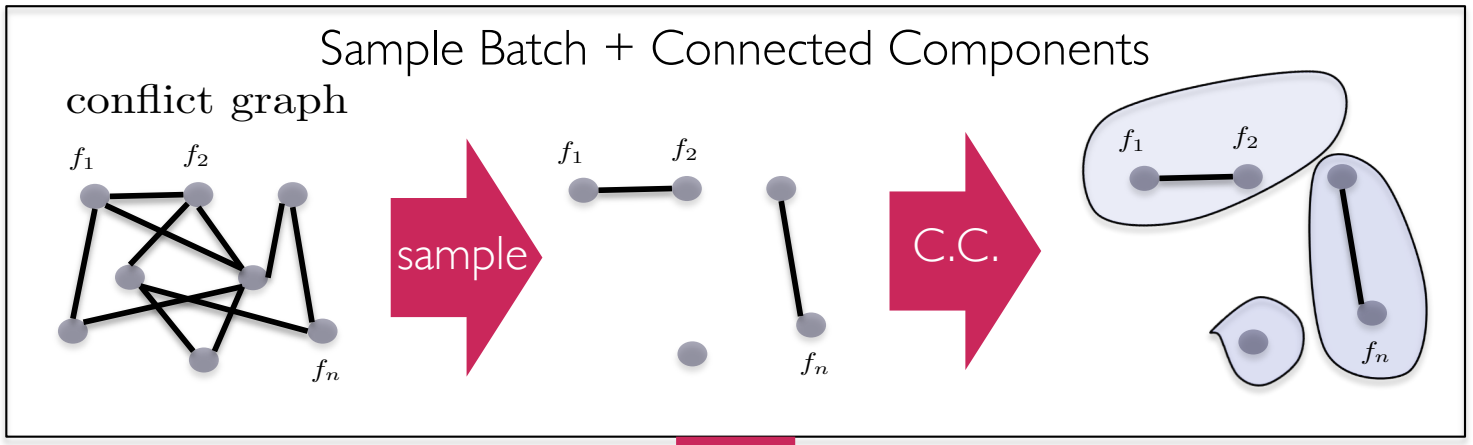
Phase II



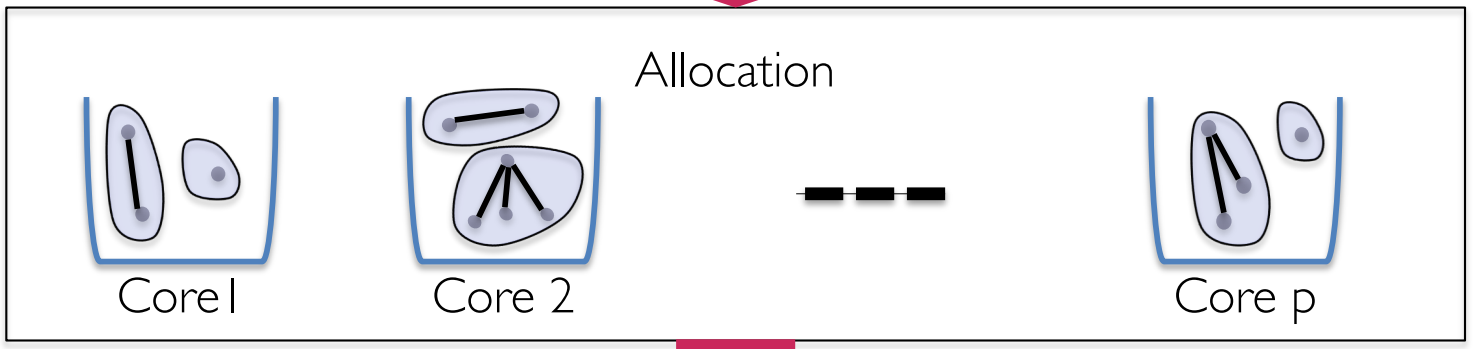
Phase III

Each core runs Asynchronously and Lock-free! (No communication!)

Phase I

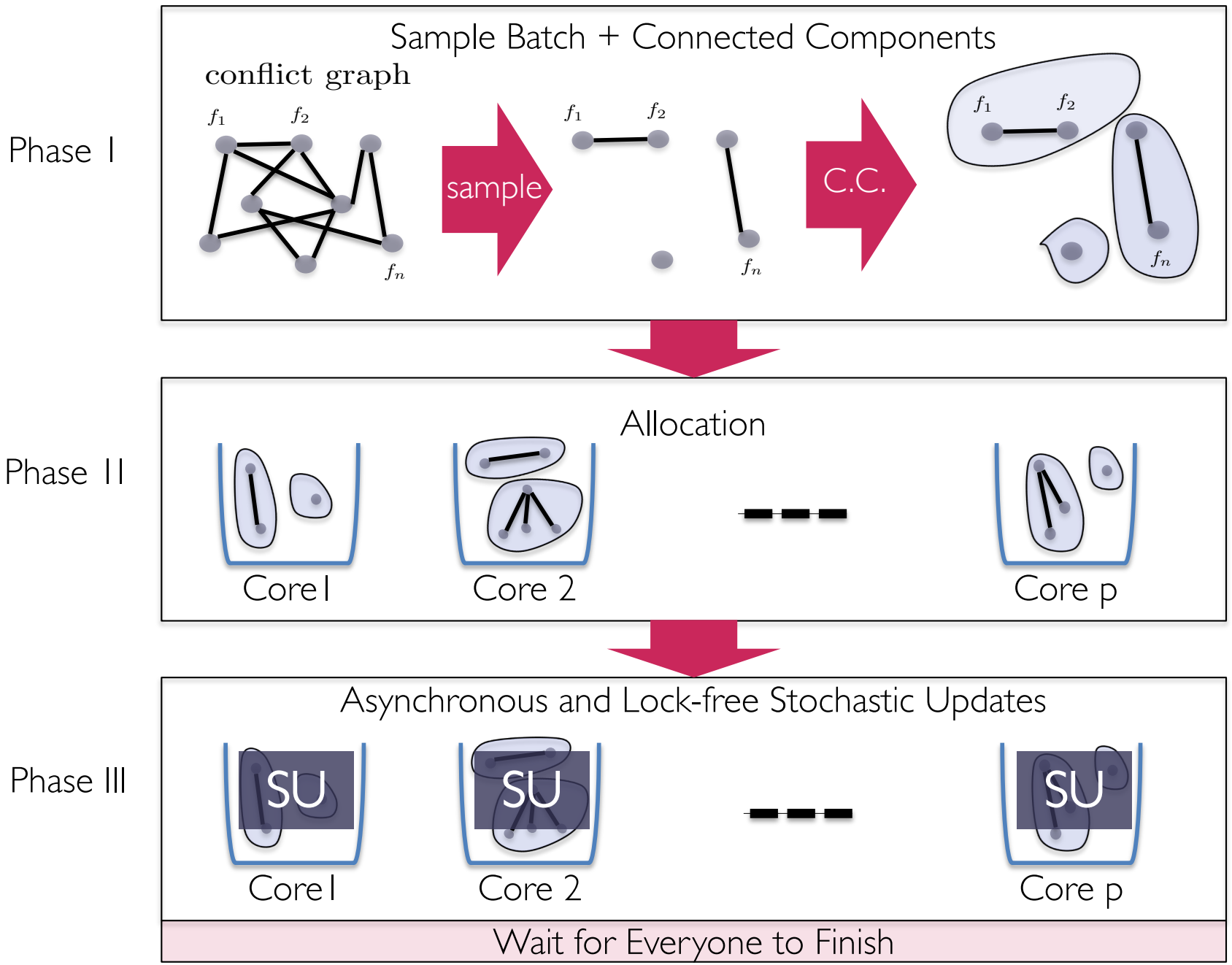


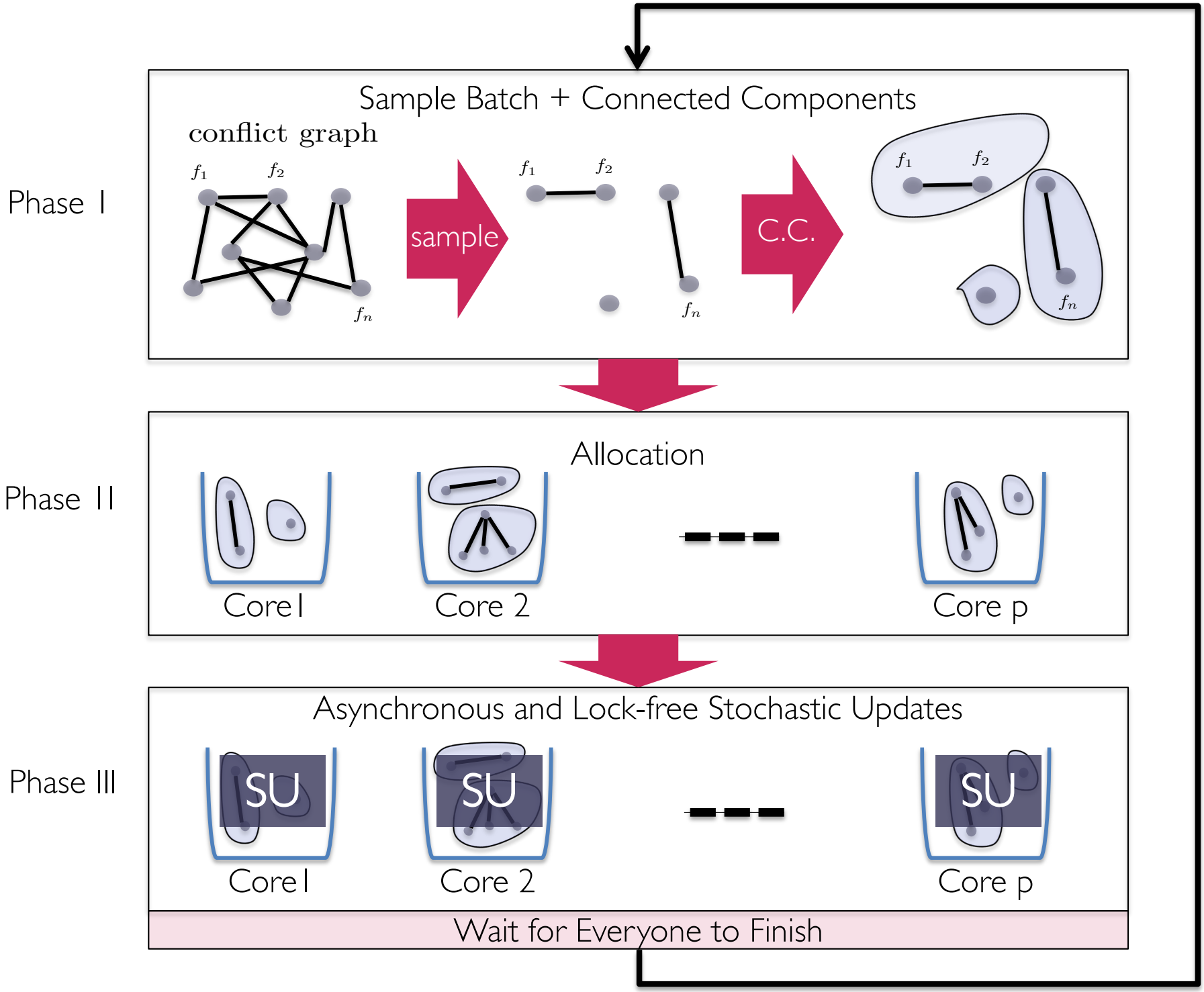
Phase II



Phase III

No memory Contention!
Not during **Reads**, neither during **Writes**!


