Those who complete this lecture will know

- basics of parallel computing
- how to parallel a bottleneck of existing sparse optimization method
- primal and dual decomposition
- GRock: greedy coordinate-block descent, and its parallel implementation
1. Background of parallel computing: benefits, speedup, and overhead

2. Parallelize existing algorithms

3. Primal decomposition / dual decomposition
   - Parallel dual gradient ascent (linearized Bregman)
   - Parallel ADMM

4. Parallel greedy coordinate descent

5. Numerical results with big data
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 ad, recommendation, 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/arithemtic 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
• 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 *increase* 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, MPMD, etc.
Parallel speedup

• **definition:**
  \[
  \text{speedup} = \frac{\text{serial time}}{\text{parallel time}}
  \]
  
  time is in the *wall-clock* 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

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

- $\rho$ may also depend on $N$
Parallel speedup

- \( \varepsilon := \) parallel overhead
- **in the real world**

\[
\text{actual speedup} = \frac{1}{(\rho/N) + (1 - \rho) + \varepsilon}
\]

<table>
<thead>
<tr>
<th>Number of processors</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>25%</td>
</tr>
<tr>
<td></td>
<td>50%</td>
</tr>
<tr>
<td></td>
<td>90%</td>
</tr>
<tr>
<td></td>
<td>95%</td>
</tr>
</tbody>
</table>

when \( \varepsilon = N \)

when \( \varepsilon = \log(N) \)
Basic types of communication

- **Broadcast**
- **Scatter**
- **Gather**
- **Reduce**

1 2 3 3

9
Allreduce sum via butterfly communication

\[ \log(N) \text{ layers, } N \text{ parallel communications per layer} \]
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: $\pi$, area computation

- box area is $1 \times 1$
- 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 $\pi$ 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: \( P = \text{find\_total\_parallel\_processors} \)
4: \( i = \text{find\_my\_processor\_id} \)
5: total = total/P
6: for \( k = 1 \) to total do
7: \( x \leftarrow \) a uniformly random point in \([0,1]^2\)
8: if \( x \) is in the disc then
9: \( \text{in\_disc}++ \)
10: end if
11: end for
12: if \( i == 1 \) then
13: total_in_disc = reduce(in_disc, SUM)
14: return \( 4*\text{total\_in\_disc}/(\text{total}*P) \)
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 synchronization.
Outline

1. Background of parallel computing: benefits, speedup, and overhead

2. Parallelize existing algorithms

3. Primal decomposition / dual decomposition
   - Parallel dual gradient ascent (linearized Bregman)
   - Parallel ADMM

4. Parallel greedy coordinate descent

5. Numerical results with big data
Decomposable objective and constraints enables parallelism

- variables $x = [x_1, x_2, \ldots, x_N]^T$

- embarrassingly parallel (not an interesting case): $\min \sum_i f_i(x_i)$

- consensus minimization, no constraints

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

**LASSO example:** $f(x) = \frac{\lambda}{2} \sum_i \|A_{(i)}x - b_i\|^2_2 + \|x\|_1$

- coupling variable $z$

$$\min_{x,z} f(x, z) = \sum_{i=1}^{N} [f_i(x_i, z) + r_i(x_i)] + r(z)$$
• coupling constraints

\[ \min f(x) = \sum_{i=1}^{N} f_i(x_i) + r_i(x_i) \]

subject to

• equality constraints: \( \sum_{i=1}^{N} A_i x_i = b \)
• inequality constraints: \( \sum_{i=1}^{N} A_i x_i \leq b \)
• graph-coupling constraints \( Ax = b \), where \( A \) is sparse

• combinations of independent variables, coupling variables and constraints

\textit{example:}

\[
\min_{x, w, z} \sum_{i=1}^{N} f_i(x_i, z)
\]

\[ \text{s.t. } \sum_{i} A_i x_i \leq b \]

\[ B_i x_i \leq w, \forall i \]

\[ w \in \Omega \]

• variable coupling \( \longleftrightarrow \) constraint coupling:

\[
\min \sum_{i=1}^{N} f_i(x_i, z) \rightarrow \min \sum_{i=1}^{N} f_i(x_i, z_i) \quad \text{s.t. } z_i = z \quad \forall i
\]
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
Outline

