More Effective Distributed ML via a Stale Synchronous Parallel Parameter Server

Q. Ho, J. Cipar, H. Cui, J.K. Kim, S. Lee,
**P.B. Gibbons, G.A. Gibson, G.R. Ganger, E.P. Xing*

Carnegie Mellon University
***Intel Labs**

SAILING LAB
Laboratory for Statistical Artificial InteLligence & INtegrative Genomics

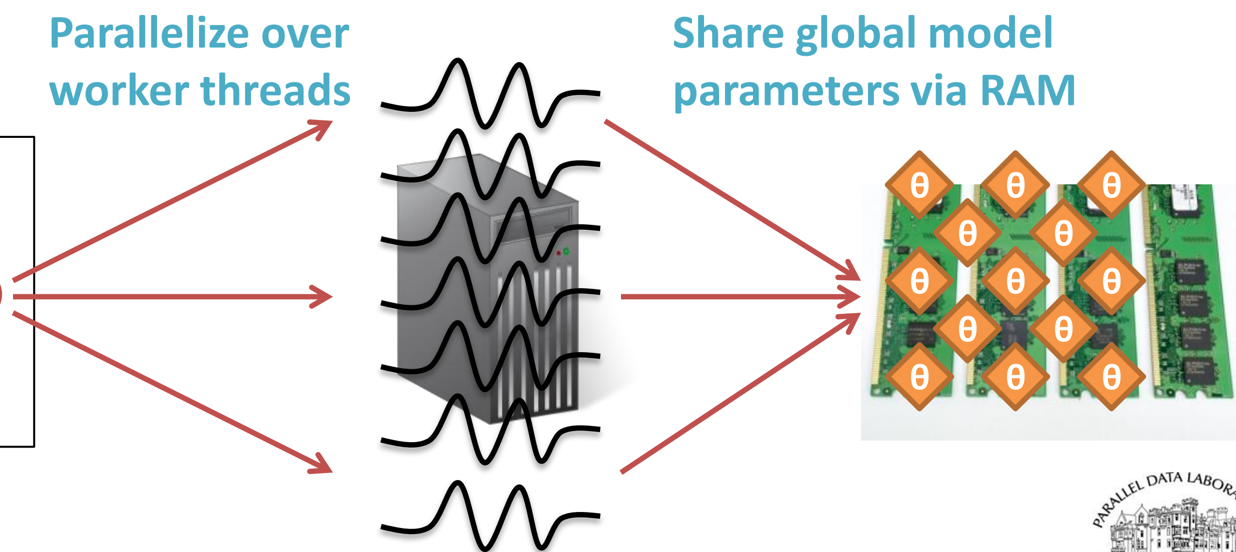
PARALLEL DATA LABORATORY
CARNEGIE MELLON UNIVERSITY

Distributed ML: one machine to many

- **Setting:** have iterative, parallel ML algorithm
  - E.g. optimization, MCMC algorithms
  - For topic models, regression, matrix factorization, SVMs, DNNs, etc.

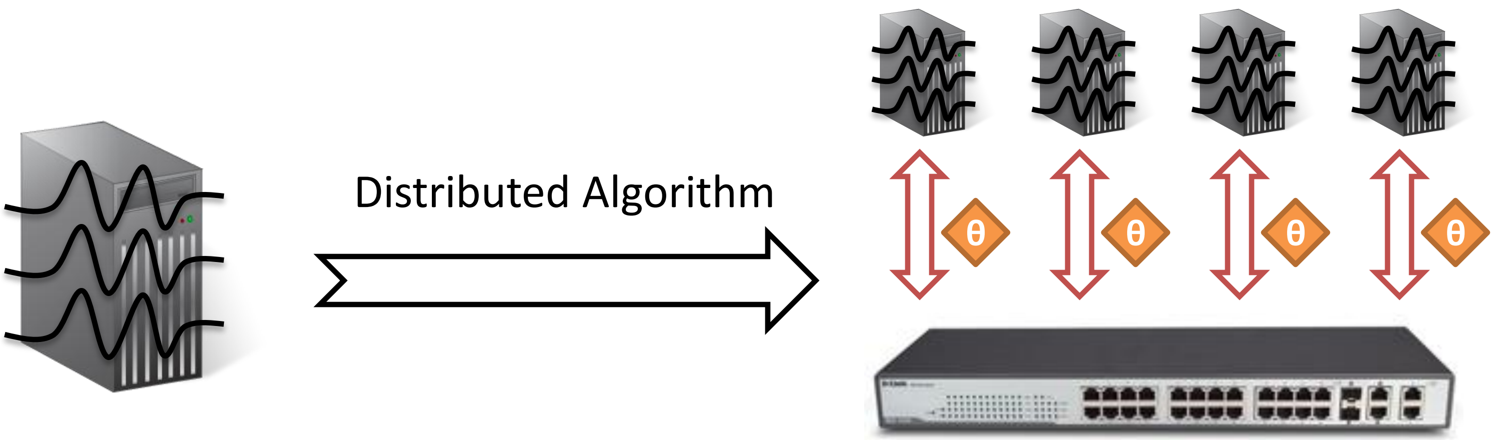
- Critical updates executed on one machine, in parallel
  - Worker threads share global model parameters θ via RAM

```
for (t = 1 to T) {
  doThings()
  parallelUpdate(x,θ)
  doOtherThings()
}
```



Distributed ML: one machine to many

- **Want:** scale up by distributing ML algorithm
  - Must now share parameters over a network
- Seems like a simple task…
  - Many distributed tools available, so just pick one and go?

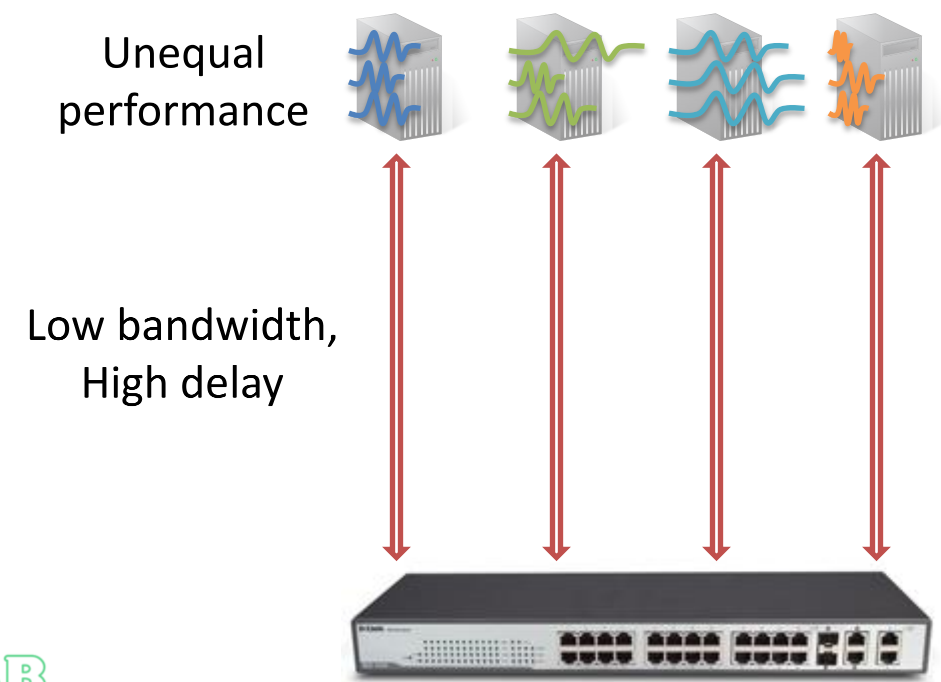


Single machine, multiple threads

Multiple machines, communicating over network switches

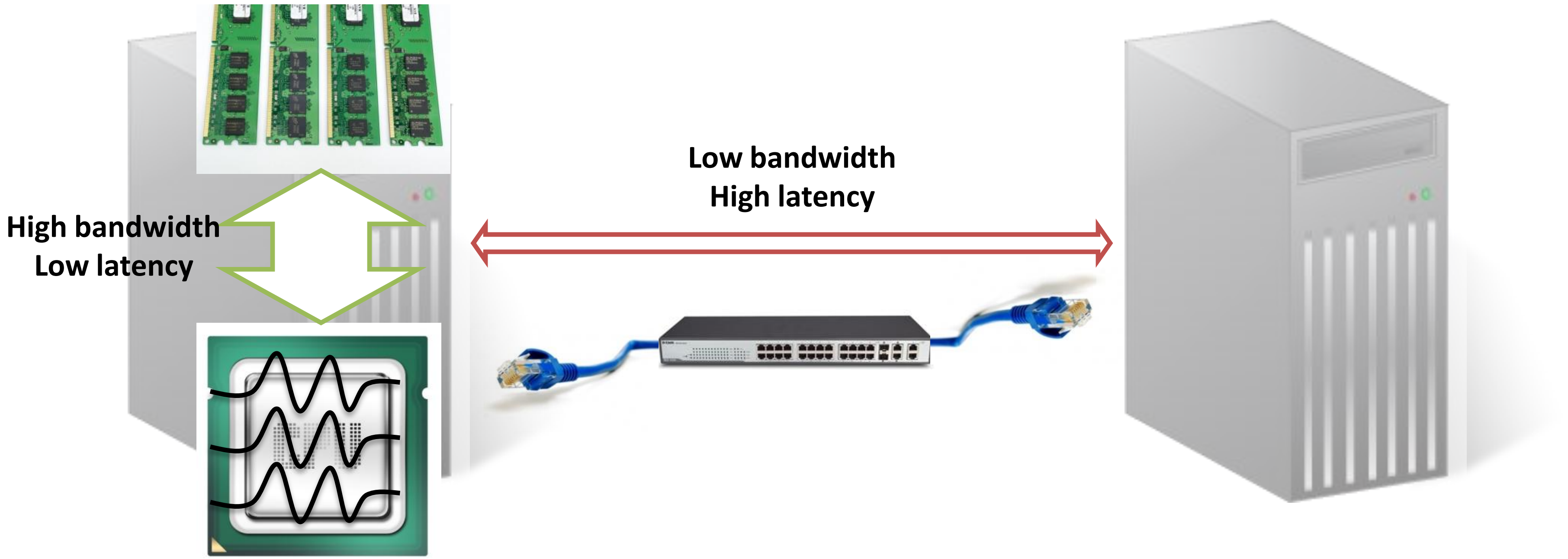
Distributed ML Challenges

- Not quite that easy…
- **Two distributed challenges:**
  - Networks are slow
  - “Identical” machines rarely perform equally



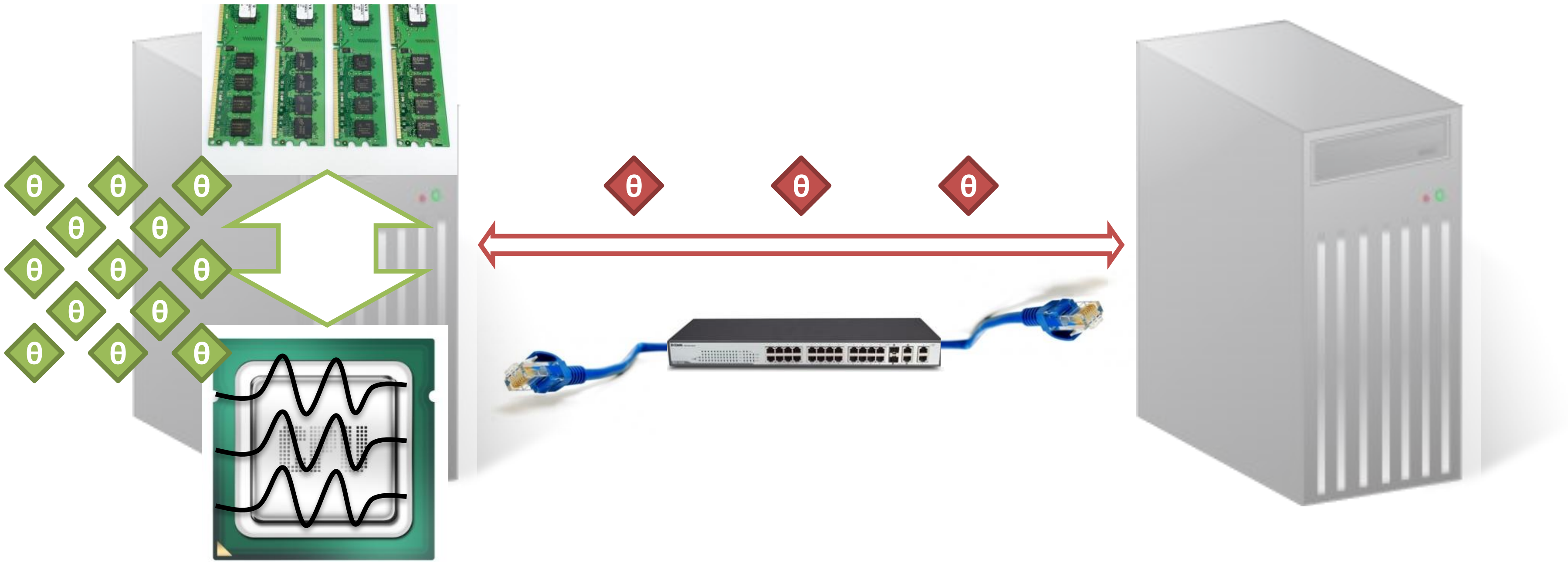
Networks are (relatively) slow

- **Low network bandwidth:**
  - 0.1-1GB/s (inter-machine) vs ≥20GB/s (CPU-RAM)
  - Fewer parameters transmitted per second
- **High network latency (messaging time):**
  - 10,000-100,000 ns (inter-machine) vs 100 ns (CPU-RAM)
  - Wait much longer to receive parameters