Phase I
Sample Batch + Connected Components
conflict graph
f_1
f_2
f_n
sample
C.C.
Phase II
Allocation
Core1
Core 2
Core p
Phase III
Asynchronous and Lock-free Stochastic Updates
SU
Core1
Core 2
Core p
Wait for Everyone to Finish


Phase I
Sample Batch + Connected Components
conflict graph
f_1
f_2
f_n
sample
C.C.
Phase II
Allocation
Core1
Core 2
Core p
Phase III
Asynchronous and Lock-free Stochastic Updates
SU
SU
SU
Core1
Core 2
Core p
Wait for Everyone to Finish

Algorithm 2 CYCLADES

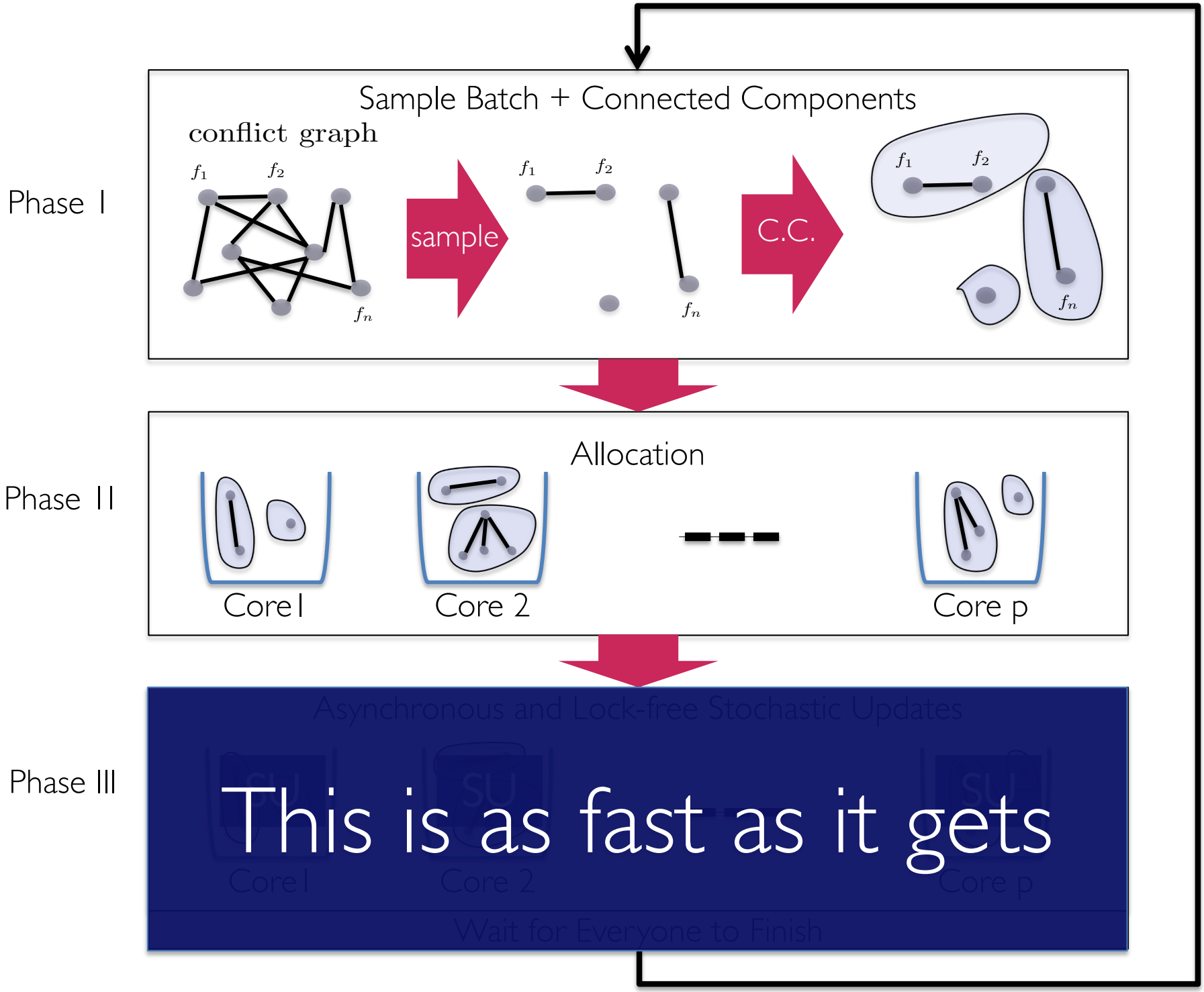
Input: G_u, T, B.
Sample $n_b = T/B$ subgraphs $G_u^1, \ldots, G_u^{n_b}$ from G_u
Cores compute in parallel CCs for sampled subgraphs
for batch $i = 1 : n_b$ **do**
    Allocation of $\mathcal{C}_1^i, \ldots \mathcal{C}_{m_i}^i$ to P cores

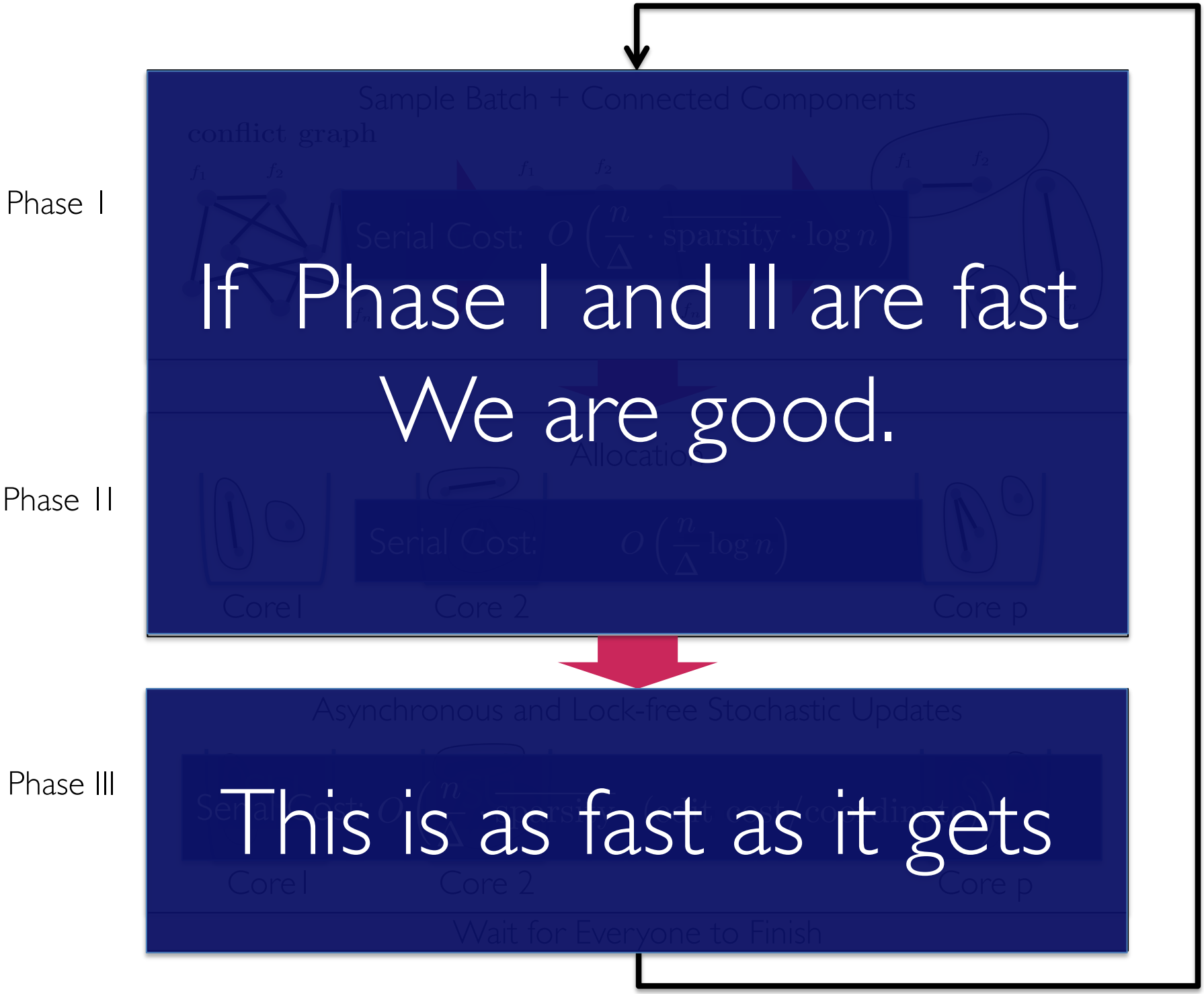
---

    for each core **in parallel do**
        for each allocated component $\mathcal{C}$ **do**
            for each update j (in order) from $\mathcal{C}$ **do**
                $\mathbf{x}_{\mathcal{S}_j} = u_j(\mathbf{x}_{\mathcal{S}_j}, f_j)$

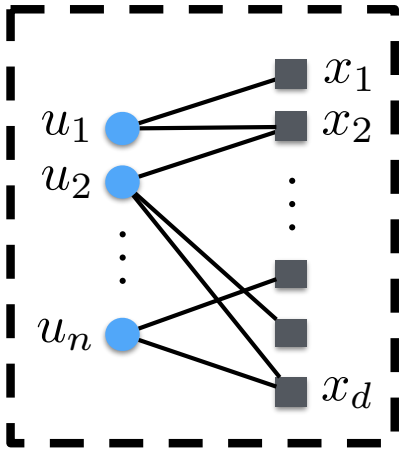
Output: x

This guarantees Serially Equivalence
But does it guarantee speedups?


Phase I
Sample Batch + Connected Components
conflict graph
f_1
f_2
f_n
sample
C.C.
Phase II
Allocation
Core1
Core 2
Core p
Phase III
Asynchronous and Lock-free Stochastic Updates
This is as fast as it gets
Wait for Everyone to Finish


Phase I
Phase II
Phase III
If Phase I and II are fast
We are good.
This is as fast as it gets

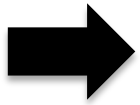
Main Theorem



Note:
Serial Cost = $E_u \cdot \kappa$
κ = cost / coordinate update

Theorem 4. *Let us assume any given update-variable graph G_u with average, and max left degree $\overline{\Delta}_L$ and Δ_L, such that $\frac{\overline{\Delta}_L}{\Delta_L} \leq \sqrt{n}$, and with induced max conflict degree Δ. Then,* CYCLADES *on $P = O(\frac{n}{\Delta \cdot \Delta_L})$ cores, with batch sizes $B = (1-\epsilon)\frac{n}{\Delta}$ can execute $T = c \cdot n$ updates, for any constant $c \geq 1$, selected uniformly at random with replacement, in time*

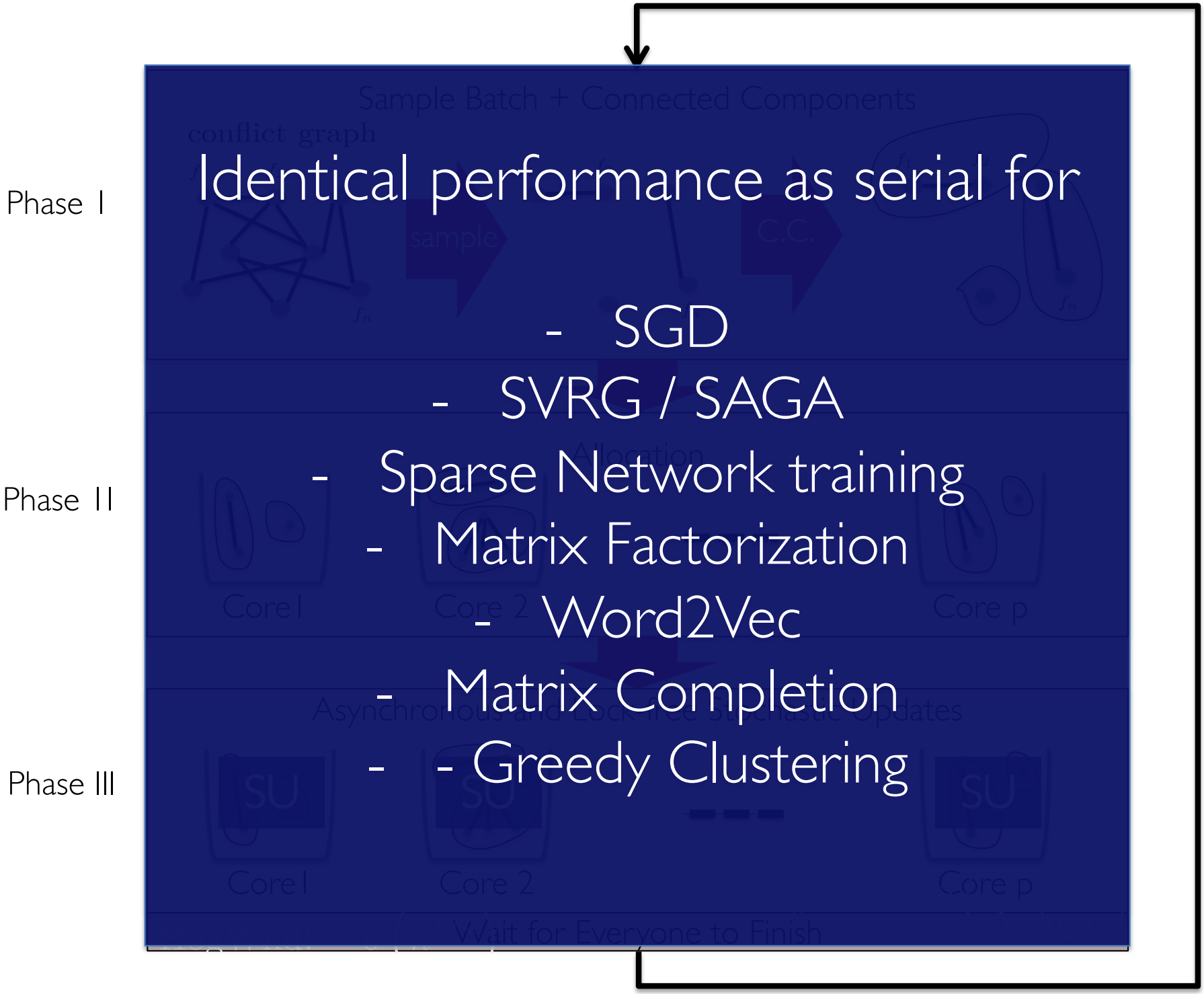
$$\mathcal{O}\left(\frac{E_u \cdot \kappa}{P} \cdot \log^2 n\right),$$



