Parallel/Distributed Optimization

http://bicmr.pku.edu.cn/~wenzw/bigdata2018.html

Acknowledgement: this slides is based on Prof. Wotao Yin's, Prof. James and Demmel's Dimitris Papailiopoulos's lecture notes as well as the HOGWILD! and CYCLADES paper

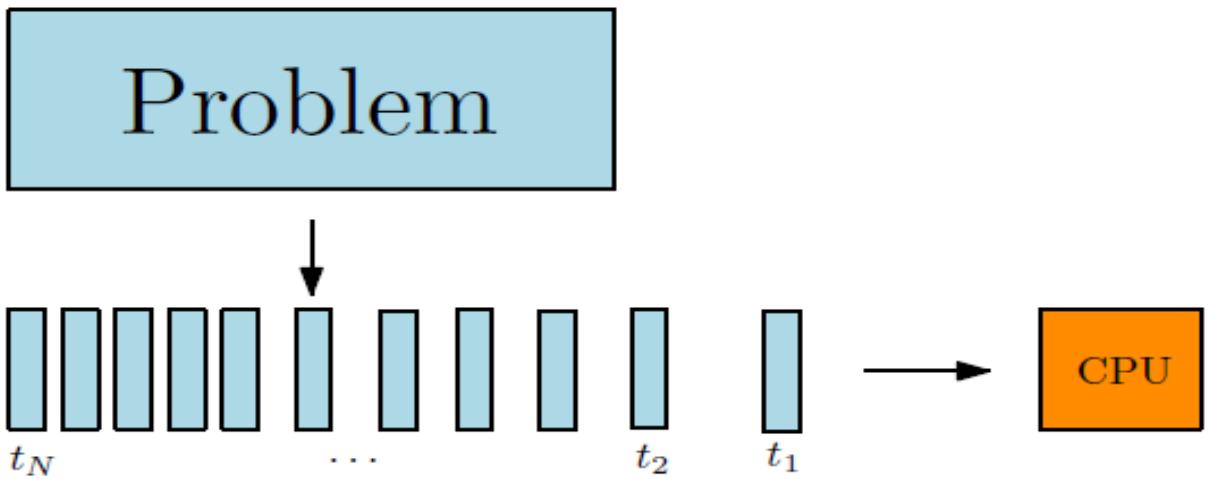
Thanks Haoyang Liu for preparing the part on HOGWILD! and CYCLADES

Outline

Serial computing

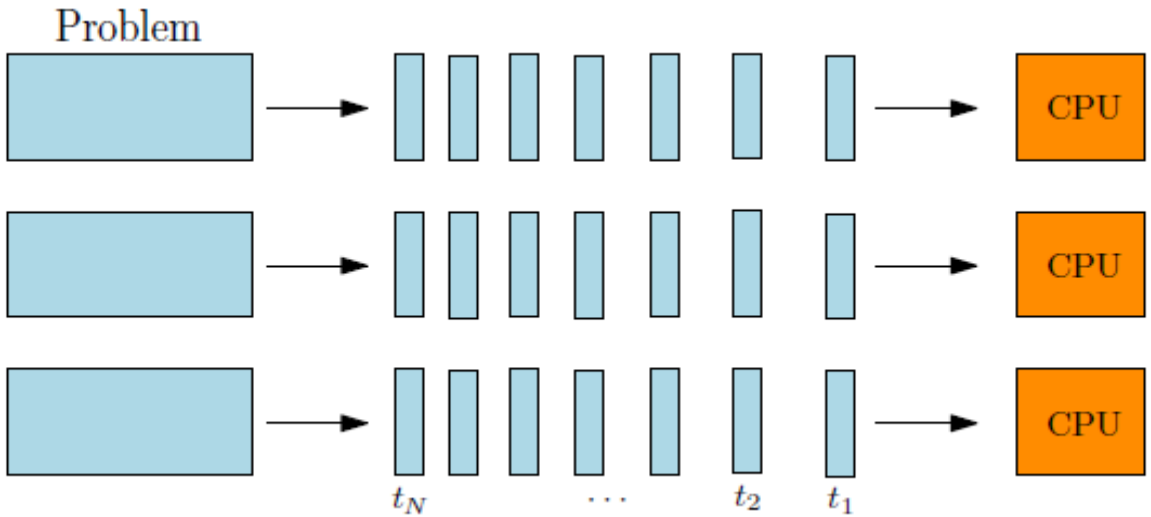
Traditional computing is **serial**

- a single CPU, one instruction is executed at a time
- a problem is broken into a series of instructions, executed one after another



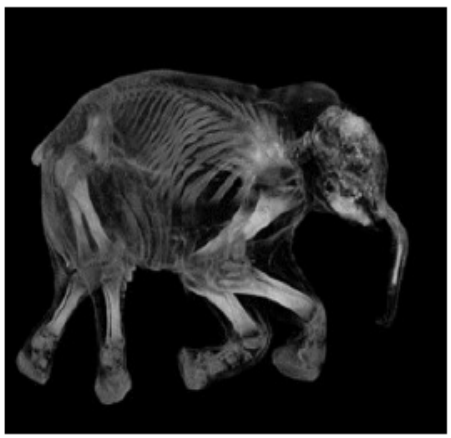
Parallel computing

- a problem is broken into concurrent parts, executed simultaneously, coordinated by a controller
- uses multiple cores/CPUs/networked computers, or their combinations
- expected to solve a problem in less time, or a larger problem



Commercial applications

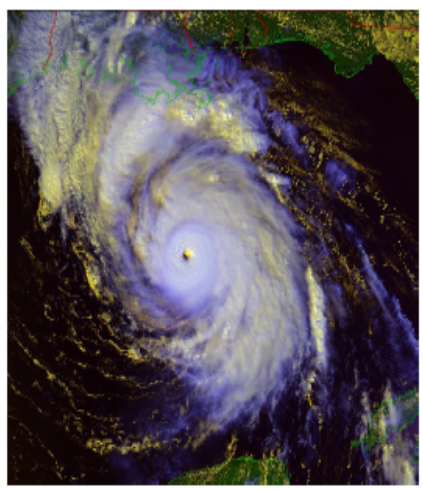
Examples: internet data mining, logistics/supply chain, finance, online advertisement, recommendation system, sales data analysis, virtual reality, computer graphics (cartoon movies), medicine design, medical imaging, network video, ...



Applications in science and engineering

- **simulation**: many real-world events happen concurrently, interrelated
  examples: galaxy, ocean, weather, traffic, assembly line, queues
- **use in science/engineering**: environment, physics, bioscience, chemistry, geoscience, mechanical engineering, mathematics (computerized proofs), defense, intelligence





Benefits of being parallel

- break single-CPU limits
- save time
- process big data
- access non-local resource
- handle concurrent data streams / enable collaboration

Parallel overhead

- **computing overhead**: start up time, synchronization wait time, data communication time, termination (data collection time)
- **I/O overhead**: slow read and write of non-local data
- **algorithm overhead**: extra variables, data duplication
- **coding overhead**: language, library, operating system, debug, maintenance

Different parallel computers

- basic element is like a single computer, a 70-year old architecture components: CPU (control/arithmetic units), memory, input/output
- **SIMD**: single instruction multiple data, GPU-like applications: image filtering, PDE on grid, ...
- **MISD**: multiple instruction single data, rarely seen conceivable applications: cryptography attack, global optimization
- **MIMD**: multiple instruction multiple data most of today's supercomputer, clusters, clouds, multi-core PCs

Tech jargons

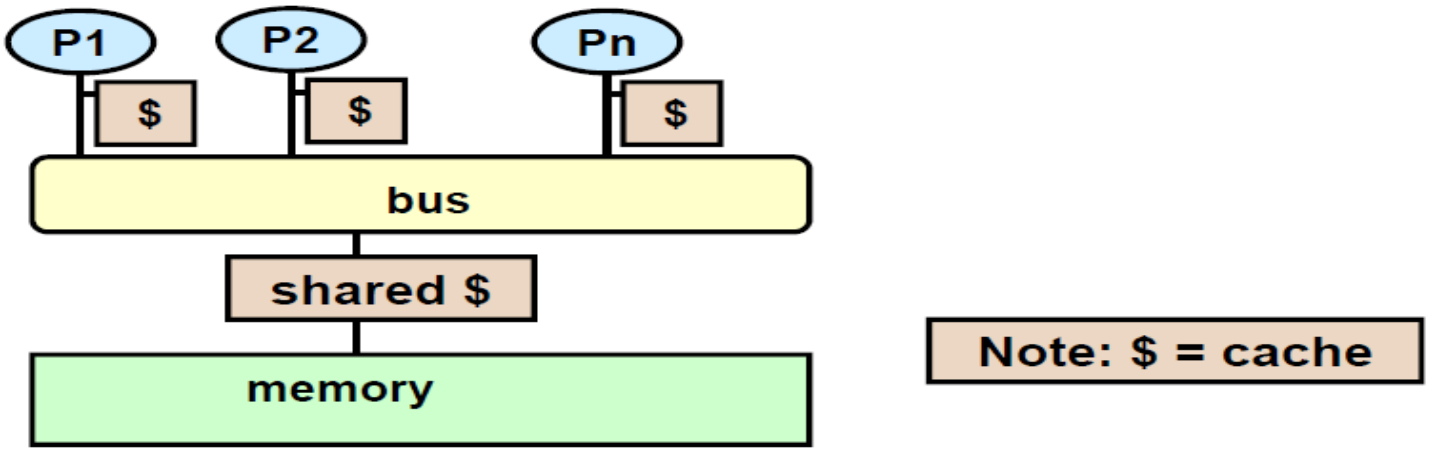
- HPC: high-performance computing
- node/CPU/processor/core
- **shared memory**: either all processors access a common physical memory, or parallel tasks have direct address of logical memory
- SMP: multiple processors share a single memory, shared-memory computing
- **distributed memory**: parallel tasks see local memory and use communication to access remote memory
- communication: exchange of data or synchronization controls; either through shared memory or network
- **synchronization**: coordination, one processor awaits others

Tech jargons

- granularity: coarse — more work between communication events; fine — less work between communication events
- Embarrassingly parallel: main tasks are independent, little need of coordination
- **speed scalability**: how speedup can increases with additional resources. **data scalability**: how problem size can increase with additional resources. **scalability affected by**: problem model, algorithm, memory bandwidth, network speed, parallel overhead. sometimes, adding more processors/memory may not save time!
- **memory architectures**: uniform or non-uniform shared, distributed, hybrid
- **parallel models**: shared memory, message passing, data parallel, SPMD (single program multiple data), MPMD, etc.

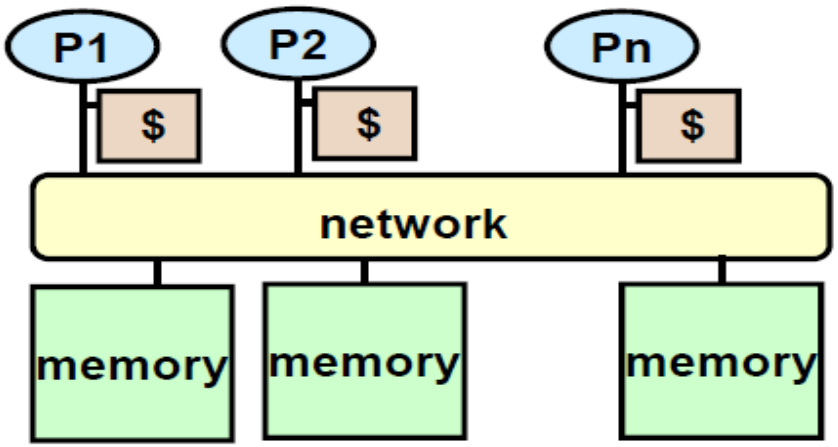
Machine Model: Shared Memory

- Processors all connected to a large shared memory.
  - Typically called Symmetric Multiprocessors (SMPs)
  - SGI, Sun, HP, Intel, IBM SMPs
  - Multicore chips, except that all caches are shared
- Advantage: uniform memory access (UMA)
- Cost: much cheaper to access data in cache than main memory
- Difficulty scaling to large numbers of processors
  $\leq$ 32 processors typical



Machine Model: Distributed Shared Memory

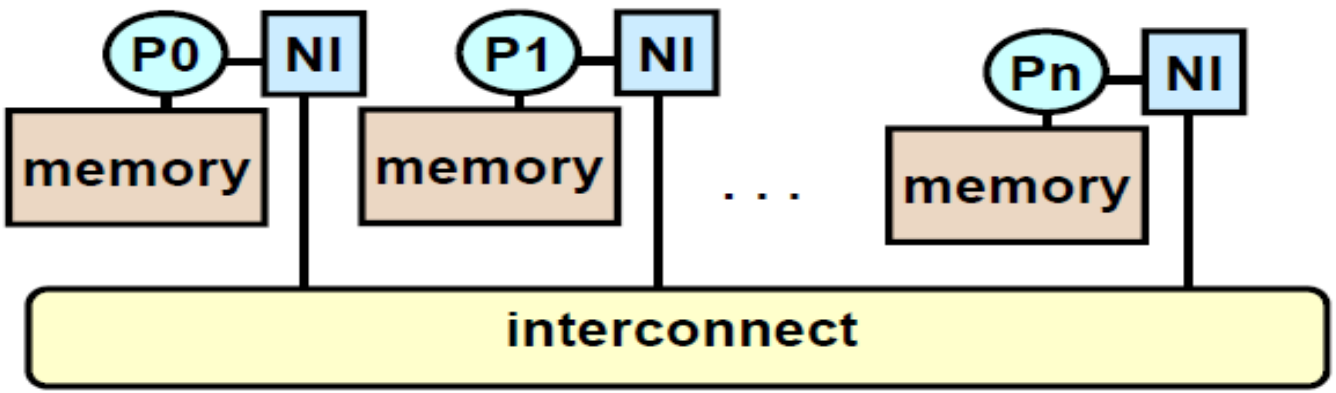
- Memory is logically shared, but physically distributed
  - Any processor can access any address in memory
  - Cache lines (or pages) are passed around machine
- SGI is canonical example (+ research machines)
  - Scales to 512 (SGI Altix (Columbia) at NASA/Ames)
  - Limitation is cache coherency protocols — how to keep cached copies of the same address consistent



Cache lines (pages) must be large to amortize overhead
➔
locality still critical to performance

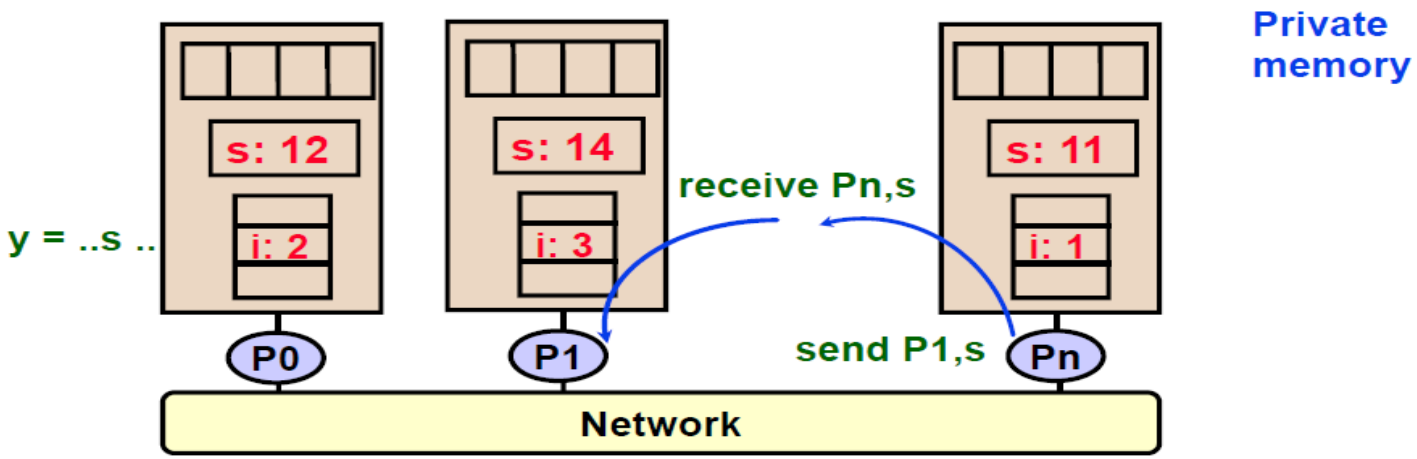
Machine Model: Distributed Memory

- Cray XE6 (Hopper), Cray XC30 (Edison)
- PC Clusters (Berkeley NOW, Beowulf)
- Edison, Hopper, most of the Top500, are distributed memory machines, but the nodes are SMPs.
- Each processor has its own memory and cache but cannot directly access another processor's memory.
- Each "node" has a Network Interface (NI) for all communication and synchronization.