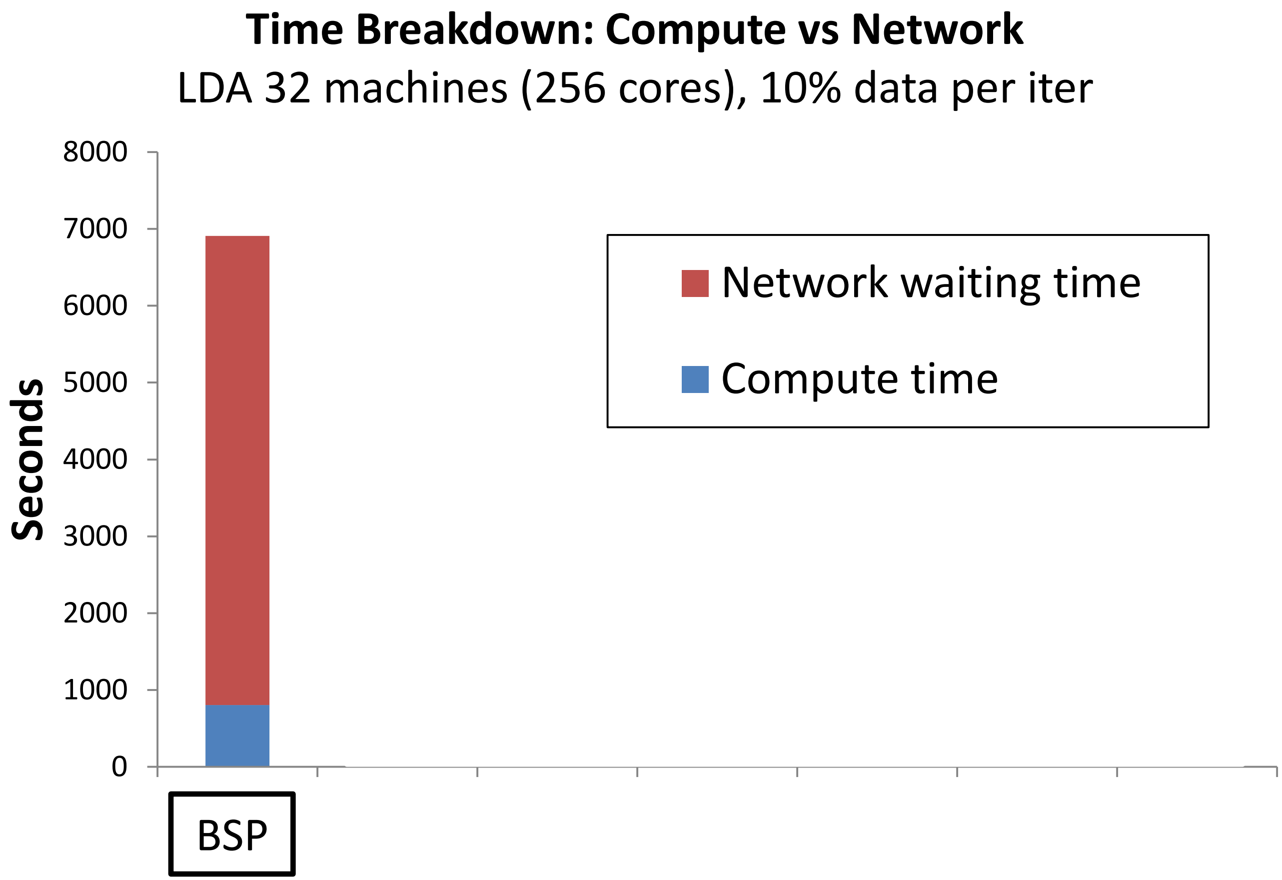


Networks are (relatively) slow

- **Parallel ML requires frequent synchronization**
  - Exchange 10-1000K scalars per second, per thread
  - Parameters not shared quickly enough → communication bottleneck
- Significant bottleneck over a network!



Networks are (relatively) slow

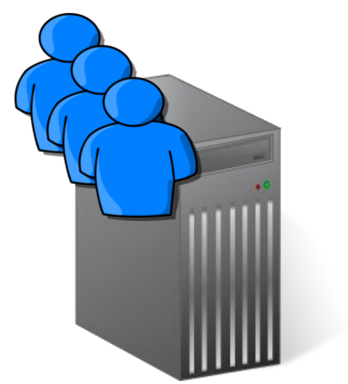


For a "clean" setting with <u>full control over machines</u> and <u>full network capacity</u>

Real clusters with many users have even worse network:compute ratios!

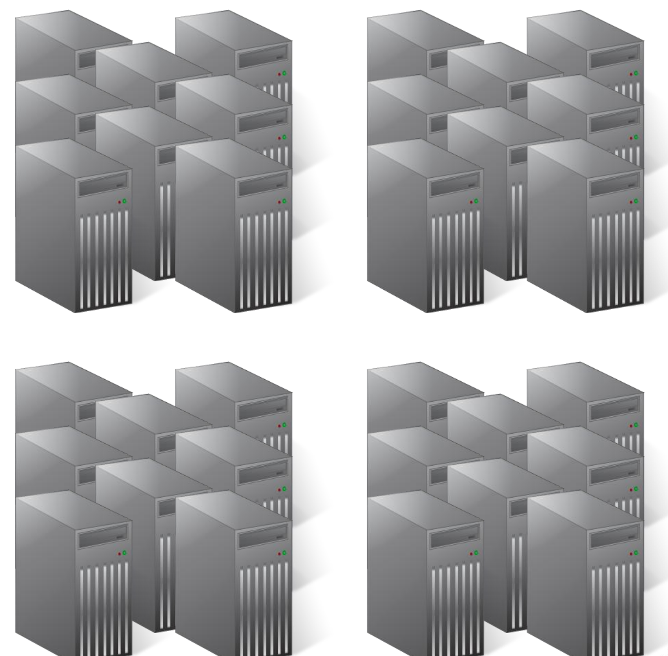
Machines don’t perform equally

- Even when configured identically
- **Variety of reasons:**
  - Vibrating hard drive
  - Background programs; part of a distributed filesystem
  - Other users
  - Machine is a VM/cloud service
- Occasional, random slowdowns in different machines



Consequence: Scaling up ML is hard!

- **Going from 1 to N machines:**
  - Naïve implementations rarely yield N-fold speedup
    - Slower convergence due to machine slowdowns, network bottlenecks
  - If not careful, even worse than a single machine!
    - Algorithm diverges due to errors from slowdowns!



Existing general-purpose scalable ML

Theory-oriented

- Focus on algorithm correctness/convergence

- Examples:
  - **Cyclic fixed-delay schemes** (Langford et al., Agarwal & Duchi)
  - **Single-machine asynchronous** (Niu et al.)
  - **Naively-parallel SGD** (Zinkevich et al.)
  - **Partitioned SGD** (Gemulla et al.)

- May oversimplify systems issues
  - e.g. need machines to perform consistently
  - e.g. need lots of synchronization
  - e.g. or even try not to communicate at all

Systems-oriented

- Focus on high iteration throughput

- Examples:
  - **MapReduce**: Hadoop and Mahout
  - **Spark**
  - **Graph-based**: GraphLab, Pregel

- May oversimplify ML issues
  - e.g. assume algorithms "just work" in distributed setting, without proof
  - e.g. must convert programs to new programming model; nontrivial effort

Existing general-purpose scalable ML

Theory-oriented

- Focus on algorithm correctness/convergence

- Examples:
  - **Cyclic fixed-delay schemes** (Langford et al., Agarwal & Duchi)
  - **Single-machine asynchronous** (Niu et al.)
  - **Naively-parallel SGD** (Zinkevich et al.)
  - **Partitioned SGD** (Gemulla et al.)

- May oversimplify systems issues
  - e.g. need machines to perform consistently
  - e.g. need lots of synchronization
  - e.g. or even try not to communicate at all

Systems-oriented

- Focus on high iteration throughput

- Examples:
  - **MapReduce**: Hadoop and Mahout
  - **Spark**
  - **Graph-based**: GraphLab, Pregel

- May oversimplify ML issues
  - e.g. assume algorithms "just work" in distributed setting, without proof
  - e.g. must convert programs to new programming model; nontrivial effort

Can we take both sides into account?

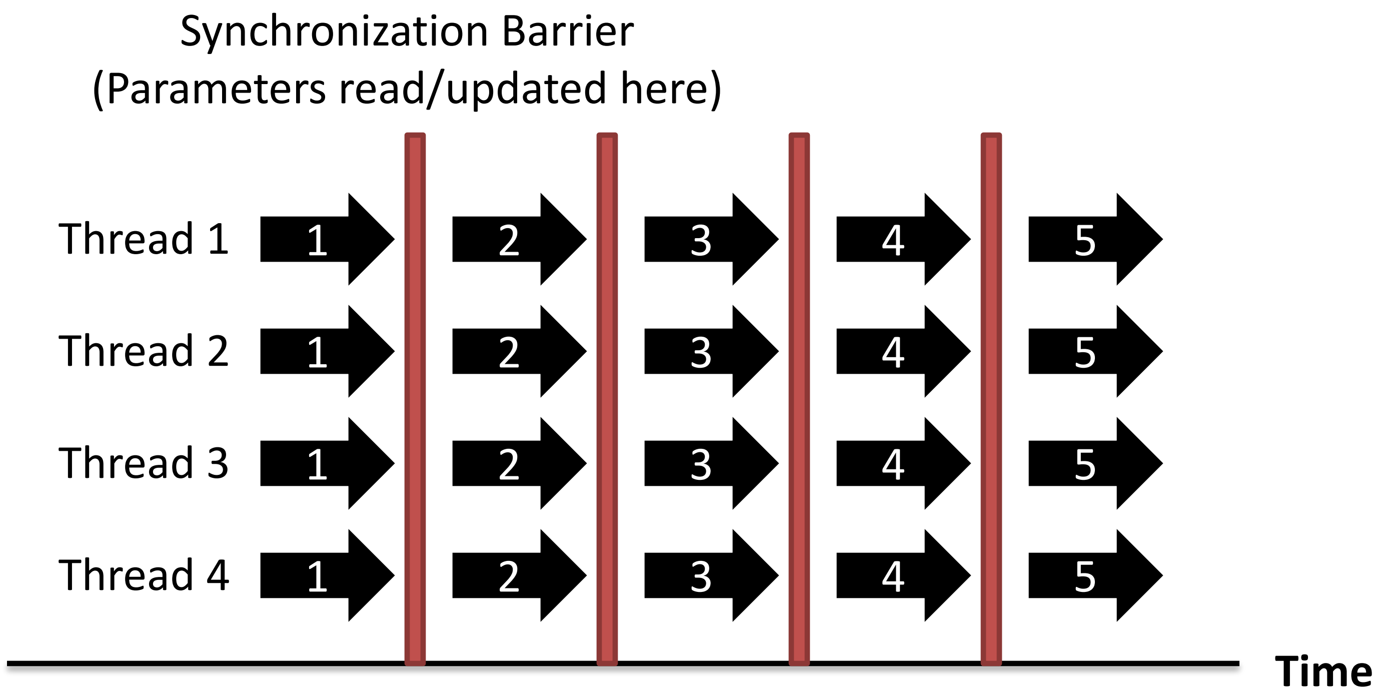
SAILING LAB
Laboratory for Statistical Artificial InteLligence & INtegrative Genomics

PARALLEL DATA LABORATORY
CARNEGIE MELLON UNIVERSITY

Middle of the road approach

- **Want:** ML algorithms converge quickly under imperfect systems conditions
  - e.g. slow network performance
  - e.g. random machine slowdowns
  - Parameters are not communicated consistently
- **Existing work:** mostly use one of two communication models
  - Bulk Synchronous Parallel (BSP)
  - Asynchronous (Async)
- **First, understand pros and cons of BSP and Async**