$$\text{Speedup} = \frac{P}{\log^2 n}$$

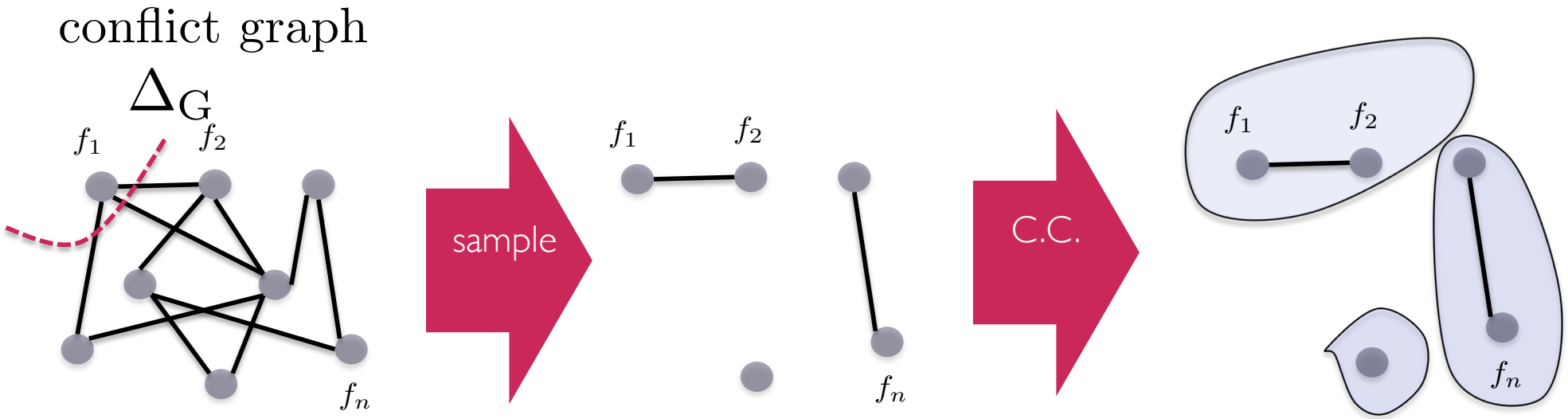
with high probability.

Assumptions:
1) Not too large max degree (approximate "regularity")
2) Not too many cores
3) Sampling according to the "Graph Theorem"


Phase I
Phase II
Phase III
Identical performance as serial for
- SGD
- SVRG / SAGA
- Sparse Network training
- Matrix Factorization
- Word2Vec
- Matrix Completion
- - Greedy Clustering

Fast Connected Components

Fast Connected Components

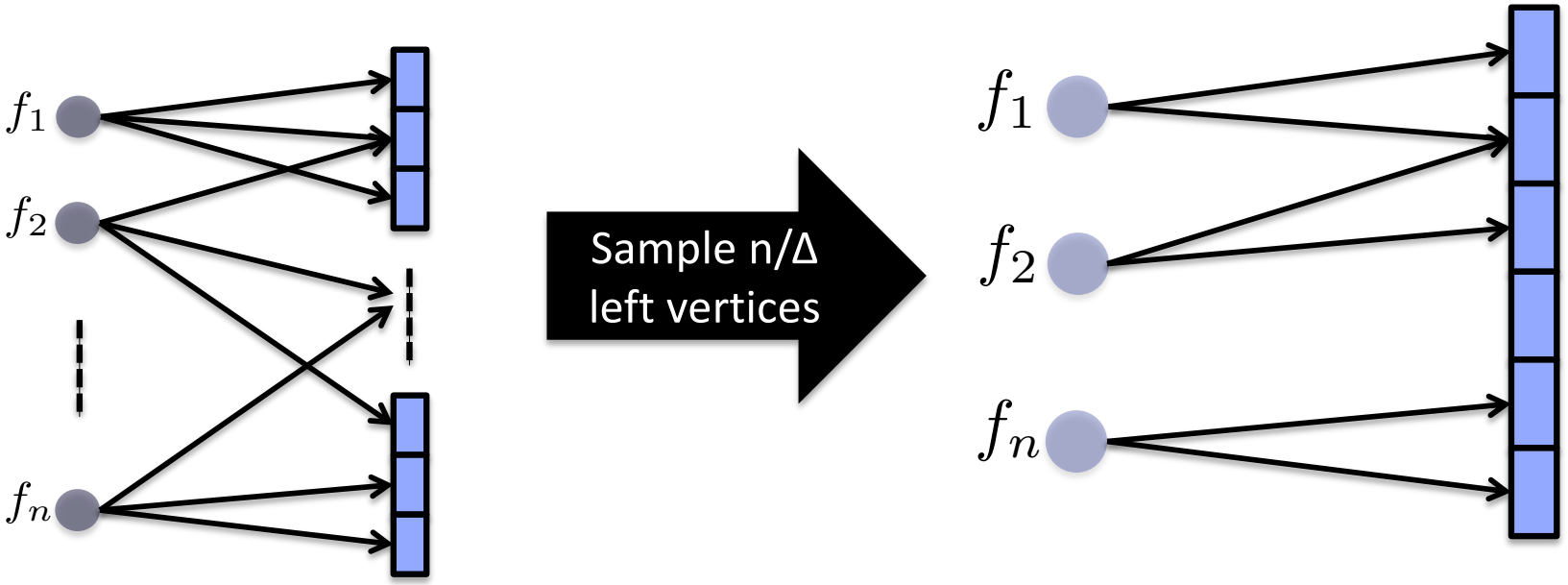


If you have the conflict graph CC is easy… O(Sampled Edges)

Building the conflict graph requires n^2 time…

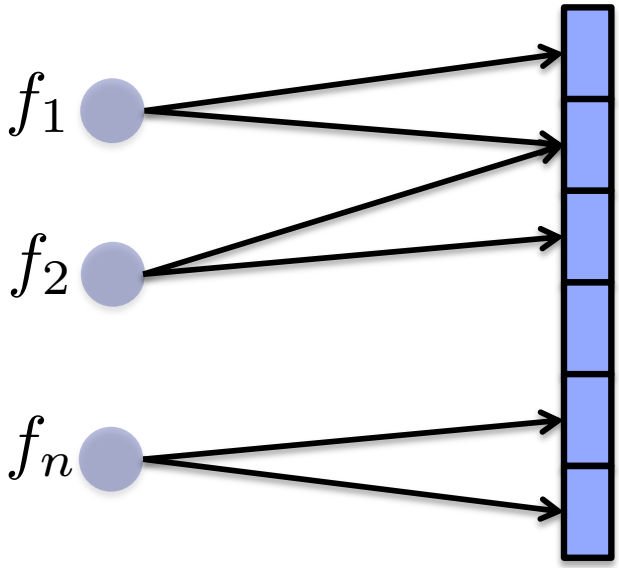
No, thanks.