Programming Model: Message Passing

- Program consists of a collection of named processes
  - Usually fixed at program startup time
  - Thread of control plus local address space — NO shared data.
  - Logically shared data is partitioned over local processes
- Processes communicate by explicit send/receive pairs
  - Coordination is implicit in every communication event.
  - MPI (Message Passing Interface) is the most commonly used SW



MPI References

- The Standard itself:
  - at `http://www.mpi-forum.org`
  - All MPI official releases, in both postscript and HTML
  - Latest version MPI 3.0, released Sept 2012
- Other information on Web
  - at `http://www.mcs.anl.gov/mpi`
  - pointers to lots of stuff, including other talks and tutorials, a FAQ, other MPI pages

Simple Example: MPI

```
#include "mpi.h"
#include <stdio.h>
int main( int argc, char *argv[] )
{
    int rank, size;
    MPI_Init( &argc, &argv );
    MPI_Comm_rank( MPI_COMM_WORLD, &rank );
    MPI_Comm_size( MPI_COMM_WORLD, &size );
    printf( "I am %d of %d\n", rank, size );
    MPI_Finalize();
    return 0;
}
```

```
mpirun -np 4 ./mpi_hello_world
I am 1 of 4
I am 0 of 4
I am 2 of 4
I am 3 of 4
```

Notes on Hello World

- All MPI programs begin with MPI_Init and end with MPI_Finalize
- MPI_COMM_WORLD is defined by mpi.h (in C) or mpif.h (in Fortran) and designates all processes in the MPI “job”
- Each statement executes independently in each process including the printf/print statements
- The MPI-1 Standard does not specify how to run an MPI program, but many implementations provide
  mpirun -np 4 a.out

MPI is Simple

Claim: most MPI applications can be written with only 6 functions (although which 6 may differ)

Using point-to-point

- MPI_INIT
- MPI_FINALIZE
- MPI_COMM_SIZE
- MPI_COMM_RANK
- MPI_SEND
- MPI_RECEIVE

Using collectives

- MPI_INIT
- MPI_FINALIZE
- MPI_COMM_SIZE
- MPI_COMM_RANK
- MPI_BCAST
- MPI_REDUCE

You may use more for convenience or performance

Parallel speedup

- definition:

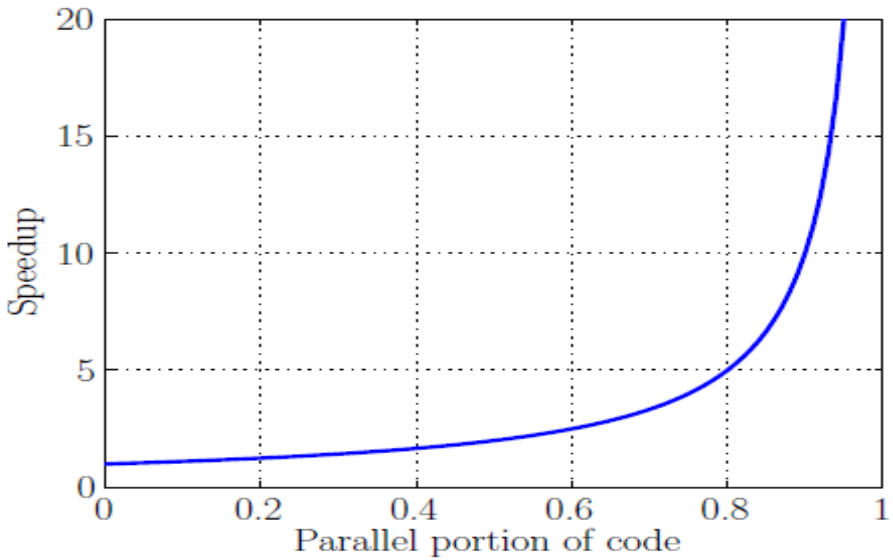
$$\text{speedup} = \frac{\text{serial time}}{\text{parallel time}}$$

  time is in the wall-clock (in contrast to CPU time) sense

- **Amdahl's Law**: assume infinite processors and no overhead

$$\text{ideal max speedup} = \frac{1}{1-\rho}$$

  where $\rho =$ percentage of parallelized code, can depend on input size

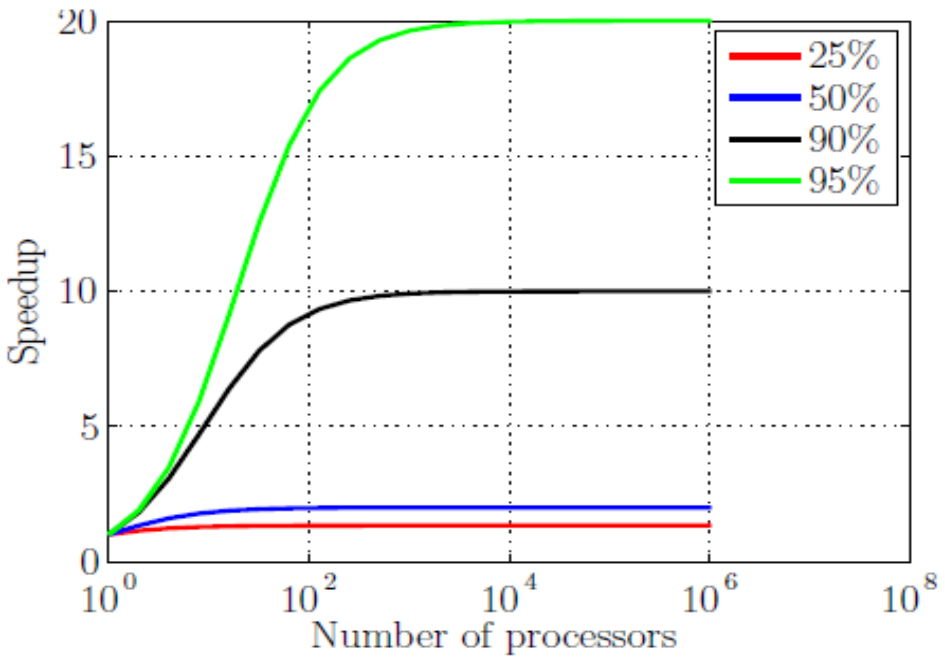


Parallel speedup

- assume N processors and no overhead:

$$\text{ideal speedup} = \frac{1}{(\rho/N) + (1-\rho)}$$

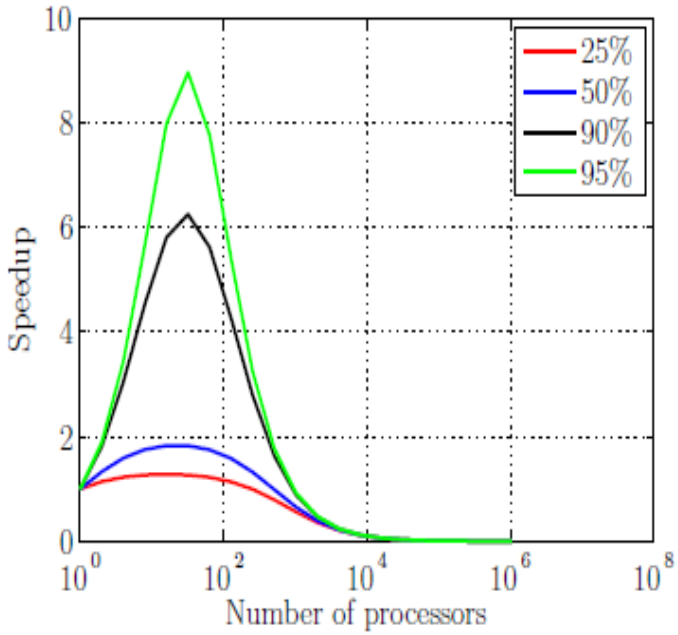
- ρ may also depend on N



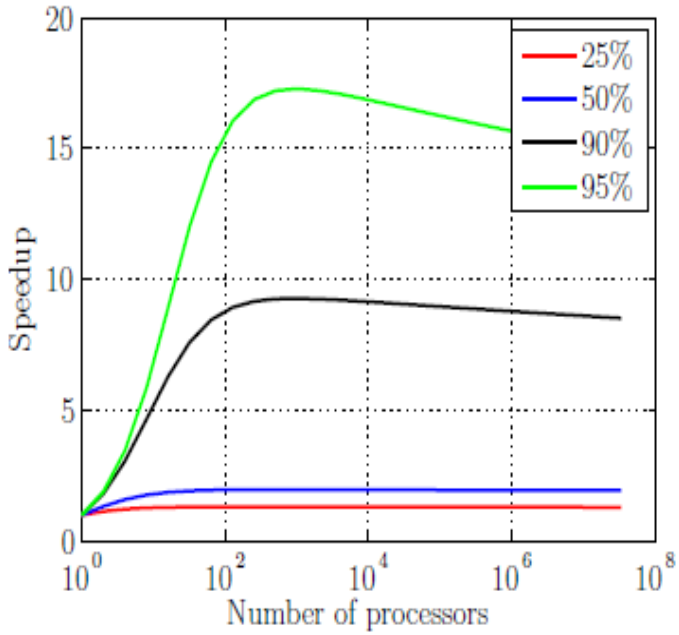
Parallel speedup

- $\epsilon :=$ parallel overhead
- in the real world

$$\text{actual speedup} = \frac{1}{(\rho/N) + (1-\rho) + \epsilon}$$

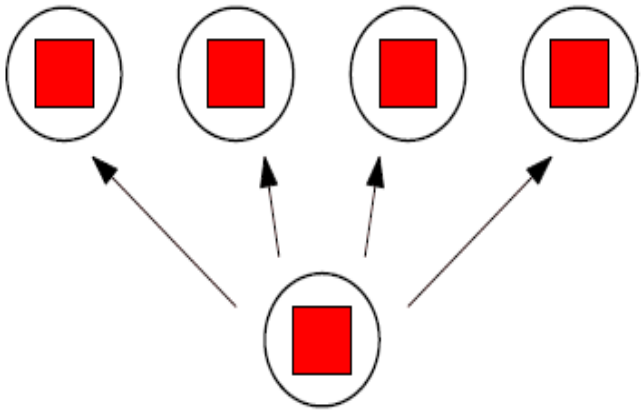


when $\varepsilon = N$

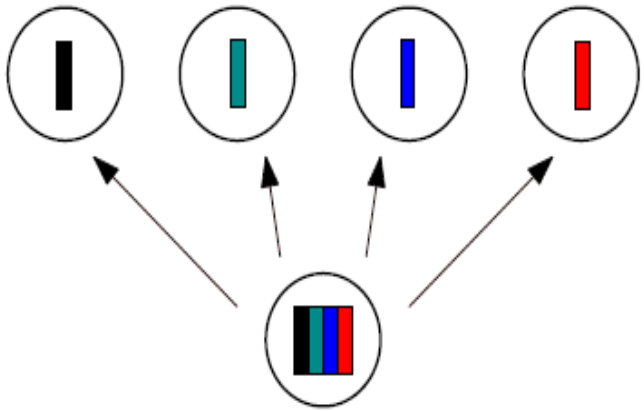


when $\varepsilon = \log(N)$

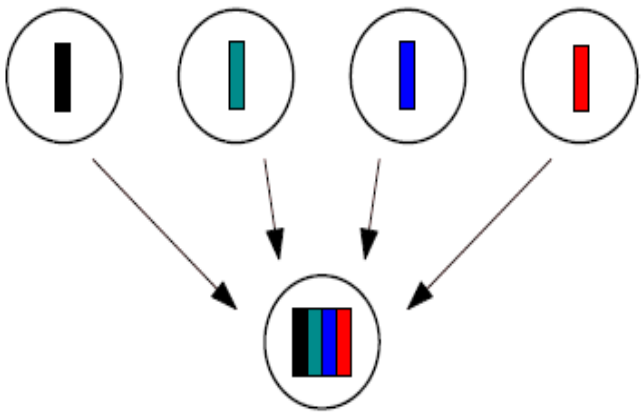
Basic types of communication



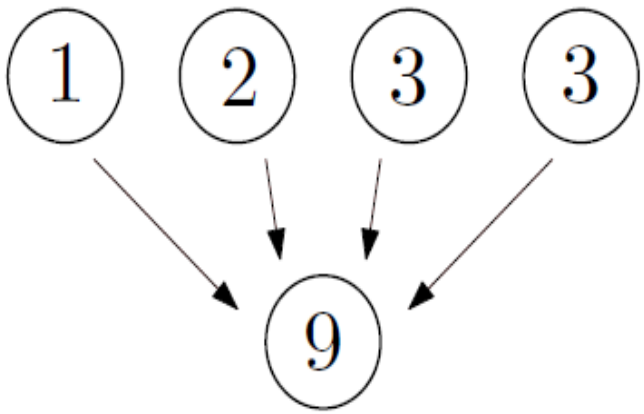
Broadcast



Scatter



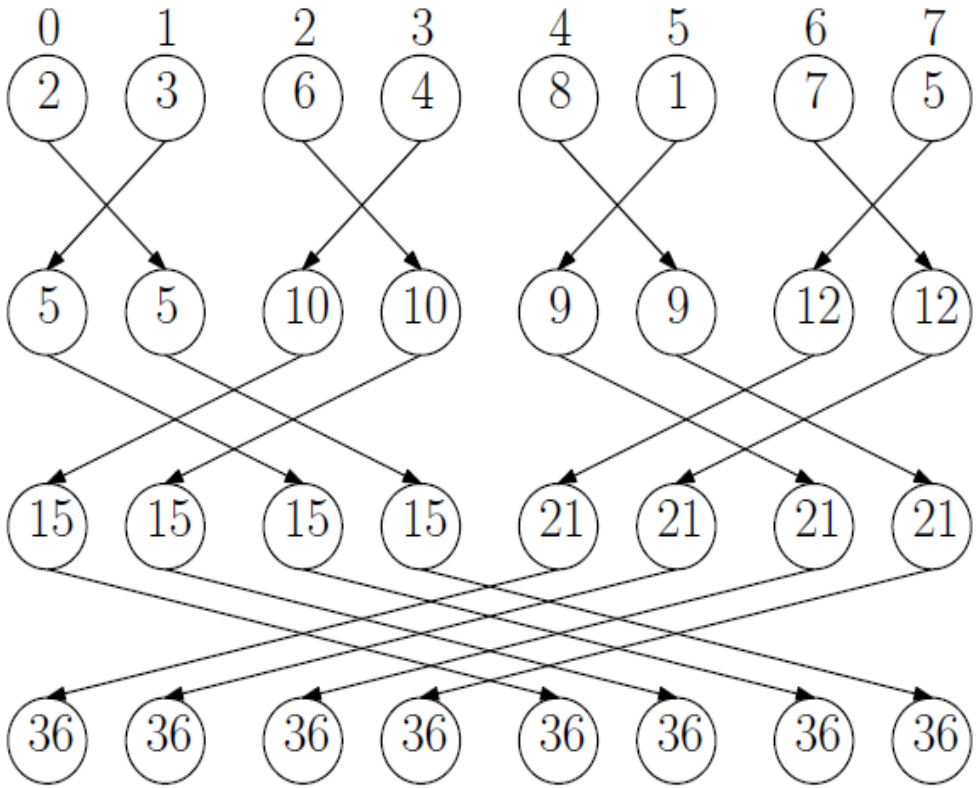
Gather



Reduce

Allreduce

Allreduce *sum* via butterfly communication

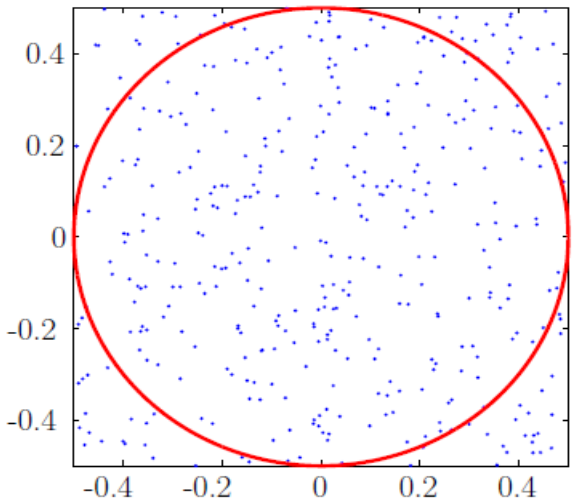


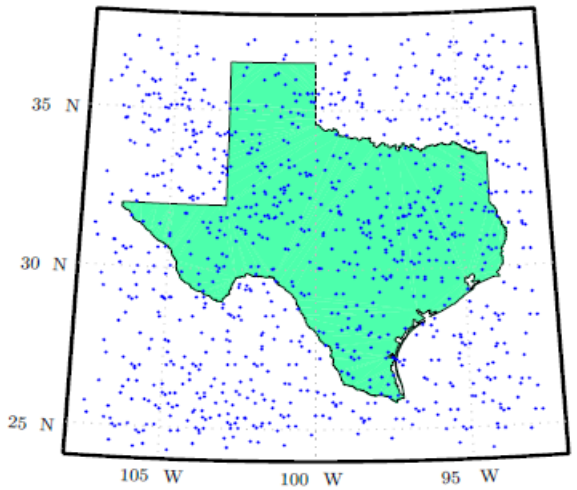
$\log(N)$ layers, N parallel communications per layer

