Parallelization Primitives for Dynamic Sparse Computations

June 24, 2013

Tsung-Han Lin, Steve Tarsa, and H.T. Kung

HotPar'13

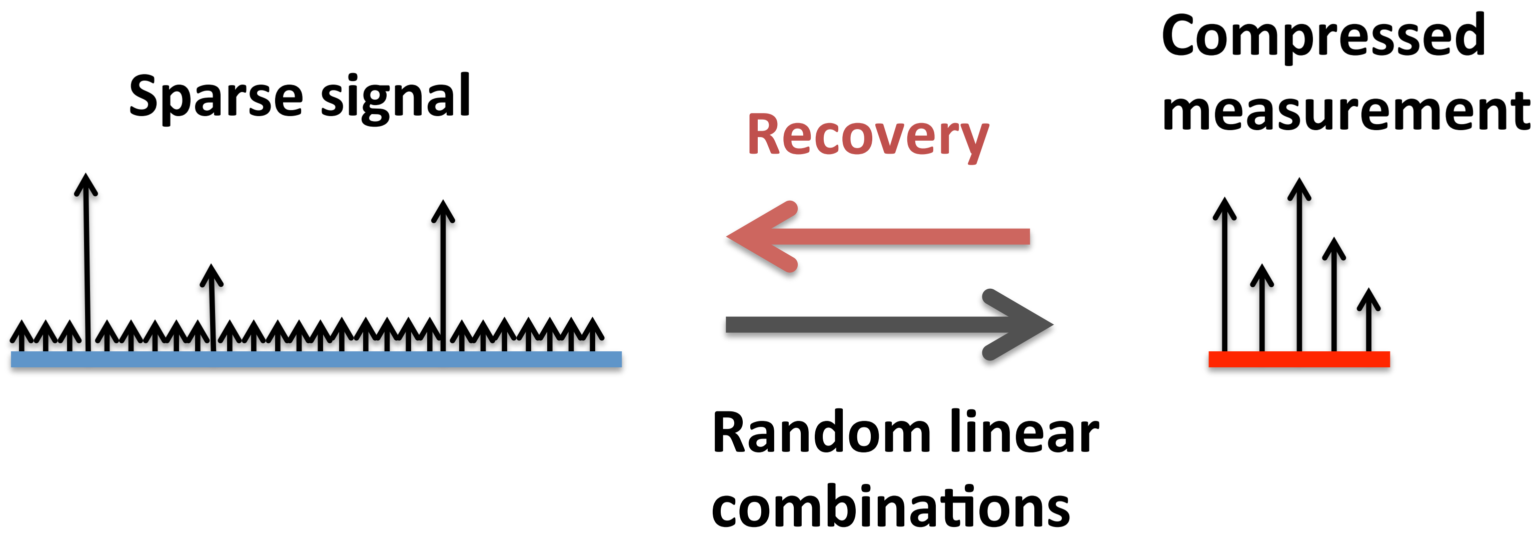
HARVARD
School of Engineering
and Applied Sciences

Dynamic Sparse Computations

- Problem: we want to parallelize sparse computations where the nonzero variables are identified only **at run time**
- Important for many applications in *signal processing* and *machine learning*

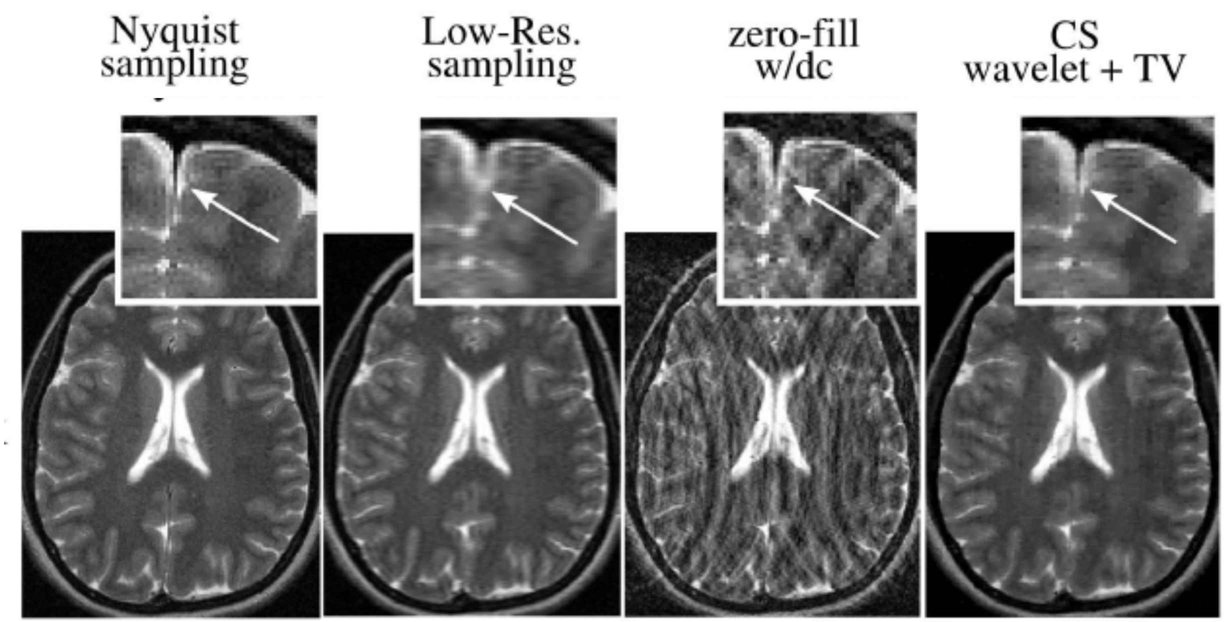
Dynamic Sparse Computations
Example 1: Compressive Sensing Recovery

- **Recover** sparse signals from dense compressed measurements



Compressive Sensing Recovery Example: Dynamic MRI

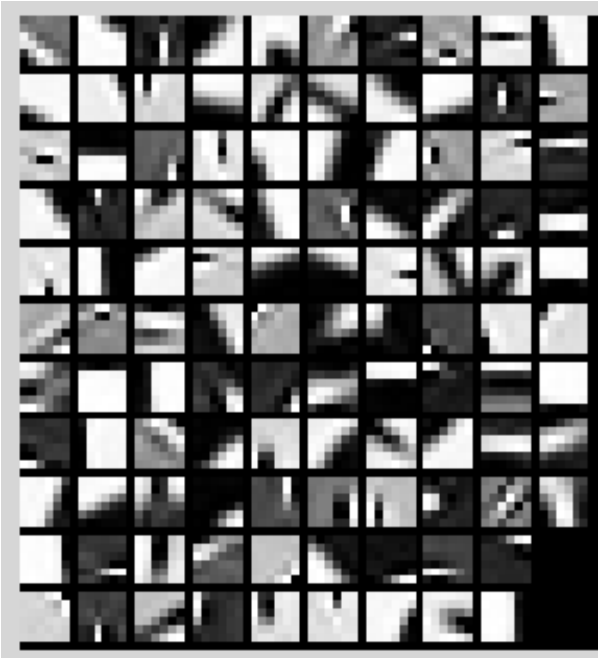
- 3D signal to be recovered is large
  - E.g., with 40 100x100 MRI time frames; signal size is 400K