Bulk Synchronous Parallel

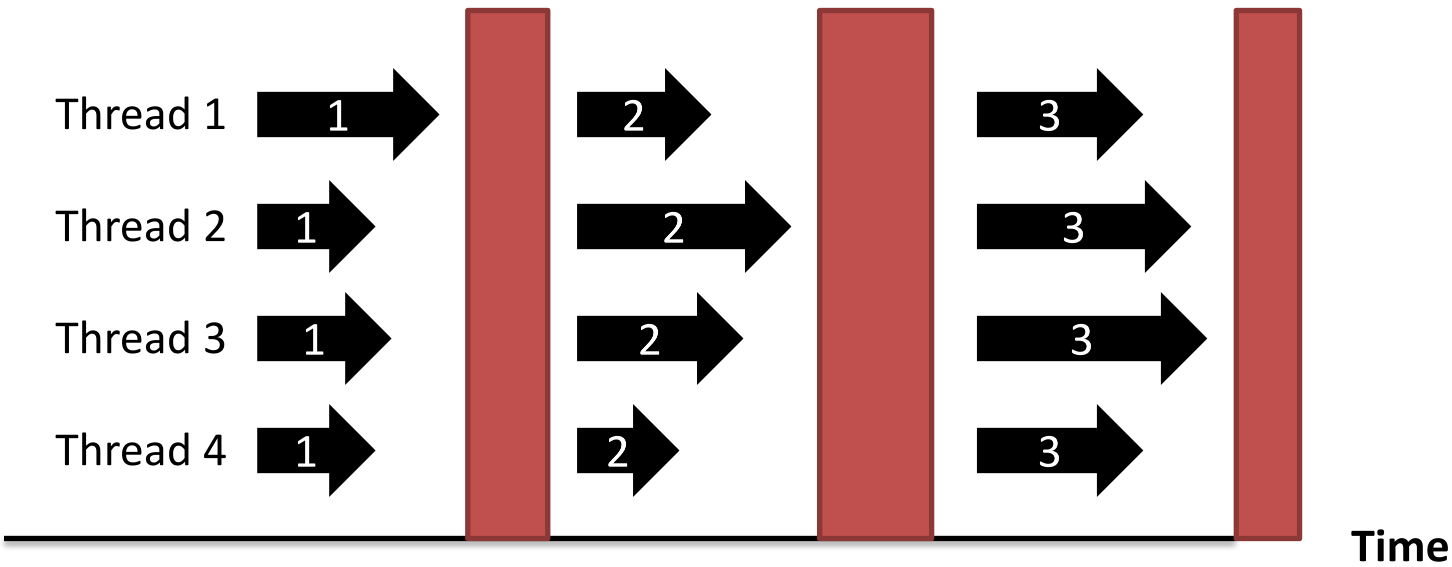


Threads synchronize (wait for each other) every iteration

Threads all on same iteration #

Parameters read/updated at synchronization barriers

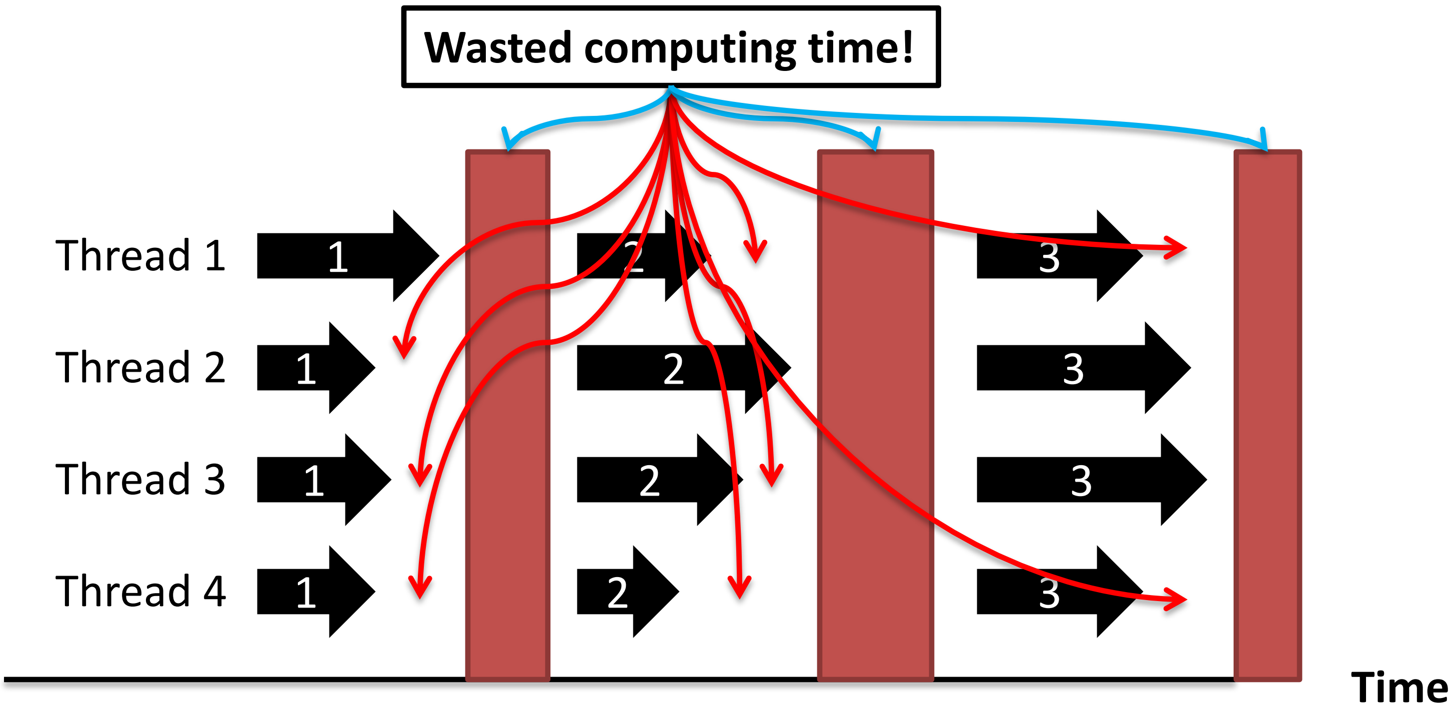
The cost of synchronicity



(a) Machines perform unequally
(b) Algorithmic workload imbalanced
So threads must wait for each other

End-of-iteration sync gets longer with larger clusters (due to slow network)

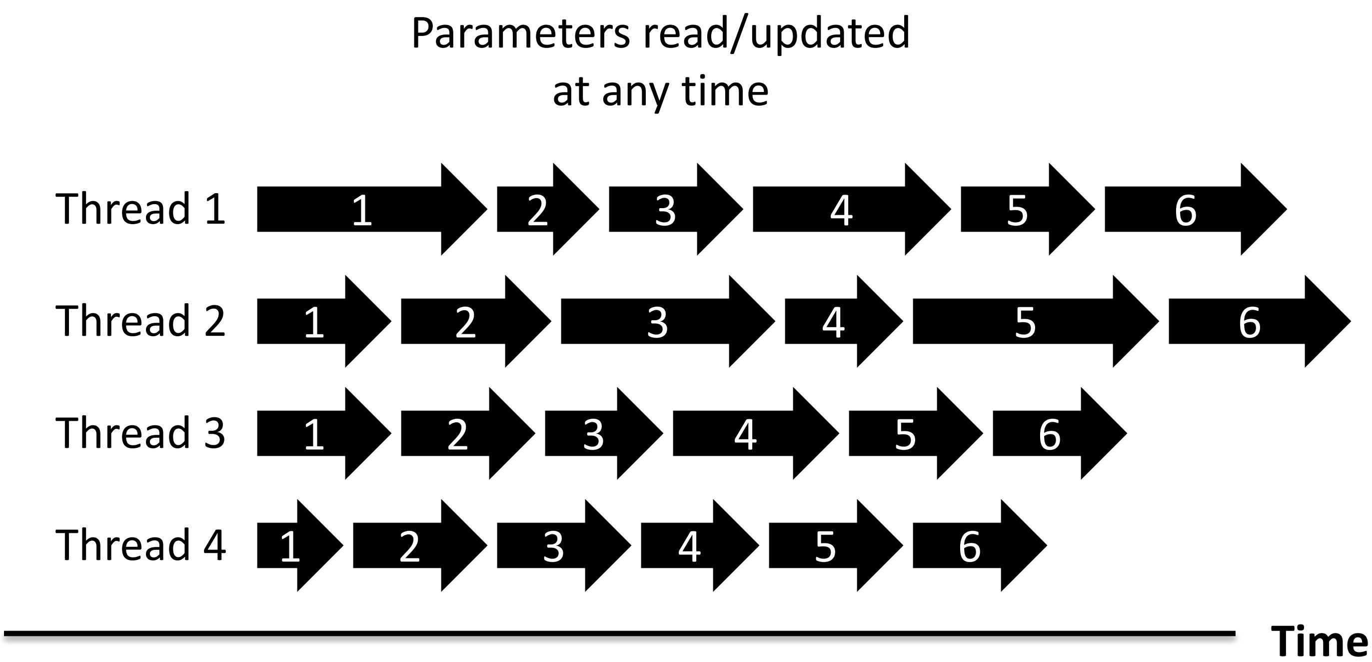
The cost of synchronicity



Threads must wait for each other
End-of-iteration sync gets longer with larger clusters

Precious computing time wasted

Asynchronous



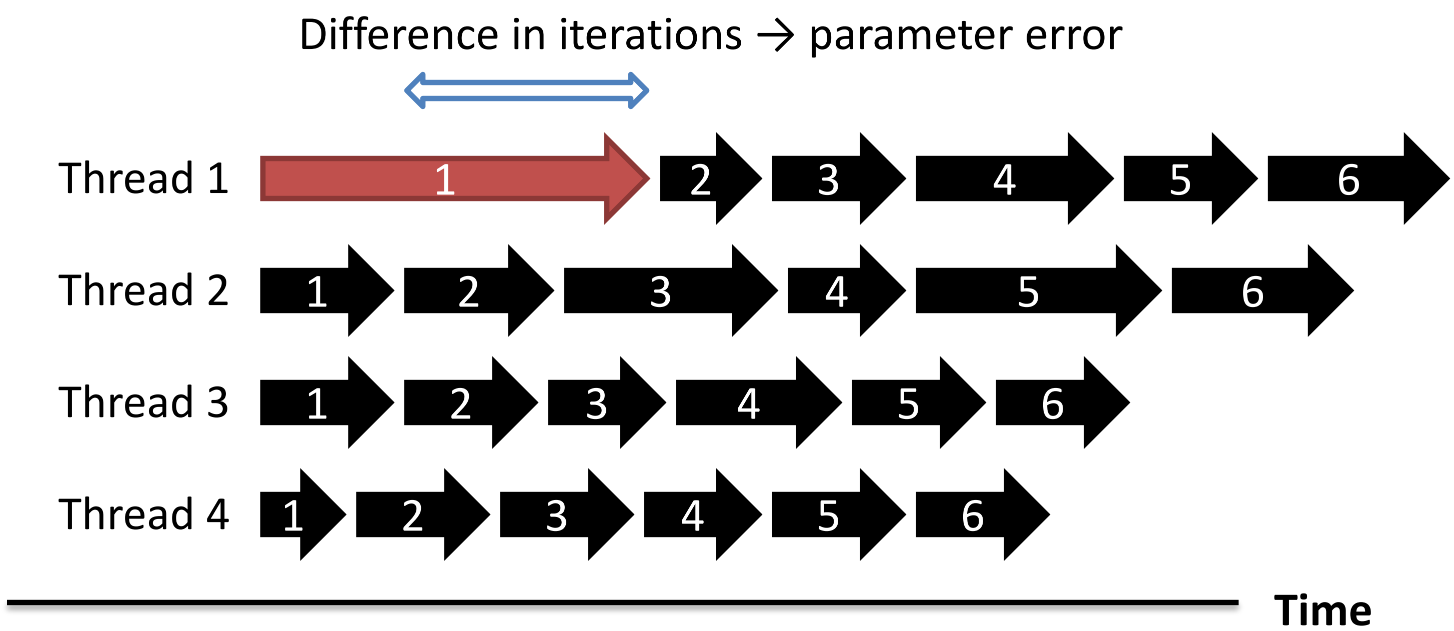
Threads proceed to next iteration without waiting

Threads not on same iteration #

Parameters read/updated any time

SAILING LAB
Laboratory for Statistical Artificial InteLligence & INtegrative Genomics

Slowdowns and Async



Machine suddenly slows down (hard drive, background process, etc.)

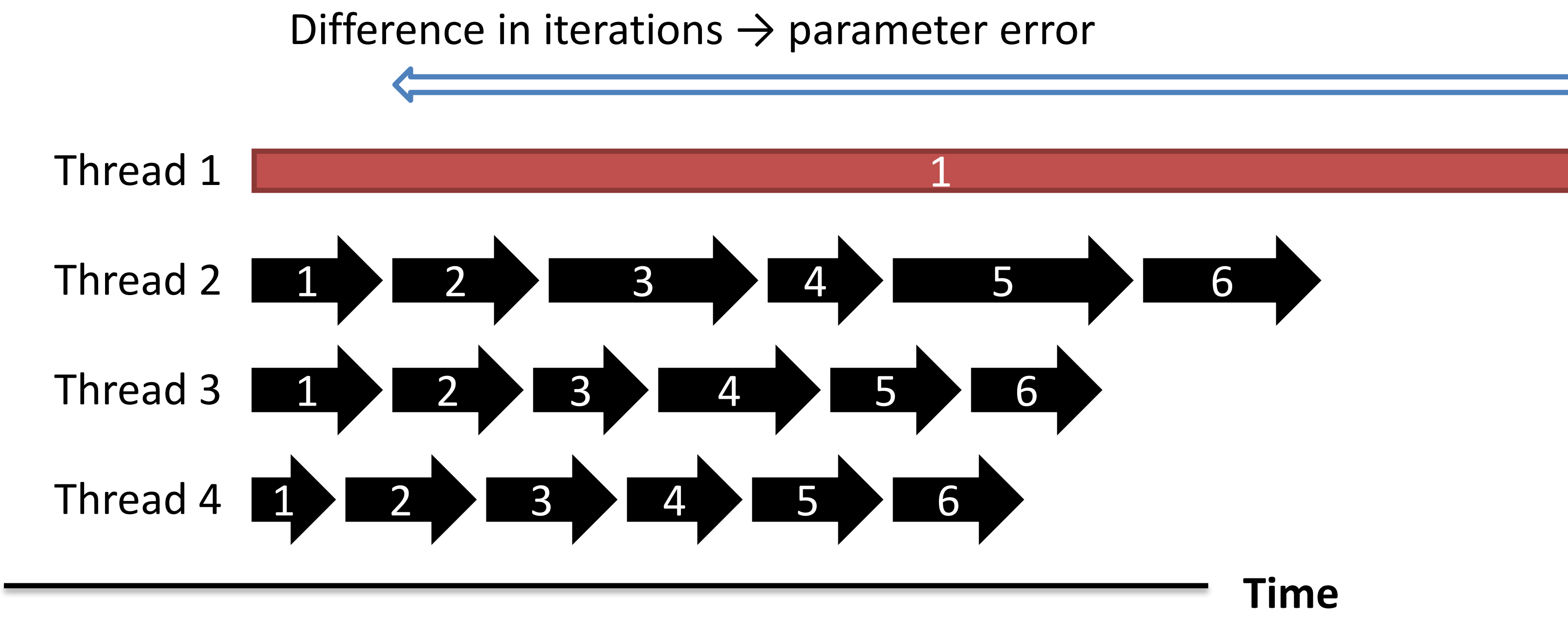
Causing iteration difference between threads

Leading to error in parameters

SAILING LAB
Laboratory for Statistical Artificial InteLligence & INtegrative Genomics

PARALLEL DATA LABORATORY
CARNEGIE MELLON UNIVERSITY

Async worst-case situation



Large clusters have arbitrarily large slowdowns!

Machines become inaccessible for extended periods

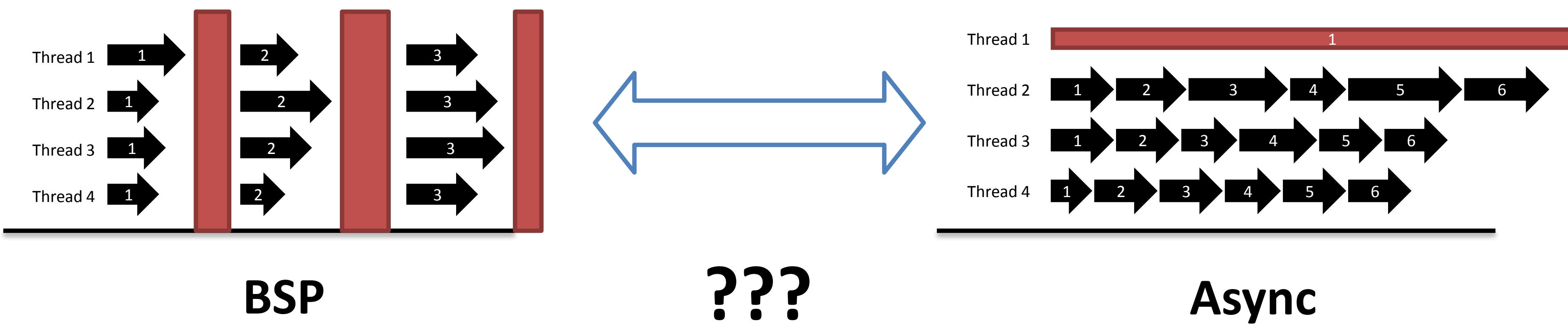
Error becomes unbounded!

What we really want

- **"Partial" synchronicity**
  - Spread network comms evenly (don't sync unless needed)
  - Threads usually shouldn't wait – but mustn't drift too far apart!