Additional important topics

- **synchronization types**: barrier, lock, synchronous communication
- **load balancing**: static or dynamic distribution of tasks/data so that all processors are busy all time
- **granularity**: fine grain vs coarse grain
- **I/O**: a traditional parallelism killer, but modern database/parallel file system alleviates the problem (e.g., Hadoop Distributed File System (HDFS))

Example: π, area computation



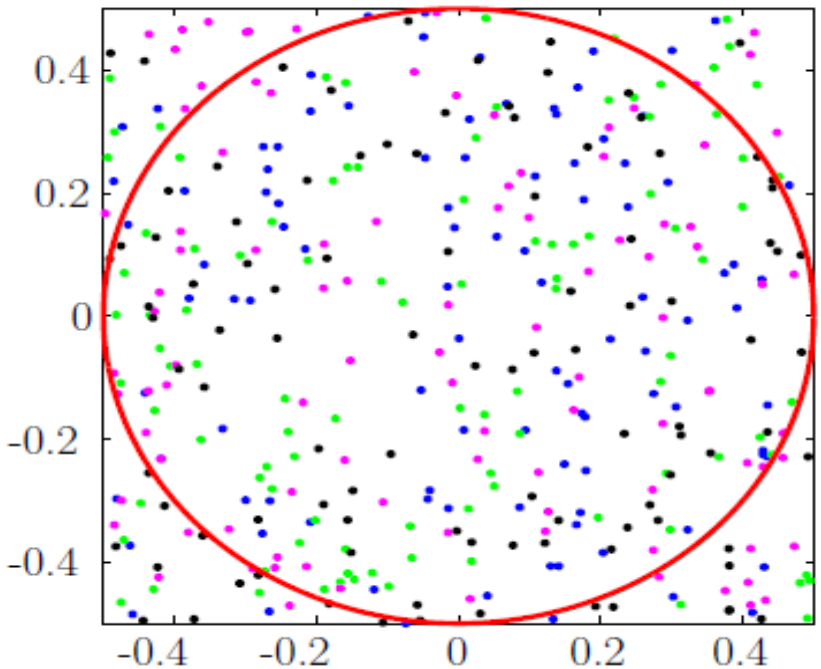


- box area is 1×1
- disc area is $\pi/4$
- ratio of area $\approx$ % inside points
- $\hat{\pi} = 4 \frac{\#\text{ points in the disc}}{\#\text{all the points}}$

Serial pseudo-code

1. total = 100000
2. in_disc = 0
3. **for** k = 1 to total do
4. x = a uniformly random point in $[-1/2, 1/2]^2$
5. **if** x is in the disc then
6. in_disc++
7. **end if**
8. **end for**
9. **return** 4*in_disc/total

Parallel π computation



- each sample is independent of others
- use multiple processors to run the for-loop
- use SPMD, process #1 collects results and return $\hat{\pi}$

SPMD pseudo-code

```
1  total = 100000
2  in_disc = 0
3  P= find total parallel processors
4  i= find my processor id
5  total = total/P
6  for k = 1 to total do
7      x = a uniformly random point in [0, 1]^2
8      if x is in the disc then
9          in_disc++
10     end if
11 end for
12 if i == 1 then
13     total_in_disc = reduce(in disc, SUM)
14     return 4*total_in_disc/total
15 end if
```

- requires only one collective communication;
- speedup = $\frac{1}{1/N+O(\log(N))} = \frac{N}{1+cN\log(N)}$, where c is a very small number;
- main loop doesn't require any synchro-nization.

Shared Memory Programming: OpenMP

- What is OpenMP?
  - Open specification for Multi-Processing, latest version 4.0, July 2013
  - "Standard" API for defining multi-threaded shared-memory programs
  - `openmp.org` — Talks, examples, forums, etc.
  - See `parlab.eecs.berkeley.edu/2012bootcampagenda` 2 OpenMP lectures (slides and video) by Tim Mattson
  - `computing.llnl.gov/tutorials/openMP/`
  - `portal.xsede.org/online-training`
  - `www.nersc.gov/assets/Uploads/XE62011OpenMP.pdf`
- High-level API
  - Preprocessor (compiler) directives (80%)
  - Library Calls (19%)
  - Environment Variables (1%)

Simple Example: OpenMP

```
int main() {
    for( i=0; i < 25; i++ ){
        printf(``Foo'');
    }
    return 0;
}
```

Parallel code by OpenMP:

```
int main() {
    omp_set_num_threads(16);
    // Do this part in parallel
    #pragma omp parallel for
    for( i=0; i < 25; i++ ){
        printf(``Foo'');
    }
    return 0;
}
```

Outline

Decomposable objective and constraints enables parallelism

- variables $\mathbf{x} = [\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_N]^T$
- embarrassingly parallel (not an interesting case): $\min \sum_i f_i(\mathbf{x}_i)$
- consensus minimization, no constraints

$$\min f(\mathbf{x}) = \sum_{i=1}^{N} f_i(\mathbf{x}) + r(\mathbf{x})$$

LASSO example: $f(x) = \frac{\lambda}{2} \sum_i \|\mathbf{A}_{(i)}\mathbf{x} - \mathbf{b}_i\|_2^2 + \|\mathbf{x}\|_1$

- coupling variable $\mathbf{z}$

$$\min_{\mathbf{x},\mathbf{z}} f(\mathbf{x}, \mathbf{z}) = \sum_{i=1}^{N} [f_i(\mathbf{x}_i, \mathbf{z}) + r_i(\mathbf{x}_i)] + r(\mathbf{z})$$

- coupling constraints

$$\min f(x) = \sum_{i=1}^{N} f_i(\mathbf{x}_i) + r_i(\mathbf{x}_i)$$

subject to

- equality constraints: $\sum_{i=1}^{N} \mathbf{A}_i \mathbf{x}_i = b$
- inequality constraints: $\sum_{i=1}^{N} \mathbf{A}_i \mathbf{x}_i \leq b$
- graph-coupling constraints $\mathbf{A}\mathbf{x} = b$, where $\mathbf{A}$ is sparse

- combinations of independent variables, coupling variables and constraints. example:

$$\begin{aligned} \min_{\mathbf{x},\mathbf{w},\mathbf{z}} \quad & \sum_{i=1}^{N} f_i(\mathbf{x}_i, \mathbf{z}) \\ \text{s.t.} \quad & \sum_{i} \mathbf{A}_i \mathbf{x}_i \leq \mathbf{b} \\ & \mathbf{B}_i \mathbf{x}_i \leq \mathbf{w}, \forall i \\ & \mathbf{w} \in \Omega \end{aligned}$$

- combinations of independent variables, coupling variables and constraints. example:

$$\begin{aligned} \min_{\mathbf{x},\mathbf{w},\mathbf{z}} \quad & \sum_{i=1}^{N} f_i(\mathbf{x}_i, \mathbf{z}) \\ \text{s.t.} \quad & \sum_{i} \mathbf{A}_i \mathbf{x}_i \leq \mathbf{b} \\ & \mathbf{B}_i \mathbf{x}_i \leq \mathbf{w}, \forall i \\ & \mathbf{w} \in \Omega \end{aligned}$$

- variable coupling $\longleftrightarrow$ constraint coupling:

$$\min \sum_{i=1}^{N} f_i(\mathbf{x}_i, \mathbf{z}) \longrightarrow \min \sum_{i=1}^{N} f_i(\mathbf{x}_i, \mathbf{z}_i), \text{ s.t. } z_i = z, \forall i$$

Approaches toward parallelism

- Keep an existing serial algorithm, parallelize its expensive part(s)
- Introduce a new parallel algorithm by
  - formulating an equivalent model
  - replacing the model by an approximate model

  where the new model lands itself for parallel computing

Example: parallel ISTA

$$\min_{\mathbf{x}} \quad \lambda\|\mathbf{x}\|_1 + f(\mathbf{x})$$

- Prox-linear approximation:

$$\arg\min_{\mathbf{x}} \lambda\|\mathbf{x}\|_1 + \left\langle \nabla f(\mathbf{x}^k), \mathbf{x} - \mathbf{x}^k \right\rangle + \frac{1}{2\delta_k}\|\mathbf{x} - \mathbf{x}^k\|_2^2$$

- Iterative soft-threshold algorithm:

$$\mathbf{x}^{k+1} = \text{shrink}(\mathbf{x}^k - \delta_k \nabla f(\mathbf{x}^k), \lambda\delta_k)$$

  - shrink(y, t) has a simple close form: max(abs(y)-t,0).*sign(y)
  - the dominating computation is $\nabla f(\mathbf{x})$

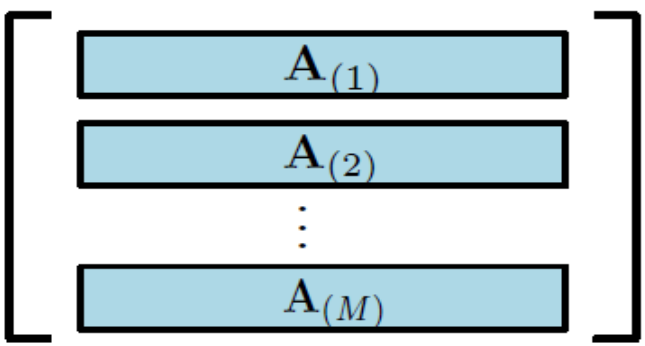
- example: $f(\mathbf{x}) = \frac{1}{2}\|\mathbf{A}\mathbf{x} - \mathbf{b}\|_2^2$, logistic loss of $\mathbf{A}\mathbf{x} - \mathbf{b}$, hinge loss, ..., where $\mathbf{A}$ is often dense and can go very large-scale

Parallel gradient computing of $f(\mathbf{x})$: row partition

Assumption: $f(\mathbf{x}) = g(\mathbf{A}\mathbf{x} + \mathbf{b})$, $\mathbf{A}$ is big and dense; $g(\mathbf{y}) = \sum_i g_i(\mathbf{y}_i)$ decomposable. Then

$$f(\mathbf{x}) = g(\mathbf{A}\mathbf{x} + \mathbf{b}) = \sum g_i(\mathbf{A}_{(i)}\mathbf{x} + \mathbf{b}_i)$$

- Let $\mathbf{A}_{(i)}$ and $\mathbf{b}_i$ be the ith row block of $\mathbf{A}$ and $\mathbf{b}$, respectively



- store $\mathbf{A}_{(i)}$ and $\mathbf{b}_i$ on node i
- goal: to compute $\nabla f(\mathbf{x}) = \sum_i \mathbf{A}_{(i)}^T \nabla g_i(\mathbf{A}_{(i)}\mathbf{x} + \mathbf{b}_i)$, or similarly, $\partial f(\mathbf{x})$

Parallel gradient computing of $f(\mathbf{x})$: row partition

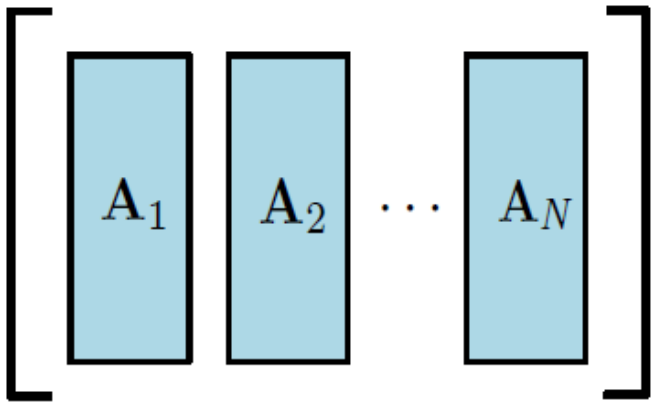
Steps:

1. Compute $\mathbf{A}_{(i)}\mathbf{x} + \mathbf{b}_i$, $\forall i$ in parallel
2. Compute $\mathbf{g}_i = \nabla g_i(\mathbf{A}_{(i)}\mathbf{x} + \mathbf{b}_i)$, $\forall i$ in parallel
3. compute $\mathbf{A}_{(i)}^T \mathbf{g}_i$, $\forall i$ in parallel
4. allreduce $\sum_{i=1}^{M} \mathbf{A}_{(i)}^T \mathbf{g}_i$

Notes:

- step 1, 2 and 3 are local computation and communication-free;
- step 4 requires an allreduce sum communication.

Parallel gradient computing of $f(\mathbf{x})$: column partition



column block partition

$$\mathbf{Ax} = \sum_{i=1}^{N} \mathbf{A}_i \mathbf{x}_i \Longrightarrow f(\mathbf{x}) = g(\mathbf{Ax} + \mathbf{b}) = g(\sum_{i} \mathbf{A}_i \mathbf{x}_i + \mathbf{b})$$

$$\nabla f(\mathbf{x}) = \begin{pmatrix} \mathbf{A}_1^T \nabla g(\mathbf{Ax} + \mathbf{b}) \\ \mathbf{A}_2^T \nabla g(\mathbf{Ax} + \mathbf{b}) \\ \cdots \\ \mathbf{A}_N^T \nabla g(\mathbf{Ax} + \mathbf{b}) \end{pmatrix}$$

Parallel gradient computing of $f(\mathbf{x})$: column partition

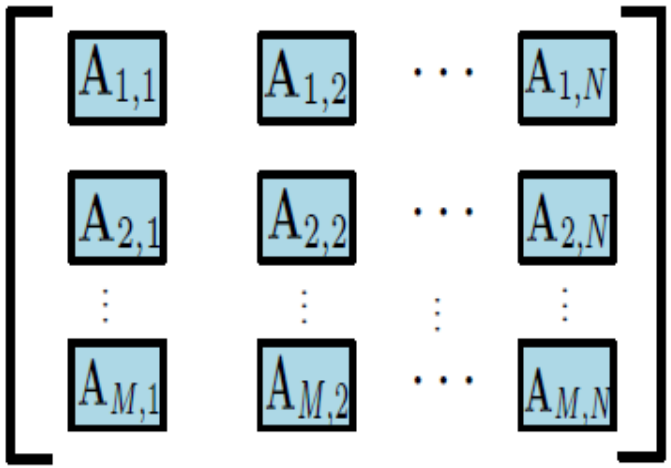
Steps:

1. Compute $\mathbf{A}_i\mathbf{x}_i + \frac{1}{N}\mathbf{b}$, $\forall i$ in parallel
2. allreduce $\mathbf{A}\mathbf{x} + \mathbf{b} = \sum_{i=1}^{N}(\mathbf{A}_i\mathbf{x}_i + \frac{1}{N}\mathbf{b})$
3. Compute $\nabla g(\mathbf{A}\mathbf{x} + \mathbf{b})$, $\forall i$ in each process
4. compute $\mathbf{A}_i^T \nabla g(\mathbf{A}\mathbf{x} + \mathbf{b})$, $\forall i$ in parallel

Notes:

- each processor keeps $\mathbf{A}_i$ and $\mathbf{x}_i$, as well as a copy of $\mathbf{b}$;
- step 2 requires an allreduce sum communication;
- step 3 involves duplicated computation in each process (hopefully, g is simple).