[Lustig 2008]

Dynamic Sparse Computations Example 2: Sparse Coding

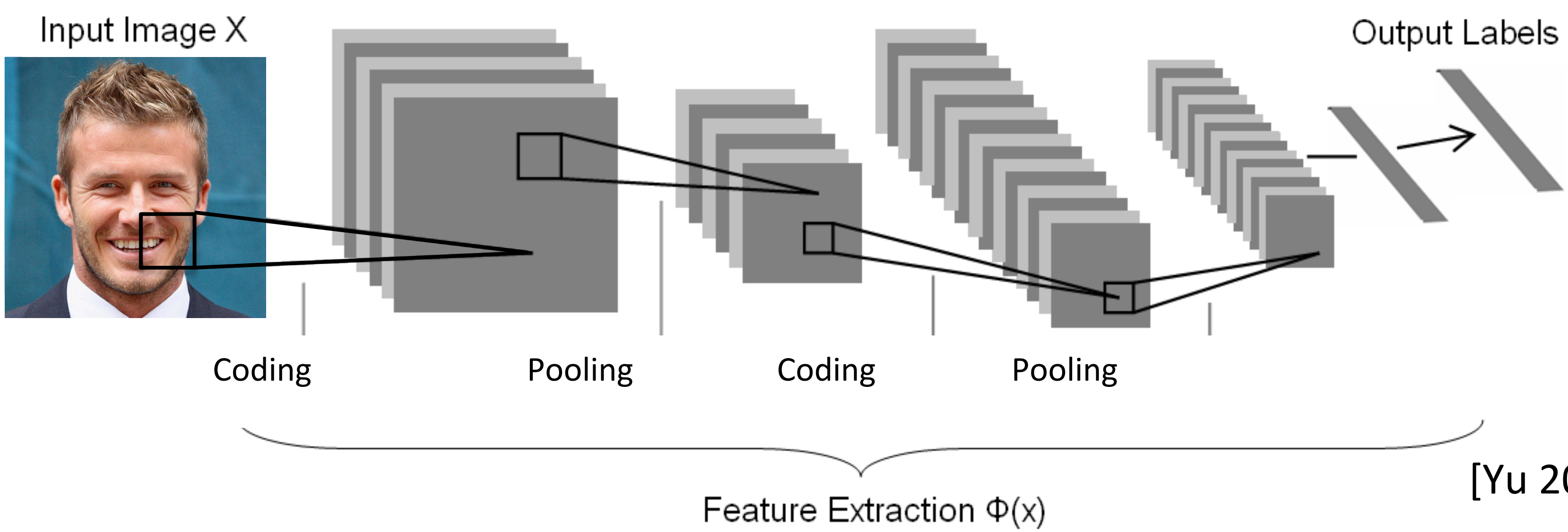
- **Extract** sparse representations based on predefined/learned dictionaries



Representing image patches as linear combinations of a few low-level features

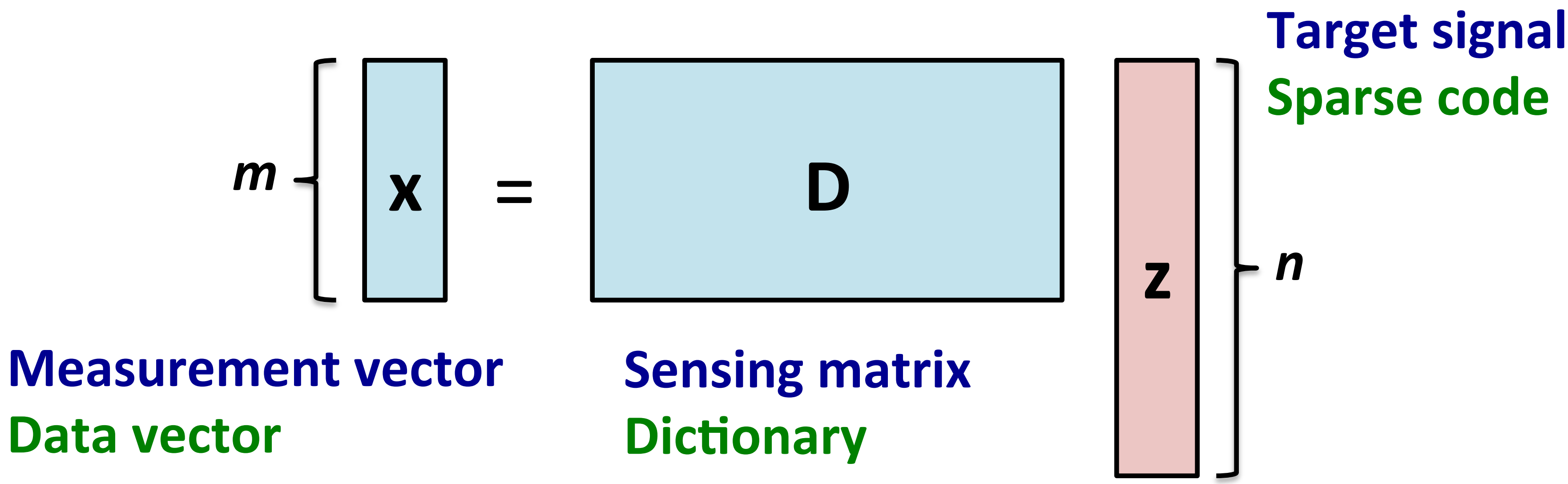
Sparse Coding Example: Convolutional Neural Networks

- Learning high-level features requires huge datasets for training
  - E.g., Google trains face detector using 10 million 200x200 images



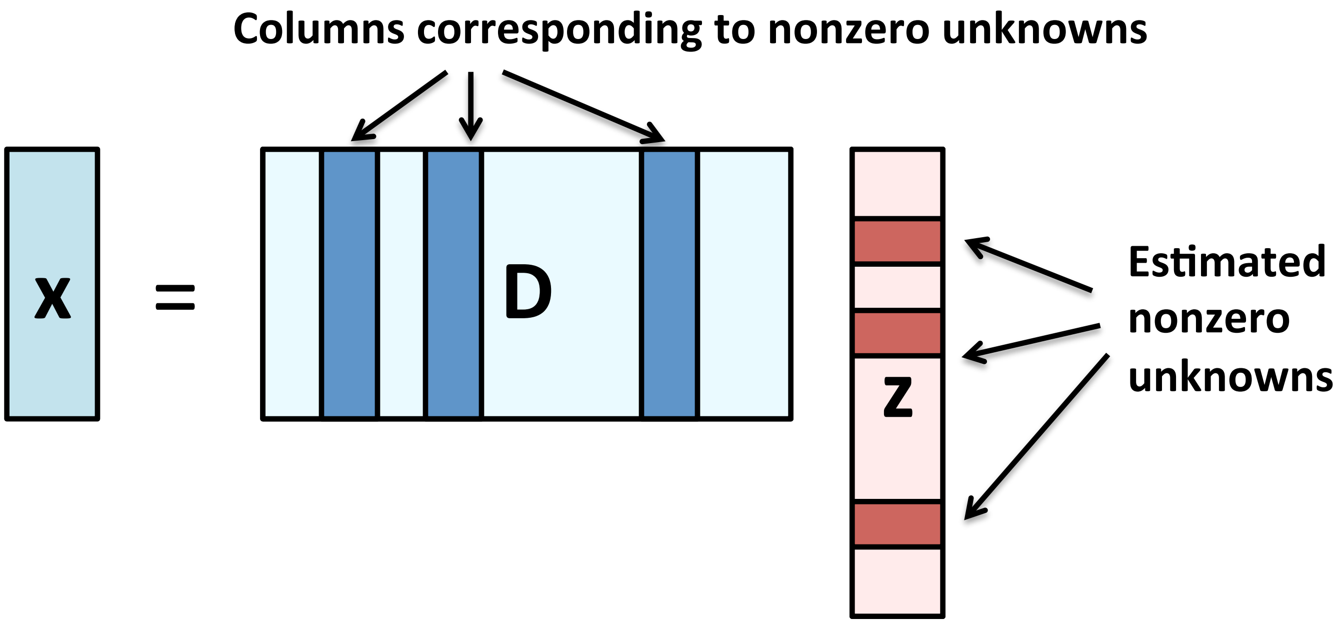
[Yu 2012]