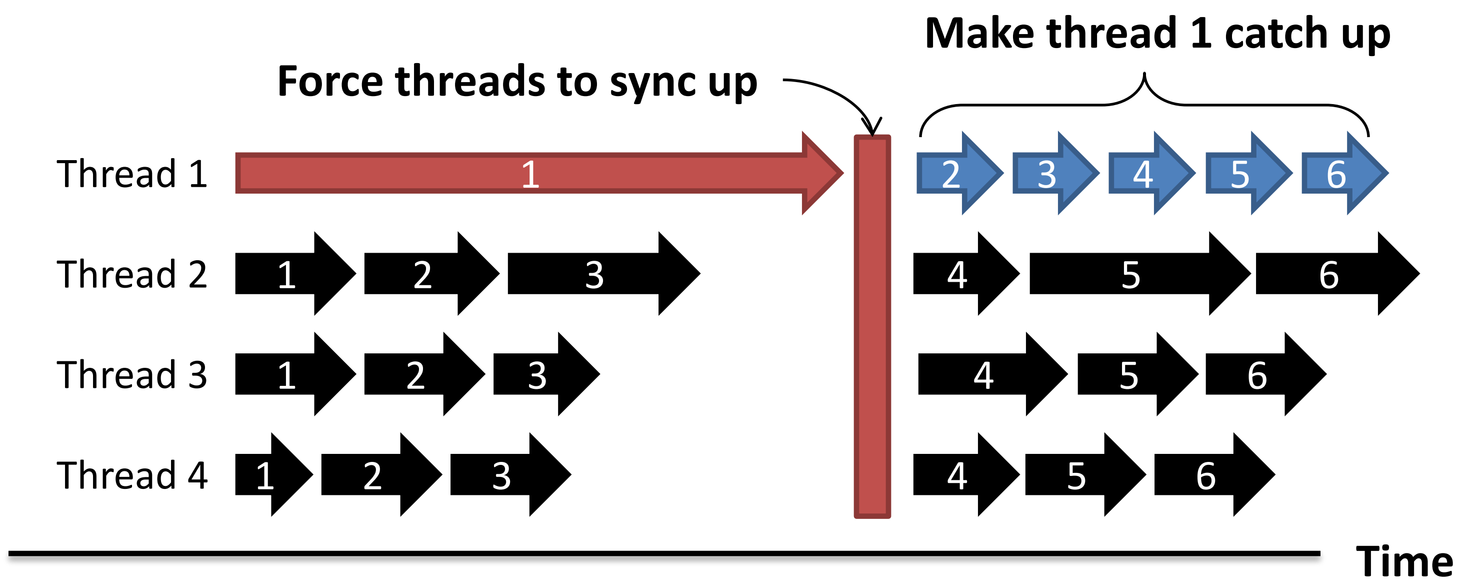
- **Straggler tolerance**
  - Slow threads must somehow catch up

- **Is there a middle ground between BSP and Async?**



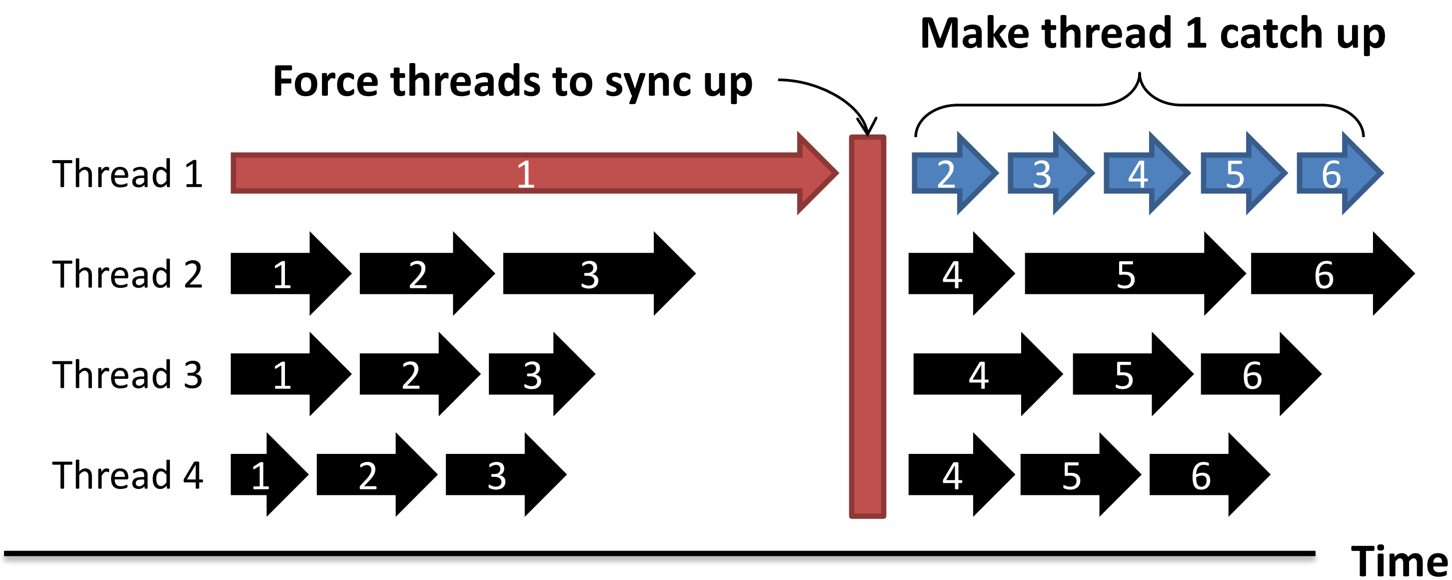
That middle ground

- **"Partial" synchronicity**
  - Spread network comms evenly (don't sync unless needed)
  - Threads usually shouldn't wait – but mustn't drift too far apart!
- **Straggler tolerance**
  - Slow threads must somehow catch up



That middle ground

How do we realize this?



Stale Synchronous Parallel

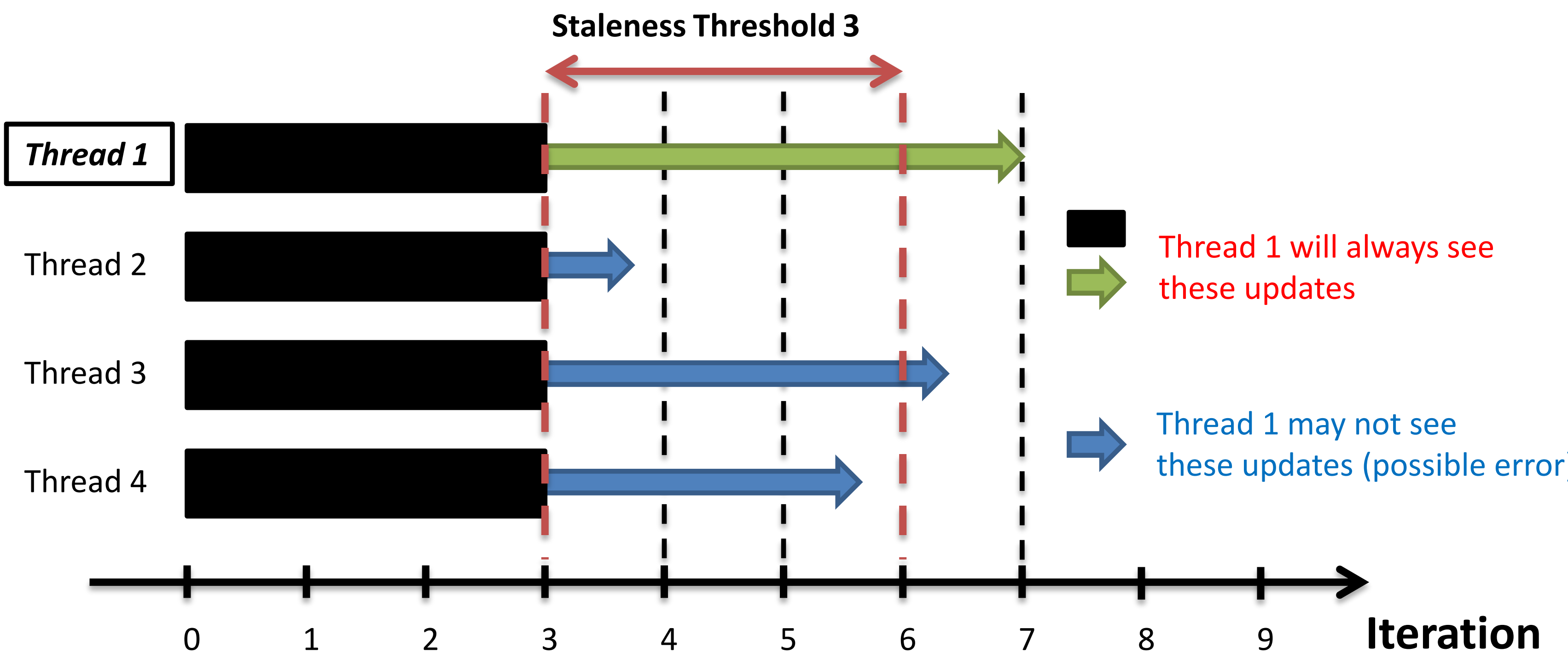
Note: x-axis is now iteration count, not time!

Allow threads to *usually* run at own pace

Fastest/slowest threads not allowed to drift >S iterations apart

Threads cache local (stale) versions of the parameters, to reduce network syncing

Stale Synchronous Parallel



A thread at iter T <u>sees all parameter updates</u> before iter T-S

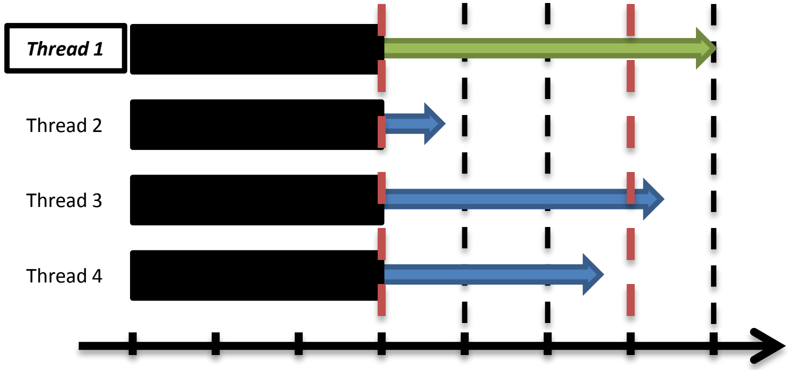
Protocol: check cache first; if too old, get latest version from network

Consequence: fast threads must check network every iteration

Slow threads only check every S iterations – fewer network accesses, so catch up!

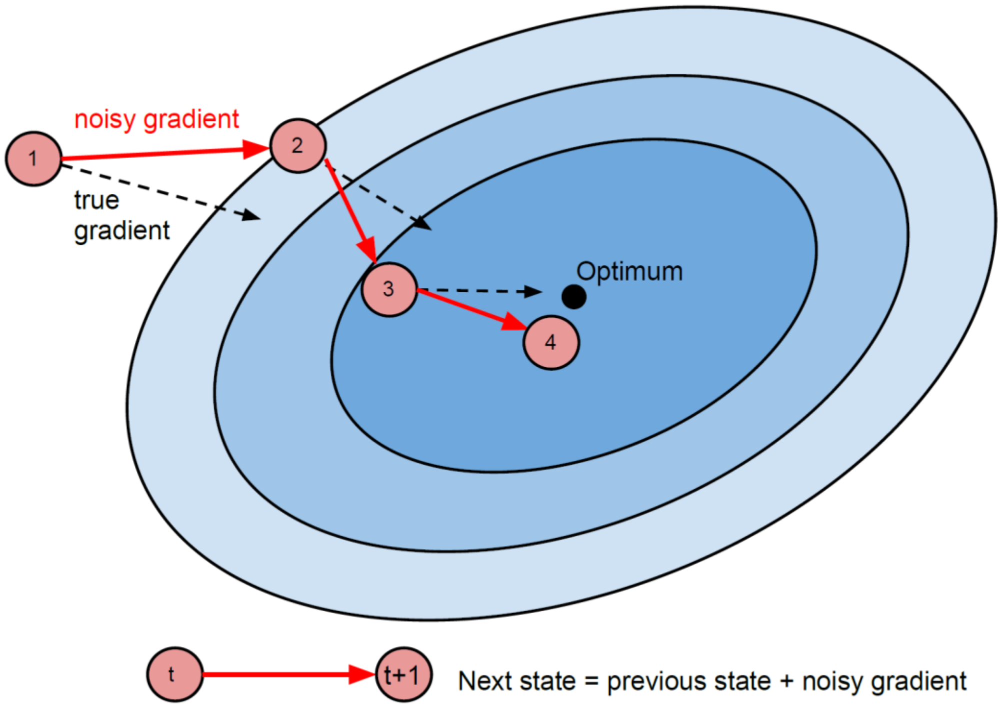
SSP provides best-of-both-worlds

- **SSP combines best properties of BSP and Async**
- BSP-like convergence guarantees
  - Threads cannot drift more than S iterations apart
  - Every thread sees all updates before iteration T-S
- Asynchronous-like speed
  - Threads usually don’t wait (unless there is drift)
  - Slower threads read from network less often, thus catching up
- **SSP is a spectrum of choices**
  - Can be fully synchronous (S = 0) or very asynchronous (S → ∞)
  - Or just take the middle ground, and benefit from both!



Laboratory for Statistical Artificial InteLligence & INtegrative Genomics

Why does SSP converge?