Fast Connected Components



Sample on the Bipartite, not on the Conflict Graphs

Fast Connected Components



Simple Message passing Idea

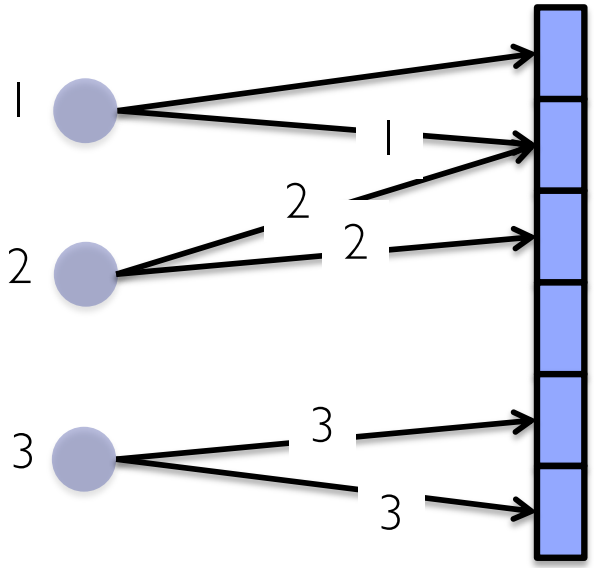
-Gradients send their IDs

-Coordinates Compute Min and Send Back

-Gradients Compute Min and Send Back

Iterate till you're done

Fast Connected Components



Simple Message passing Idea

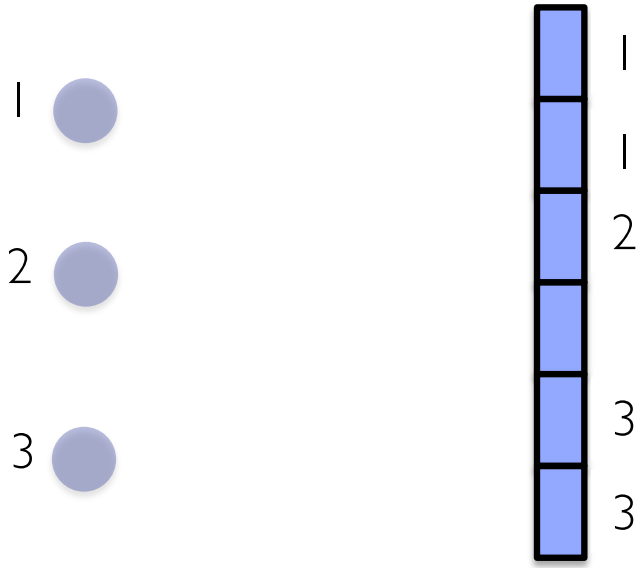
-Gradients send their IDs

-Coordinates Compute Min and Send Back

-Gradients Compute Min and Send Back

Iterate till you're done

Fast Connected Components



Simple Message passing Idea

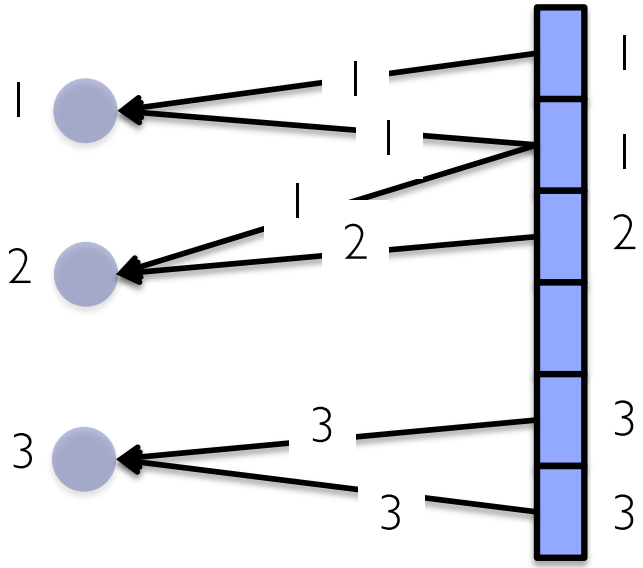
-Gradients send their IDs

-Coordinates Compute Min and Send Back

-Gradients Compute Min and Send Back

Iterate till you're done

Fast Connected Components



Simple Message passing Idea

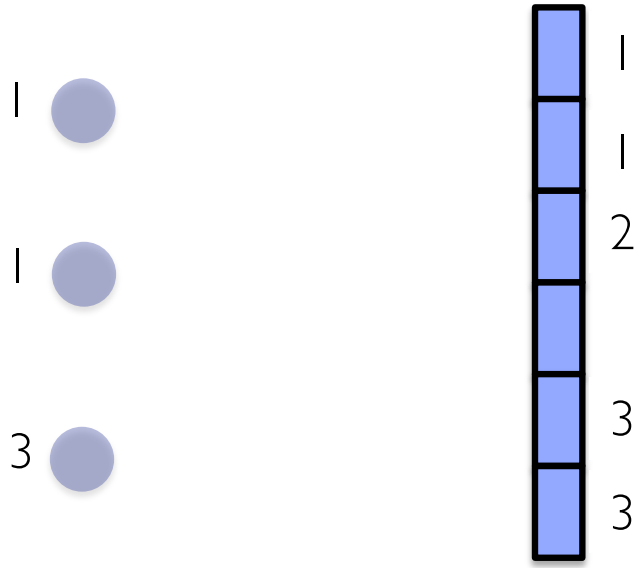
-Gradients send their IDs

-Coordinates Compute Min and Send Back

-Gradients Compute Min and Send Back

Iterate till you're done

Fast Connected Components



Simple Message passing Idea

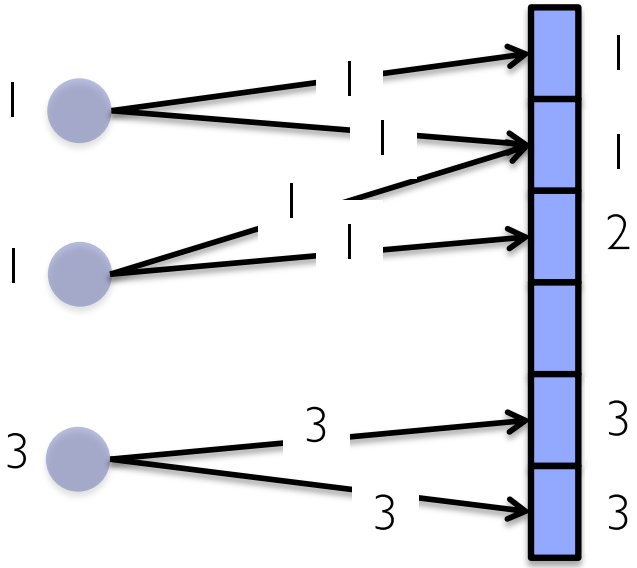
-Gradients send their IDs

-Coordinates Compute Min and Send Back

-Gradients Compute Min and Send Back

Iterate till you're done

Fast Connected Components



Simple Message passing Idea

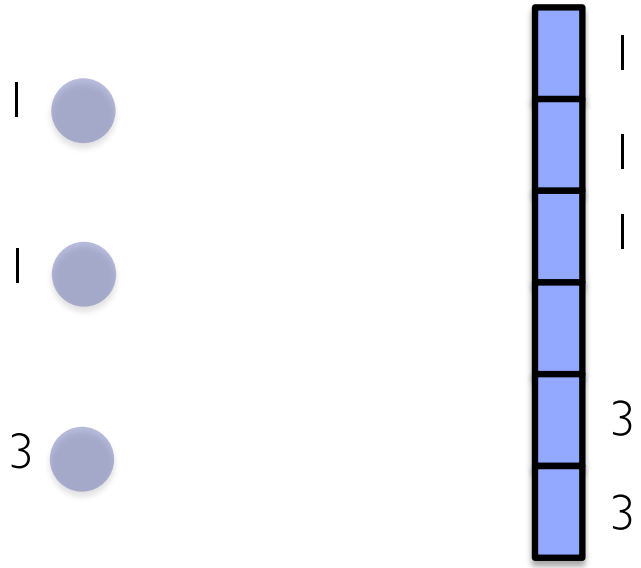
-Gradients send their IDs

-Coordinates Compute Min and Send Back

-Gradients Compute Min and Send Back

Iterate till you're done

Fast Connected Components



Simple Message passing Idea

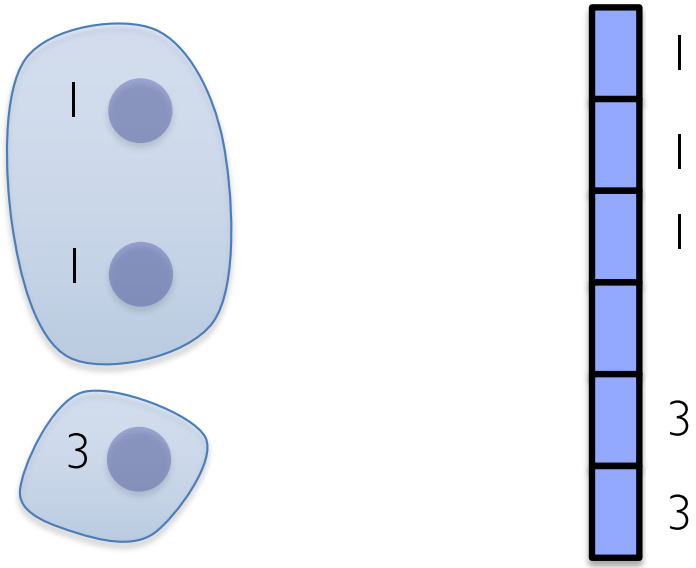
-Gradients send their IDs

-Coordinates Compute Min and Send Back

-Gradients Compute Min and Send Back

Iterate till you're done

Fast Connected Components



Simple Message passing Idea

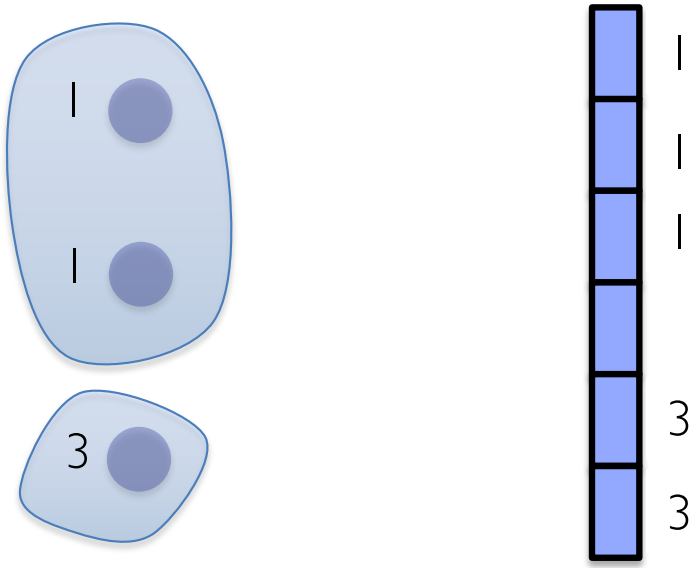
-Gradients send their IDs

-Coordinates Compute Min and Send Back

-Gradients Compute Min and Send Back

Iterate till you're done

Fast Connected Components

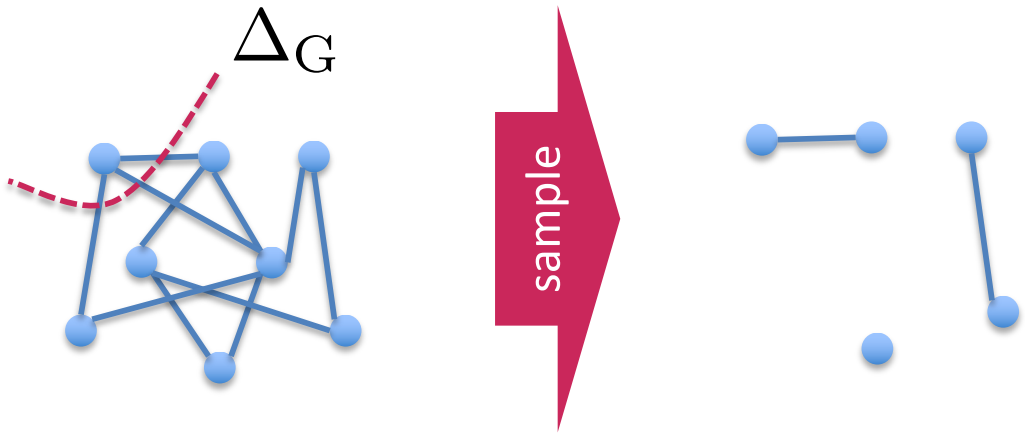


$$\text{Cost} = O\left(\frac{E_u \log^2 n}{P}\right)$$

Same assumptions as main theorem

Proof of "The Theorem"

The Theorem



Lemma:

Activate each vertex with probability

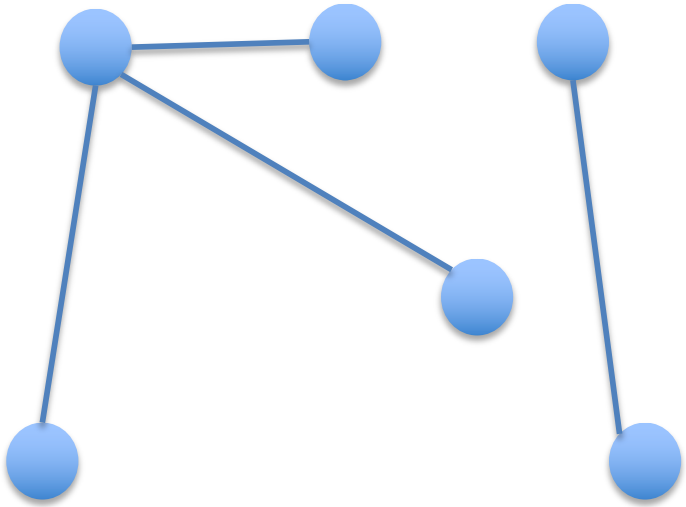
$$p = (1-\varepsilon)/\Delta$$

Then, the induced subgraph shatters,
and the largest connected component has size