A Canonical Example of Dynamic Sparse Computation: Solving Under-constrained Linear Systems



- Given **x** and **D**, infinitely many solutions for **z**
- Suppose **D** is well-behaved (e.g., satisfies RIP), we can recover the correct **z** by minimizing $\|\mathbf{z}\|_1$
- Efficient iterative algorithms, such as orthogonal matching pursuit (OMP), are available for recovering **z**

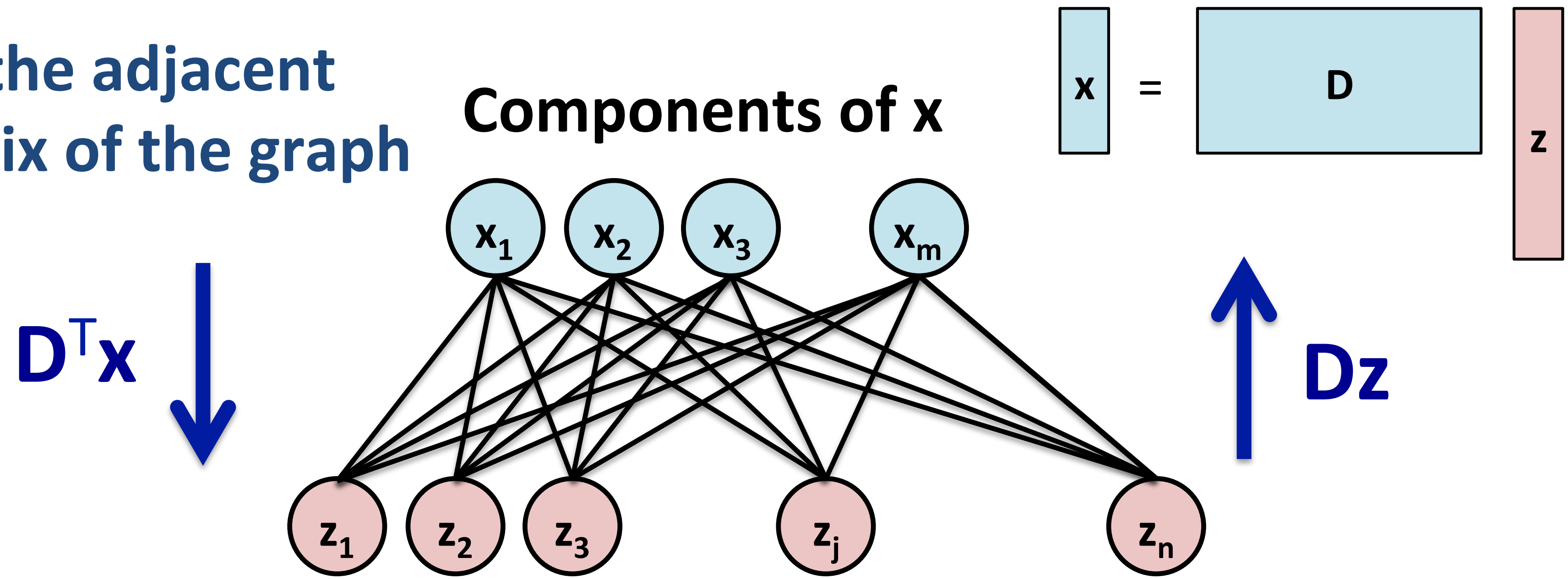
OMP for Sparse Recovery



- OMP is an iterative algorithm, which is greedy, simple, fast
- It iteratively refines the sparse solution vector
  - Outer loop **identifies** nonzero unknowns and reduces the problem to be over-constrained
  - Inner loop **estimates** values by solving the over-constrained system via, e.g., least squares

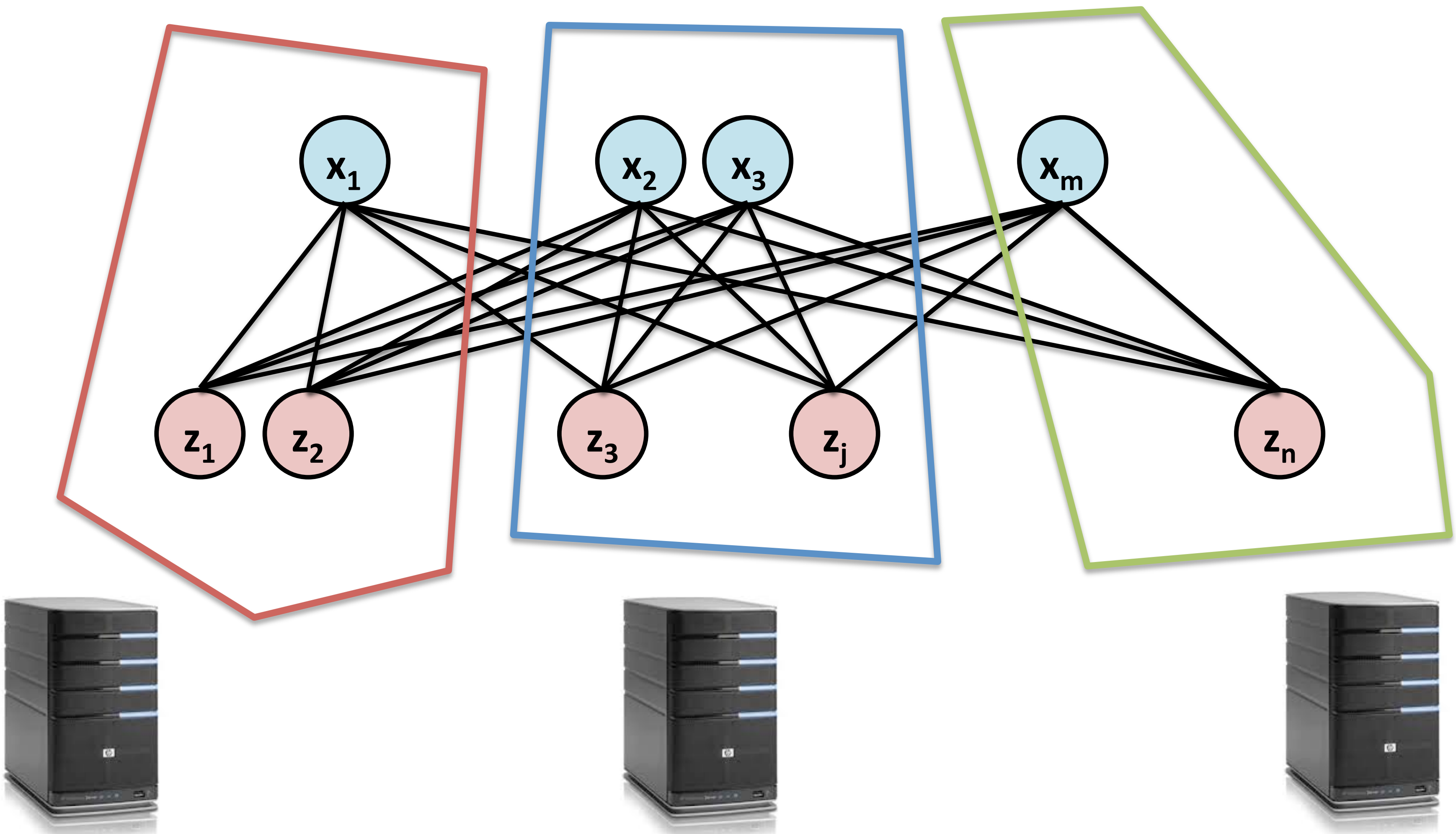
Parallelizing OMP: Ping-Ponging on a Bipartite Graph