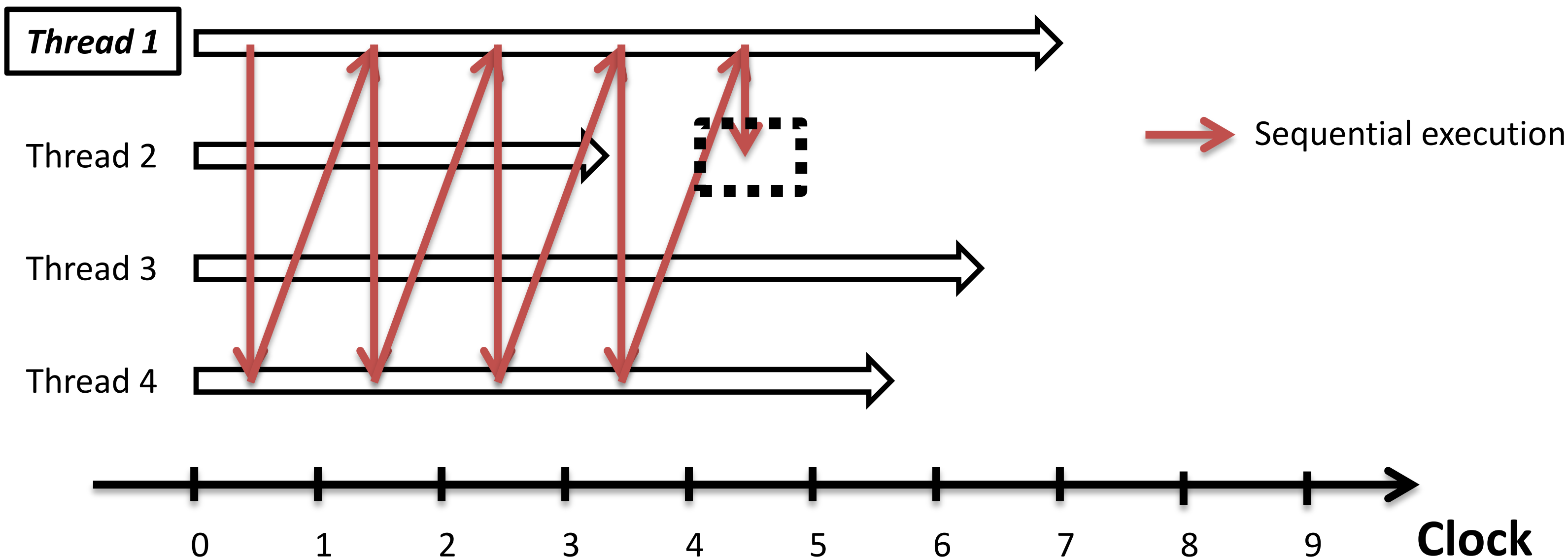
Instead of $\boldsymbol{x}_{\textit{true}}$, SSP sees $\boldsymbol{x}_{\textit{stale}} = \boldsymbol{x}_{\textit{true}}$ + ***error***

The *error* caused by staleness is <u>bounded</u>

Over many iterations, average error goes to zero

Why does SSP converge?

SSP approximates sequential execution



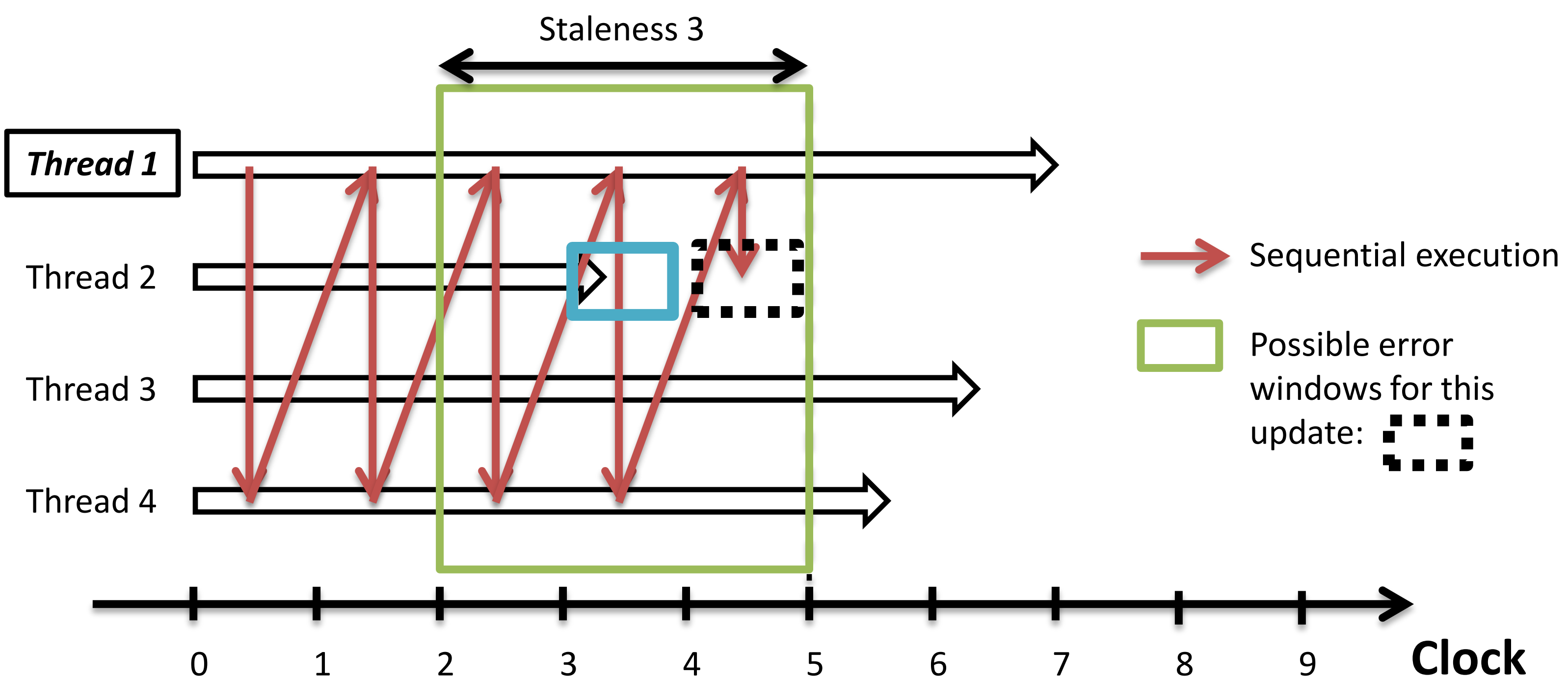
Compare actual update order to ideal sequential execution

SAILING LAB
Laboratory for Statistical Artificial InteLligence & INtegrative Genomics

PARALLEL DATA LABORATORY
CARNEGIE MELLON UNIVERSITY

Why does SSP converge?

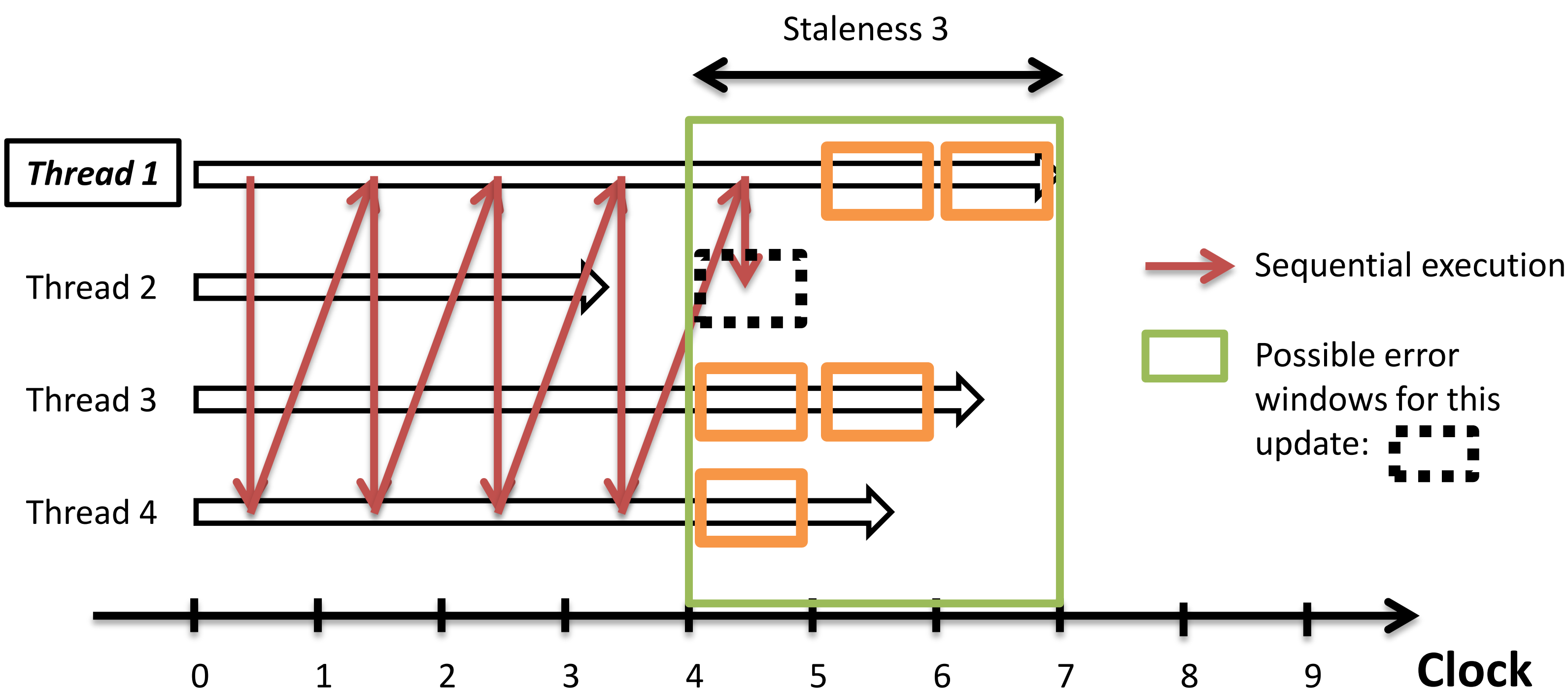
SSP approximates sequential execution



SSP may lose up to S iterations of updates to the left…

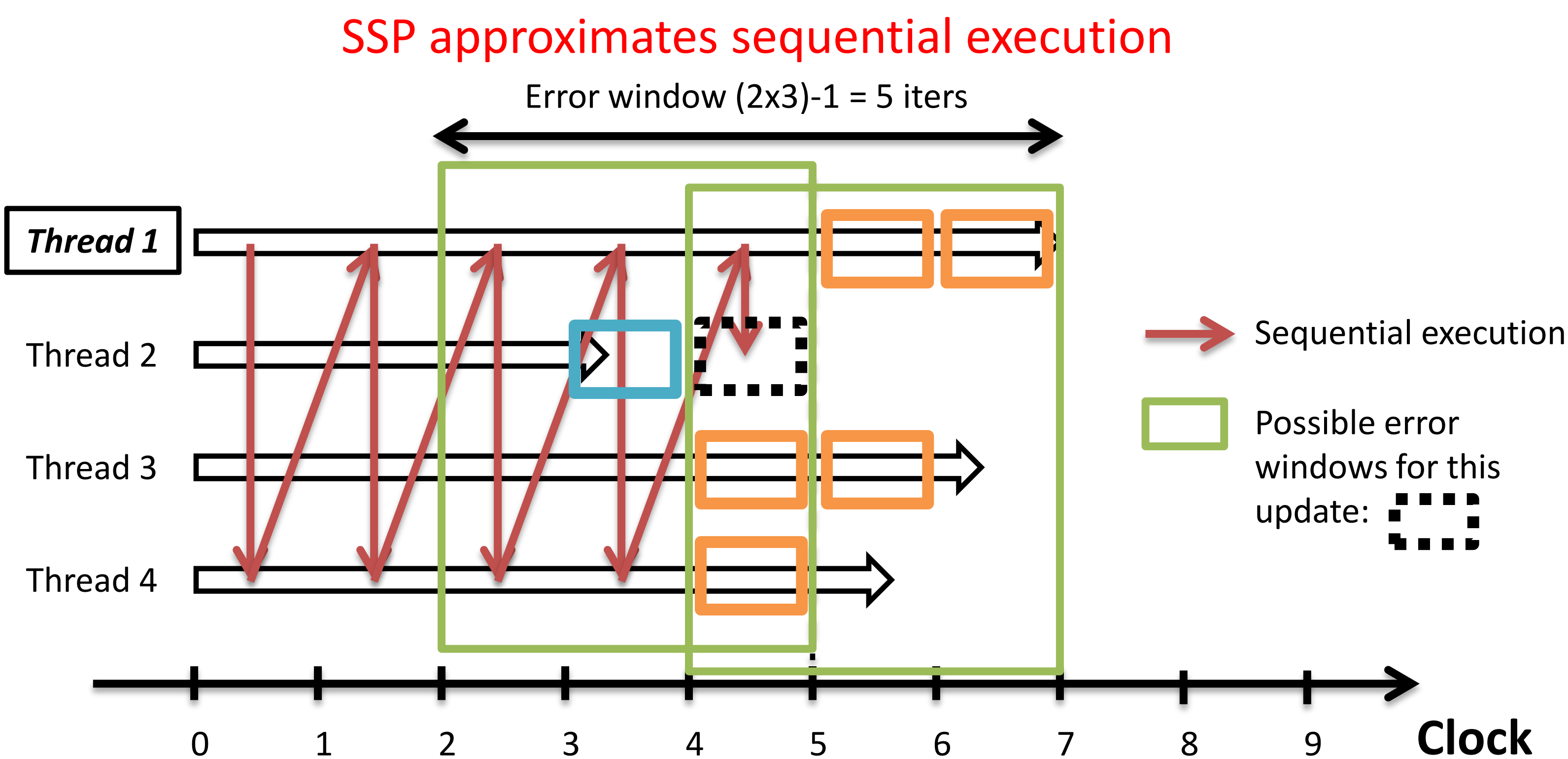
Why does SSP converge?

SSP approximates sequential execution



… as well as gain up to S iterations of updates to the right

Why does SSP converge?