Parallel gradient computing of $f(\mathbf{x})$: Two-way partition



$$f(\mathbf{x}) = \sum_i g_i(\mathbf{A}_{(i)}\mathbf{x} + \mathbf{b}_i) = \sum_i g_i(\sum_j \mathbf{A}_{ij}\mathbf{x}_j + \mathbf{b}_i)$$

$$\nabla f(\mathbf{x}) = \sum_i \mathbf{A}_{(i)}^T \nabla g_i(\mathbf{A}_{(i)}\mathbf{x} + \mathbf{b}_i) = \sum_i \mathbf{A}_{(i)}^T \nabla g_i(\sum_j \mathbf{A}_{ij}\mathbf{x}_j + \mathbf{b}_i)$$

Parallel gradient computing of $f(\mathbf{x})$: Two-way partition

Step:

1. Compute $\mathbf{A}_{ij}\mathbf{x}_j + \frac{1}{N}\mathbf{b}_i$, $\forall i$ in parallel
2. allreduce $\sum_{j=1}^{N} \mathbf{A}_{ij}\mathbf{x}_j + \mathbf{b}_i$ for every i
3. compute $\mathbf{g}_i = \nabla g_i(\mathbf{A}_{(i)}\mathbf{x} + \mathbf{b}_i)$, $\forall i$ in parallel
4. compute $\mathbf{A}_{j,i}^T \mathbf{g}_i$, $\forall i$ in parallel
5. allreduce $\sum_{i=1}^{N} \mathbf{A}_{j,i}^T \mathbf{g}_i$ for every j

Notes:

- processor (i,j) keeps $\mathbf{A}_{i,j}, \mathbf{x}_j$ and $\mathbf{b}_i$;
- step 2 and 5 require parallel allreduce sum communication;

Example: parallel ISTA for LASSO

LASSO

$$\min f(\mathbf{x}) = \lambda \|\mathbf{x}\|_1 + \frac{1}{2}\|\mathbf{A}\mathbf{x} - \mathbf{b}\|_2^2$$

algorithm:

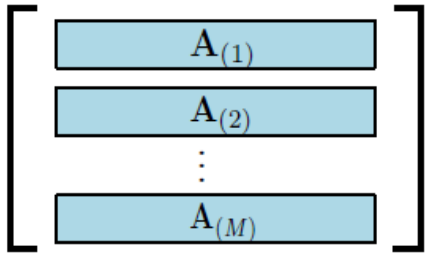
$$\mathbf{x}^{k+1} = \text{shrink}(\mathbf{x}^k - \delta_k \mathbf{A}^T \mathbf{A}\mathbf{x}^k + \delta_k \mathbf{A}^T \mathbf{b}, \lambda \delta_k)$$

Serial pseudo-code

1. initialize $\mathbf{x} = 0$ and δ
2. pre-compute $\mathbf{A}^T\mathbf{b}$
3. **while** not converged **do**
4. $\quad \mathbf{g} = \mathbf{A}^T\mathbf{A}\mathbf{x} - \mathbf{A}^T\mathbf{b}$
5. $\quad \mathbf{x} = \text{shrink}(\mathbf{x} - \delta\mathbf{g}, \lambda\delta)$
6. **end while**

Example: parallel ISTA for LASSO

distribute blocks of rows to M nodes

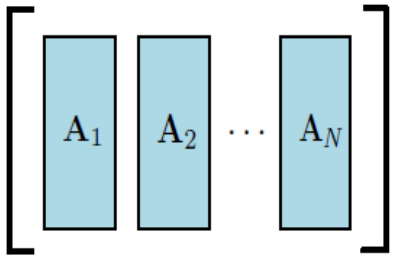


1. initialize $\mathbf{x} = 0$ and δ
2. i=find my processor id
3. processor i load $\mathbf{A}_{(i)}$
4. pre-compute $\mathbf{A}^T\mathbf{b}$
5. **while** not converged **do**
6. processor i computes $\mathbf{c}_i = \mathbf{A}_{(i)}^T\mathbf{A}_{(i)}\mathbf{x}$
7. y=allreduce($\mathbf{c}_i$, SUM)
8. $\mathbf{x} = \text{shrink}(\mathbf{x} - \delta\mathbf{c} + \delta\mathbf{A}^T\mathbf{b}, \lambda\delta)$
9. **end while**

- Assume $\mathbf{A} \in \mathbb{R}^{m\times n}$
- one allreduce per iteration
- speedup $\approx \frac{1}{\rho/M+(1-\rho)+O(\log(M))}$
- P is close to 1
- requires synchronization

Example: parallel ISTA for LASSO

distribute blocks of columns to N nodes



1. initialize $\mathbf{x} = 0$ and δ
2. i=find my processor id
3. processor i load $\mathbf{A}_i$
4. pre-compute $\mathbf{A}^T\mathbf{b}$
5. **while** not converged **do**
6. processor i computes $\mathbf{y}_i = \mathbf{A}_i\mathbf{x}_i$
7. y=allreduce($\mathbf{y}_i$, SUM)
8. processor i computes $\mathbf{z}_i = \mathbf{A}_i^T\mathbf{y}$
9. $\mathbf{x}_i^{k+1} = \text{shrink}(\mathbf{x} - \delta\mathbf{z}_i + \delta\mathbf{A}_i^T\mathbf{b}, \lambda\delta)$
10. **end while**

- one allreduce per iteration
- speedup $\approx \frac{1}{\rho/N+(1-\rho)+O(\frac{m}{n}\log(N)}$
- P is close to 1
- requires synchronization
- if $m \ll n$, this approach is faster

Outline

Unconstrained minimization with coupling variables

$\mathbf{x} = [\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_N]^T$.
$\mathbf{z}$ is the **coupling/bridging/complicating variable**

$$\min_{\mathbf{x},\mathbf{z}} f(\mathbf{x}, \mathbf{z}) = \sum_{i=1}^{N} f_i(\mathbf{x}_i, \mathbf{z})$$

equivalently to separable objective with **coupling constraints**

$$\min_{\mathbf{x},\mathbf{z}} \sum_{i=1}^{N} f_i(\mathbf{x}_i, \mathbf{z}_i), \text{ s.t. } \mathbf{z}_i = \mathbf{z}, \forall i$$

- network utility maximization (NUM), extend to multiple layers
- domain decomposition ($\mathbf{z}$ are overlapping boundary variables)
- system of equations: $\mathbf{A}$ is block-diagonal but with a few dense columns

Primal decomposition (bi-level optimization)

Introduce **easier subproblems**

$$\mathbf{g}_i(\mathbf{z}) = \min_{\mathbf{x}_i} f_i(\mathbf{x}_i, \mathbf{z}), \quad i = 1, \ldots, N$$

then

$$\min_{\mathbf{x},\mathbf{z}} \sum_{i=1}^{N} f_i(\mathbf{x}_i, \mathbf{z}) \Longleftrightarrow \min_{\mathbf{z}} \sum_{i} g_i(\mathbf{z})$$

the RHS is called the **master problem**.
parallel computing: iterate

- fix $\mathbf{z}$, update $\mathbf{x}_i$ and compute $g_i(\mathbf{z})$ in parallel
- update $\mathbf{z}$ using a standard method (typically, subgradient descent)

Notes:

- good for small-dimensional $\mathbf{z}$
- fast if the master problem converges fast

Dual decomposition

Consider

$$\min_{\mathbf{x},\mathbf{z}} \sum_i f_i(\mathbf{x}_i, \mathbf{z}_i)$$

$$\text{s.t.} \sum_i \mathbf{A}_i \mathbf{z}_i = \mathbf{b}$$

Introduce:

$$\begin{aligned} g_i(\mathbf{z}) &= \min_{\mathbf{x}_i} f(\mathbf{x}_i, \mathbf{z}) \\ g_i^*(y) &= \max_{\mathbf{z}} \mathbf{y}^T \mathbf{z} - g_i(z) \quad \text{convex conjugate} \end{aligned}$$

$$\min_{\mathbf{x},\mathbf{z}} \max_{\mathbf{y}} \sum_i f_i(\mathbf{x}_i, \mathbf{z}_i) + \mathbf{y}^T (\mathbf{b} - \sum_i \mathbf{A}_i \mathbf{z}_i)$$

$$\iff \max_{\mathbf{y}_i} \min_{\mathbf{x}_i,\mathbf{z}_i} \left\{ \sum_i (f_i(\mathbf{x}_i, \mathbf{z}_i) - \mathbf{y}^T \mathbf{A}_i \mathbf{z}_i) + \mathbf{y}^T \mathbf{b} \right\}$$

Dual decomposition

$$\min_{\mathbf{x},\mathbf{z}} \max_{\mathbf{y}} \sum_i f_i(\mathbf{x}_i, \mathbf{z}_i) + \mathbf{y}^T(\mathbf{b} - \sum_i \mathbf{A}_i \mathbf{z}_i)$$

$$\iff \max_{\mathbf{y}_i} \min_{\mathbf{x}_i,\mathbf{z}_i} \left\{ \sum_i (f_i(\mathbf{x}_i, \mathbf{z}_i) - \mathbf{y}^T \mathbf{A}_i \mathbf{z}_i) + \mathbf{y}^T \mathbf{b} \right\}$$

$$\iff \max_{\mathbf{y}_i} \left\{ \min_{\mathbf{z}_i} \sum_i (g_i(\mathbf{z}_i) - \mathbf{y}^T \mathbf{A}_i \mathbf{z}_i) + \mathbf{y}^T \mathbf{b} \right\}$$

$$\iff \min_{\mathbf{y}} \sum_i g_i^*(\mathbf{A}_i^T \mathbf{y}) - \mathbf{y}^T b$$

Example: consensus and exchange problems

Consensus problem

$$\min_{\mathbf{z}} \sum_i g_i(\mathbf{z})$$

Reformulate as

$$\min_{\mathbf{z}} \sum_i g_i(\mathbf{z}_i), \text{ s.t. } \mathbf{z}_i = \mathbf{z}, \forall i$$

Its dual problem is the exchange problem

$$\min_{\mathbf{y}} \sum_i g_i^*(\mathbf{y}_i), \text{ s.t. } \sum_i \mathbf{y}_i = 0$$

Primal-dual objective properties

Constrained separable problem

$$\min_{\mathbf{z}} \sum_i g_i(\mathbf{z}) \text{ s.t. } \sum_i \mathbf{A}_i \mathbf{z}_i = \mathbf{b}$$

Dual problem

$$\min_{\mathbf{y}} \sum_i g_i^*(\mathbf{A}_i^T \mathbf{y}) - \mathbf{b}^T \mathbf{y}$$

- If g_i is proper, closed, convex, $g_i^*(\mathbf{a}_i^T \mathbf{y})$ is sub-differentiable
- If g_i is strictly convex, $g_i^*(\mathbf{a}_i^T \mathbf{y})$ is differentiable
- If g_i is strongly convex, $g_i^*(\mathbf{a}_i^T \mathbf{y})$ is differentiable and has Lipschitz gradient
- Obtaining $\mathbf{z}^*$ from $\mathbf{y}^*$ generally requires strict convexity of g_i or strict complementary slackness

Example:

- $g_i(\mathbf{z}) = |\mathbf{z}_i|$ and $g_i^*(\mathbf{a}_i^T\mathbf{y}) = \iota\{-1 \leq \mathbf{a}_i^T\mathbf{y} \leq 1\}$
- $g_i(\mathbf{z}) = \frac{1}{2}|\mathbf{z}_i|^2$ and $g_i^*(\mathbf{a}_i^T\mathbf{y}) = \frac{1}{2}(\mathbf{a}_i^T\mathbf{y})^2$
- $g_i(\mathbf{z}) = \exp(\mathbf{z}_i)$ and $g_i^*(\mathbf{a}_i^T\mathbf{y}) = (\mathbf{a}_i^T\mathbf{y})\ln(\mathbf{a}_i^T\mathbf{y}) - \mathbf{a}_i^T\mathbf{y} + \iota\{\mathbf{a}_i^T\mathbf{y} \geq 0\}$

Example: augmented ℓ_1

Change $|z_i|$ into strongly convex $g_i(z_i) = |z_i| + \frac{1}{2\alpha}|z_i|^2$.
Dual problem:

$$\min_{\mathbf{y}} \sum_i g_i^*(\mathbf{A}_i^T\mathbf{y}) - \mathbf{b}^T\mathbf{y} = \sum_i \frac{\alpha}{2}|\text{shrink}(\mathbf{A}_i^T\mathbf{y})|^2 - \mathbf{b}^T\mathbf{y}$$

Dual gradient iteration (equivalent to linearized Bregman):

$$\mathbf{y}^{k+1} = \mathbf{y}^k - t^k\left(\sum_i \nabla h_i(\mathbf{y}^k) - \mathbf{b}\right)$$

where $\nabla h_i(\mathbf{y}) = \alpha\mathbf{A}_i\text{shrink}(\mathbf{A}_i^T\mathbf{y})$.

- Dual is C^1 and unconstrained. One can apply (accelerated) gradient descent, line search, quasi-Newton method, etc.
- Recover $\mathbf{x}^* = \alpha\text{shrink}(\mathbf{A}_i^T\mathbf{y})$
- Easy to parallelize. Node i computes $\nabla h_i(y)$. One collective communication per iteration.

Alternating direction method of multipliers (ADMM)

- **step 1**: turn problem into the form of

$$\begin{aligned} \min_{\mathbf{x},\mathbf{y}} \quad & f(\mathbf{x}) + g(\mathbf{y}) \\ \text{s.t.} \quad & \mathbf{A}\mathbf{x} + \mathbf{B}\mathbf{y} = \mathbf{b} \end{aligned}$$

  f and g are **convex**, maybe **nonsmooth**, can **incorporate constraints**

- **step 2**:

$$\begin{aligned} \mathbf{x}^{k+1} &= \arg\min_{\mathbf{x}} f(\mathbf{x}) + g(\mathbf{y}^k) + \frac{\beta}{2}\|\mathbf{A}\mathbf{x} + \mathbf{B}\mathbf{y}^k - \mathbf{b} - \mathbf{z}^k\|_2^2 \\ \mathbf{y}^{k+1} &= \arg\min_{\mathbf{y}} f(\mathbf{x}^{k+1}) + g(\mathbf{y}) + \frac{\beta}{2}\|\mathbf{A}\mathbf{x}^{k+1} + \mathbf{B}\mathbf{y} - \mathbf{b} - \mathbf{z}^k\|_2^2 \\ \mathbf{z}^{k+1} &= \mathbf{z}^k - \beta(\mathbf{A}x^{k+1} + \mathbf{B}\mathbf{y}^{k+1} - b) \end{aligned}$$

history: dates back to 60s, formalized 80s, parallel versions appeared late 80s, revived very recently, new convergence results recently

Block separable ADMM

Suppose $\mathbf{x} = (\mathbf{x}_1, \mathbf{x}_2, \cdots, \mathbf{x}_N)$ and f is separable, i.e.,

$$f(\mathbf{x}) = f_1(\mathbf{x}_1) + f_2(\mathbf{x}_2) + \cdots + f_N(\mathbf{x}_N).$$

Model

$$\begin{array}{ll} \min\limits_{\mathbf{x},\mathbf{z}} & f(\mathbf{x}) + g(\mathbf{z}) \\ \text{s.t.} & \mathbf{A}\mathbf{x} + \mathbf{B}\mathbf{z} = \mathbf{b}. \end{array}$$

where

$$\mathbf{A} = \begin{bmatrix} \mathbf{A}_1 & & & \mathbf{0} \\ & \mathbf{A}_2 & & \\ & & \ddots & \\ \mathbf{0} & & & \mathbf{A}_N \end{bmatrix}$$

Block separable ADMM

The $\mathbf{x}-$update

$$\mathbf{x}^{k+1} \leftarrow \min\, f(\mathbf{x}) + \frac{\beta}{2}\|\mathbf{A}\mathbf{x} + \mathbf{B}\mathbf{y}^k - \mathbf{b} - \mathbf{z}^k\|_2^2$$

is separable to N independent subproblems

$$\mathbf{x}_1^{k+1} \leftarrow \min\, f_1(\mathbf{x}_1) + \frac{\beta}{2}\|\mathbf{A}_1\mathbf{x}_1 + (\mathbf{B}\mathbf{y}^k - \mathbf{b} - \mathbf{z}^k)_1\|_2^2,$$

$$\vdots$$

$$\mathbf{x}_N^{k+1} \leftarrow \min\, f_N(\mathbf{x}_N) + \frac{\beta}{2}\|\mathbf{A}_N\mathbf{x}_N + (\mathbf{B}\mathbf{y}^k - \mathbf{b} - \mathbf{z}^k)_N\|_2^2.$$

No coordination is required.