D is the adjacent matrix of the graph

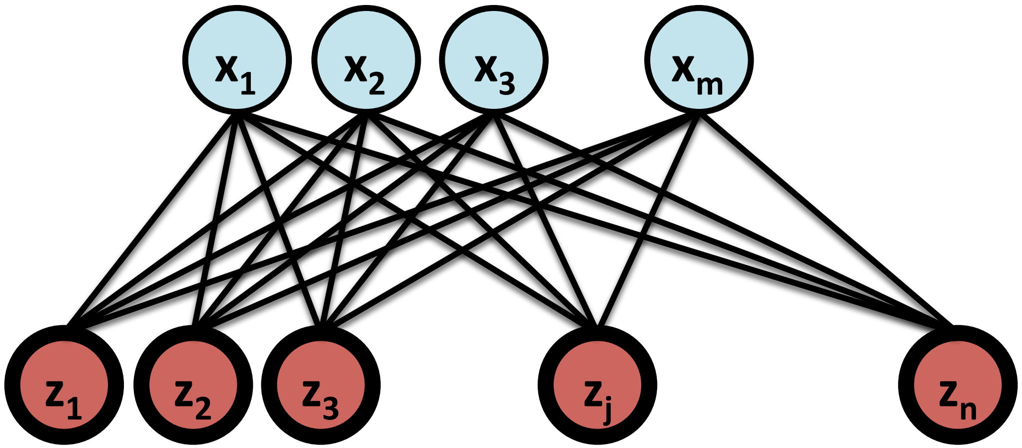


Components of z

Parallelizing OMP: Splitting Graph to Multiple Machines

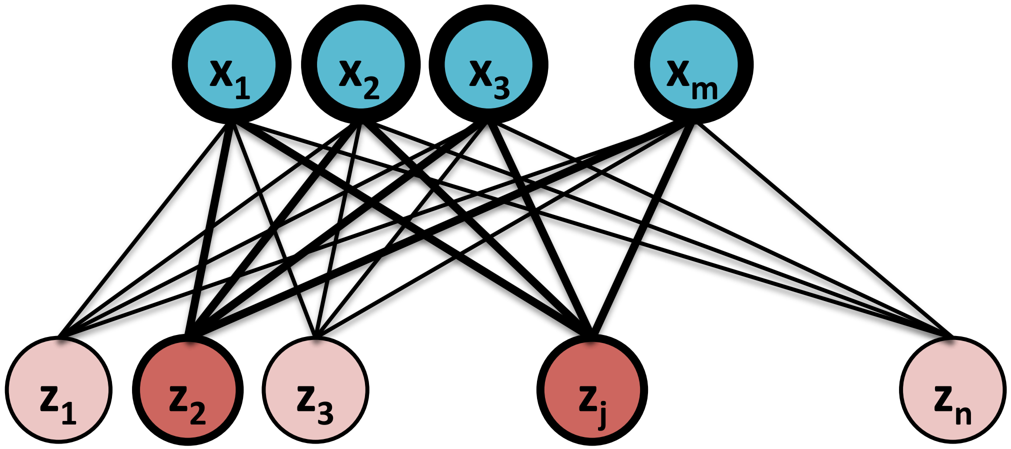


Ping-Ponging in Outer Loop



Compute a “score” for every z_i (highlighted) to identify nonzeros

Ping-Ponging in Inner Loop



Select the nonzeros and compute their values using the corresponding subgraph (highlighted)

Two **Parallelization Primitives** for efficient Parallel Execution of Dynamic Sparse Computation

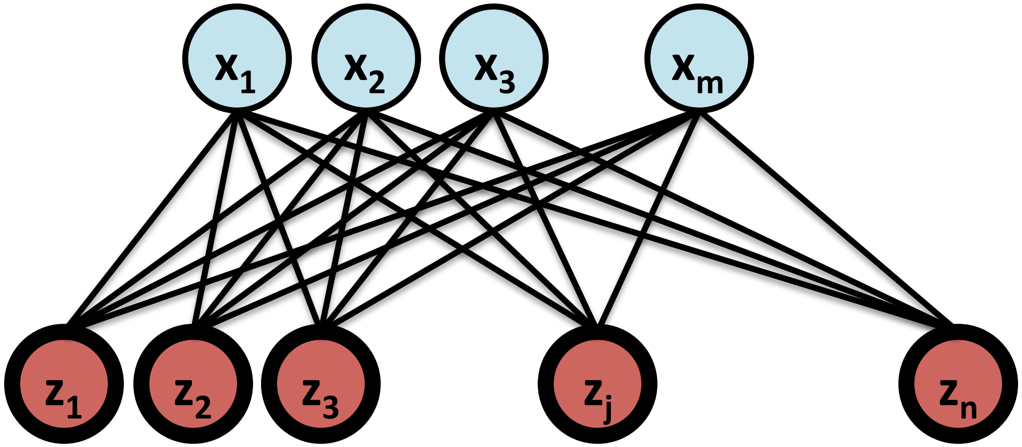
Challenge: For efficiency we must limit computation only to the subgraph corresponding to nonzero unknowns, but we don't know them at the outset; they are determined at run time

We propose the following two primitives for the efficient identification of these nonzeros in parallel:

- **Statistical barrier** to identify nonzero unknowns without having to wait for stragglers
- **Selective push-pull** to focus computations only on the selected subgraph