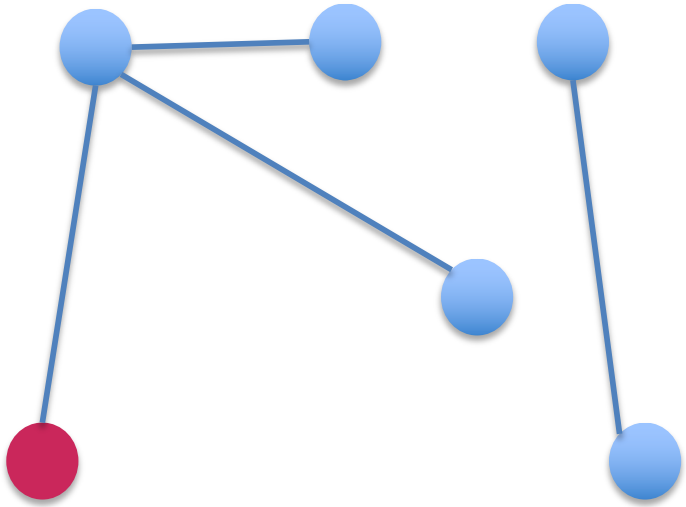
$$\frac{4}{\epsilon^2} \cdot \log n$$

DFS 101

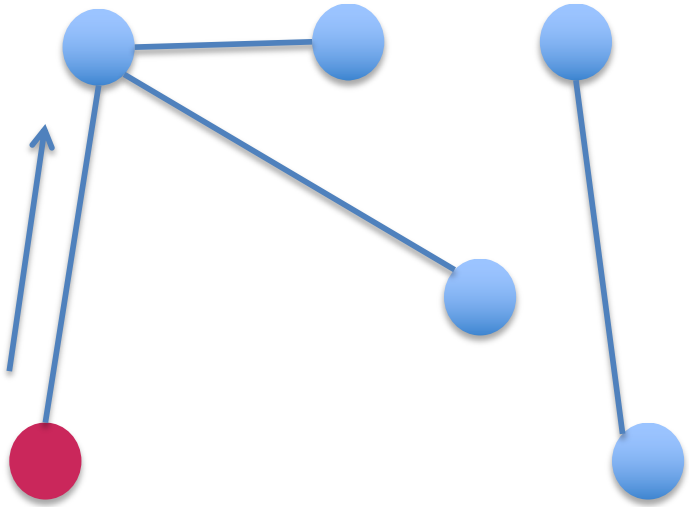
DFS



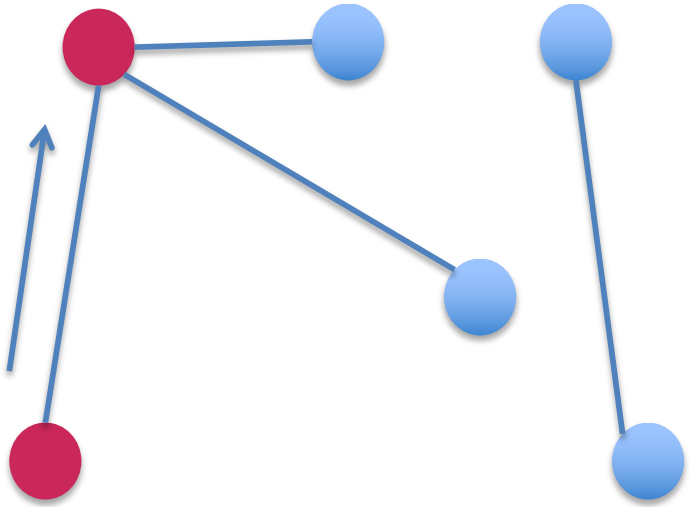
DFS



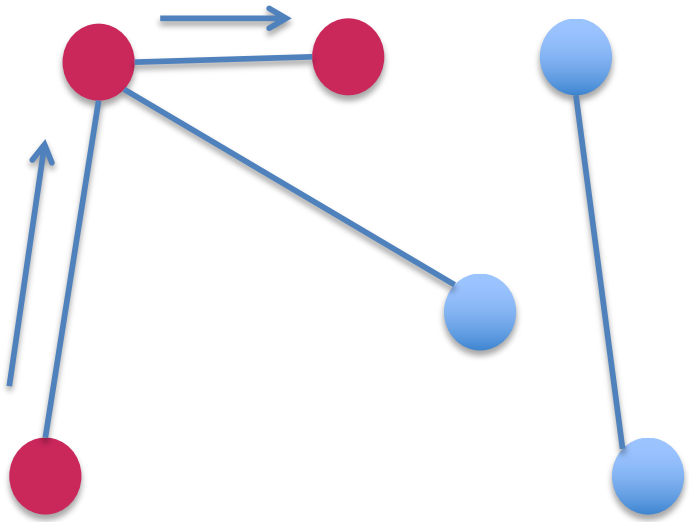
DFS



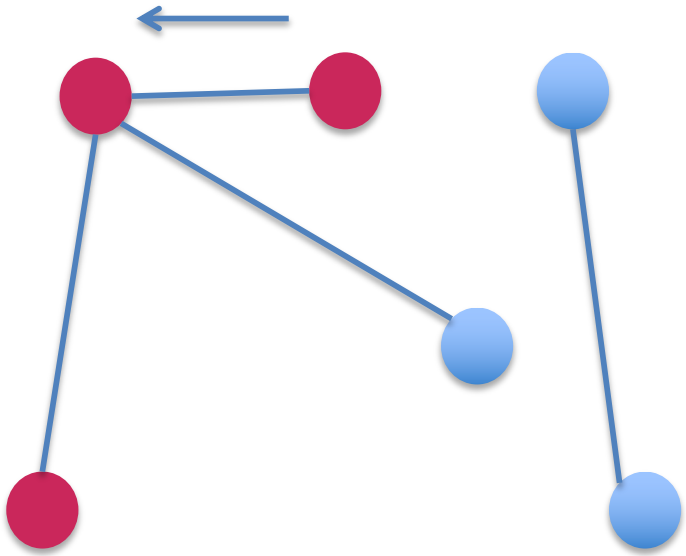
DFS



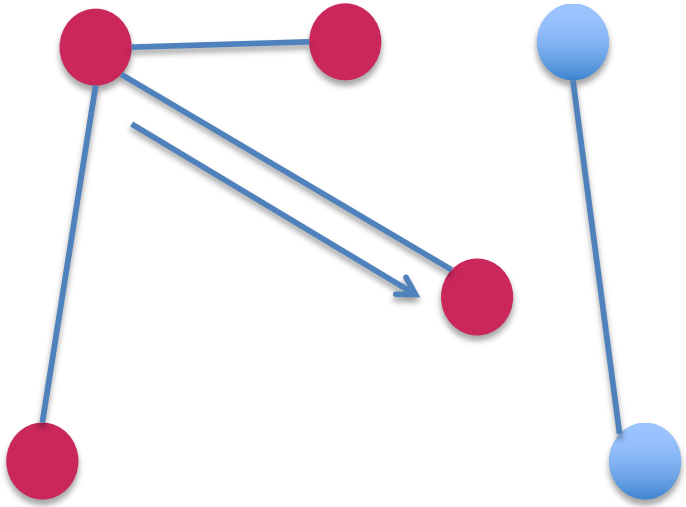
DFS



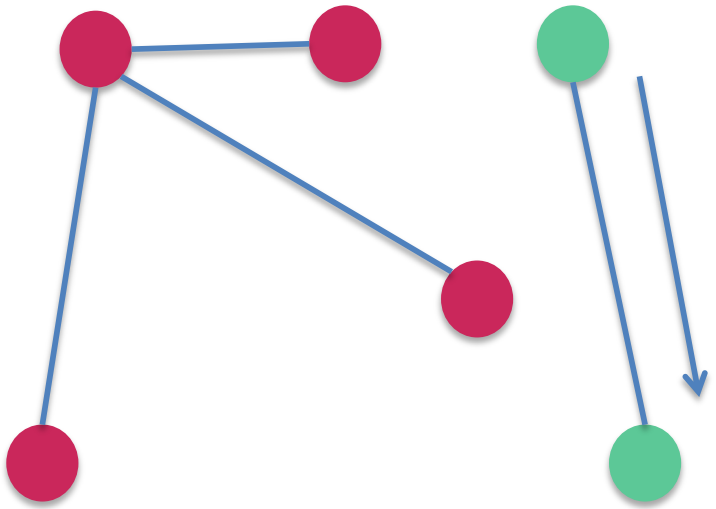
DFS



DFS

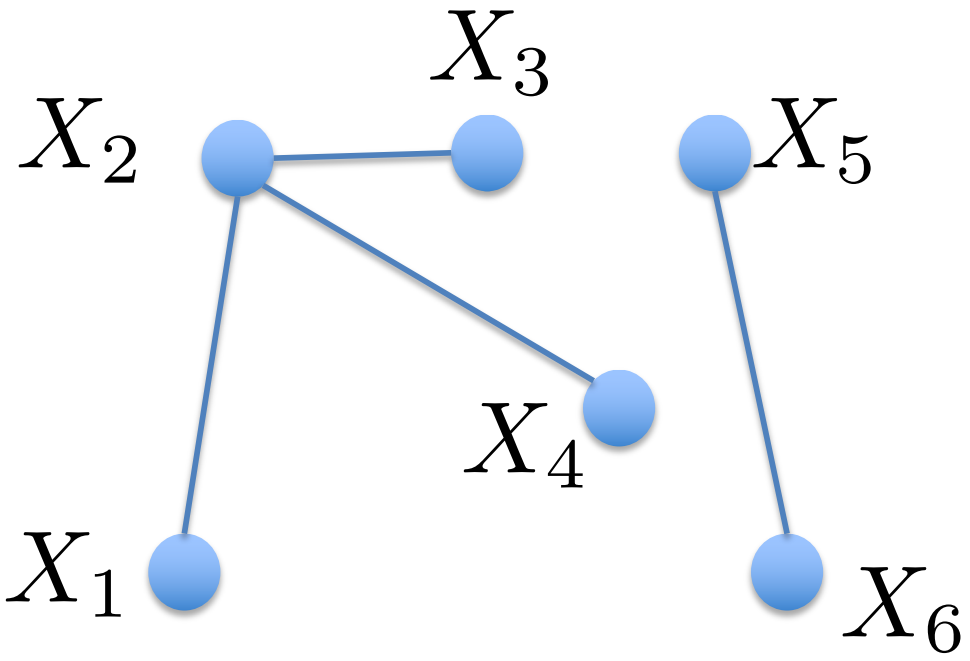


DFS



Probabilistic DFS

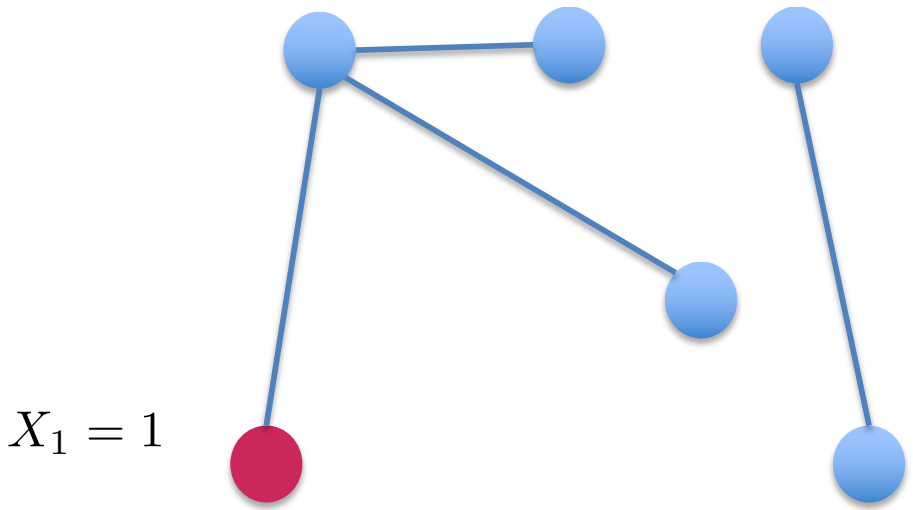
DFS with random coins



Algorithm:

- Flip a coin for each vertex DFS wants to visit
- If 1 visit, if 0 don't visit and delete with its edges

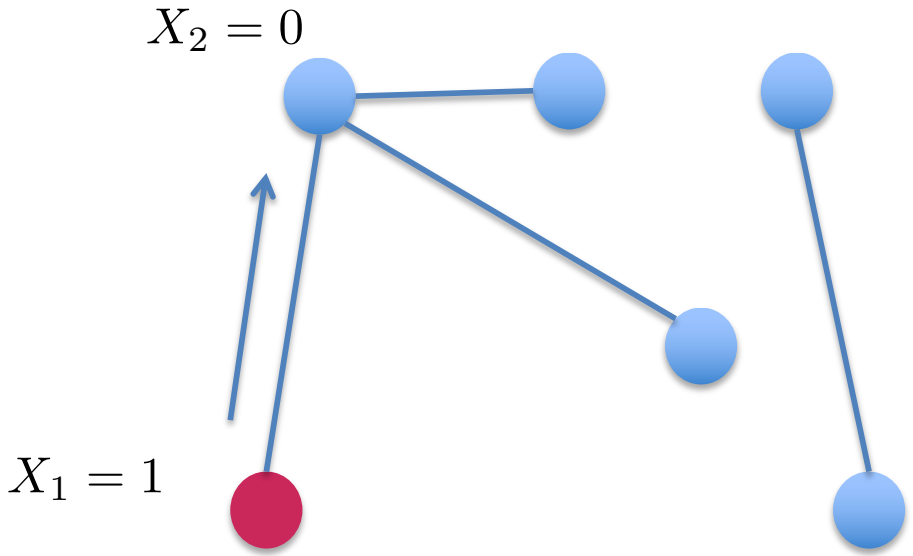
DFS with random coins



Algorithm:

- Flip a coin for each vertex DFS wants to visit
- If 1 visit, if 0 don't visit and delete with its edges