Example: consensus optimization

Model

$$\min \sum_{i=1}^{N} f_i(\mathbf{x})$$

the objective is partially separable.
Introduce N copies $\mathbf{x}_1, \cdots, \mathbf{x}_N$ of $\mathbf{x}$. They have the same dimensions.
New model:

$$\min_{\{\mathbf{x}_i\},\mathbf{z}} \sum_{i=1}^{N} f_i(\mathbf{x}_i), \text{ s.t. } \mathbf{x}_i - \mathbf{z} = \mathbf{0}, \forall i.$$

A more general objective with function g is $\sum_{i=1}^{N} f_i(\mathbf{x}) + g(\mathbf{z})$.
New model:

$$\min_{\{\mathbf{x}_i\},\mathbf{y}} \sum_{i=1}^{N} f_i(\mathbf{x}_i) + g(\mathbf{z}), \text{ s.t. } \begin{bmatrix} I & & \\ & \ddots & \\ & & I \end{bmatrix} \begin{bmatrix} \mathbf{x}_1 \\ \vdots \\ \mathbf{x}_N \end{bmatrix} - \begin{bmatrix} I \\ \vdots \\ I \end{bmatrix} \mathbf{z} = \mathbf{0}.$$

Example: consensus optimization

Lagrangian

$$L(\{\mathbf{x}_i\}, \mathbf{z}; \{\mathbf{y}_i\}) = \sum_i (f_i(\mathbf{x}_i) + \frac{\beta}{2}\|\mathbf{x}_i - \mathbf{z} - \mathbf{y}_i\|_2^2))$$

where $\mathbf{y}_i$ is the Lagrange multipliers to $\mathbf{x}_i - \mathbf{z} = 0$.
ADMM

$$\begin{aligned}
\mathbf{x}_i^{k+1} &= \operatorname*{argmin}_{\mathbf{x}_i} f_i(\mathbf{x}_i) + \frac{\beta}{2}\|\mathbf{x}_i - \mathbf{z}^k - \mathbf{y}_i^k\|_2, i = 1, \cdots, N, \\
\mathbf{z}^{k+1} &= \frac{1}{N}\sum_{i=1}^{N}(\mathbf{x}_i^{k+1} - \beta^{-1}\mathbf{y}_i^k), \\
\mathbf{y}_i^{k+1} &= \mathbf{y}_i^k - (\mathbf{x}_i^{k+1} - \mathbf{z}^{k+1}), i = 1, \cdots, N.
\end{aligned}$$

The exchange problem

Model $\mathbf{x}_1, \cdots, \mathbf{x}_N \in \mathbb{R}^n$,

$$\min \sum_{i=1}^{N} f_i(\mathbf{x}_i), \text{ s.t. } \sum_{i=1}^{N} \mathbf{x}_i = \mathbf{0}.$$

- it is the dual of the consensus problem
- exchanging n goods among N parties to minimize a total cost
- our goal: to decouple $\mathbf{x}_i-$updates

An equivalent model

$$\min \sum_{i=1}^{N} f_i(\mathbf{x}_i), \text{ s.t. } \mathbf{x}_i - \mathbf{x}_i^{'} = \mathbf{0}, \forall i, \sum_{i=1}^{N} \mathbf{x}_i^{'} = \mathbf{0}.$$

The exchange problem

ADMM after consolidating the $\mathbf{x}_i'$ update:

$$\begin{aligned}\mathbf{x}_i^{k+1} &= \underset{\mathbf{x}_i}{\operatorname{argmin}} f_i(\mathbf{x}_i) + \frac{\beta}{2}\|\mathbf{x}_i - (\mathbf{x}_i^k - \text{mean}\{\mathbf{x}_i^k\} - \mathbf{u}^k)\|_2^2, \\ \mathbf{u}^{k+1} &= \mathbf{u}^k + \text{mean}\{\mathbf{x}_i^{k+1}\}.\end{aligned}$$

Applications: distributed dynamic energy management

Distributed ADMM I

$$\min_{\{\mathbf{x}_i\},\mathbf{y}} \sum_{i=1}^{N} f_i(\mathbf{x}_i) + g(\mathbf{z}), \text{ s.t. } \begin{bmatrix} I & & \\ & \ddots & \\ & & I \end{bmatrix} \begin{bmatrix} \mathbf{x}_1 \\ \vdots \\ \mathbf{x}_N \end{bmatrix} - \begin{bmatrix} I \\ \vdots \\ I \end{bmatrix} \mathbf{z} = \mathbf{0}.$$

Consider N computing nodes with MPI (message passing interface).

- $\mathbf{x}_i$ are local variables; $\mathbf{x}_i$ is stored and updated on node i only
- $\mathbf{z}$ is the global variable; computed and communicated by MPI
- $\mathbf{y}_i$ are dual variables, stored and updated on node i only

At each iteration, given $\mathbf{y}^k$ and $\mathbf{z}_i^k$

- each node i computes $\mathbf{x}_i^{k+1}$
- each node i computes $\mathbf{P}_i := (\mathbf{x}_i^{k+1} - \beta^{-1}\mathbf{y}_i^k)$
- MPI gathers $\mathbf{P}_i$ and scatters its mean, $\mathbf{z}^{k+1}$, to all nodes
- each node i computes $\mathbf{y}_i^{k+1}$

Example: distributed LASSO

Model

$$\min \|\mathbf{x}\|_1 + \frac{\beta}{2}\|\mathbf{A}\mathbf{x} - \mathbf{b}\|_2^2.$$

Decomposition

$$\mathbf{A}\mathbf{x} = \begin{bmatrix} \mathbf{A}_1 \\ \mathbf{A}_2 \\ \vdots \\ \mathbf{A}_N \end{bmatrix} \mathbf{x}, \mathbf{b} = \begin{bmatrix} \mathbf{b}_1 \\ \mathbf{b}_2 \\ \vdots \\ \mathbf{b}_N \end{bmatrix}.$$

$\Rightarrow$

$$\frac{\beta}{2}\|\mathbf{A}\mathbf{x} - \mathbf{b}\|_2^2 = \sum_{i=1}^{N} \frac{\beta}{2}\|\mathbf{A}_i\mathbf{x} - \mathbf{b}_i\|_2^2 =: \sum_{i=1}^{N} f_i(\mathbf{x}).$$

LASSO has the form

$$\min \sum_{i=1}^{N} f_i(\mathbf{x}) + g(\mathbf{x})$$

and thus can be solved by distributed ADMM.

Example: dual of LASSO

LASSO

$$\min \|\mathbf{x}\|_1 + \frac{\beta}{2}\|\mathbf{A}\mathbf{x} - \mathbf{b}\|_2^2.$$

Lagrange dual

$$\min_{\mathbf{y}} \{\mathbf{b}^T\mathbf{y} + \frac{\mu}{2}\|\mathbf{y}\|_2^2 : \|\mathbf{A}^T\mathbf{y}\|_\infty \leq 1\}$$

equivalently,

$$\min_{\mathbf{y},\mathbf{z}} \{-\mathbf{b}^T\mathbf{y} + \frac{\mu}{2}\|\mathbf{y}\|_2^2 + l_{\{\|\mathbf{z}\|_\infty \leq 1\}} : \mathbf{A}^T\mathbf{y} + \mathbf{z} = \mathbf{0}\}$$

Standard ADMM:

- primal $\mathbf{x}$ is the multipliers to $\mathbf{A}^T\mathbf{y} + \mathbf{z} = \mathbf{0}$
- $\mathbf{z}-$update is projection to $l_\infty-$ball; easy and separable
- $\mathbf{y}-$update is quadratic

Example: dual of LASSO

- Dual augmented Lagrangian (the scaled form):

$$L(\mathbf{y}, \mathbf{z}; \mathbf{x}) = \mathbf{b}^T\mathbf{y} + \frac{\mu}{2}\|\mathbf{y}\|_2^2 + l_{\|\mathbf{z}\|_\infty \le 1} + \frac{\beta}{2}\|\mathbf{A}^T\mathbf{y} + \mathbf{z} - \mathbf{x}\|_2^2$$

- Dual ADMM iterations:

$$\begin{aligned}
\mathbf{z}^{k+1} &= Proj_{\|\cdot\|_\infty \le 1}(\mathbf{x}^k - \mathbf{A}^T\mathbf{y}^k), \\
\mathbf{y}^{k+1} &= (\mu I + \beta\mathbf{A}\mathbf{A}^T)^{-1}(\beta\mathbf{A}(\mathbf{x}^k - \mathbf{z}^{k+1}) - \mathbf{b}), \\
\mathbf{x}^{k+1} &= \mathbf{z}^k - \gamma(\mathbf{A}^T\mathbf{y}^{k+1} + \mathbf{z}^{k+1}).
\end{aligned}$$

  and upon termination at step K, return primal solution

$$\mathbf{x}^* = \beta\mathbf{x}^K \text{ (de-scaling).}$$

- Computation bottlenecks:
  - $(\mu I + \beta\mathbf{A}\mathbf{A}^T)^{-1}$, unless $\mathbf{A}\mathbf{A}^T = I$ or $\mathbf{A}\mathbf{A}^T \approx I$
  - $\mathbf{A}(\mathbf{x}^k - \mathbf{z}^{k+1})$ and $\mathbf{A}^T\mathbf{y}^k$, unless $\mathbf{A}$ is small or has structures

Example: dual of LASSO

Observe

$$\min_{\mathbf{y},\mathbf{z}}\{\mathbf{b}^T\mathbf{y} + \frac{\mu}{2}\|\mathbf{y}\|_2^2 + l_{\{\|\mathbf{z}_\infty \leq 1\}} : \mathbf{A}^T\mathbf{y} + \mathbf{z} = \mathbf{0}\}$$

- All the objective terms are perfectly separable
- The constraints cause the computation bottlenecks
- We shall try to decouple the blocks of $\mathbf{A}^T$

Distributed ADMM II

A general form with inseparable f and separable g

$$\min_{\mathbf{x},\mathbf{z}} \sum_{l=1}^{L} (f_l(\mathbf{x}) + g_l(\mathbf{z}_l)), \text{ s.t. } \mathbf{A}\mathbf{x} + \mathbf{z} = \mathbf{b}$$

- Make L copies $\mathbf{x}_1, \mathbf{x}_2, \cdots, \mathbf{x}_L$ of $\mathbf{x}$
- Decompose

$$\mathbf{A} = \begin{bmatrix} \mathbf{A}_1 \\ \vdots \\ \mathbf{A}_L \end{bmatrix}, \mathbf{z} = \begin{bmatrix} \mathbf{z}_1 \\ \vdots \\ \mathbf{z}_L \end{bmatrix}, \mathbf{b} = \begin{bmatrix} \mathbf{b}_1 \\ \vdots \\ \mathbf{b}_L \end{bmatrix}$$

- Rewrite $\mathbf{A}\mathbf{x} + \mathbf{z} = \mathbf{0}$ as

$$\mathbf{A}_l\mathbf{x}_l + \mathbf{z}_l = \mathbf{b}_l, \mathbf{x}_l - \mathbf{x} = \mathbf{0}, l = 1, \cdots, L.$$

Distributed ADMM II

New model:

$$\begin{aligned} \min_{\mathbf{x},\{\mathbf{x}_l\},\mathbf{z}} \quad & \sum_{l=1}^{L}(f_l(\mathbf{x}_l)+g_l(\mathbf{z}_l)) \\ \text{s.t.} \quad & \mathbf{A}_l\mathbf{x}_l+\mathbf{z}_l=\mathbf{b}_l, \mathbf{x}_l-\mathbf{x}=\mathbf{0}, l=1,\cdots,L. \end{aligned}$$

- $\mathbf{x}_l$'s are copies of $\mathbf{x}$
- $\mathbf{z}_l$'s are sub-blocks of $\mathbf{z}$
- Group variables $\{\mathbf{x}_l\}, \mathbf{z}, \mathbf{x}$ into two sets
  - $\{\mathbf{x}_l\}$: given $\mathbf{z}$ and $\mathbf{x}$, the updates of $\mathbf{x}_l$ are separable
  - $(\mathbf{z}, \mathbf{x})$: given $\{\mathbf{x}_l\}$, the updates of $\mathbf{z}_l$ and $\mathbf{x}$ are separable
    Therefore, standard (2-block) ADMM applies.
- One can also add a simple regularizer $h(\mathbf{x})$

Distributed ADMM II

Consider L computing nodes with MPI.

- $\mathbf{A}_l$ is local data store on node l only
- $\mathbf{x}_l, \mathbf{z}_l$ are local variables; $\mathbf{x}_l$ is stored and updated on node l only
- $\mathbf{x}$ is the global variable; computed and dispatched by MPI
- $\mathbf{y}_l, \bar{\mathbf{y}}_l$ are Lagrange multipliers to $\mathbf{A}_l\mathbf{x}_l + \mathbf{z}_l = \mathbf{b}_l$ and $\mathbf{x}_l - \mathbf{x} = \mathbf{0}$,respectively, stored and updated on node l only

At each iteration,

- each node l computes $\mathbf{x}_l^{k+1}$, using data $\mathbf{A}_l$
- each node l computes $\mathbf{z}_l^{k+1}$, prepares $\mathbf{P}_l = (\cdots)$
- MPI gathers $\mathbf{P}_l$ and scatters its mean, $\mathbf{x}^{k+1}$, to all nodes l
- each node l computes $\mathbf{y}_l^{k+1}, \bar{\mathbf{y}}_l^{k+1}$

Example: distributed dual LASSO

Recall

$$\min_{\mathbf{y},\mathbf{z}}\{\mathbf{b}^T\mathbf{y} + \frac{\mu}{2}\|\mathbf{y}\|_2^2 + l_{\{\|\mathbf{z}\|_\infty \le 1\}} : \mathbf{A}^T\mathbf{y} + \mathbf{z} = \mathbf{0}\}$$

Apply distributed ADMM II

- decompose $\mathbf{A}^T$ to row blocks, equivalently, $\mathbf{A}$ to column blocks.
- make copies of $\mathbf{y}$
- parallel computing $+$ MPI(gathering and scatting vectors of size dim($\mathbf{y}$))

Recall distribute ADMM I

- decompose $\mathbf{A}$ to row blocks.
- make copies of $\mathbf{x}$
- parallel computing $+$ MPI (gathering and scatting vectors of size dim($\mathbf{x}$))

Between I and II, which is better?

- If $\mathbf{A}$ is fat
  - column decomposition in approach II is more efficient
  - the global variable of approach II is smaller
- If $\mathbf{A}$ is tall
  - row decomposition in approach I is more efficient
  - the global variable of approach I is smaller

Distributed ADMM II

A formulation with separable f and separable g

$$\min \sum_{j=1}^{N} f_j(\mathbf{x}_j) + \sum_{i=1}^{M} g_i(\mathbf{z}_i), \text{ s.t. } \mathbf{A}\mathbf{x} + \mathbf{z} = \mathbf{b},$$

where

$$\mathbf{x} = (\mathbf{x}_1, \mathbf{x}_2, \cdots, \mathbf{x}_N), \mathbf{z} = (\mathbf{y}_1, \mathbf{y}_2, \cdots, \mathbf{y}_M).$$

Decompose $\mathbf{A}$ in both directions as

$$\mathbf{A} = \begin{bmatrix} \mathbf{A}_{11} & \mathbf{A}_{12} & \cdots & \mathbf{A}_{1N} \\ \mathbf{A}_{21} & \mathbf{A}_{22} & \cdots & \mathbf{A}_{2N} \\ & & \cdots & \\ \mathbf{A}_{M1} & \mathbf{A}_{M2} & \cdots & \mathbf{A}_{MN} \end{bmatrix}, also\ \mathbf{b} = \begin{bmatrix} \mathbf{b}_1 \\ \mathbf{b}_2 \\ \vdots \\ \mathbf{b}_M \end{bmatrix}.$$

Same model:

$$\min \sum_{j=1}^{N} f_j(\mathbf{x}_j) + \sum_{i=1}^{M} g_i(\mathbf{z}_i), \text{ s.t. } \sum_{j=1}^{N} \mathbf{A}_{ij}\mathbf{x}_j + \mathbf{z}_i = \mathbf{b}_i, i = 1, \cdots, M.$$

Distributed ADMM III

$\mathbf{A}_{ij}\mathbf{x}_j'$s are coupled in the constraints. Standard treatment:

$$\mathbf{p}_{ij} = \mathbf{A}_{ij}\mathbf{x}_j.$$

New model:

$$\min \sum_{j=1}^{N} f_j(\mathbf{x}_j) + \sum_{i=1}^{M} g_i(\mathbf{z}_i), \text{ s.t. } \begin{array}{l} \sum_{j=1}^{N} \mathbf{p}_{ij} + \mathbf{z}_i = \mathbf{b}_i, \forall i, \\ \mathbf{p}_{ij} - \mathbf{A}_{ij}\mathbf{x}_j = 0, \forall i, j. \end{array}$$

ADMM

- alternate between $\{\mathbf{p}_{ij}\}$ and $(\{\mathbf{x}_j\}, \{\mathbf{z}_i\})$
- $\mathbf{p}_{ij}-$subproblems have closed-form solutions
- $(\{\mathbf{x}_j\}, \{\mathbf{z}_i\})$-subproblem are separable over all $\mathbf{x}_j$ and $\mathbf{z}_i$
  - $\mathbf{x}_j-$update involves f_j and $\mathbf{A}_{1j}^T\mathbf{A}_{1j}, \cdots, \mathbf{A}_{Mj}^T\mathbf{A}_{Mj}$;
  - $\mathbf{z}_i-$update involves g_i.
- ready for distributed implementation

Question: how to further decouple f_j and $\mathbf{A}_{1j}^T\mathbf{A}_{1j}, \cdots, \mathbf{A}_{Mj}^T\mathbf{A}_{Mj}$?