Thus, at most 2S-1 iterations of erroneous updates

Hence numeric error in parameters is also bounded

Partial, but bounded, loss of serializability

Convergence Theorem

- **Want:** minimize convex $f(\mathbf{x}) = \frac{1}{T}\sum_{t=1}^{T} f_t(\mathbf{x})$ (Example: Stochastic Gradient)
  - L-Lipschitz, problem diameter bounded by F^2
  - Staleness s, using P threads across all machines
  - Use step size $\eta_t = \frac{\sigma}{\sqrt{t}}$ with $\sigma = \frac{F}{L\sqrt{2(s+1)P}}$

Convergence Theorem

- **Want:** minimize convex $f(\mathbf{x}) = \frac{1}{T}\sum_{t=1}^{T} f_t(\mathbf{x})$ (Example: Stochastic Gradient)
  - $\boldsymbol{L}$-Lipschitz, problem diameter bounded by $\boldsymbol{F^2}$
  - Staleness $\boldsymbol{s}$, using $\boldsymbol{P}$ threads across all machines
  - Use step size $\eta_t = \frac{\sigma}{\sqrt{t}}$ with $\sigma = \frac{F}{L\sqrt{2(s+1)P}}$

- **SSP converges according to** $$R[\mathbf{X}] := \underbrace{\left[\frac{1}{T}\sum_{t=1}^{T} f_t(\tilde{\mathbf{x}}_t)\right] - f(\mathbf{x}^*)}_{\text{Difference between SSP estimate and true optimum}} \leq 4FL\sqrt{\frac{2(s+1)P}{T}}$$
  - Where $\boldsymbol{T}$ is the number of iterations

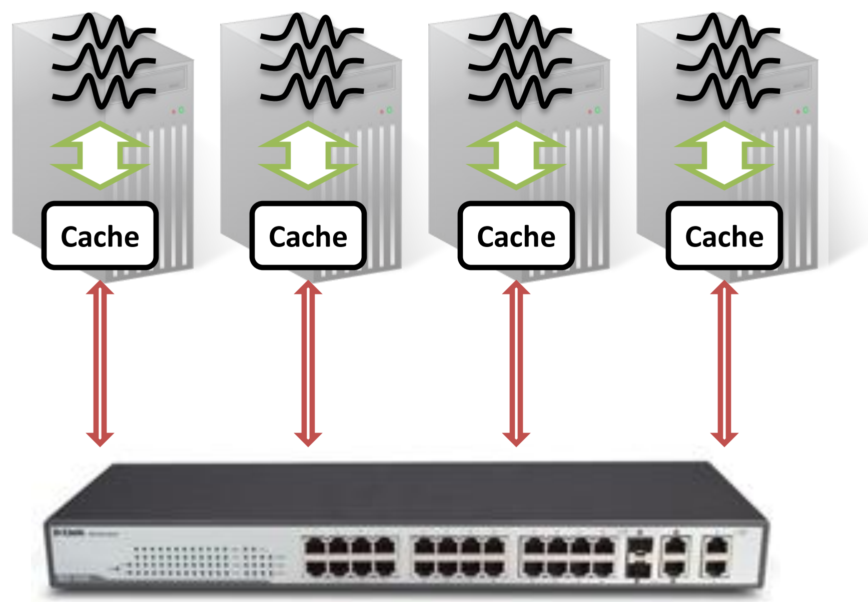
- Note: RHS bound contains $(\boldsymbol{L}, \boldsymbol{F})$ and $(\boldsymbol{s}, \boldsymbol{P})$
  - The interaction between theory and systems parameters

SAILING LAB
Laboratory for Statistical Artificial InteLligence & INtegrative Genomics

PARALLEL DATA LABORATORY CARNEGIE MELLON UNIVERSITY

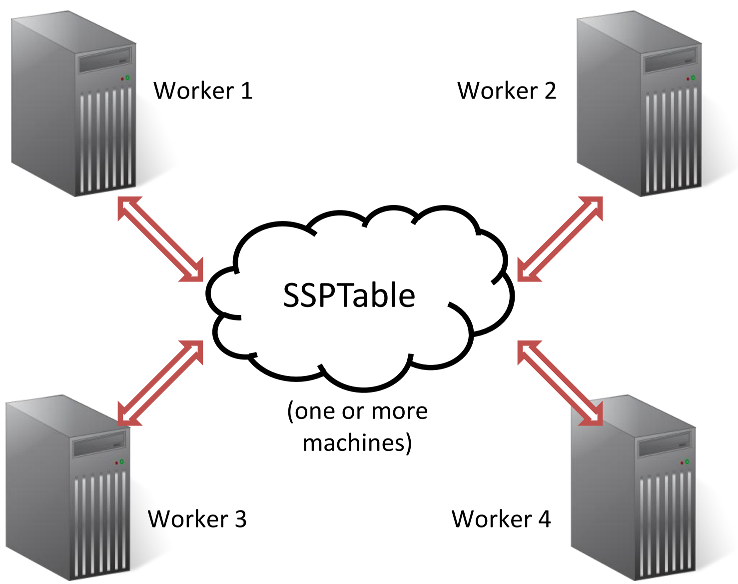
SSP solves Distributed ML challenges

- **SSP is a synchronization model for fast and correct distributed ML**
  - For "abelian" parameter updates of the form $\theta_{new} = \theta_{old} + \Delta$
- **SSP reduces network traffic**
  - Threads use stale local cache whenever possible
  - Addresses slow network and occasional machine slowdowns



SSP + Parameter Server = Easy Distributed ML

- **We implement SSP as a "parameter server" (PS)†, called SSPTable**
  - Provides all machines with convenient access to global model parameter
  - Can be run on multiple machines – reduces load per machine
- **SSPTable allows easy conversion of single-machine parallel ML algorithms**
  - "Distributed shared memory" programming style
  - No need for complicated message passing
  - Replace local memory access with PS access



Single Machine Parallel

```
UpdateVar(i) {
  old = y[i]
  delta = f(old)
  y[i] += delta
}
```

Distributed with SSPTable

```
UpdateVar(i) {
  old = PS.read(y,i)
  delta = f(old)
  PS.inc(y,i,delta)
}
```

† **Ahmed et al. (WSDM 2012), Power and Li (OSDI 2010)**

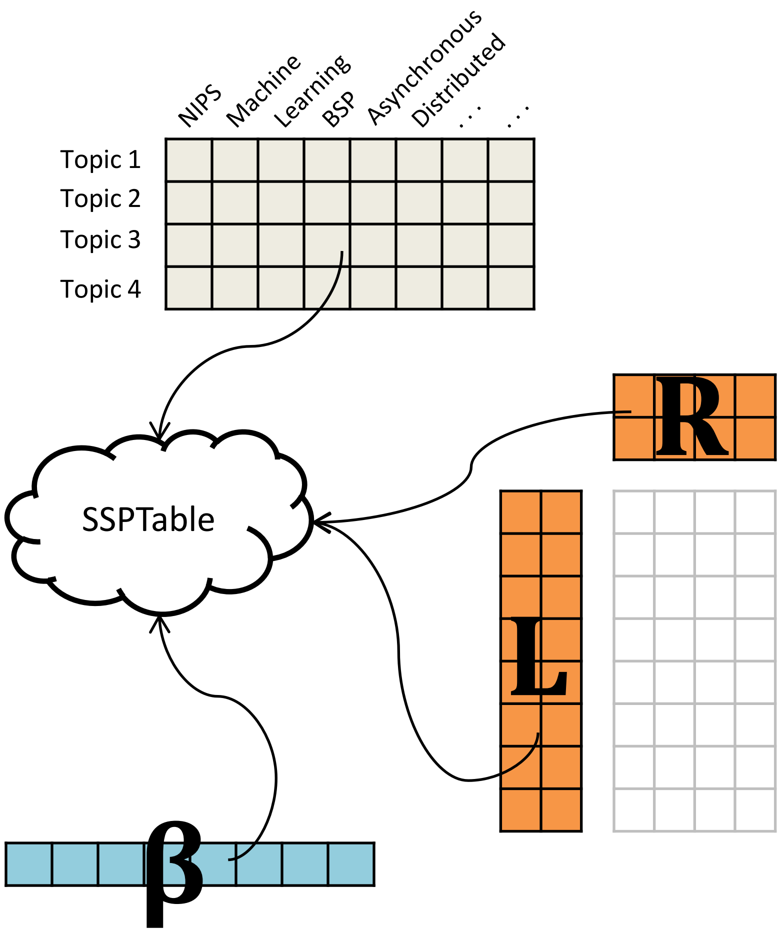
SSPTable Programming

- Easy, table-based programming – just 3 commands!
  - No message passing, barriers, locks, etc.

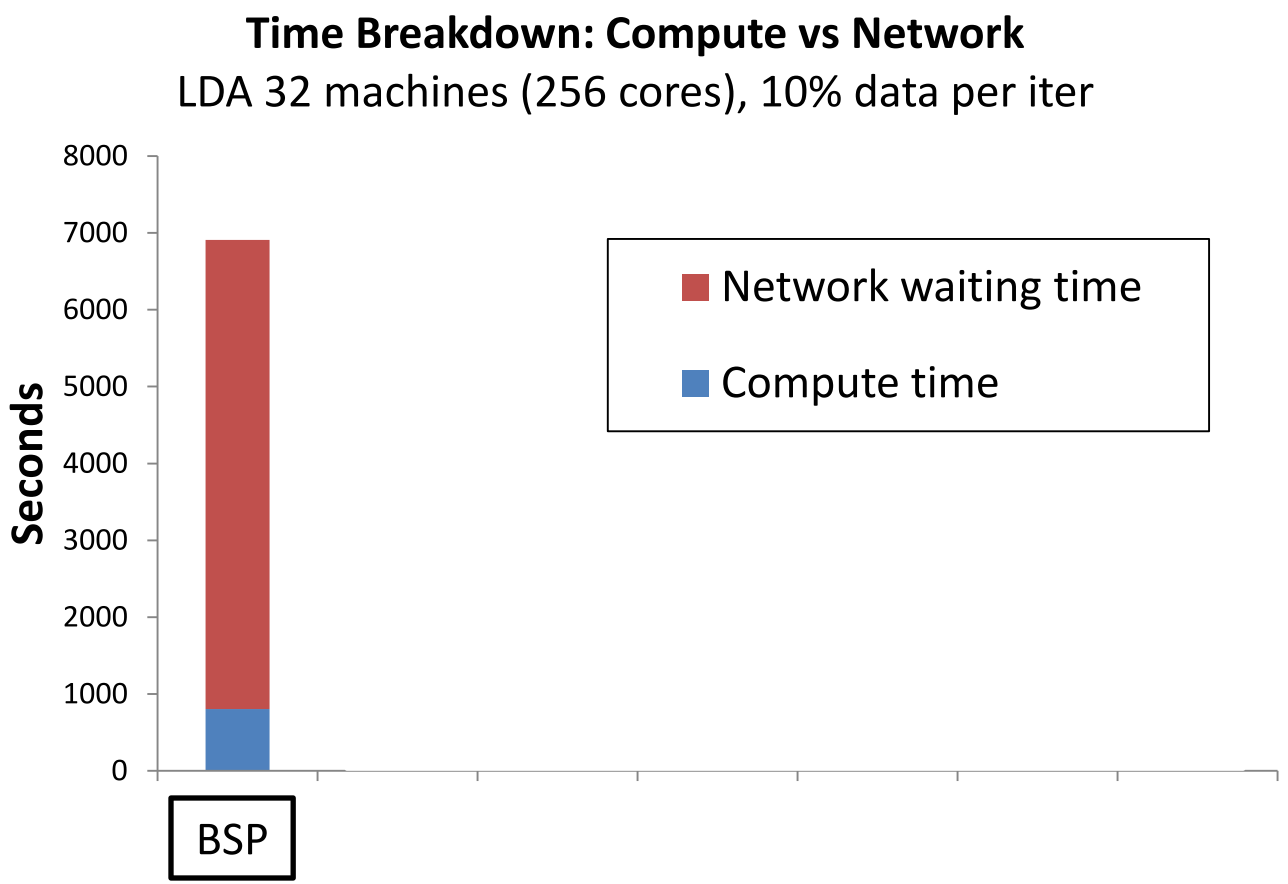
- `read_row(table,row,s)`
  - Retrieve a table row with staleness s
- `inc(table,row,col,value)`
  - Increment table's (row,col) by value
- `clock()`
  - Inform PS that this thread is advancing to the next iteration

SSPTable Programming

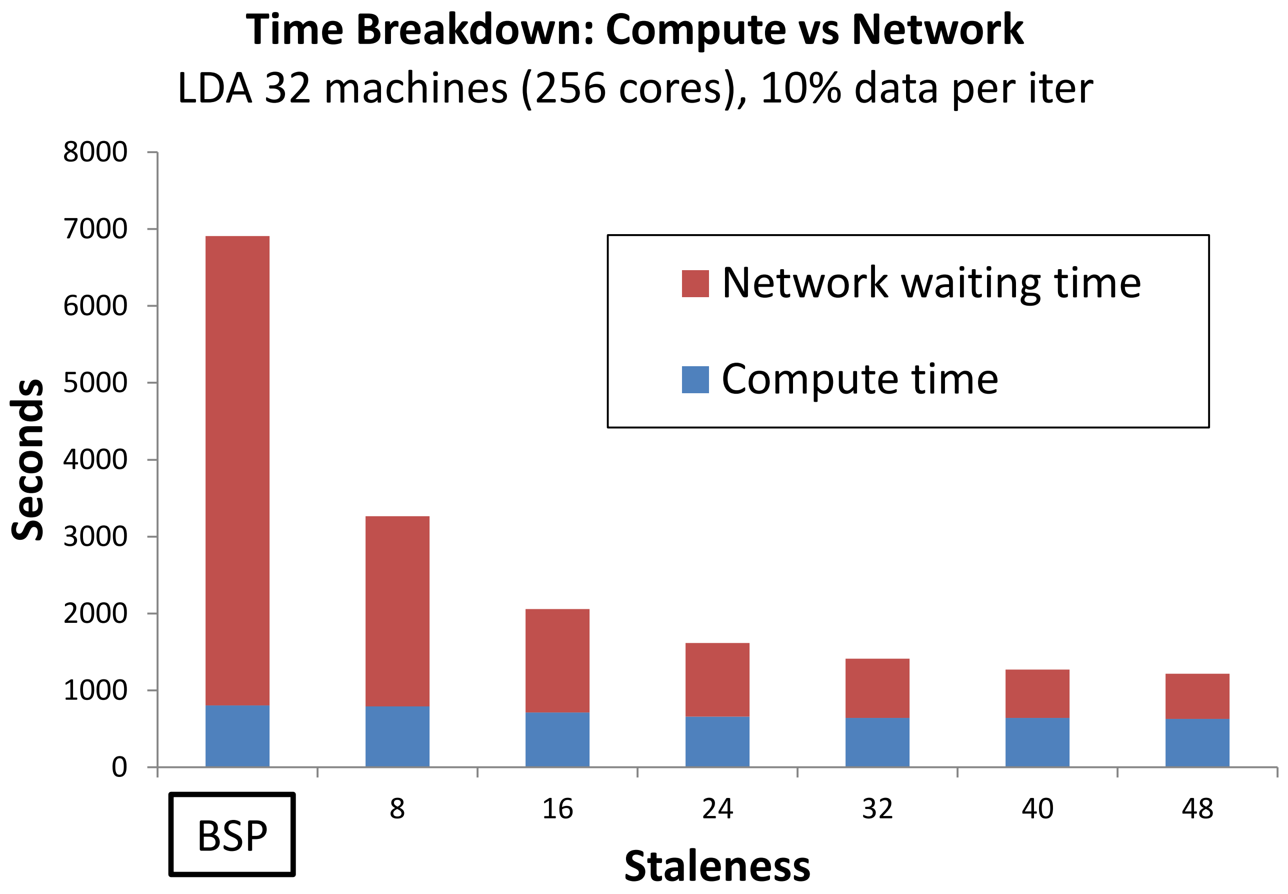
- Just put global parameters in SSPTable! Examples:
- **Topic Modeling (MCMC)**
  - Topic-word table
- **Matrix Factorization (SGD)**
  - Factor matrices L, R
- **Lasso Regression (CD)**
  - Coefficients β
- SSPTable supports generic classes of algorithms
  - With these models as examples