1. Background of parallel computing: benefits, speedup, and overhead

2. **Parallelize existing algorithms**

3. Primal decomposition / dual decomposition
   - Parallel dual gradient ascent (linearized Bregman)
   - Parallel ADMM

4. Parallel greedy coordinate descent

5. Numerical results with big data
Example: parallel ISTA

\[
\text{minimize } \lambda \|x\|_1 + f(x)
\]

- Prox-linear approximation:

\[
\arg \min_x \lambda \|x\|_1 + \langle \nabla f(x), x - x^k \rangle + \frac{1}{2\delta_k} \|x - x^k\|^2_2
\]

- Iterative soft-threshold algorithm:

\[
x^{k+1} = \text{shrink} \left( x^k - \delta_k \nabla f(x), \lambda \delta_k \right)
\]

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

- examples: \( f(x) = \frac{1}{2} \|Ax - b\|^2_2 \), logistic loss of \( Ax - b \), hinge loss ... where \( A \) is often dense and can go very large-scale
Parallel gradient computing of $f(x) = g(Ax + b)$: row partition

**Assumption:** $A$ is big and dense; $g(y) = \sum g_i(y_i)$ decomposable

Then $f(x) = g(Ax + b) = \sum g_i(A_i x + b_i)$

- let $A_i$ and $b_i$ be the $i$th row block of $A$ and $b$, respectively

\[
\begin{bmatrix}
A(1) \\
A(2) \\
\vdots \\
A(M)
\end{bmatrix}
\]

- store $A_i, b_i$ on node $i$

- goal: to compute $\nabla f(x) = \sum A_i^T \nabla g_i(A_i x + b_i)$, or similarly $\partial f(x)$
Parallel gradient computing of \( f(x) = g(Ax + b) \): row partition

**Steps**

1. compute \( A_i x + b_i \) \( \forall i \) in parallel;
2. compute \( g_i = \nabla g_i (A_i x + b_i) \) \( \forall i \) in parallel;
3. compute \( A_i^T g_i \) \( \forall i \) in parallel;
4. allreduce \( \sum_{i=1}^{M} A_i^T g_i \).

- step 1, 2 and 3 are local computation and communication-free;
- step 4 requires an allreduce sum communication.
Parallel gradient computing of $f(x) = g(Ax + b)$: column partition

\[
\begin{bmatrix}
A_1 & A_2 & \cdots & A_N
\end{bmatrix}
\]

column block partition

\[
Ax = \sum_{i=1}^{N} A_i x_i \implies f(x) = g(Ax + b) = g(\sum A_i x_i + b)
\]

\[
\nabla f(x) = \begin{pmatrix}
A_1^T \nabla g(Ax + b) \\
A_2^T \nabla g(Ax + b) \\
\vdots \\
A_N^T \nabla g(Ax + b)
\end{pmatrix}, \text{ similar for } \partial f(x)
\]
Parallel computing of $\nabla f(x) = g(Ax + b)$: column partition

Steps

1. compute $A_i x_i + \frac{1}{N} b \ \forall i$ in parallel
2. allreduce $Ax + b = \sum_{i=1}^{N} A_i x_i + \frac{1}{N} b$;
3. evaluate $\nabla g(Ax + b) \ \forall i$ in each process;
4. compute $A_i^T \nabla g(Ax + b) \ \forall i$ in parallel.

- each processor keeps $A_i$ and $x_i$, as well as a copy of $b$;
- step 2 requires an allreduce sum communication;
- step 3 involves duplicated computation in each process (hopefully, $g$ is simple).
Two-way partition

\[
\begin{bmatrix}
A_{1,1} & A_{1,2} & \cdots & A_{1,N} \\
A_{2,1} & A_{2,2} & \cdots & A_{2,N} \\
\vdots & \vdots & \ddots & \vdots \\
A_{M,1} & A_{M,2} & \cdots & A_{M,N}
\end{bmatrix}
\]

\[
f(x) = \sum g_i (A_{(i)} x + b_i) = \sum g_i \left( \sum_{j} A_{i,j} x_j + b_i \right)
\]

\[
\nabla f(x) = \sum_i A_{(i)}^T \nabla g_i (A_{(i)} x + b_i) = \sum_i A_{(i)}^T \nabla g_i \left( \sum_{j} A_{i,j} x_j + b_i \right)
\]
Two-way partition

Step

1. compute \( A_{i,j} x_j + \frac{1}{N} b_i \) \( \forall i \) in parallel;
2. allreduce \( \sum_{j=1}^{N} A_{i,j} x_j + b_i \) for every \( i \);
3. compute \( g_i = \nabla g_i (A_{(i)} x + b_i) \) \( \forall i \) in parallel;
4. compute \( A_{j,i}^T g_i \) \( \forall i \) in parallel;
5. allreduce \( \sum_{i=1}^{N} A_{j,i}^T g_i \) for every \( j \).