Distributed ADMM IV

For each $\mathbf{x}_j$, make M identical copies:$\mathbf{x}_{1j}, \mathbf{x}_{2j}, \cdots, \mathbf{x}_{Mj}$.
New model:

$$\min \sum_{j=1}^{N} f_j(\mathbf{x}_j) + \sum_{i=1}^{M} g_i(\mathbf{z}_i), \text{ s.t. } \begin{array}{ll} \sum_{j=1}^{N} \mathbf{p}_{ij} + \mathbf{z}_i = \mathbf{b}_i, & \forall i, \\ \mathbf{p}_{ij} - \mathbf{A}_{ij}\mathbf{x}_{ij} = \mathbf{0}, & \forall i, j, \\ \mathbf{x}_j - \mathbf{x}_{ij} = \mathbf{0}, & \forall i, j. \end{array}$$

ADMM

- alternate between $(\{\mathbf{x}_j\}, \{\mathbf{p}_{ij}\})$ and $(\{\mathbf{x}_j\}, \{\mathbf{z}_i\})$
- $(\{\mathbf{x}_j\}, \{\mathbf{p}_{ij}\})$-subproblem are separable
  - $\mathbf{x}_j-$update involves f_j only; computes prox_{f_j}
  - $\mathbf{p}_{ij}-$update is in closed form
- $(\{\mathbf{x}_{ij}\}, \{\mathbf{z}_i\})$-subproblem are separable
  - $\mathbf{x}_{ij}-$update involves $(\alpha I + \beta \mathbf{A}_{ij}^T \mathbf{A}_{ij})$;
  - $\mathbf{y}_i-$update involves g_i only; computes prox_{g_i}.
- ready for distributed implementation

Outline

Stochastic Gradient Descent

Consider

$$\min_{x \in \mathbb{R}^n} f(x) = \sum_{e \in E} f_e(x_e)$$

- e denotes a small subset of $\{1, 2, \ldots, n\}$.
- In many of machine learning problems, n and $|E|$ are both large. **Note**: the 'set' E can contain a few copies of the same element e.
- f_e only acts on a few components of x, say x_e.
- f_e can easily be regarded as a function over $\mathbb{R}^n$, just by ignoring the components not in the subset e.

Problem Definition

Example: machine learning applications

- Minimize the empirical risk

$$\min_x \frac{1}{n} \sum_{i=1}^{n} l_i(a_i^T x)$$

- a_i represents the ith data point, x is the model. l_i is a loss function.
- Logistic regression, least squares, SVM ...
- If each a_i is sparse, then it becomes our problem today.

Problem Definition

Example: generic minimization problem

- Minimize the following function

$$\min_{x_1,\ldots,x_{m_1}} \min_{y_1,\ldots,y_{m_2}} \sum_{i=1}^{m_1} \sum_{j=1}^{m_2} \phi_{i,j}(x_i, y_j)$$

- $\phi_{i,j}$ is a convex scalar function. x_i and y_j are vectors.
- For matrix completion and matrix factorization:

$$\phi_{i,j} = (A_{i,j} - x_i^T y_j)^2$$

- $n = m_1 m_2$ functions, each of which interacts with only two variables.
- A variable is shared among at most $m_1 + m_2$ functions.

Stochastic Gradient Descent

- Gradient method is given by

$$x_{k+1} = x_k - \gamma_k \nabla f(x_k)$$

  where $\nabla f(x_k) = \sum_{e \in E} \nabla f_e(x_k)$.

- Stochastic Gradient Descent(SGD)

$$x_{k+1} = x_k - \gamma_k \nabla f_{s_k}(x_k)$$

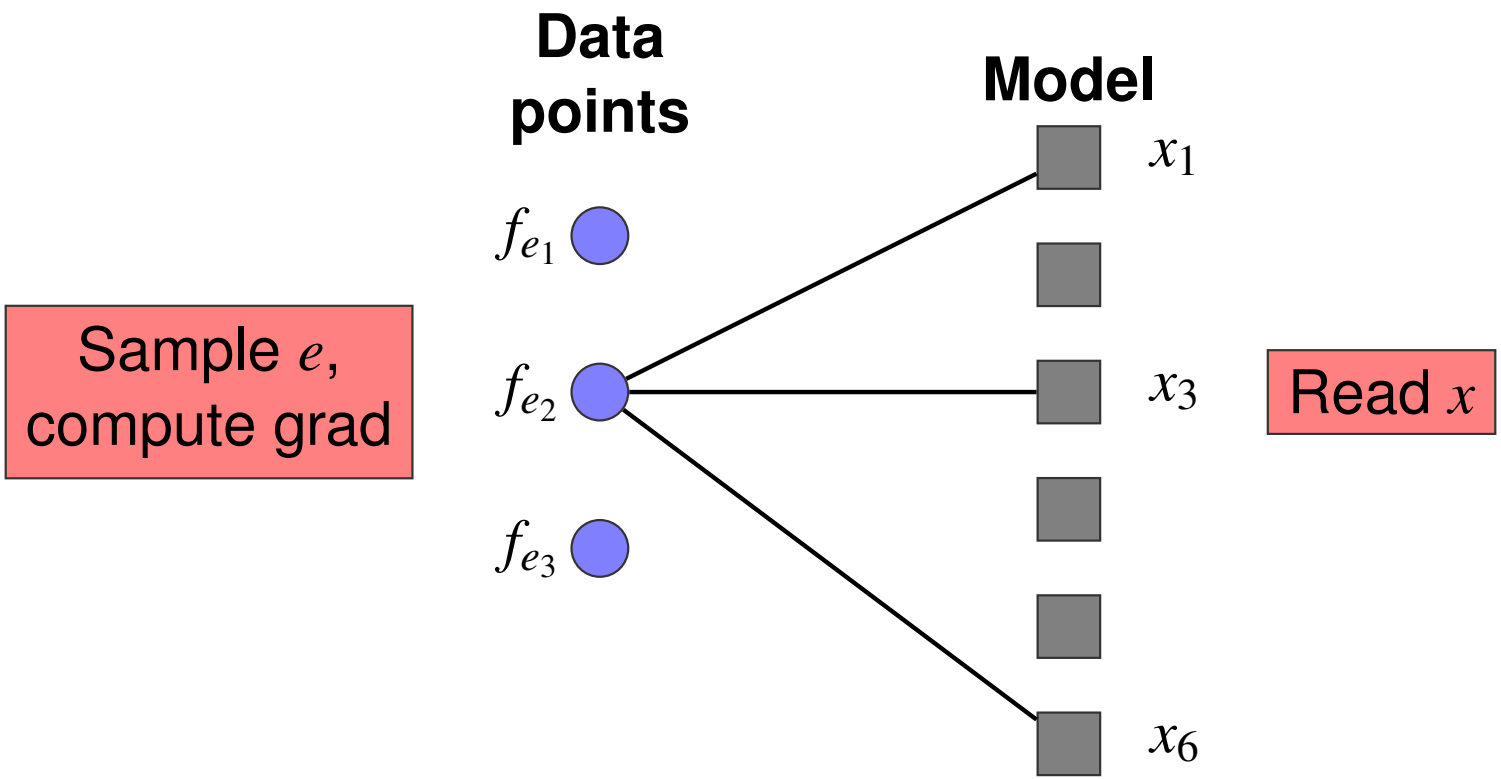
  where s_k is randomly sampled from E.

- γ_k is the step size(or learning rate). Can be a constant or shrinking.

- Idea of SGD: computing the entire $\nabla f(x)$ may be too costly for large $|E|$ and n. Can we compute just a small proportion of $\nabla f(x)$ and ensure the convergence?

Stochastic Gradient Descent

One step of SGD:



SGD: Advantages

SGD has been a round for a while, for good reasons:

- Less computation than classic GD.
- Robust to noise.
- Simple to implement.
- Near-optimal learning performance.
- Small computational foot-print.
- Stable.

Parallel SGD

- It may takes a **HUGE** number of updates for SGD on large datasets.
- **Goal**: a parallel version of SGD.
- **How to parallelize SGD?**
  - Parallelize one update – computing $\nabla f_{s_k}(x_k)$ is cheap, even for deep nets. Thus it may be not worth working out parallel codes on a cheap subroutine.
  - Parallelize the updates – SGD is sequential, making it nearly impossible for the parallel stuff.
  - Can we parallelize a sequential algorithm?

Parallel SGD

Can we parallelize a sequential algorithm?

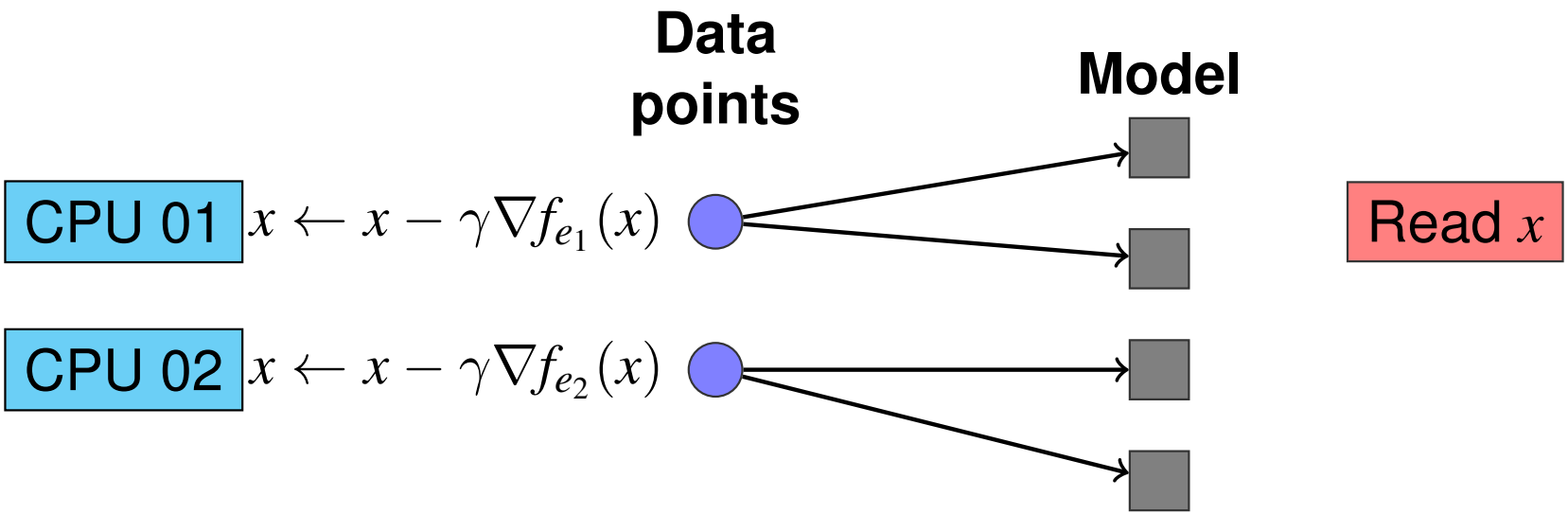
- No – for most cases.
- Almost yes – for problems with structures of sparsity.
- For our problem:

$$\min_{x \in \mathbb{R}^n} f(x) = \sum_{e \in E} f_e(x_e)$$

  If most of f_e's don't share the same component of x, may be we can exploit the sparsity to parallelize SGD.

Parallel SGD

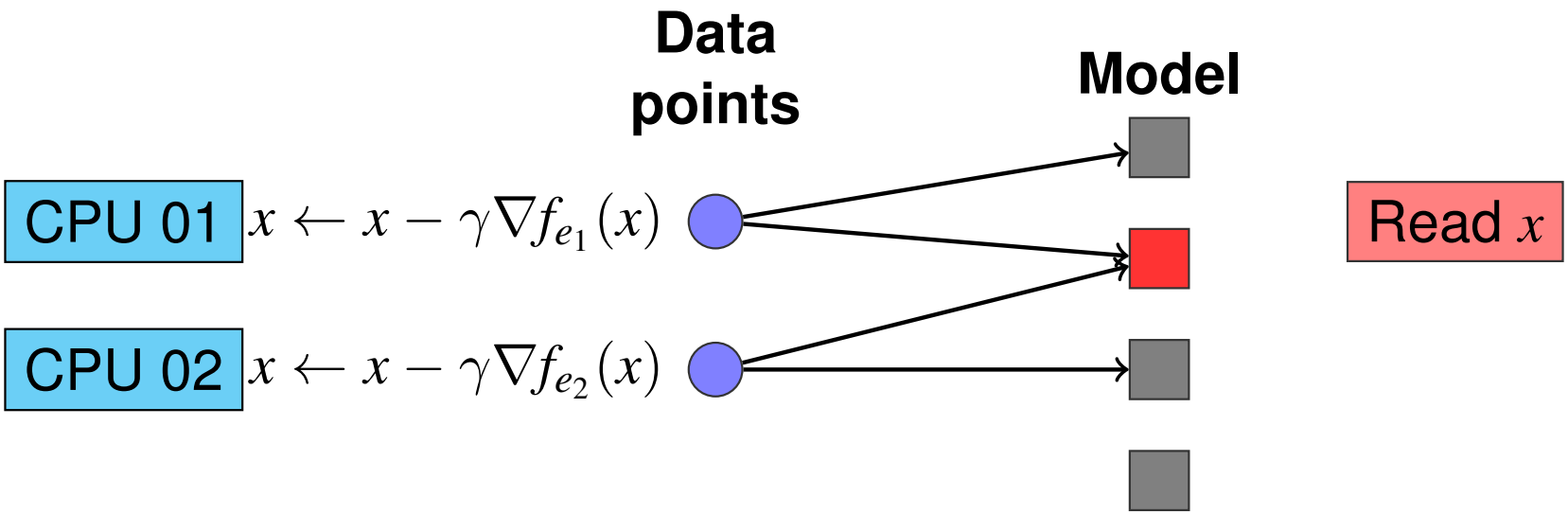
Good Case:



- The components of f_{e_1} and f_{e_2} don't overlap.
- Two parallel updates means two serial updates.
- No conflicts means speed up!

Parallel SGD

Bad Case:



- The components of f_{e_1} and f_{e_2} overlap at x_2.
- Conflicts mean less parallelization.
- What should be done to handle the conflicts?

Parallel SGD

Why conflicts affect so much?

- CPUs don't have direct access to the memory.
- Computations can only be done on CPU cache.
- Data are read from the memory to CPU cache. After the computations are finished, results are pushed back to the memory.
- A CPU has no idea whether its 'colleagues' have local updates which have not been pushed to the memory.

Parallel SGD

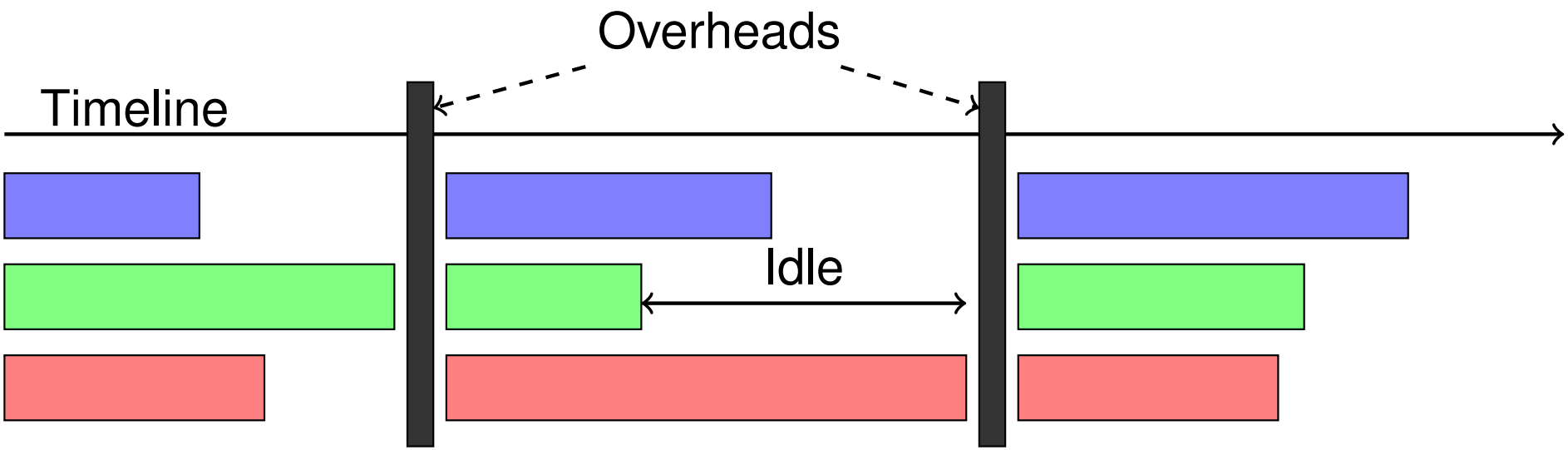
How to deal with the conflicts?