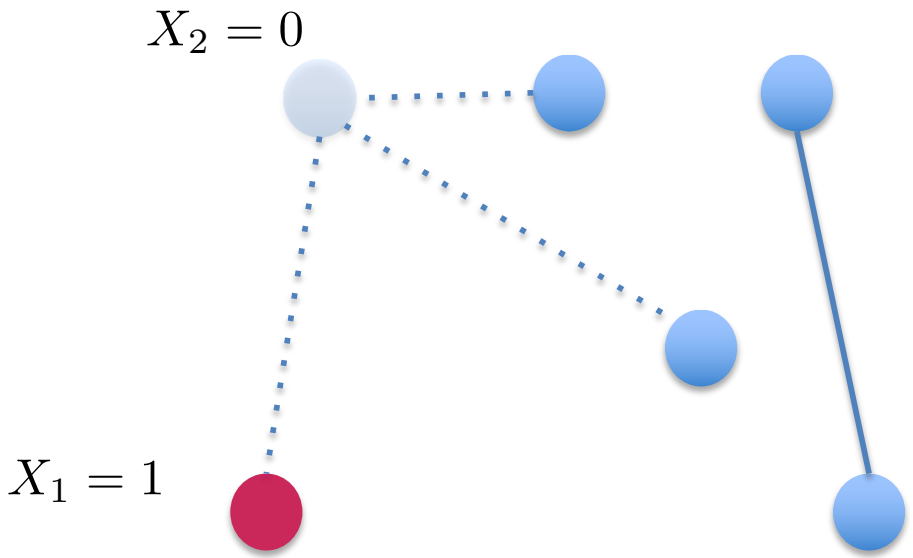
DFS with random coins



Algorithm:

- Flip a coin for each vertex DFS wants to visit
- If 1 visit, if 0 don't visit and delete with its edges

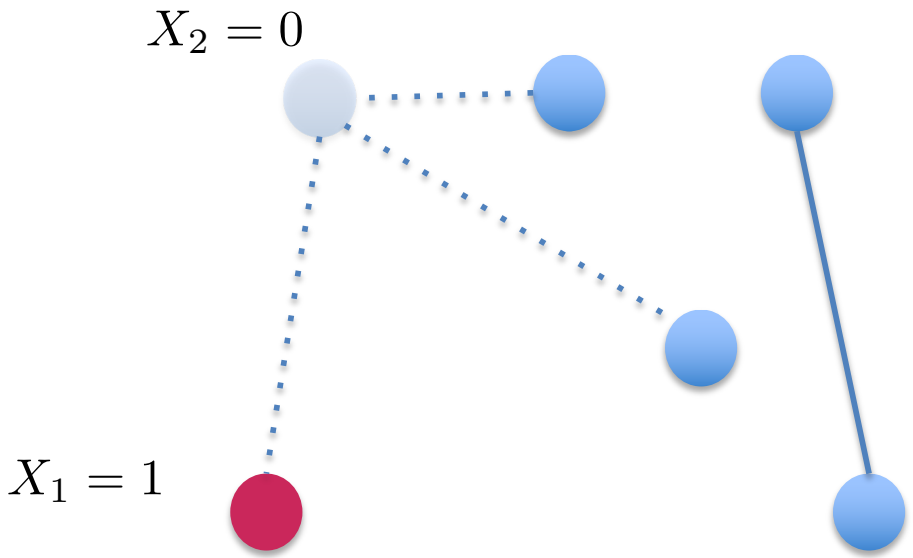
DFS with random coins



Algorithm:

- Flip a coin for each vertex DFS wants to visit
- If 1 visit, if 0 don't visit and delete with its edges

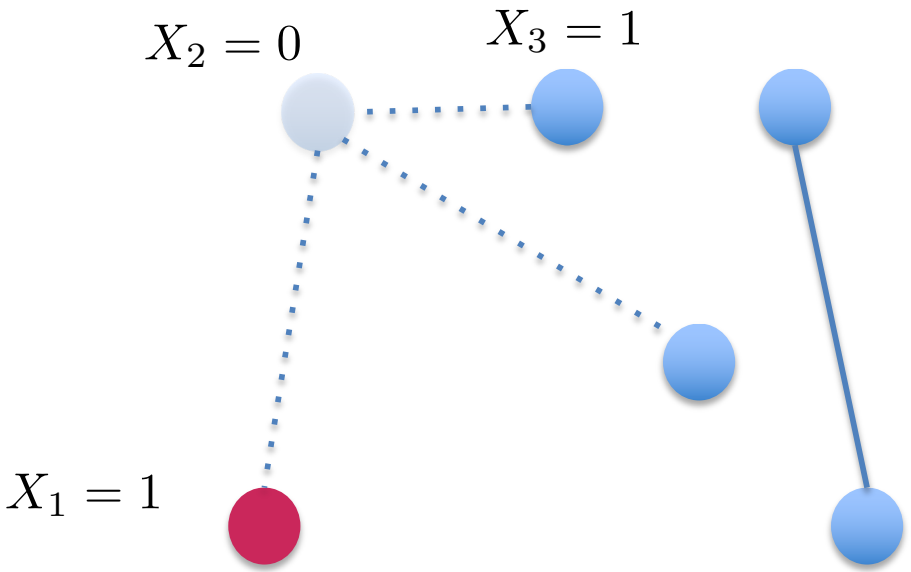
DFS with random coins



Algorithm:

- Flip a coin for each vertex DFS wants to visit
- If 1 visit, if 0 don't visit and delete with its edges

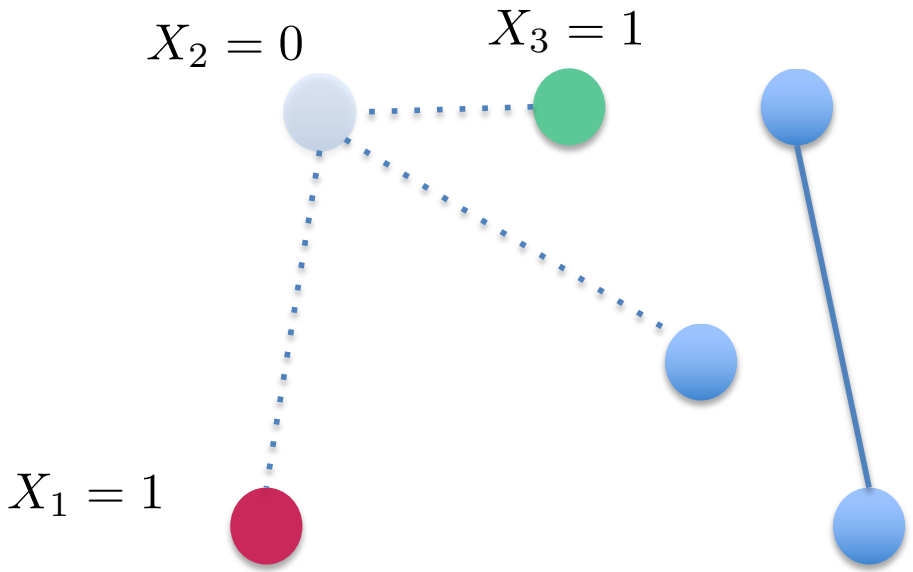
DFS with random coins



Algorithm:

- Flip a coin for each vertex DFS wants to visit
- If 1 visit, if 0 don't visit and delete with its edges

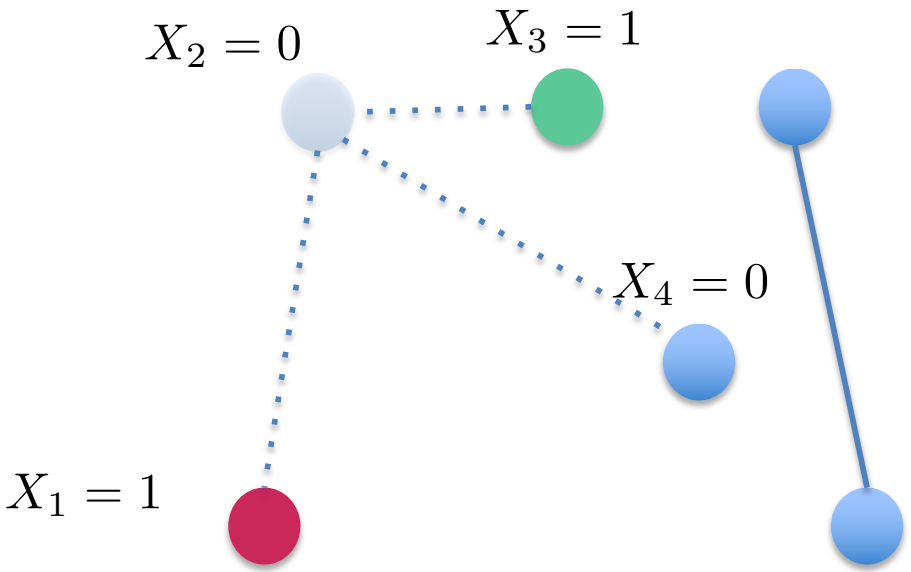
DFS with random coins



Algorithm:

- Flip a coin for each vertex DFS wants to visit
- If 1 visit, if 0 don't visit and delete with its edges

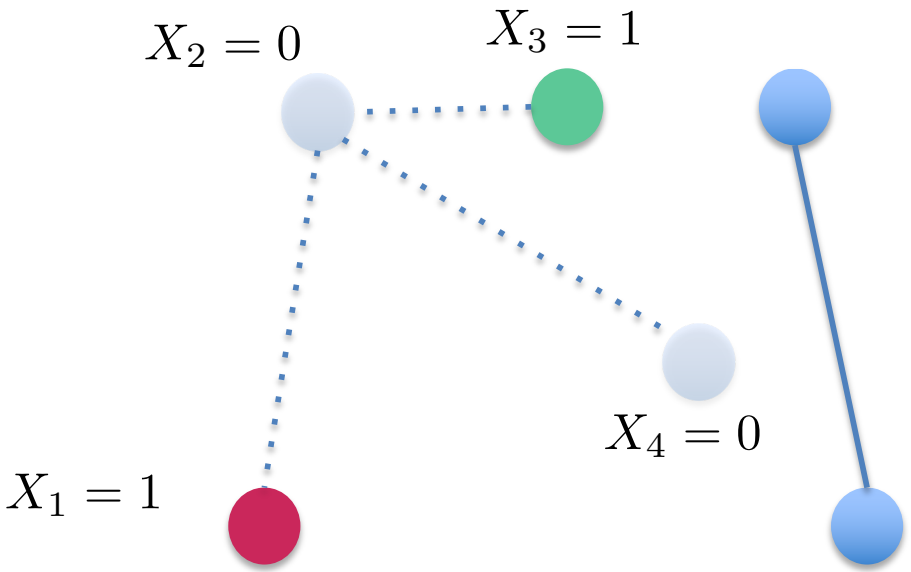
DFS with random coins



Algorithm:

- Flip a coin for each vertex DFS wants to visit
- If 1 visit, if 0 don't visit and delete with its edges

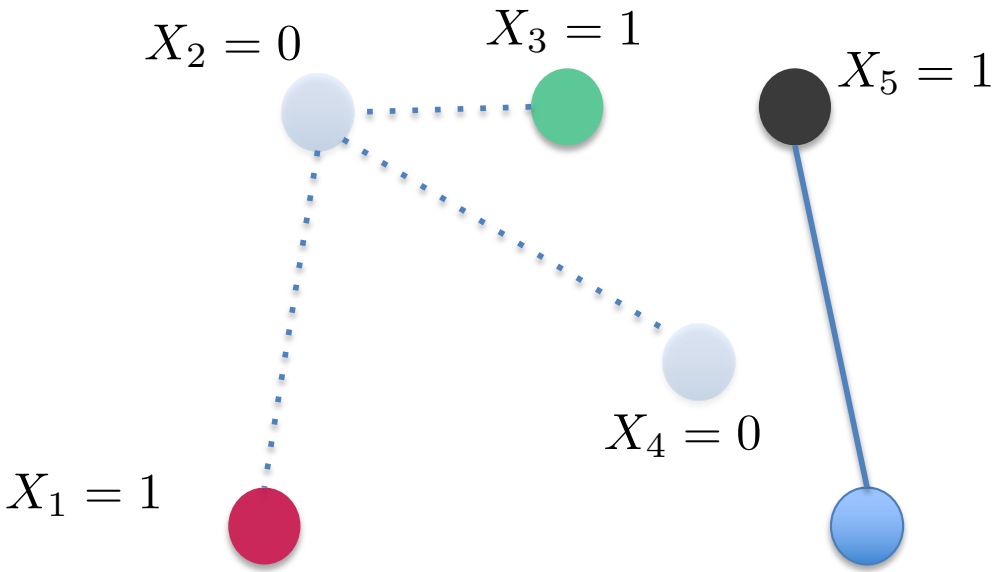
DFS with random coins



Algorithm:

- Flip a coin for each vertex DFS wants to visit
- If 1 visit, if 0 don't visit and delete with its edges

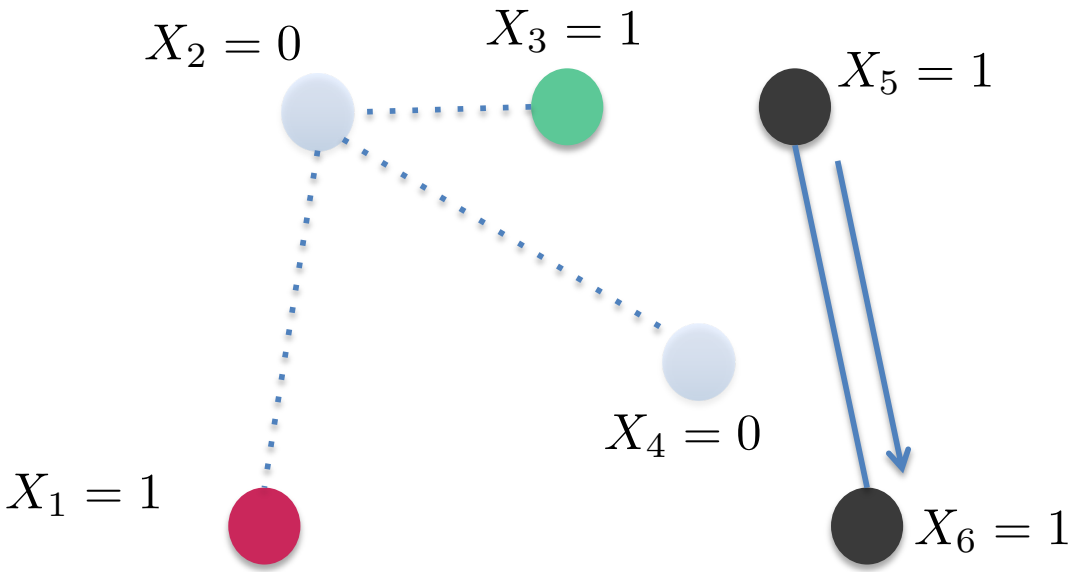
DFS with random coins



Algorithm:

- Flip a coin for each vertex DFS wants to visit
- If 1 visit, if 0 don't visit and delete with its edges

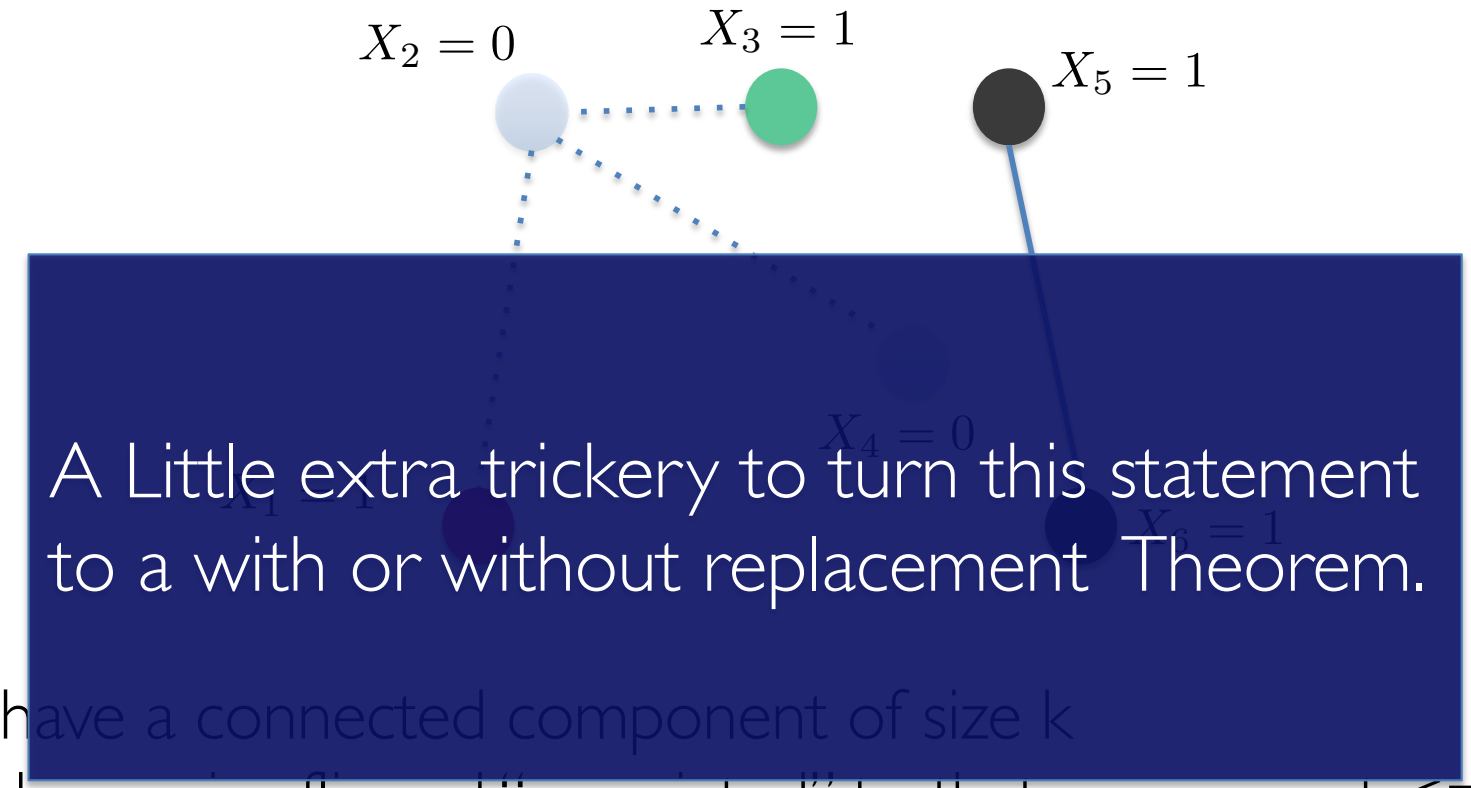
DFS with random coins



Algorithm:

- Flip a coin for each vertex DFS wants to visit
- If 1 visit, if 0 don't visit and delete with its edges

DFS with random coins



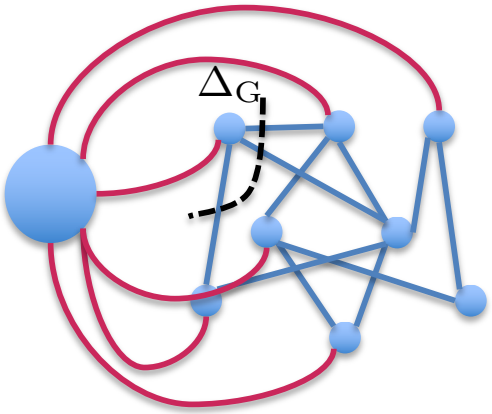
A Little extra trickery to turn this statement to a with or without replacement Theorem.

- Say I have a connected component of size k
- #random coins flipped "associated" to that component <= k *Δ
- Since I have a size k component it means that I had the event "at least k coins are "ON" in a set of k * Δ coins"

$$(n - kd + 1)Pr[B(kd, p) \geq k] < n \cdot e^{-\frac{\epsilon^2}{3}(1-\epsilon)k} < n \cdot e^{-\frac{\epsilon^2(1-\epsilon)}{3}\frac{4}{\epsilon^2}\ln n} :$$

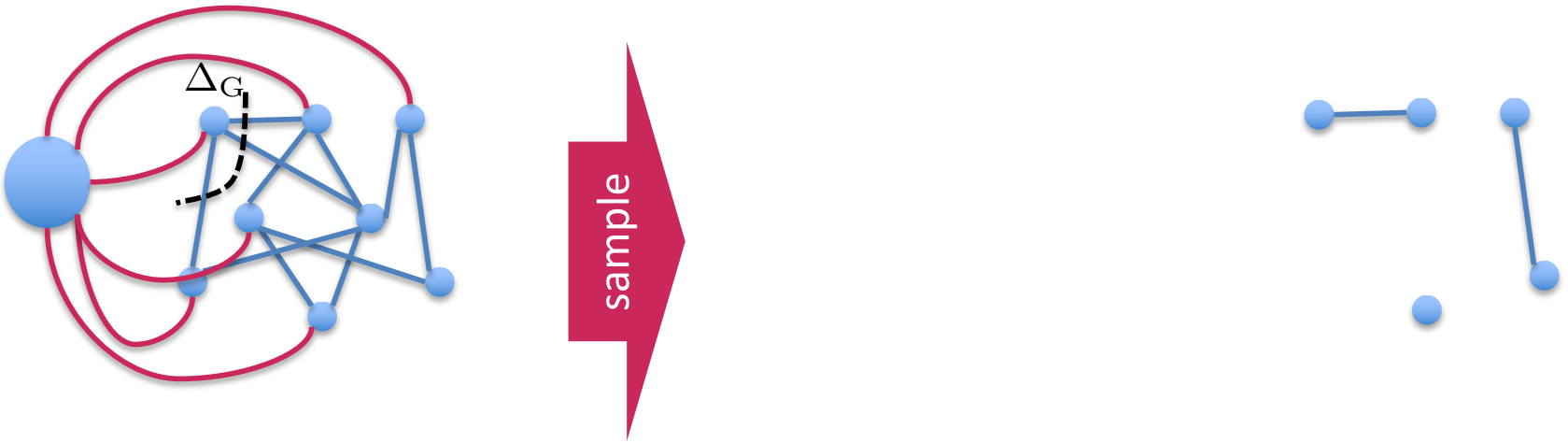
Is Max.Degree really an issue?

High Degree Vertices (outliers)



Say we have a graph with a low-degree component + some high degree vertices

High Degree Vertices (outliers)



Lemma:

If you sample uniformly less than $P \leq (1-\epsilon)\frac{n}{\Delta_G}$ vertices

Then, the induced subgraph of the low-degree (!) part shatters, and the largest connected component has size (whp)

$$\frac{4}{\epsilon^2} \cdot \log n$$

Δ_G is the max degree is the low-degree subgraph

High Degree Vertices (outliers)

Lemma 7. *Let us assume that there are $O(n^\delta)$ outlier vertices in the original conflict graph G with degree at most Δ_o, and let the remaining vertices have degree (induced on the remaining graph) at most Δ. Let the induced update-variable graph on these low degree vertices abide to the same graph assumptions as those of Theorem 4. Moreover, let the batch size be bounded as*

$$B \leq \min\left\{(1-\epsilon)\frac{n-O(n^\delta)}{\Delta},\ O\left(\frac{n^{1-\delta}}{P}\right)\right\}.$$

Then, the expected runtime of CYCLADES *will be* $O\left(\frac{E_u \cdot \kappa}{P} \cdot \log^2 n\right)$.

Experiments

Experiments

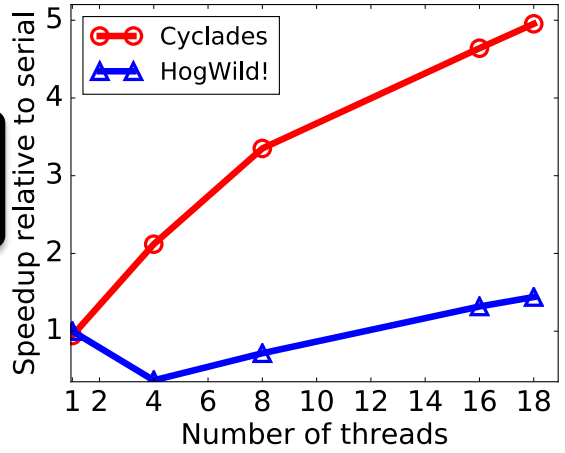
Implementation in **C++**
Experiments on **Intel Xeon CPU E7-8870 v3**
1TB RAM

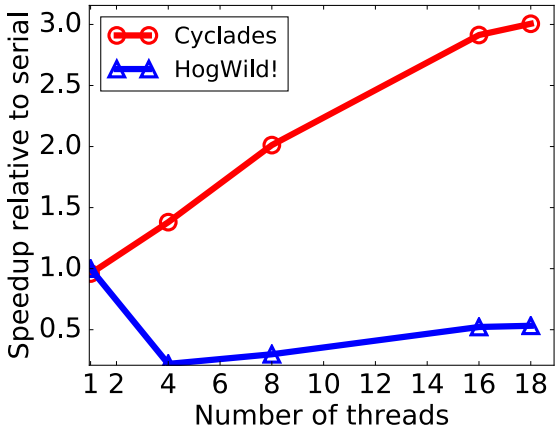
	Dataset	# datapoints	# features	Density (average number of features per datapoint)
SAGA	NH2010	48,838	48,838	4.8026
SVRG	DBLP	5,425,964	5,425,964	3.1880
L2-SGD	MovieLens	~10M	82,250	200
SGD	EN-Wiki	20,207,156	213,272	200

Full asynchronous (Hogwild!)
vs
CYCLADES

Speedups

SVRG/SAGA



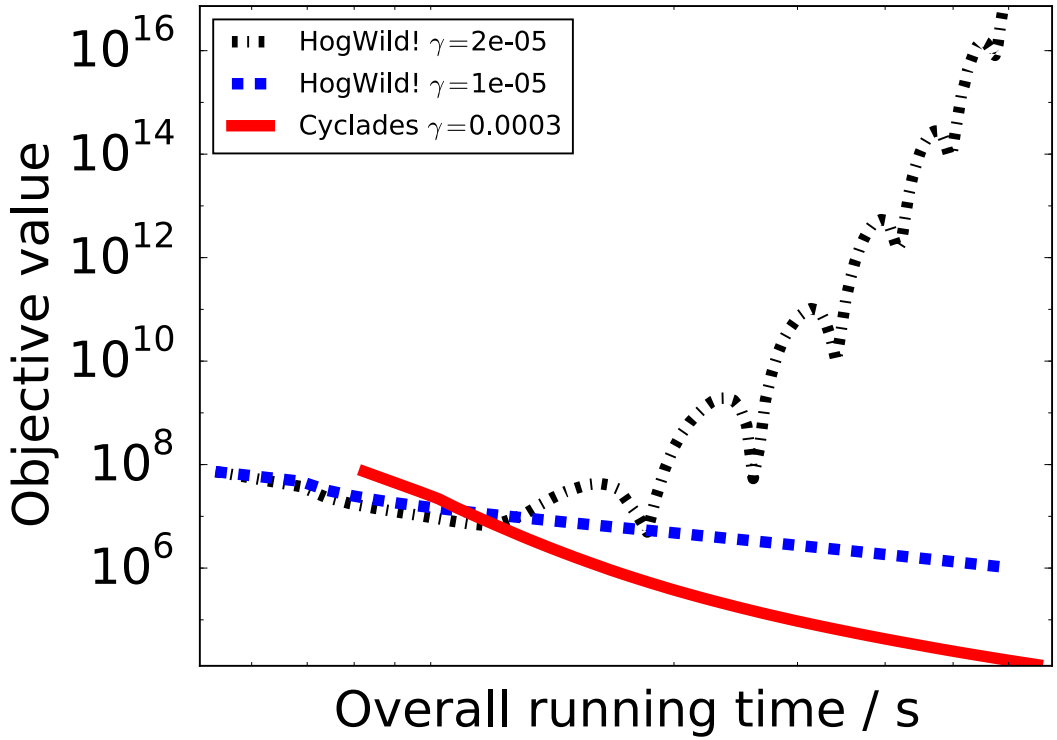


~ 500% gain

(a) Least squares, DBLP, SAGA (b) Graph Eig., NH2010, SVRG

Convergence

Least Squares SAGA
16 threads



Open Problems

Assumptions: Sparsity is Key

O.P. :
Can we handle Dense Data?

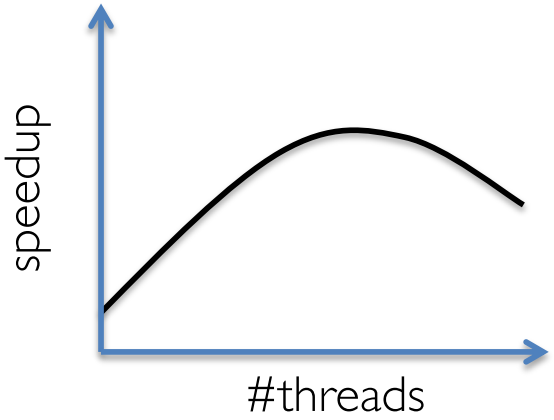
O.P. :
Data sparsification
for f(<a,x>) problems?

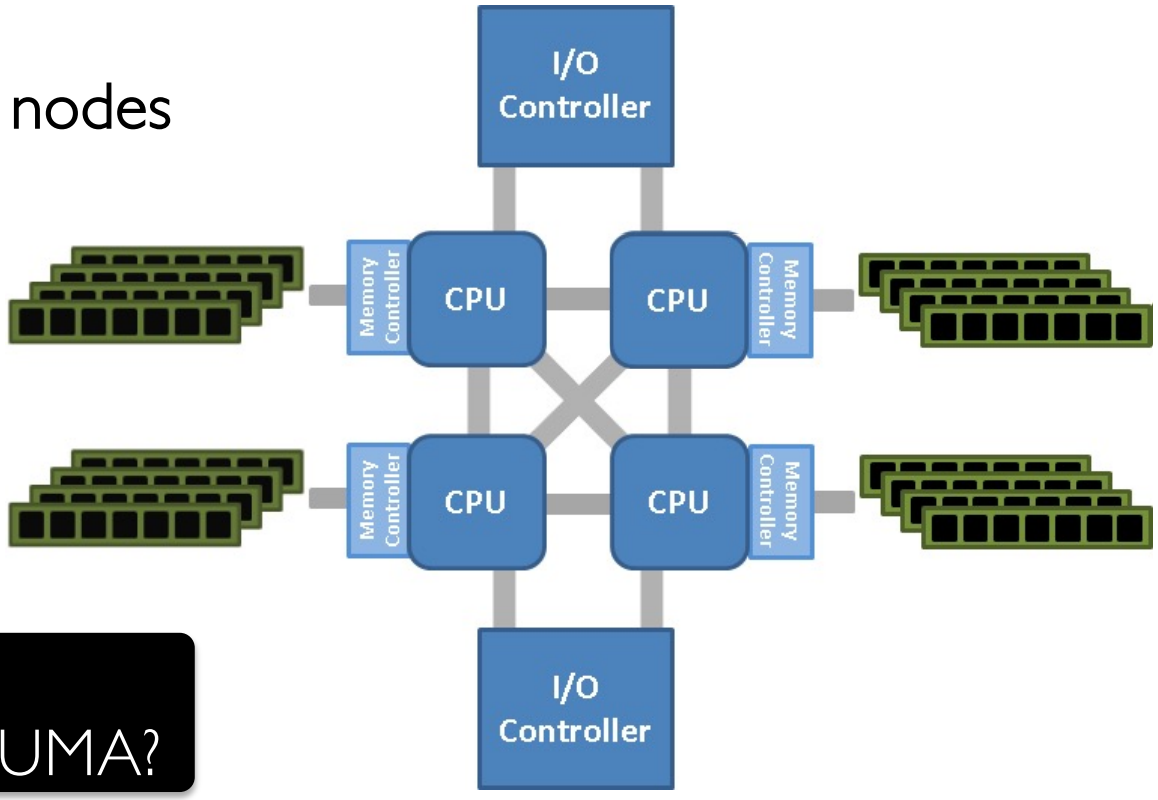
maybe…
We can relax serial equivalence to an "expected" one?

Open Problems

Asynchronous algorithms great for Shared Memory Systems

- Issues when scaling across nodes



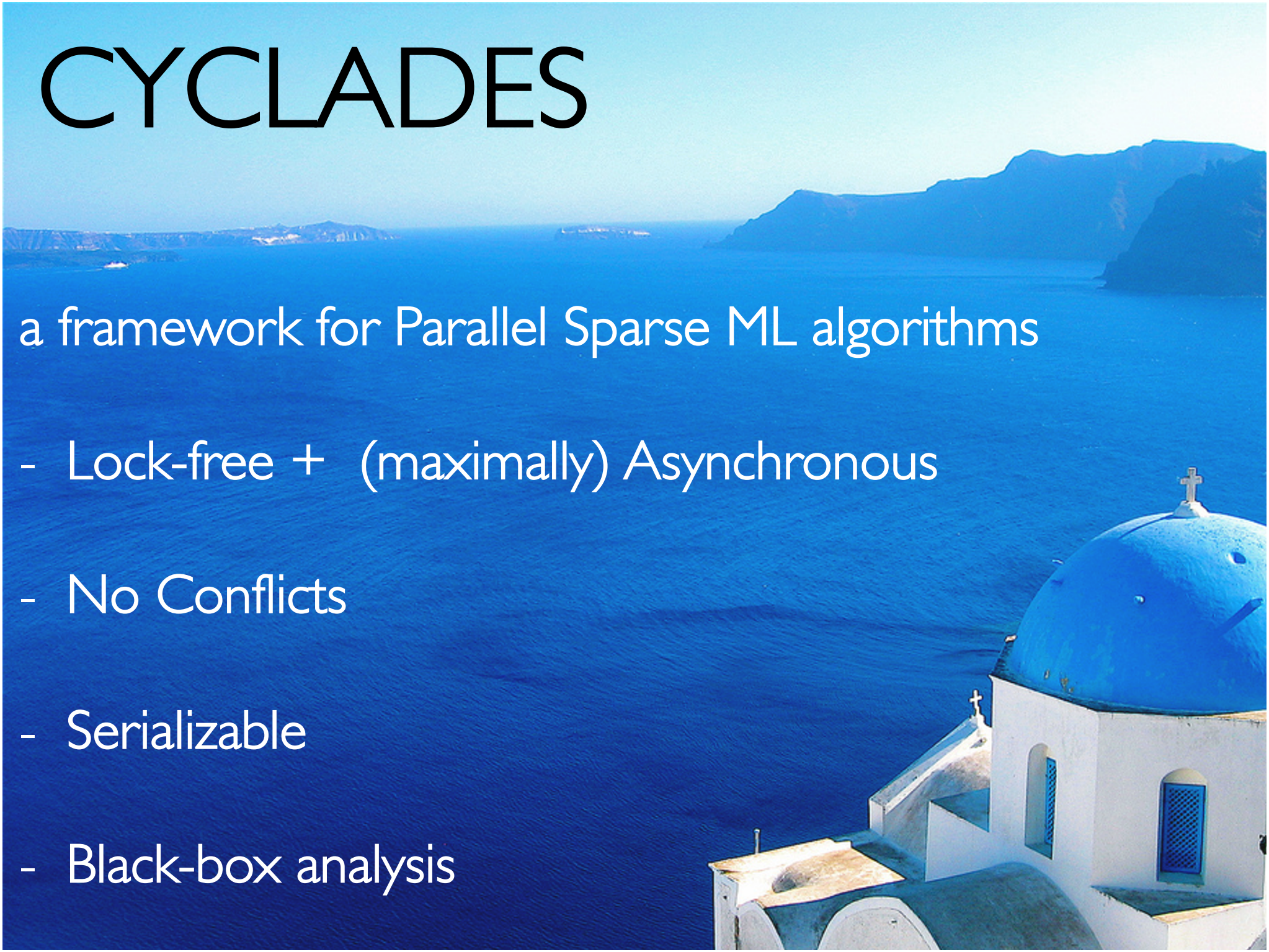


O.P. :
How to provably scale on NUMA?

- Similar Issues for Distributed:

O.P. :
What is the right ML Paradigm for Distributed?

CYCLADES



a framework for Parallel Sparse ML algorithms

- Lock-free + (maximally) Asynchronous
- No Conflicts
- Serializable
- Black-box analysis

Next Time

- Communication Bottlenecks
- Compressed Gradients
- Quantization

Reading List

- Krivelevich, M., 2014. The phase transition in site percolation on pseudo-random graphs. arXiv preprint arXiv:1404.5731.

- Pan, X., Lam, M., Tu, S., Papailiopoulos, D., Zhang, C., Jordan, M.I., Ramchandran, K. and Ré, C., 2016. Cyclades: Conflict-free asynchronous machine learning. Advances in Neural Information Processing Systems, 29.

- Pan, X., Papailiopoulos, D., Oymak, S., Recht, B., Ramchandran, K. and Jordan, M.I., 2015. Parallel correlation clustering on big graphs. Advances in Neural Information Processing Systems, 28.