- processor \((i, j)\) keeps \( A_{i,j}, x_j \) and \( b_i \);
- step 2 and 5 require parallel allreduce sum communication;
Example: parallel ISTA for LASSO

LASSO

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

algorithm

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

Serial pseudo-code

1: initialize $x = 0$ and $\delta$;
2: pre-compute $A^T b$
3: while not converged do
4:   $g = A^T Ax - A^T b$
5:   $x = \text{shrink} \left( x - \delta g, \lambda \delta \right)$;
6: end while
Example: parallel ISTA for LASSO

distribute blocks of rows to $M$ nodes

\[
\begin{bmatrix}
A(1) \\
A(2) \\
\vdots \\
A(M)
\end{bmatrix}
\]

1: initialize $x = 0$ and $\delta$;
2: $i = \text{find\_my\_processor\_id}$
3: processor $i$ loads $A(i)$
4: pre-compute $\delta A^T b$
5: while not converged do
6: processor $i$ computes $c_i = A^T(i) A(i) x$
7: $y = \text{allreduce}(c_i, \text{SUM})$
8: $x = \text{shrink}(x^k - \delta c + \delta A^T b, \lambda \delta)$;
9: end while

- assume $A \in \mathbb{R}^{m \times n}$
- 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

\[
\begin{bmatrix}
A_1 & A_2 & \cdots & A_N
\end{bmatrix}
\]

1: initialize $x = 0$ and $\delta$;
2: $i = \text{find\_my\_processor\_id}$
3: processor $i$ loads $A_i$
4: pre-compute $\delta A_i^T b$
5: while not converged do
6: processor $i$ computes $y_i = A_i x_i$
7: $y = \text{allreduce}(y_i, \text{SUM})$
8: processor $i$ computes $z_i = A_i^T y$
9: $x_{i}^{k+1} = \text{shrink}(x_i^{k} - \delta z_i + \delta A_i^T b, \lambda \delta)$;
10: end while

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

1. Background of parallel computing: benefits, speedup, and overhead

2. Parallelize existing algorithms

3. Primal decomposition / dual decomposition\(^3\)
   - Parallel dual gradient ascent (linearized Bregman)
   - Parallel ADMM

4. Parallel greedy coordinate descent

5. Numerical results with big data

---

\(^3\)Dantzig and Wolfe [1960], Spingarn [1985], Bertsekas and Tsitsiklis [1997]
Unconstrained minimization with coupling variables

\[ \mathbf{x} = [x_1, x_2, \ldots, x_N]^T. \]

\( z \) is the coupling/bridging/complicating variable.

\[
\min_{\mathbf{x}, z} f(\mathbf{x}, z) = \sum_{i=1}^{N} f_i(x_i, z)
\]
equivalently to separable objective with coupling constraints

\[
\min_{\mathbf{x}, z} \sum_{i=1}^{N} f_i(x_i, z_i) \quad \text{s.t. } z_i = z \quad \forall i
\]

Examples

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

---

\(^4\) see tutorial: Palomar and Chiang [2006]
Primal decomposition (bi-level optimization)

Introduce easier subproblems

\[ g_i(z) = \min_{x_i} f_i(x_i, z), \quad i = 1, \ldots, N \]

then

\[ \min_{x, z} \sum_{i=1}^{N} f_i(x_i, z) \iff \min_{z} \sum g_i(z) \]

the RHS is called the **master problem**

**parallel computing**: iterate

1. fix \( z \), update \( x_i \) and compute \( g_i(z) \) in parallel
2. update \( z \) using a standard method (typically, subgradient descent)

- good for small-dimensional \( z \)
- fast if the master problem converges fast
Dual decomposition

Consider

$$\min_{x, z} \sum_i f_i(x_i, z_i)$$

s.t. $$\sum_i A_i z_i = b$$

Introduce

$$g_i(z) = \min_{x_i} f(x_i, z)$$

$$g_i^*(y) = \max_z y^T z - g_i(z) \quad // \text{convex conjugate}$$

Dualization:

$$\Leftrightarrow \min_{x, z} \max_y \sum_i f_i(x_i, z_i) + y^T (b - \sum_i A_i z_i)$$