- By ‘coordinate’ or ‘memory lock’ approach.
- **Lock type**: exclusive lock.
- Tell others: I’m updating the variable NOW. Please don’t make any changes until I have finished.
- Ensure the correctness when performing parallel SGD.
- Provide only limited speedup for making parallel SGD as a ‘sequential’ program. Things are even worse when conflicts occur too often – even slower than the sequential version of SGD!

Asynchronous Updates

Q: How to deal with the conflicts?
A: Asynchronous programming tells us to just ignore it.

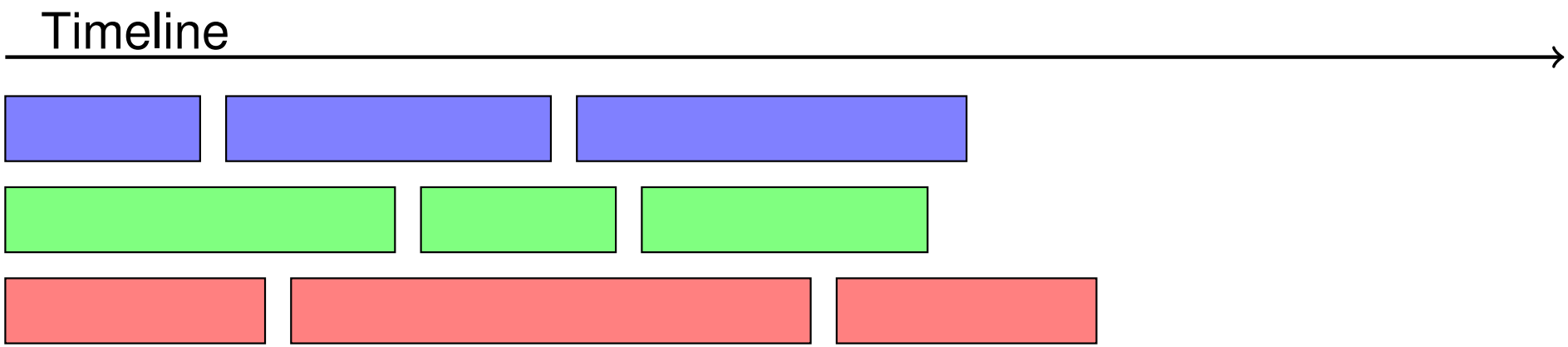
The synchronous world:



- Load imbalance causes the idle.
- Correct but slow.

Asynchronous Updates

The asynchronous world:



- No synchronizations among the workers.
- No idle time – every worker is kept busy.
- High scalability.
- Noisy but fast.

Hogwild! the Asynchronous SGD

If we remove all locking and synchronizing processes in parallel SGD. Then we obtain the Hogwild! algorithm.

Algorithm 1 Hogwild! the Asynchronous SGD

Each processor asynchronously do
loop
    Sample e randomly from E.
    Read the current point x_e from the memory.
    Compute $G_e(x) := |E|\nabla f_e(x)$.
    for $v \in e$ **do**
        $x_v \leftarrow x_v - \gamma b_v^T G_e(x)$.
    end for
end loop

Hogwild! the Asynchronous SGD

A variant of Hogwild!:

Algorithm 2 Hogwild! the Asynchronous SGD(Variant 1)

1: Each processor asynchronously do
2: **loop**
3: Sample e randomly from E.
4: Read the current point x_e from the memory.
5: Compute $G_e(x) := |E|\nabla f_e(x)$.
6: Sample $v \in e$, then $x_v \leftarrow x_v - \gamma |e| b_v^T G_e(x)$.
7: **end loop**

Note:

- The entire gradient is computed. Only one component is updated.
- The $|e|$ factor ensures $\mathbb{E}[|e| b_v^T G_e(x)] = \nabla f(x)$.
- Seems wasteful, but easy to analyze.

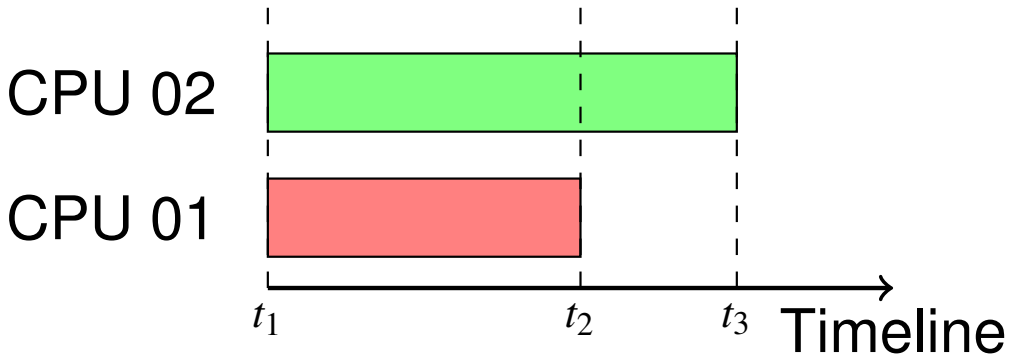
Issues of the Asynchronous Updates

- The inconsistency between the CPU cache and the memory makes the results incorrect.
- Older updates can be overwritten by newer ones.
- Suppose we want to perform two updates with two threads

$$\begin{aligned} x_3 &= x_2 - \gamma \nabla f_{e_2}(x_2) \\ &= x_1 - \gamma(\nabla f_{e_2}(x_2) + \nabla f_{e_1}(x_1)) \end{aligned}$$

- The computation of $\nabla f_{e_1}(x_1)$ and $\nabla f_{e_2}(x_2)$ is assigned to CPU1 and CPU2 respectively.

Issues of the Asynchronous Updates



time	x in ∇f		x in mem	∇f in cache		what	
t_1	x_1	x_1	x_1	—	—	read x_1 from memory	
t_2	x_2	x_1	$x_1 - \gamma \nabla f_{e_1}(x_1)$	$\nabla f_{e_1}(x_1)$	—	Update mem.	Computing
t_3	x_2	x_2	$x_2 - \gamma \nabla f_{e_2}(x_1)$	$\nabla f_{e_1}(x_1)$	$\nabla f_{e_2}(x_1)$	Idle	Update mem.

- What we should get: $x_1 - \gamma(\nabla f_{e_1}(x_1) + \nabla f_{e_2}(x_2))$.
- What we actually get: $x_1 - \gamma(\nabla f_{e_1}(x_1) + \nabla f_{e_2}(x_1))$.

Analysis of Hogwild!

Assumptions

- ∇f is Lipschitz continuous.

$$\|\nabla f(x) - \nabla f(y)\| \leqslant L\|x - y\|$$

- f is strongly convex with modulus c. Each f_e is convex.

$$f(y) \geqslant f(x) + (y - x)^T \nabla f(x) + \frac{c}{2}\|y - x\|^2$$

- The gradient of f_e is bounded. Recall that $G_e(x) = |E|\nabla f_e(x)$.

$$\|G_e(x)\| \leqslant M, \ \forall \ x$$

- $\gamma c < 1$, otherwise even gradient descent alg will fail.
- Based on the Hogwild!(Variant 1) algorithm.

Proposition 4.1 (Main result).

Suppose the lag between when a gradient is computed and when it is used in step j – namely $j - k(j)$ – is always less or equal than τ, and γ is defined to be

$$\gamma = \frac{\theta \varepsilon c}{2LM^2\Omega(1 + 6\rho\tau + 4\tau^2\Omega\Delta^{1/2})} \tag{1}$$

for some $\varepsilon > 0$ and $\theta \in (0, 1)$. Define $D_0 = \|x_0 - x_\star\|^2$ and let k be an integer satisfying

$$k \geqslant \frac{2LM^2\Omega(1 + 6\tau\rho + 6\tau^2\Omega\Delta^{1/2}) \log(LD_0/\varepsilon)}{c^2\theta\varepsilon} \tag{2}$$

Then after k component updates of x, we have $\mathbb{E}[f(x_k) - f_\star] \leqslant \varepsilon$.

Remarks on Prop 4.1

- Consider the Hogwild! algorithm as an SGD with lags.

$$x_{j+1} \leftarrow x_j - \gamma |e| \mathcal{P}_v G_e(x_{k(j)})$$

- The lags should be bounded by τ. That is, at the jth update, the point used to compute the gradient is within the last τ steps.
- τ is proportional to the number of threads.
- The step size γ should be $\mathcal{O}(\varepsilon)$ to ensure the convergence.
- To obtain an accuracy of ε, we need at least $\mathcal{O}(1/\varepsilon)$ updates, modulo the $\log(1/\varepsilon)$ factor.

Outline

Problem Definition

Consider

$$\min_{x \in \mathbb{R}^d} f(x) = \sum_{e \in E} f_e(x_e)$$

- e denotes a small subset of $\{1, 2, \ldots, d\}$.
- In many of machine learning problems, d and $n = |E|$ are both large.
- f_e only acts on a few components of x, say x_e.
- f_e can easily be regarded as a function over $\mathbb{R}^d$, just by ignoring the components not in the subset e.

Algorithms to Solve the Problem

- HOGWILD! (Asynchronous SGD):

$$x_{k+1} \leftarrow x_k - \gamma_k \nabla f_{s_k}(x_k)$$

- All cores perform the updates asynchronously with no memory locking.
- Ignores the conflict variables.
- No convergence in some cases. (Lose some of the properties of SGD)
- Complicated theoretical analysis |>_<|

Another Way?

To deal with conflicts:

- Traditional parallelizations
  - Lock the conflict variables when updating.
  - Ensure the correctness, but only limited speedup.
- HOGWILD!
  - Ignore the conflicts when updating.
  - Better speedup, but higher risk of not converging.
- CYCLADES
  - Avoid the conflicts by rearranging the updates.
  - Better speedup.
  - Preserve the property of SGD in high probability.

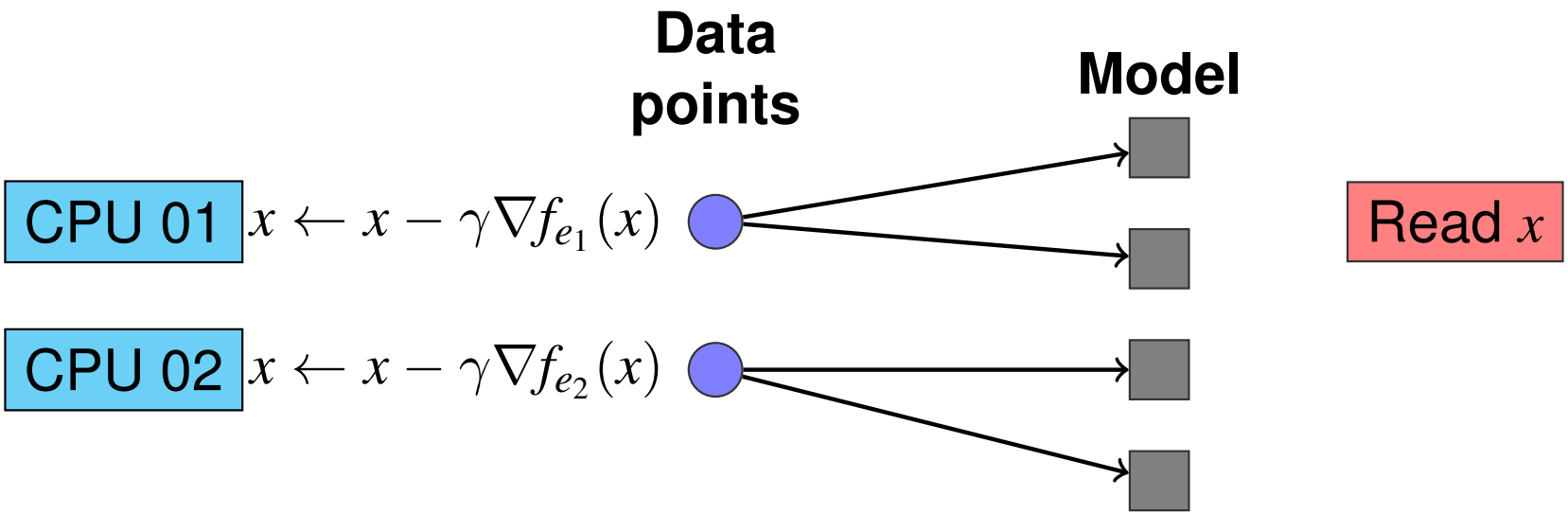
About the Conflicts

Why conflicts affect so much?

- CPUs don't have direct access to the memory.
- Computations can only be done on CPU cache.
- Data are read from the memory to CPU cache. After the computations are finished, results are pushed back to the memory.
- A CPU has no idea whether its 'colleagues' have local updates which have not been pushed to the memory.

About the Conflicts

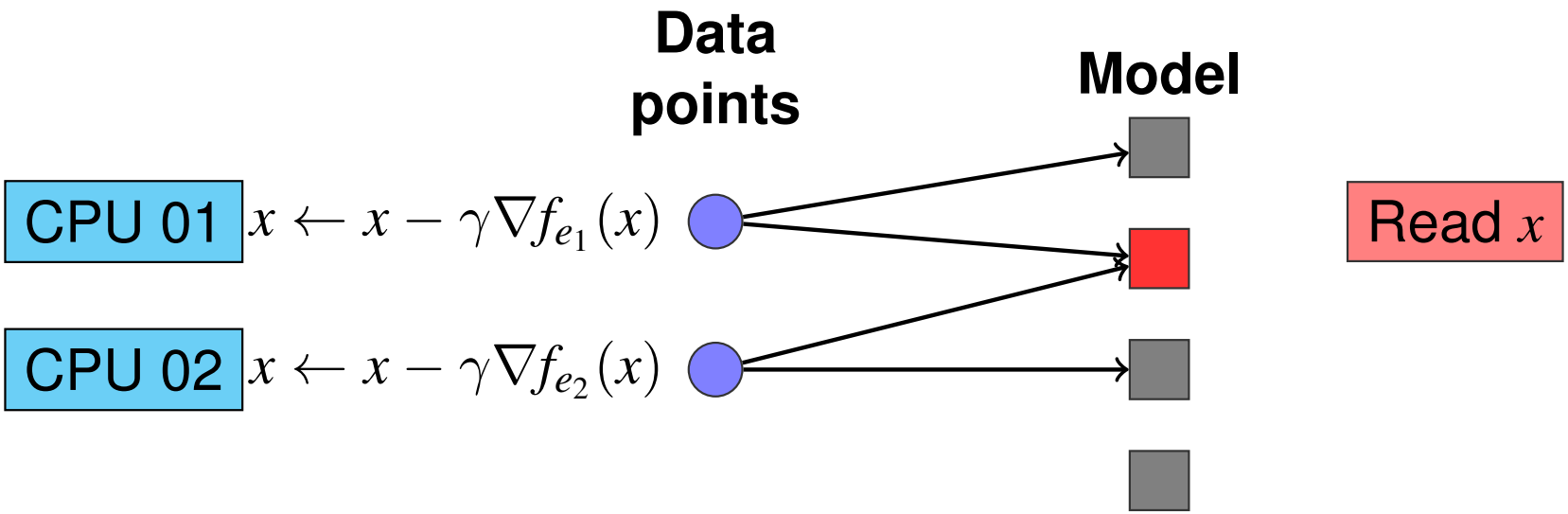
Good Case:



- The components of f_{e_1} and f_{e_2} don't overlap.
- Two parallel updates means two serial updates.
- No conflicts means speed up!