SSPTable uses networks efficiently



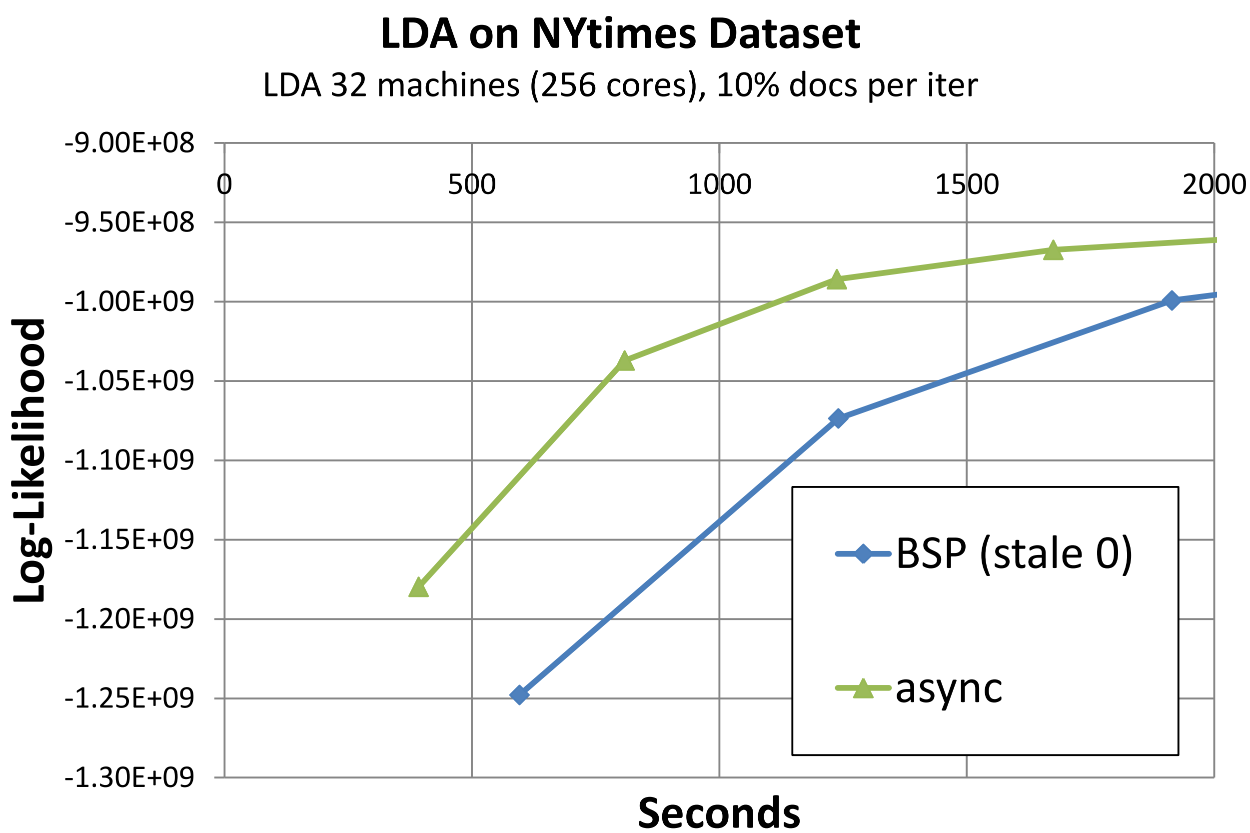
SSPTable uses networks efficiently



Network communication is a huge bottleneck with many machines

SSP balances network and compute time

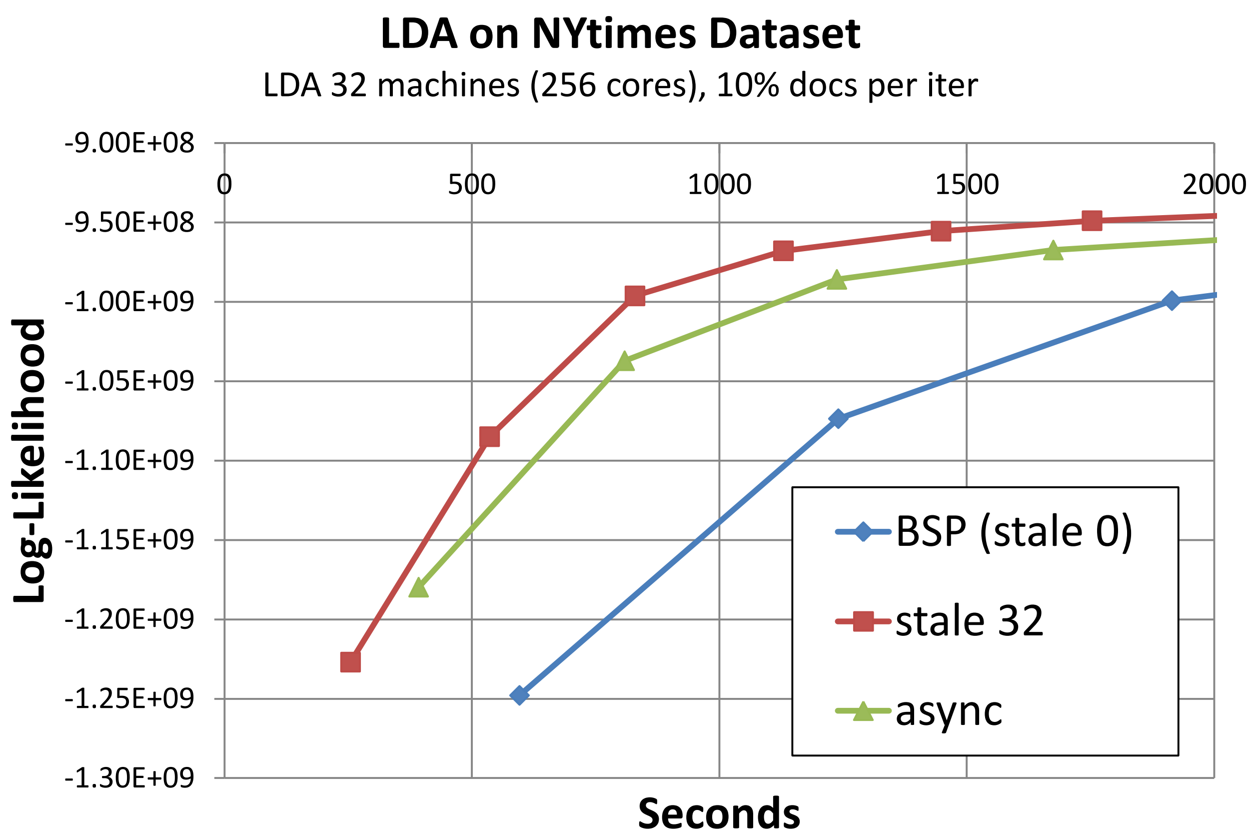
SSPTable vs BSP and Async



NYtimes data
N = 100M tokens
K = 100 topics
V = 100K terms

BSP has strong convergence guarantees but is slow
Asynchronous is fast but has weak convergence guarantees

SSPTable vs BSP and Async



NYtimes data
N = 100M tokens
K = 100 topics
V = 100K terms

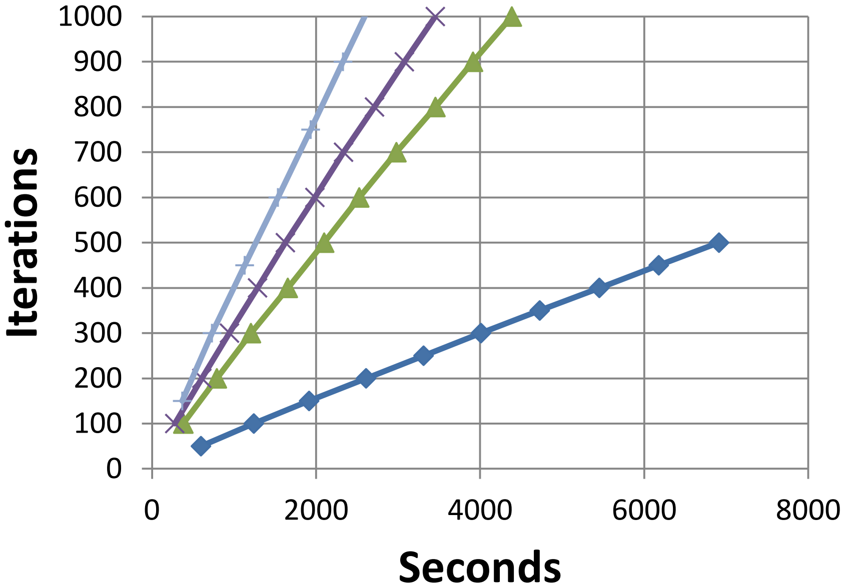
BSP has strong convergence guarantees but is slow

Asynchronous is fast but has weak convergence guarantees

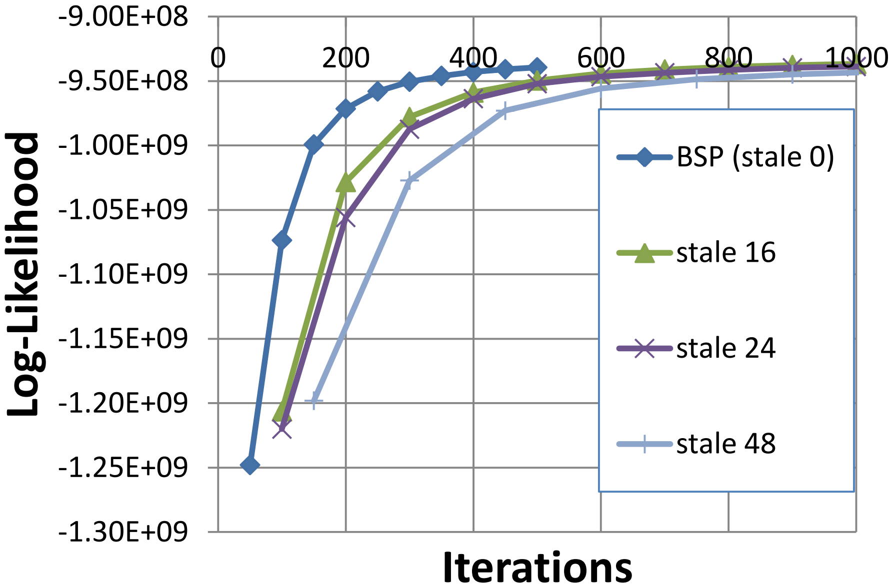
SSPTable is fast and has strong convergence guarantees

The Quality vs Quantity tradeoff

Quantity: iterations versus time
LDA 32 machines, 10% data



Quality: objective versus iterations
LDA 32 machines, 10% data

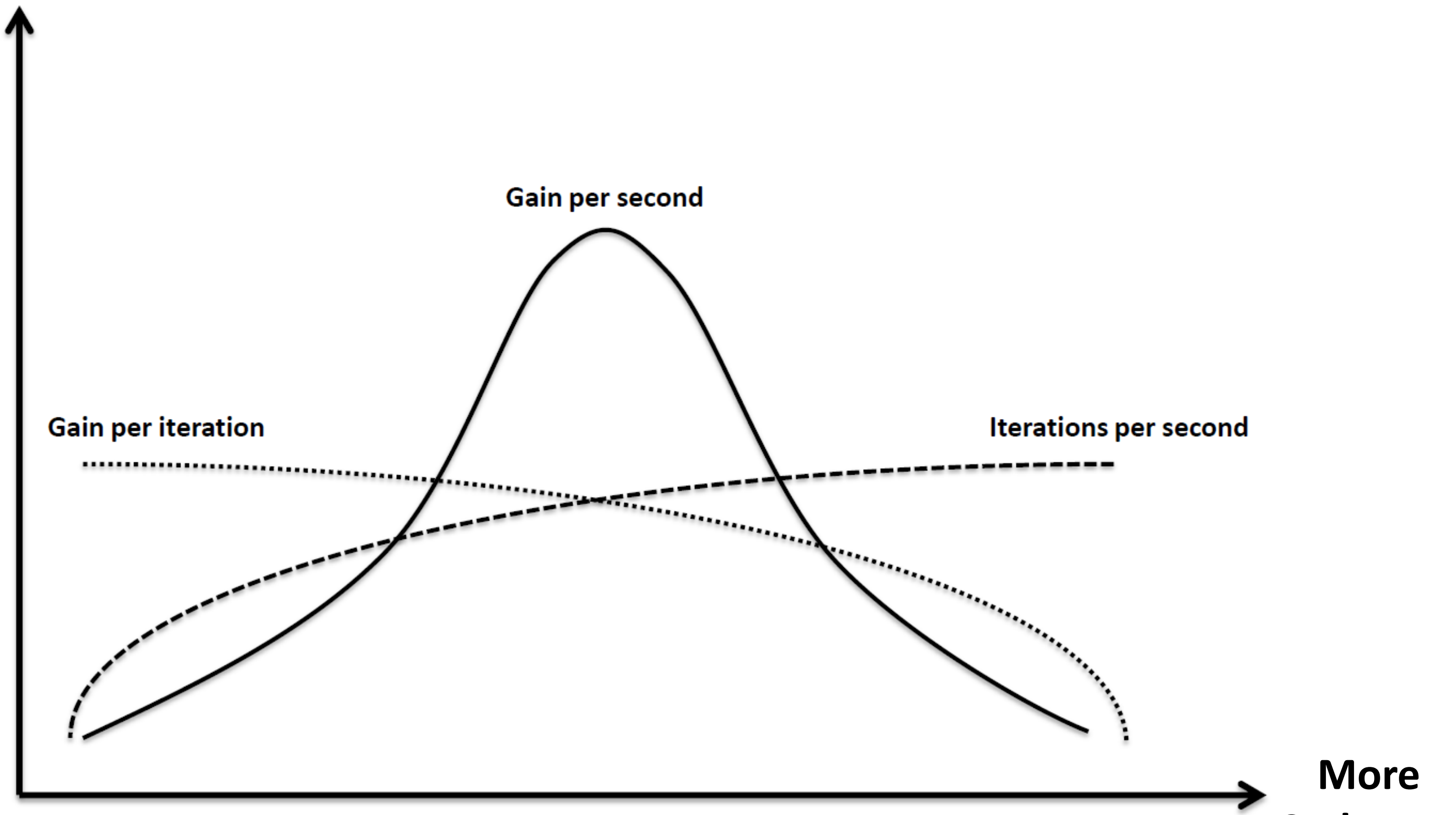


Progress per time is (iters/sec) * (progress/iter)

High staleness yields more iters/sec, but lowers progress/iter

Find the sweet spot staleness >0 for maximum progress per second

The Quality vs Quantity tradeoff



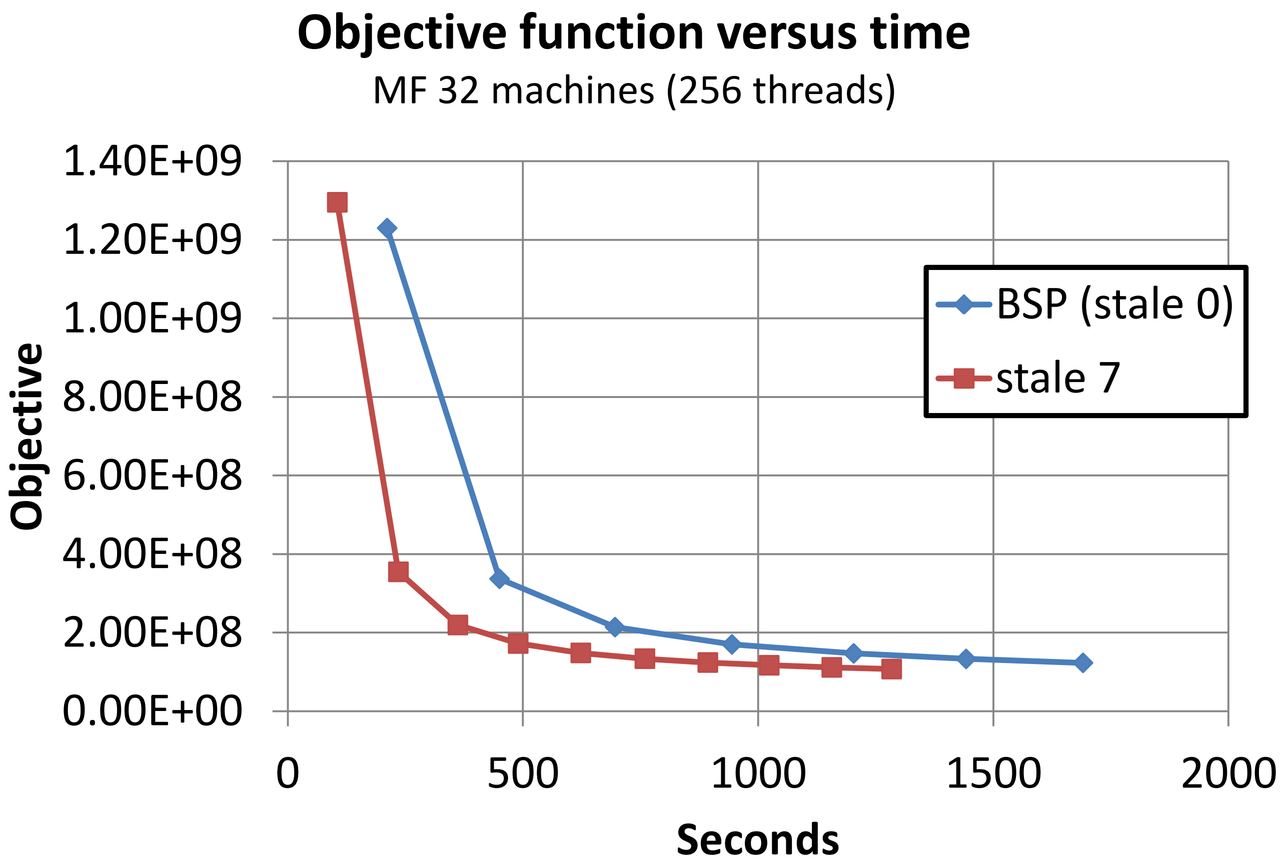
Progress per time is (iters/sec) * (progress/iter)

High staleness yields more iters/sec, but lowers progress/iter

Find the sweet spot staleness >0 for maximum progress per second

Matrix Factorization (Netflix)

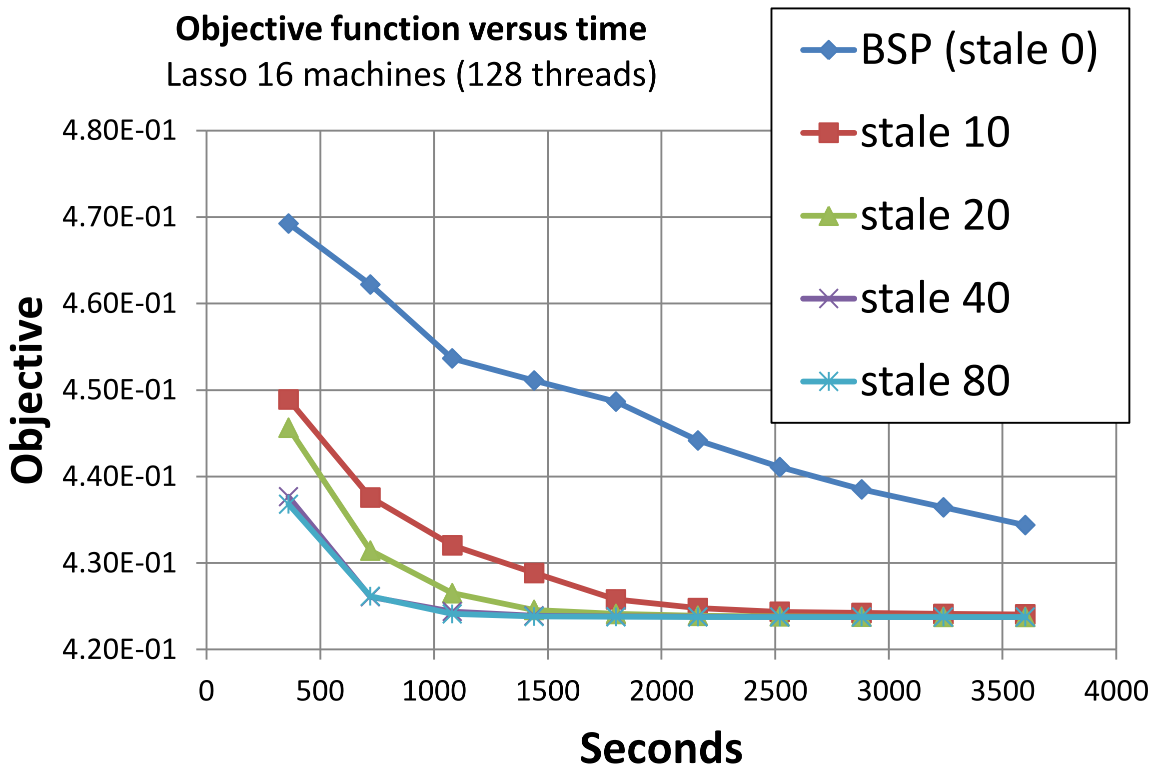
Netflix data
100M nonzeros
480K rows
18K columns
rank 100



Laboratory for Statistical Artificial InteLligence & INtegrative Genomics

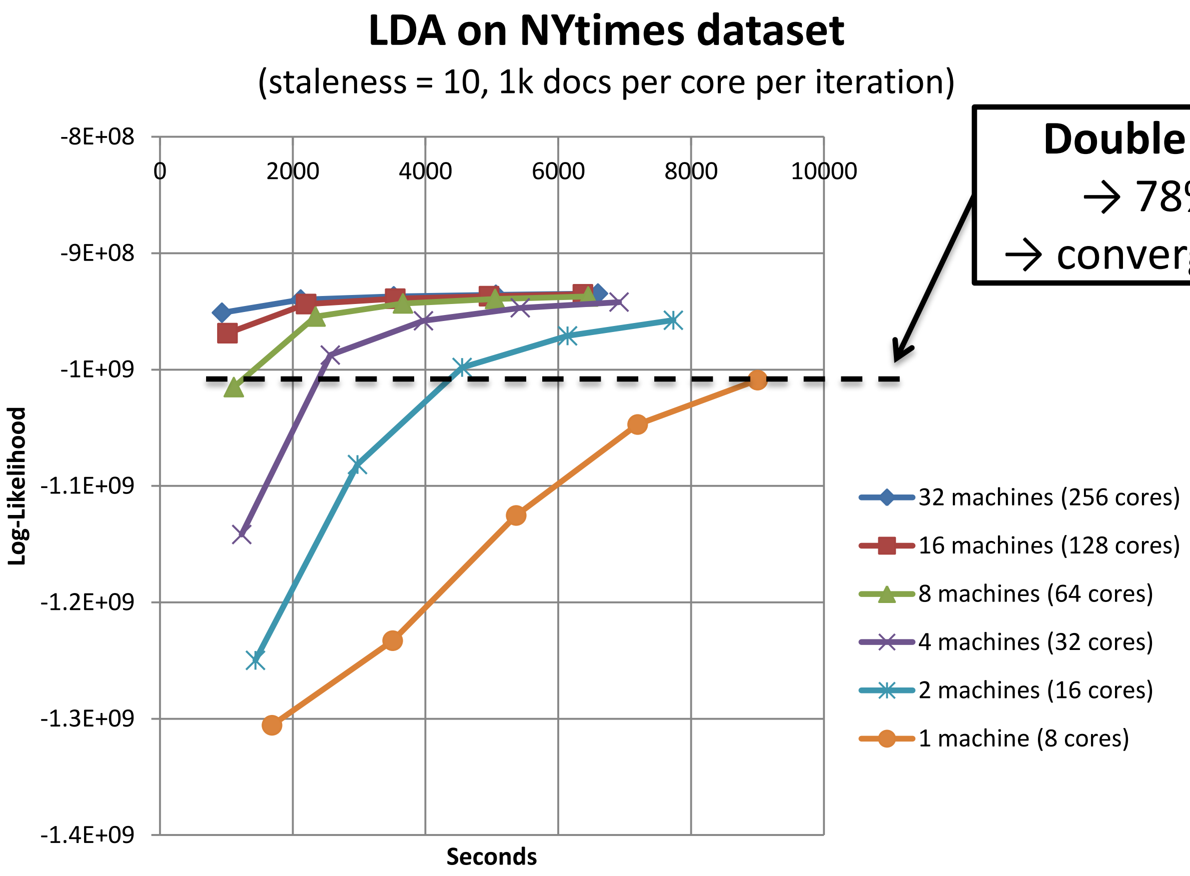
LASSO (Synthetic)

<u>Synthetic data</u>
N = 500 samples
P = 400K features

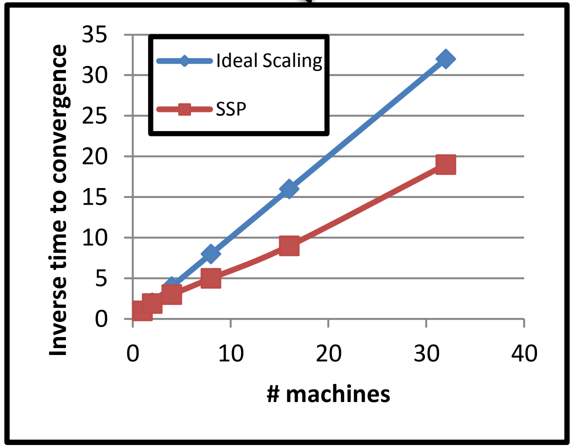


Laboratory for Statistical Artificial InteLligence & INtegrative Genomics

SSPTable scaling with # machines



Double # machines:
→ 78% speedup
→ converge in 56% time



SSP computational model scales with increasing # machines
(given a fixed dataset)

Recent Results

- Using 8 machines * 16 cores = 128 threads
  - 128GB RAM per machine
- Latent Dirichlet Allocation
  - NYTimes dataset (100M tokens, 100K words, **10K topics**)
    - **SSP 100K tokens/s**
    - GraphLab 80K tokens/s
  - PubMed dataset (**7.5B tokens**, 141K words, 100 topics)
    - **SSP 3.3M tokens/s**
    - GraphLab 1.8M tokens/s
- Network latent space role modeling
  - Friendster network sample (**39M nodes**, 180M edges)
  - 50 roles: **SSP takes 14h to converge** (vs 5 days on one machine)

Future work

- **Theory**
  - SSP for MCMC
  - Automatic staleness tuning
  - Average-case analysis for better bounds
- **Systems**
  - Load balancing
  - Fault tolerance
  - Prefetching
  - Other consistency schemes
- **Applications**
  - Hard-to-parallelize ML models
  - DNNs, Regularized Bayes, Network Analysis models

Coauthors


James Cipar


Henggang Cui


Jin Kyu Kim


Seunghak Lee


Phillip B. Gibbons


Garth A. Gibson


Gregory R. Ganger


Eric P. Xing

Workshop Demo

- **SSP is part of a bigger system: Petuum**
  - SSP parameter server
  - STRADS dynamic variable scheduler
  - More features in the works

- We have a demo!
  - **Topic modeling** (8.2M docs, 7.5B tokens, 141K words, 10K topics)
  - **Lasso regression** (100K samples, 100M dimensions, 5 billion nonzeros)
  - **Network latent space modeling** (39M nodes, 180M edges, 50 roles)

- **At BigLearning 2013 workshop (Monday)**
  - http://biglearn.org/

Summary

- **Distributed ML is nontrivial**
  - Slow network
  - Unequal machine performance

- **SSP addresses those problems**
  - Efficiently use network resources; reduces waiting time
  - Allows slow machines to catch up
  - Fast like Async, converges like BSP

- **SSPTable parameter server provides easy table interface**
  - Quickly convert single-machine parallel ML algorithms to distributed

- Slides: `www.cs.cmu.edu/~qho/ssp_nips2013.pdf`

SAILING LAB
Laboratory for Statistical Artificial InteLligence & INtegrative Genomics

PARALLEL DATA LABORATORY
CARNEGIE MELLON UNIVERSITY