(generalized minimax thm.) $$\Leftrightarrow \max_y \min_{y_i, x_i, z_i} \left\{ \sum_i \left( f_i(x_i, z_i) - y^T A_i z_i \right) + y^T b \right\}$$

$$\Leftrightarrow \max_y \left\{ \min_{z_i} \sum_i \left( g_i(z_i) - y^T A_i z_i \right) + y^T b \right\}$$

$$\Leftrightarrow \min_y \sum_i g_i^*(A_i^T y) - y^T b$$
Example: consensus and exchange problems

Consensus problem

\[
\min_{\mathbf{z}} \sum_{i} g_i(\mathbf{z})
\]

Reformulate as

\[
\min_{\mathbf{z}} \sum_{i} g_i(\mathbf{z}_i) \quad \text{s.t.} \quad \mathbf{z}_i = \mathbf{z} \quad \forall i
\]

Its dual problem is the exchange problem

\[
\min_{\mathbf{y}} \sum_{i} g_i^*(\mathbf{y}_i) \quad \text{s.t.} \quad \sum_{i} \mathbf{y}_i = 0
\]
Primal-dual objective properties

Constrained separable problem

$$\min_z \sum_i g_i(z) \quad \text{s.t.} \quad \sum_i A_i z_i = b$$

Dual problem

$$\min_y \sum_i g_i^*(A_i^T y) - b^T y$$

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

Example:

- $g_i(z) = |z_i|$ and $g_i^*(a_i^T y) = \nu\{-1 \leq a_i^T y \leq 1\}$
- $g_i(z) = \frac{1}{2}|z_i|^2$ and $g_i^*(a_i^T y) = \frac{1}{2}|a_i^T y|^2$
- $g_i(z) = \exp(z_i)$ and $g_i^*(a_i^T y) = (a_i^T y) \ln(a_i^T y) - a_i^T y + \nu\{a_i^T y \geq 0\}$
Outline

1. Background of parallel computing: benefits, speedup, and overhead

2. Parallelize existing algorithms

3. **Primal decomposition / dual decomposition**
   - Parallel dual gradient ascent (linearized Bregman\(^5\))
   - Parallel ADMM

4. Parallel greedy coordinate descent

5. Numerical results with big data

\(^5\)Yin, Osher, Goldfarb, and Darbon [2008]
Example: augmented $\ell_1$

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

Dual problem

$$\min_y \sum_i g_i^*(A_i^T y) - b^T y = \sum_i \frac{\alpha}{2} |\text{shrink}(A_i^T y)|^2 - b^T y$$

Dual gradient iteration (equivalent to linearized Bregman):

$$y^{k+1} = y^k - t^k \left( \sum_i \nabla h_i(y^k) - b \right), \quad \text{where} \ \nabla h_i(y) = \alpha A_i \text{shrink}(A_i^T y).$$

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

1. Background of parallel computing: benefits, speedup, and overhead

2. Parallelize existing algorithms

3. Primal decomposition / dual decomposition
   - Parallel dual gradient ascent (linearized Bregman)
   - Parallel ADMM\(^8\)

4. Parallel greedy coordinate descent

5. Numerical results with big data

---

\(^8\) Bertsekas and Tsitsiklis [1997]
Alternating direction method of multipliers (ADMM)

step 1: turn problem into the form of

$$\min_{x,y} \ f(x) + g(y)$$

s.t. \( Ax + By = b. \)

\( f \) and \( g \) are convex, maybe nonsmooth, can incorporate constraints

step 2: iterate

1. \( x^{k+1} \leftarrow \min_x f(x) + g(y^k) + \frac{\beta}{2} \| Ax + By^k - b - z^k \|_2^2, \)
2. \( y^{k+1} \leftarrow \min_y f(x^{k+1}) + g(y) + \frac{\beta}{2} \| Ax^{k+1} + By - b - z^k \|_2^2, \)
3. \( z^{k+1} \leftarrow z^k - (Ax^{k+1} + By^{k+1} - b). \)

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

Two approaches:

- parallelize the serial algorithm(s) for ADMM subproblem(s),
- turn problem into a parallel-ready form\(^9\)

\[
\begin{align*}
\min_{x,y} & \quad \sum_{i} f_i(x_i) + g(y) \\
\text{s.t.} & \quad \begin{bmatrix} A_1 & \ldots & A_M \\ \vdots & \ddots & \vdots \\ A_M \end{bmatrix} \begin{bmatrix} x_1 \\ \vdots \\ x_M \end{bmatrix} + \begin{bmatrix} B_1 \\ \vdots \\ B_M \end{bmatrix} y = \begin{bmatrix} b_1 \\ \vdots \\ b_M \end{bmatrix}
\end{align*}
\]

consequently, computing \(x^{k+1}\) reduces to parallel subproblems

\[
x_i^{k+1} \leftarrow \min_{x} f_i(x_i) + \frac{\beta}{2} \| A_i x_i + B_i y^k - b_i - z_i^k \|_2^2, \quad i = 1, \ldots, M
\]

**Issue:** to have block diagonal \(A\) and simple \(B\), additional variables are often needed, thus increasing the problem size and parallel overhead.

\(^9\)a recent survey Boyd, Parikh, Chu, Peleato, and Eckstein [2011]
Parallel Linearized Bregman vs ADMM on basis pursuit

Basis pursuit:
\[
\min_x \|x\|_1 \quad \text{s.t. } Ax = b.
\]

- example with dense \( A \in \mathbb{R}^{1024 \times 2048} \);
- distribute \( A \) to \( N \) computing nodes
  \[
  A = [A_1 \ A_2 \ \cdots \ A_N];
  \]
- compare
  1. parallel ADMM in the survey paper Boyd, Parikh, Chu, Peleato, and Eckstein [2011];
  2. parallel linearized Bregman (dual gradient descent), un-accelerated.
- tested \( N = 1, 2, 4, \ldots, 32 \) computing nodes;
parallel ADMM v.s. parallel linearized Bregman

parallel linearized Bregman scales much better
Outline

1. Background of parallel computing: benefits, speedup, and overhead
2. Parallelize existing algorithms
3. Primal decomposition / dual decomposition
   - Parallel dual gradient ascent (linearized Bregman)
   - Parallel ADMM
4. Parallel greedy coordinate descent
5. Numerical results with big data
(Block)-coordinate descent/update

**Definition:** update one block of variables each time, keeping others fixed

**Advantage:** update is simple

**Disadvantages:**
- more iterations (with exceptions)
- may stuck at non-stationary points if problem is non-convex or non-smooth (with exceptions)

**Block selection:** cycle (Gauss-Seidel), parallel (Jacobi), random, greedy

**Specific for sparse optimization:** greedy approach\(^\text{10}\) can be exceptionally effective (because most time correct variables are selected to update; most zero variables are never touched; “reducing” the problem to a much smaller one.)

---

\(^{10}\)Li and Osher [2009]
**GRock: greedy coordinate-block descent**

**Example:**

\[
\min \lambda \|x\|_1 + f(Ax - b)
\]

- decompose \(Ax = \sum_j A_j x_j\); block \(A_j\) and \(x_j\) are kept on node \(j\)

Parallel GRock:

1. *(parallel)* compute a merit value for each coordinate \(i\)
   
   \[
   d_i = \arg \min_d \lambda \cdot r(x_i + d) + g_i d + \frac{1}{2} d^2, \quad \text{where } g_i = a_T^{(i)} \nabla f(Ax - b)
   \]

2. *(parallel)* compute a merit value for each block \(j\)
   
   \[
   m_j = \max\{|d| : d \text{ is an element of } d_j\}
   \]
   
   let \(s_j\) be the index of the maximal coordinate within block \(j\)

3. *(allreduce)* \(P \leftarrow \text{select } P \text{ blocks with largest } m_j, 2 \leq P \leq N\)

4. *(parallel)* update \(x_{s_j} \leftarrow x_{s_j} + d_{s_j}\) for all \(j \in P\)

5. *(allreduce)* update \(Ax\)
How large $P$ can be?