About the Conflicts

Bad Case:



- The components of f_{e_1} and f_{e_2} overlap at x_2.
- Conflicts mean less parallelization.
- What should be done to handle the conflicts? HOGWILD! tells us to ignore it. CYCLADES tells us to avoid it.

The Updates Conflict Graph

Definition 1 (Bipartite update-variable graph).

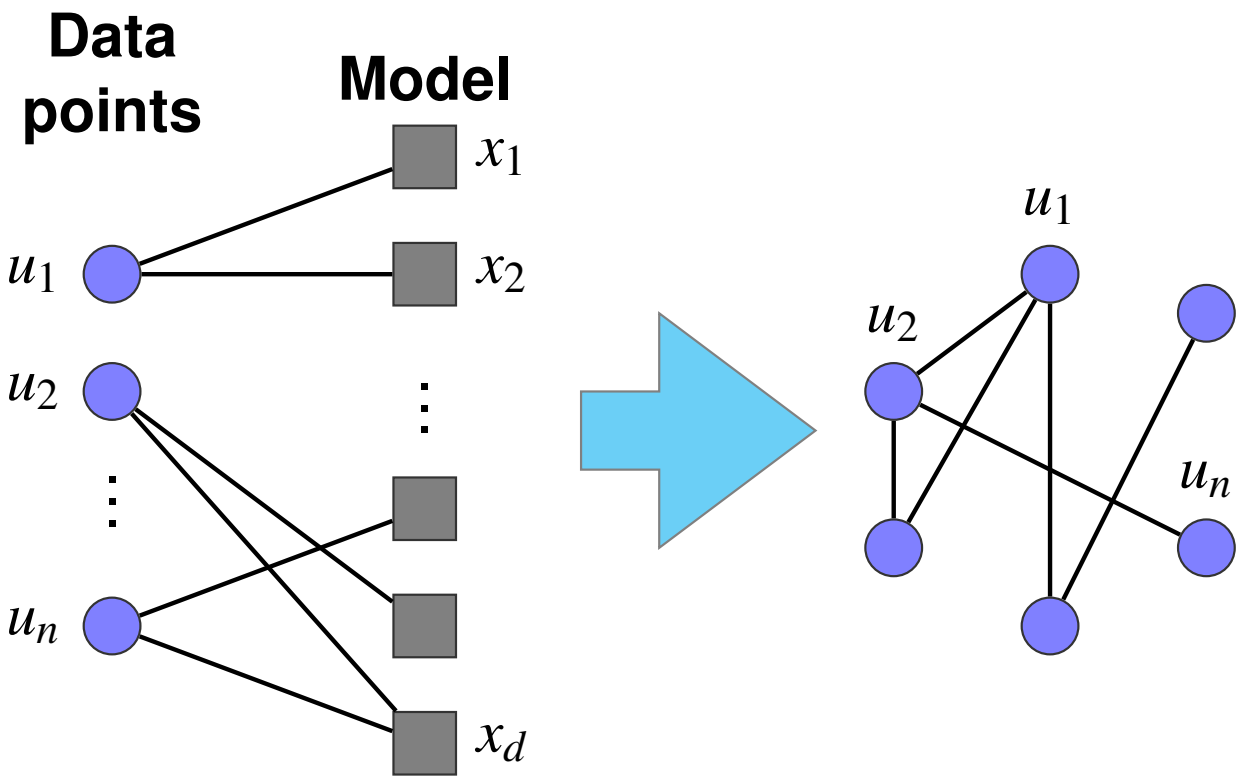
We denote as G_u the bipartite update-variable graph between the updates $u_1, \ldots, u_n$ and the d variables. In G_u an update u_i is linked to a variable x_j, if u_i requires to read and write x_j. E_u denotes the number of edges in the bipartite graph. Δ_L denotes the left max vertex degree of G_u. $\bar{\Delta}_L$ denotes its average left degree.

Definition 2 (Conflict Graph).

We denot by G_c a conflict graph on n vertices, each corresponding to an update u_i. Two vertices of G_c are linked with an edge, if and only if the corresponding updates share at least one variable in the bipartite-update graph G_u. Δ is denoted as the max vertex degree of G_c.

The Updates Conflict Graph

From a bipartite graph to a conflict graph



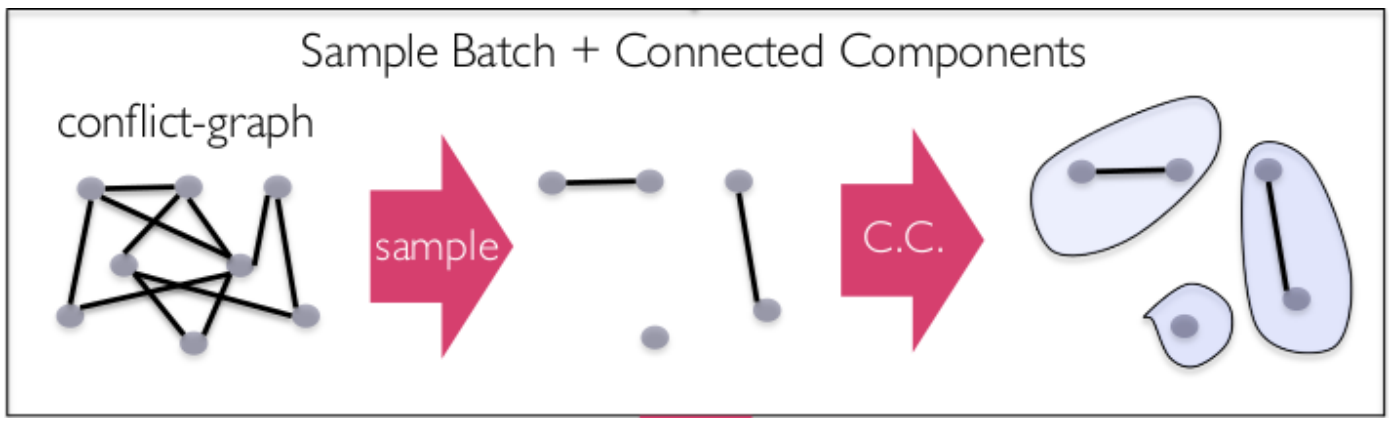
The Updates Conflict Graph

Remarks

- In the conflict graph, if two vertices are linked with an edge, then their corresponding updates conflict with each other.
- G_c is introduced to analyze the CYCLADES algorithm. In practice we never construct it.
- Suppose the number of connected components(CC) is N_{cc}, then we can employ N_{cc} workers to process the updates asynchronously. No conflicts occur. **However,** N_{cc} **equals to 1 for most problems!**

The Idea of CYCLADES

- We need more CCs to perform parallelization.
- Just sample part of the updates.



How many CCs we can get if we perform the sampling?

The Number of CCs

Theorem 1.

Let G be a graph on n vertices, with maximum vertex degree Δ. Let us sample each vertex independently with probability $p = \frac{1-\varepsilon}{\Delta}$ and define G' as the induced subgraph on sampled vertices. Then, the largest CC of G' has the size at most $\frac{4}{\varepsilon^2} \log n$, with high probability.

Theorem 2.

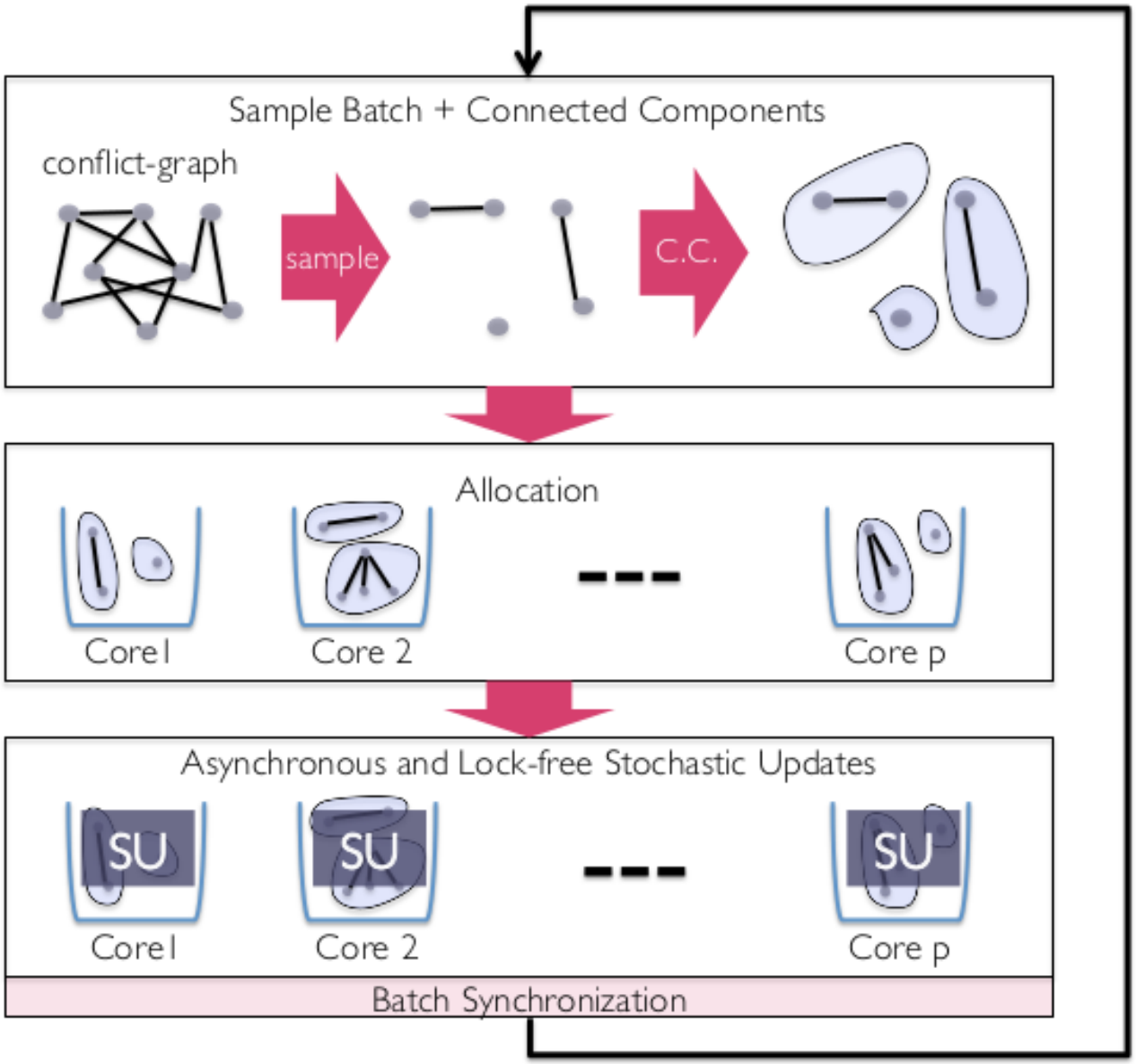
Let G be a graph on n vertices, with maximum vertex degree Δ. Let us sample $B = (1-\varepsilon)\frac{n}{\Delta}$, with or without replacement, and define G' as the induced subgraph on sampled vertices. Then, the largest CC of G' has the size at most $\mathcal{O}\left(\frac{\log n}{\varepsilon^2}\right)$, with high probability.

The Number of CCs

Remarks

- Theorem 2 can be regard as a corollary of Theorem 1(Not obvious!). We'll give a brief proof of Theorem 1 later.
- From Theorem 2, if one samples $B = (1-\varepsilon)\frac{n}{\Delta}$ vertices, then there will be at least $\mathcal{O}(\varepsilon^2 B/\log n)$ CCs, each of size at most $\mathcal{O}(\log n/\varepsilon^2)$.
- The number of CCs is increased a lot – OK. Good for parallelization.

The Idea of CYCLADES



The Idea of CYCLADES

- After sampling, use certain algorithm to find all CCs in a parallel way(to be continued).
- When all CCs are determined, allocate them to the workers(consider the load balance).
- Each worker performs stochastic updates asynchronously and individually, with no conflicts and false sharing issues.
- Repeat the three phases until finished.

CYCLADES Outline

Algorithm 3 CYCLADES

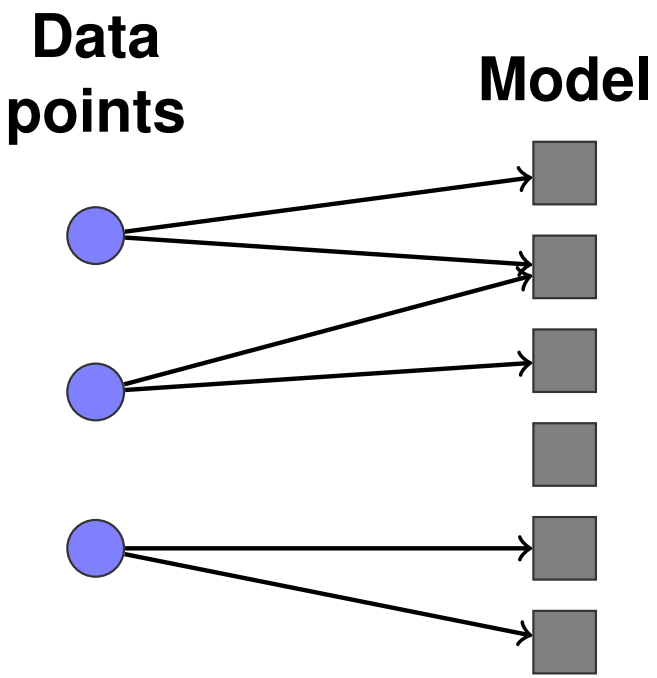
1: **Input**: G_u, T, B.
2: Sample $n_b = T/B$ subgraphs from G_u.
3: **for** batch $i = 1 : n_b$ **do**
4: Allocate CCs to P cores.
5: **for** each core asynchronously **do**
6: Perform local updates in allocated CCs.
7: **end for**
8: **end for**

- T is the total updates you'd like to run.
- One synchronization at the end of each batch.
- G_u is determined by the structure of problem.

Compute CCs of All Batches in Parallel

- If we use the conflict graph G_c to compute the CCs...
- The cost is the order of sampled edges.
- However, constructing G_c requires n^2 time. A high cost – no, thanks.
- Compute CCs using the bipartite graph only!

Compute CCs of All Batches in Parallel



Simple message passing idea

- Data points send their label.
- Variables compute min and send back.
- Data points compute min.
- Iterate until you're done.

Compute CCs of All Batches in Parallel

Remarks

- The iterations needed are bounded with the length of the longest-shortest path(i.e. the diameter of G_c).
- Overall complexity: $\mathcal{O}(E_u \log n/P)$, P is the number of cores. Note: useful when P is large.
- Only need the bipartite graph G_u, not the conflict graph G_c.

Allocating the Conflict Groups

Once we obtain the CCs of the batch, we have to allocate these CCs to P cores.

- w_i is the cost of updating with the ith data point.
- $W_{C(i)} = \sum_{j \in C(i)} w_j$ is the cost of updating the ith CC.
- For each batch update, we need to

$$\min \max W_{C(i)}$$

  to get the best load balance.
- An NP hard problem. We can utilize an approximation algorithm to obtain a sub-optimal plan.

Allocating the Conflict Groups

Algorithm 4 Greedy allocation

1: Estimate $W_1, \ldots, W_m$, the cost of each CC.
2: Sort the W_i's in the descending order.
3: **for** $i = 1 : m$ **do**
4: Choose the currently largest W_i.
5: Add W_i to the cores with least sum of cost currently.
6: **end for**

- A $4/3$-approximation algorithm.
- w_i is proportional to the out-degree of that update.

Main Results

Theorem 3 (Main result).

Let us assume any given update-variable graph G_u with average, and max left degree $\bar{\Delta}_L$ and Δ_L, such that $\Delta_L/\bar{\Delta}_L \leqslant \sqrt{n}$, and with induced max conflict degree Δ. Then, CYCLADES on $P = \mathcal{O}(n/\Delta\Delta_L)$ cores with batch sizes $B = (1-\varepsilon)\frac{n}{\Delta}$ can execute $T = cn$ updates, for any $c > 1$, in time

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

with high probability.

- κ is the cost to update one edge in E_u.
- CYCLADES can achieve the same result as the serial algorithms with high probability. Thus it requires similar properties of the objective function f and its components f_e.

Main Results

Remarks of Thm 3

- The sampling of each batch can be with or without replacement.
- The number of cores is bounded, for it's hard to allocate the work of computing CCs evenly if P is too large.
- The batch size B should be bounded. This is very important. Small B will induce a subgraph with many CCs.
- Theorem 1 is the most important foundation to develop the main result.