Statistical Barrier

- Continue computation without waiting for the last straggler

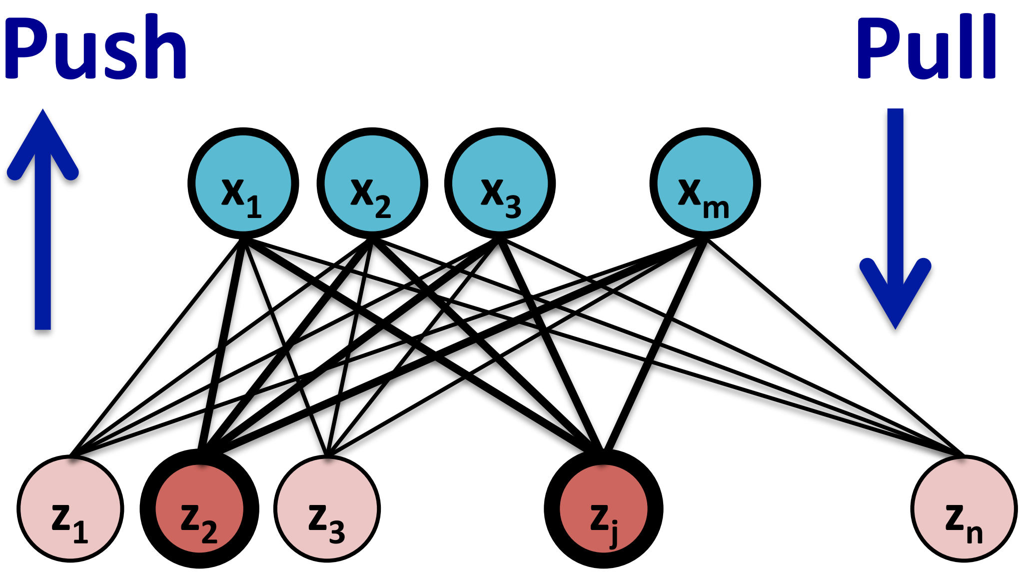


- Finishing a large fraction of z_i is likely to capture sparse nonzeros
- Algorithm is robust to missing values, which can be fixed in the next iteration

E.g. Leave the barrier when 80% of z_i complete

Selective Push-Pull

- Support computation on dynamically selected subgraph



No edge selection needed!

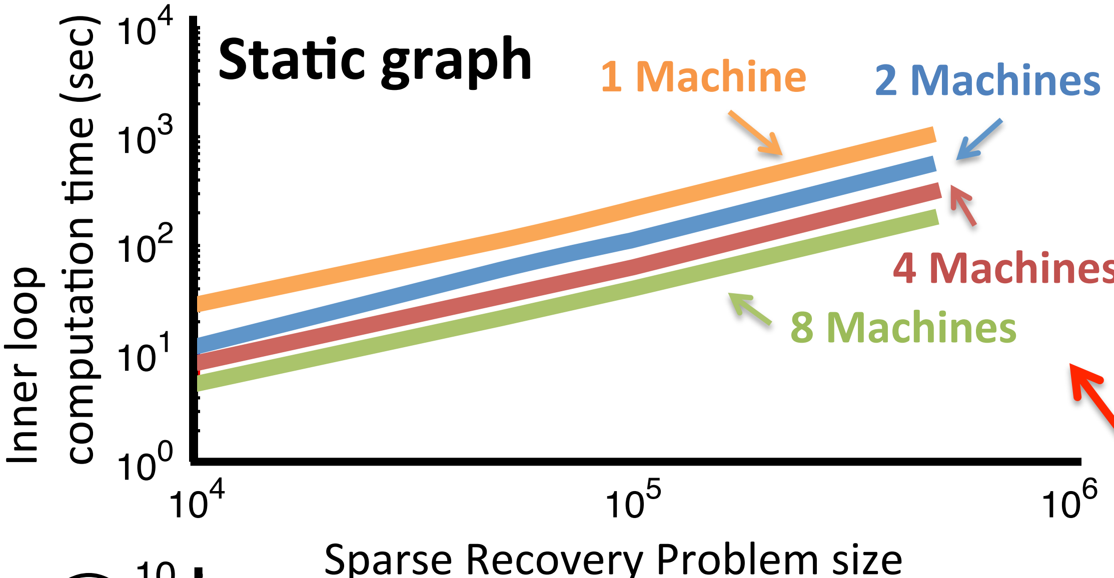
(1) Select a subset of active z_i

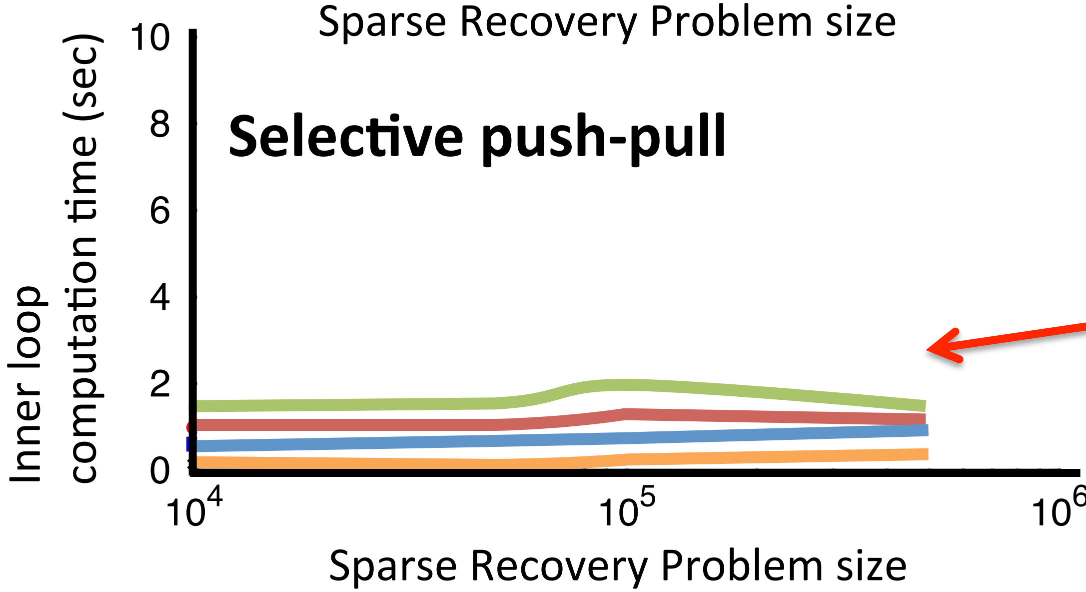
(2) z_i compute and **push** update to x_i

(3) x_i compute using incoming updates

(4) z_i **pull** updates and continue computation

Performance Gains of Selective Push-Pull on EC2





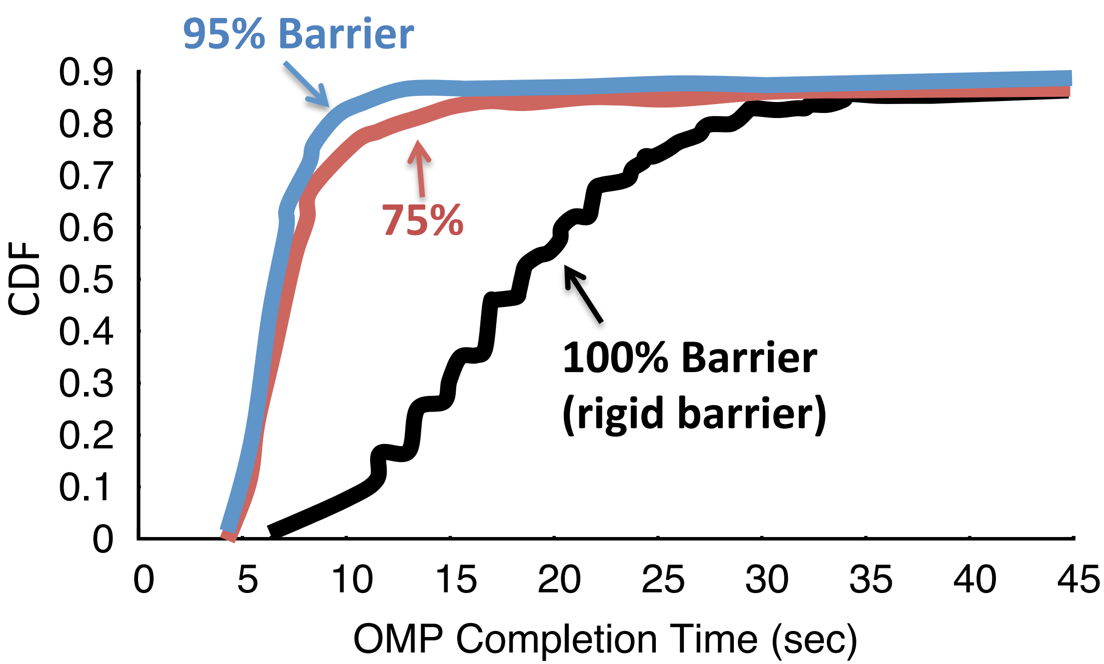
- **Built selective push-pull on GraphLab**
- **Vary sparse recovery problem size but with constant sparsity**

Computation time grows due to unnecessary computation

Computation time remains constant for constant subgraph size

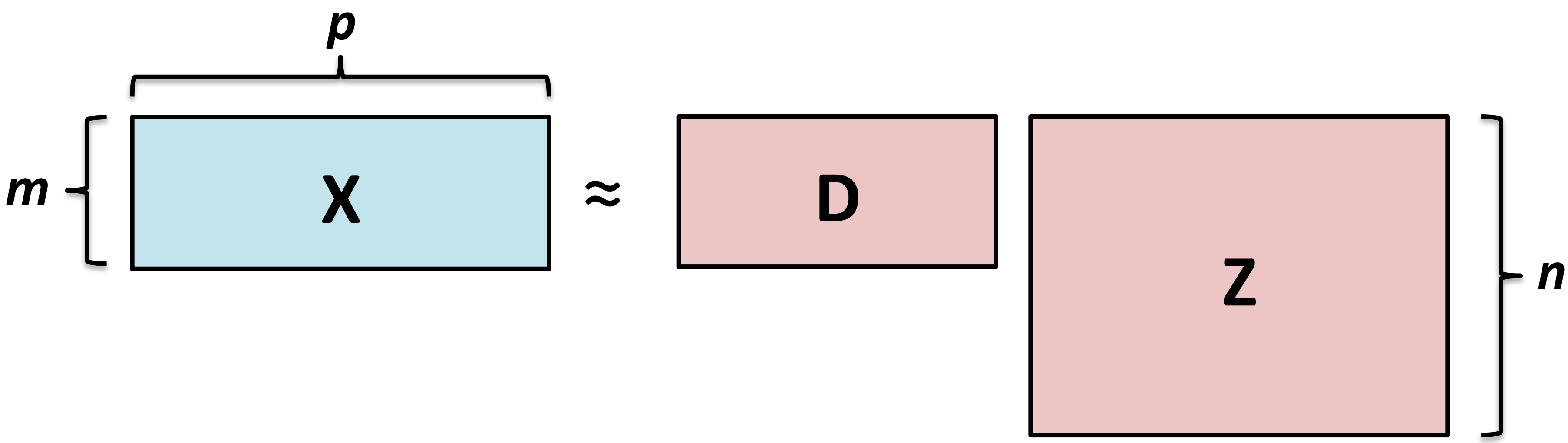
Performance Gains of Statistical Barrier in Simulation

Straggler stats from MS's Bing cluster: 25% jobs see high prop. of stragglers, up to 10x median completion time



- **95% Barrier trims worst stragglers → improves average time by 2.5x, and worst case by 4x over rigid**
- **75% is too aggressive, and extra iterations hurt**

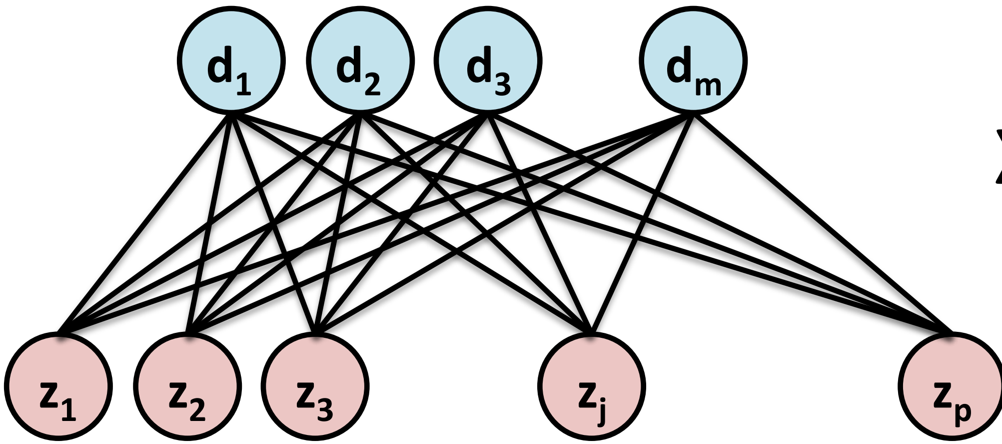
Parallel Ping-Pong Applicable to Other Applications, e.g., Dictionary Learning for Feature Extraction



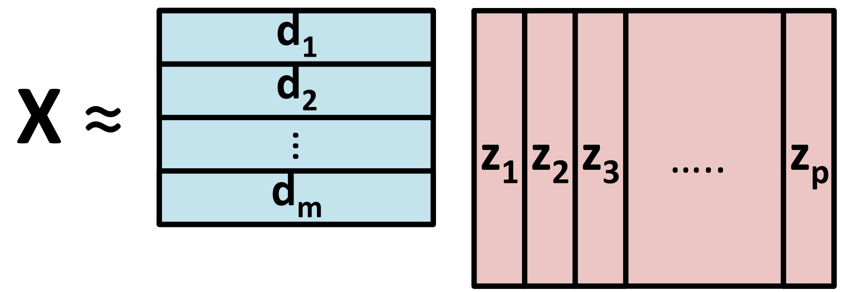
- Learn an **overcomplete** dictionary that can represent data vectors using only a few atoms
- **K-SVD** alternates between optimizing **Z** and **D**

Express Dictionary Learning Using Graphical Model

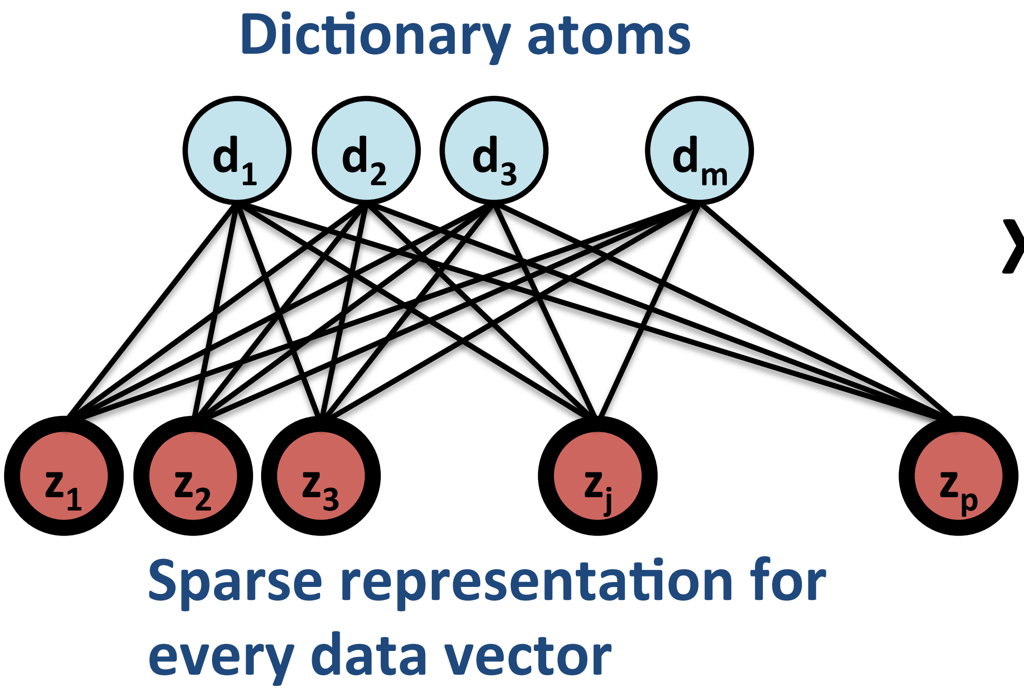
Dictionary atoms

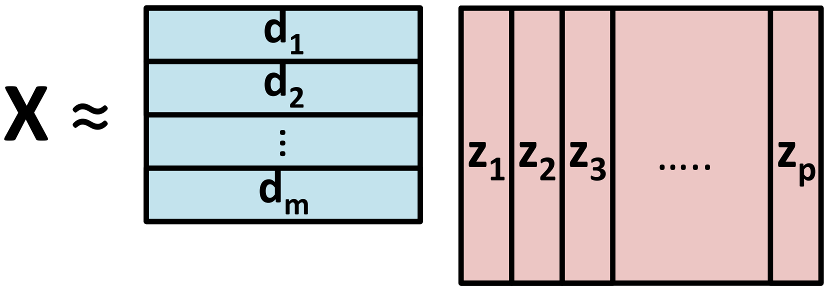


Sparse representation for every data vector



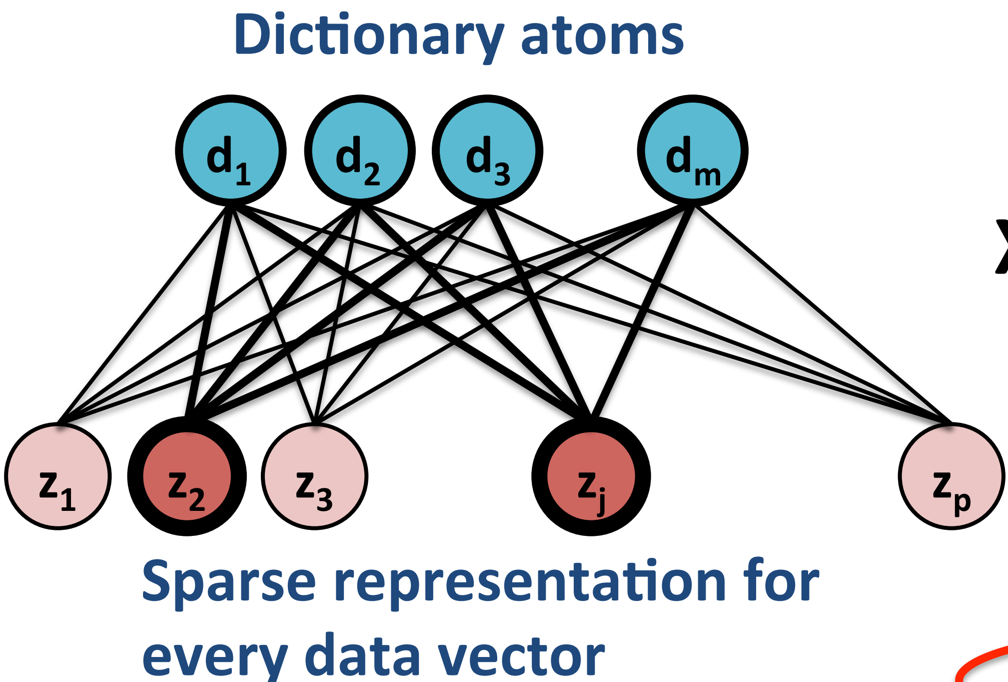
Express Dictionary Learning Using Graphical Model

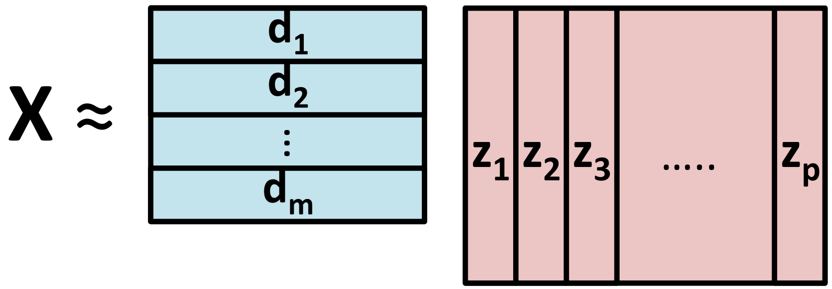




- **Update Z**: compute sparse code for every data vector
- **Update D**: for a given atom, optimize using associated data vectors (has nonzero coefficient for the atom)

Express Dictionary Learning Using Graphical Model





Statistical barrier **to skip stragglers**

- **Update Z**: compute sparse code for every data vector
- **Update D**: for a given atom, optimize using associated data vectors (has nonzero coefficient for the atom)

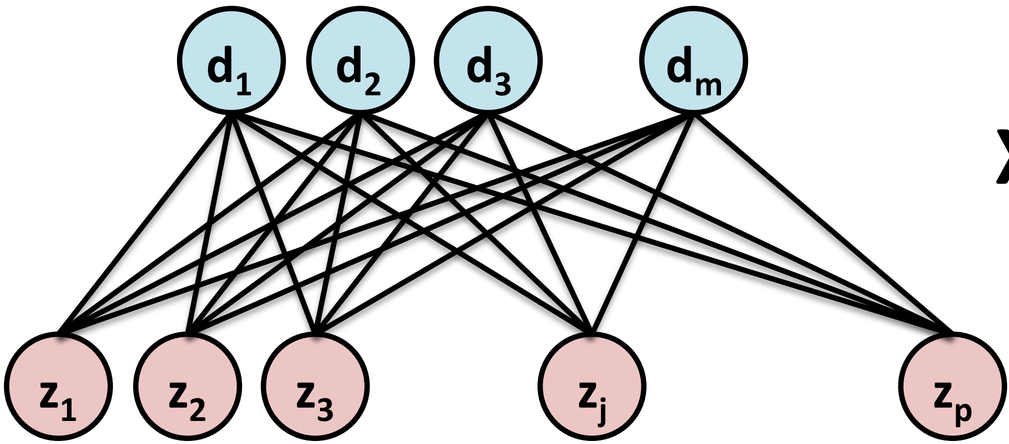
Selective push-pull **to activate only associated data vertices**

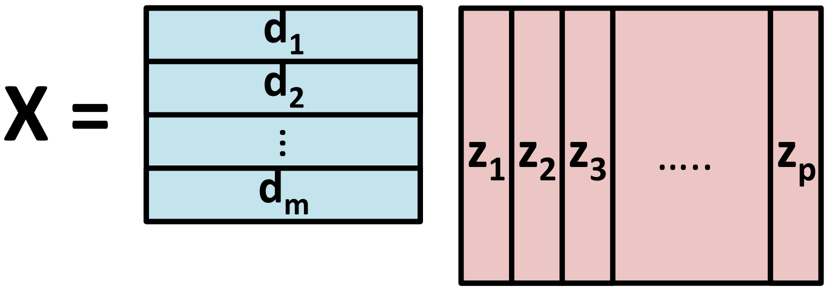
Conclusion

- We have identified an important class of dynamic sparse computations on bipartite graphs
- This class of computations can benefit from a flexible execution model supported by two new primitives
  - Statistical barrier
  - Selective push-pull
- There are important applications for machine learning and signal processing

Express Dictionary Learning Using Graphical Model

Dictionary atoms



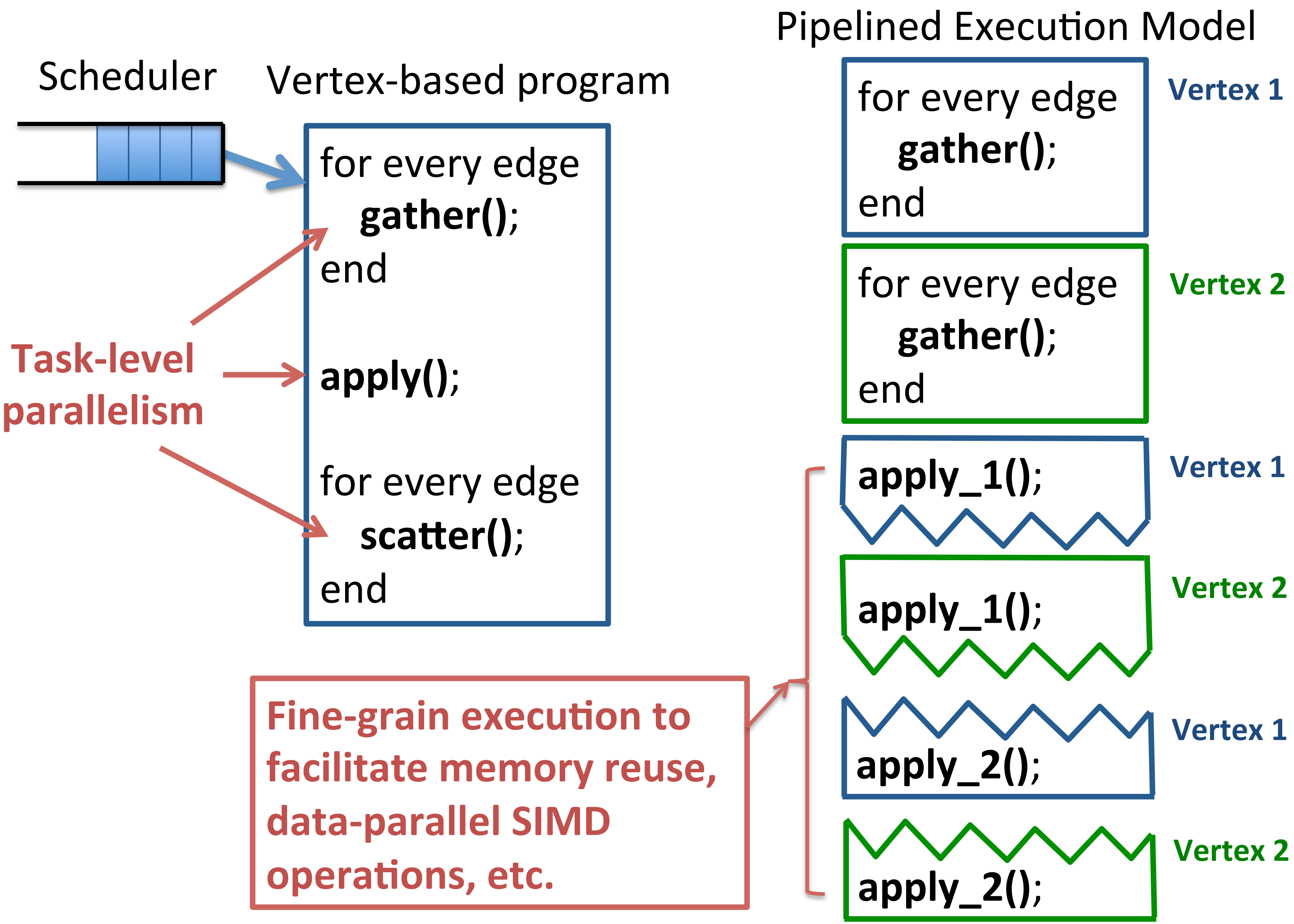


Sparse representation for every data vector

Statistical barrier **to skip stragglers**

- **Update z**: compute sparse code for every data vector
- **Update d**: for a given atom, optimize using associated data vectors (has nonzero coefficient for the atom)

Selective push-pull **to activate only associated data vertices**


Pipelined Execution Model
Scheduler
Vertex-based program
for every edge
gather();
end
apply();
for every edge
scatter();
end
Task-level parallelism
for every edge
gather();
end
Vertex 1
for every edge
gather();
end
Vertex 2
apply_1();
Vertex 1
apply_1();
Vertex 2
apply_2();
Vertex 1
apply_2();
Vertex 2
Fine-grain execution to facilitate memory reuse, data-parallel SIMD operations, etc.