$P$ depends on the block spectral radius

$$
\rho_P = \max_{M \in \mathcal{M}} \rho(M),
$$

where $\mathcal{M}$ is the set of all $P \times P$ submatrices that we can obtain from $A^T A$ corresponding to selecting exactly one column from each of the $P$ blocks

**Theorem**

Assume each column of $A$ has unit 2-norm. If $\rho_P < 2$, GRock with $P$ parallel updates per iteration gives

$$
\mathcal{F}(x + d) - \mathcal{F}(x) \leq \frac{\rho_P - 2}{2} \beta \|d\|_2^2
$$

moreover,

$$
\mathcal{F}(x^k) - \mathcal{F}(x^*) \leq \frac{2C^2 \left(2L + \beta \sqrt{\frac{N}{P}}\right)^2}{(2 - \rho_P)\beta} \cdot \frac{1}{k}.
$$
Compare different block selection rules

- LASSO test with $\mathbf{A} \in \mathbb{R}^{512 \times 1024}$, $N = 64$ column blocks
- GRock uses $P = 8$ updates each iteration
- greedy $\text{CD}^{11}$ uses $P = 1$
- mixed $\text{CD}^{12}$ selects $P = 8$ random blocks and best coordinate in each iteration

\[^{11}\text{Li and Osher [2009]}\]
\[^{12}\text{Scherrer, Tewari, Halappanavar, and Haglin [2012]}\]
Outline

1. Background of parallel computing: benefits, speedup, and overhead

2. Parallelize existing algorithms

3. Primal decomposition / dual decomposition
   - Parallel dual gradient ascent (linearized Bregman)
   - Parallel ADMM

4. Parallel greedy coordinate descent

5. **Numerical results with big data**
Test on Rice cluster STIC

specs:

- 170 Appro Greenblade E5530 nodes each with two quad-core 2.4GHz Xeon (Nahalem) CPUs
- each node has 12GB of memory shared by all 8 cores
- # of processes used on each node = 8

test dataset:

<table>
<thead>
<tr>
<th></th>
<th>A type</th>
<th>A size</th>
<th>λ</th>
<th>sparsity of x*</th>
</tr>
</thead>
<tbody>
<tr>
<td>dataset I</td>
<td>Gaussian</td>
<td>1024 × 2048</td>
<td>0.1</td>
<td>100</td>
</tr>
<tr>
<td>dataset II</td>
<td>Gaussian</td>
<td>2048 × 4096</td>
<td>0.01</td>
<td>200</td>
</tr>
</tbody>
</table>
iterations vs cores

Note: we set $P = \text{number of cores}$
Note: we set \( P = \text{number of cores} \)
Note: we set $P = \text{number of cores}$
big data LASSO on Amazon EC2

Amazon EC2 is an elastic, pay-as-you-use cluster;
advantage: no hardware investment, everyone can have an account

test dataset

- A dense matrix, 20 billion entries, and **170GB** size
- x: 200K entries, 4K nonzeros, Gaussian values

requested system

- 20 “high-memory quadruple extra-large instances”
- each instance has 8 cores and 60GB memory

code

- written in C using GSL (for matrix-vector multiplication) and MPI
parallel Dual-ADMM vs FISTA\textsuperscript{13} vs GRock

<table>
<thead>
<tr>
<th></th>
<th>p D-ADMM</th>
<th>p FISTA</th>
<th>GRock</th>
</tr>
</thead>
<tbody>
<tr>
<td>estimate stepsizes (min.)</td>
<td>n/a</td>
<td>1.6</td>
<td>n/a</td>
</tr>
<tr>
<td>matrix factorization (min.)</td>
<td>51</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td>iteration time (min.)</td>
<td>105</td>
<td>40</td>
<td>1.7</td>
</tr>
<tr>
<td>number of iterations</td>
<td>2500</td>
<td>2500</td>
<td>104</td>
</tr>
<tr>
<td>communication time (min.)</td>
<td>30.7</td>
<td>9.5</td>
<td>0.5</td>
</tr>
<tr>
<td>stopping relative error</td>
<td>1E-1</td>
<td>1E-3</td>
<td>1E-5</td>
</tr>
<tr>
<td>total time (min.)</td>
<td>156</td>
<td>41.6</td>
<td>1.7</td>
</tr>
<tr>
<td>Amazon charge</td>
<td>$85</td>
<td>$22.6</td>
<td>$0.93</td>
</tr>
</tbody>
</table>

- ADMM’s performance depends on penalty parameter $\beta$
  - we picked $\beta$ as the best out of only a few trials (we cannot afford more)
- parallel Dual ADMM and FISTA were capped at 2500 iterations
- GRock used adaptive $P$ and stopped at relative error 1E-5

\textsuperscript{13}Beck and Teboulle [2009]
Software codes can be found in instructor’s website.

Acknowledgements: Zhimin Peng (Rice), Ming Yan (UCLA). Also, Qing Ling (USTC) and Zaiwen Wen (SJTU)

References:


