CS 598sm
Probabilistic & Approximate Computing

http://misailo.web.engr.Illinois.edu/courses/cs598
Nondeterministic Approximation in Parallel Computations

Removing synchronization and reading stale data
Various techniques over the years:

• Dropping tasks (Rinard 2006 ICS)
• Removing barriers (Rinard 2007 OOPSLA)
• Reading stale data (Thies et al. PLDI 2011)
• Removing locks
• Parallelizing with data races (Misailovic et al. 2012, 2013)
• Breaking data dependencies
• ...
Some Early Insights

Figure 4. Pseudocode of the best-effort iterative-convergence template.

```
iterate
{
    mask[1:M] = filter(...);
    parallel_iterate (i = 1 to M with mask[1:M] batch P)
    {
        ...
    }
} until converged(...);
```

We observe that the proposed iterative convergence template can be used to explore best-effort computing in three different ways.

- The selection of appropriate filtering criteria that reduce the computations performed in each iteration.
- The selection of convergence criteria that decide when the iterations can be terminated.
- The use of the batch operator to relax data dependencies in the body of the parallel_iterate.
Some Early Insights

Convergence-based pruning: Use converging data structures to speculatively identify computations that have minimal impact on results and eliminate them.

Staged Computation: Consider fewer points in early stages; gradually use more points in later stages to improve accuracy.

Early Termination: Aggregate statistics to estimate accuracy and terminate before full convergence.

Sampling: Select a random subset of input data and use it to compute the results.

Dependency Relaxation: Ignore potentially redundant dependencies across iterations. Leads to more degree of parallelism or coarser granularity.
Data Dependence

A data dependence from statement $S_1$ to statement $S_2$ exists if

1. there is a feasible execution path from $S_1$ to $S_2$, and
2. an instance of $S_1$ references the same memory location as an instance of $S_2$ in some execution of the program, and
3. at least one of the references is a store.
Kinds of Data Dependence

**Direct Dependence**

\[ X = \ldots \]
\[ \ldots = X + \ldots \]

**Anti-dependence**

\[ \ldots = X \]
\[ X = \ldots \]

**Output Dependence**

\[ X = \ldots \]
\[ X = \ldots \]
Dependence Graph

A dependence graph is a graph with:

- Each node represents a statement, and
- Each directed edge from S1 to S2, if there is a data dependence between S1 and S2 (where the instance of S2 follows the instance of S1 in the relevant execution).
  - S1 is known as a source node
  - S2 is known as a sink node
Kinds of Data Dependence

**Direct Dependence**

S1: \( X = \ldots \)
S2: \( \ldots = X + \ldots \)

**Anti-dependence**

S1: \( \ldots = X \)
S2: \( X = \ldots \)

**Output Dependence**

S1: \( X = \ldots \)
S2: \( X = \ldots \)
A dependence graph is a graph with:

- one node per statement, and
- a directed edge from S1 to S2 if there is a data dependence between S1 and S2 (where the instance of S2 follows the instance of S1 in the relevant execution).

For loops: dependence graph is a summary of unrolled dependencies for different iterations

- Some (detailed) information may be lost
Dependence in Loops

def X(), Y(), a(), i;
    do i = 1 to N
        S1: \[ X(i) = a(i) + 2 \]
        S2: \[ Y(i) = X(i) + 1 \]
    enddo
Dependence in Loops

def X(), Y(), a(), i;
doi = 1 to N

S1: \[ X(i+1) = a(i) + 2 \]
S2: \[ Y(i) = X(i) + 1 \]
enddo
Dependence in Loops

def X(), Y(), a(), i;

do i = 2 to N

S1: \[ X(i) = a(i) + 2 \]
S2: \[ Y(i) = X(i-1) + 1 \]

enddo
Dependence in Loops

def X(), Y(), a(), i;

do i = 1 to N

S1:       X(i) = a(i) + 2

S2:       Y(i) = X(i+1) + 1

enddo
Dependence in Loops

def X(), Y(), a(), i, t;
do i = 1 to N
    S1: t = a(i) + 2
    S2: Y(i) = t + 1
enddo
Dependence in Loops

def X(), Y(), a(), i, t();
doi = 1 to N

S1: t(i) = a(i) + 2
S2: Y(i) = t(i) + 1

enddo
STOCHASTIC GRADIENT DESCENT (SGD)

Slides based on Linyi Li’s Talk in CS 598 Last Year
MACHINE LEARNING AS OPTIMIZATION PROBLEM

\[
\text{minimize}_\theta \ L(\theta) = \mathbb{E}_{x \sim D} \ell(x, f(x, \theta))
\]

- \( \theta \in \mathbb{R}^P \) : model parameter
- \( D \) : data distribution
- \( x \in \mathbb{R}^n \) : data sample
- \( f(\cdot, \cdot) \) : the model output given input and parameters
- \( \ell(\cdot, \cdot) \) : loss function;

it’s smaller, closer \( f(x, \theta) \) it gets to the ground truth
FINITE DATASET

- Usually, the dataset is finite.
- Suppose there are $N$ data samples, then it becomes

$$\text{minimize}_\theta \ L(\theta) = \frac{1}{N} \sum_{i=1}^{N} \ell(x_i, f(x_i, \theta))$$
EXAMPLE

- Each sample $x_i$ is given a true label $y_i \in \{0, \ldots, 9\}$.
- Model outputs 10-dimension confidence vector in $[0,1]^{10}$ summing up to 1.
- The cross-entropy loss on the sample:

$$\ell(x_i, f(x_i, \theta)) = - \sum_{k=1}^{C} \mathbf{1}[k = y_i] \log(f(x_i, \theta))_k$$

$$= \log(1/f(x_i, \theta)_{y_i})$$

Smaller loss, higher confidence on the correct label, and higher accuracy.
A common way to solve the problem, is by using SGD:

- Take the gradient of $L$ with respect to $\theta$: $\nabla_\theta L(\theta) \in \mathbb{R}^P$
- To minimize $L$, we move the $\theta$ along the opposite direction:
  $$\theta \leftarrow \theta - \gamma \nabla_\theta L(\theta)$$
  - $\gamma$: step size, a constant, positive small number
- Take sufficient such small steps, until $L(\theta)$ does not change much.
In our MNIST task, \( f(x_i, \theta)_{y_i} \) is model confidence score for correct label.

**Loss function:** \( \ell(x_i, f(x_i, \theta)) = \log(1/f(x_i, \theta)_{y_i}) \)

**Gradient:** \( \nabla_\theta \ell(x_i, f(x_i, \theta)) = -\frac{\nabla_\theta f(x_i, \theta)_{y_i}}{f(x_i, \theta)_{y_i}} \)

**Parameter update by SGD:** \( \delta = \gamma \frac{\nabla_\theta f(x_i, \theta)_{y_i}}{f(x_i, \theta)_{y_i}} \)

\( \delta \): model parameter change

**Direction:** move towards larger confidence;
  - smaller confidence, sharper change.
LEADING TO
HOGWILD!
ALGORITHM
DECOMPOSE $\nabla_\theta L(\theta)$

$$L(\theta) = \frac{1}{N} \sum_{i=1}^{N} \ell(x_i, f(x_i, \theta))$$

$$\nabla L(\theta) = \frac{1}{N} \sum_{i=1}^{N} \nabla_\theta \ell(x_i, f(x_i, \theta))$$

- A serial algorithm:
  
  while (!converged($\theta$))
  for (int i=0; i<N; ++i)
      $\theta = \theta - 1/N \times \nabla_\theta \ell(x_i, f(x_i, \theta))$
DECOMPOSE $\nabla_\theta L(\theta)$

\[
L(\theta) = \frac{1}{N} \sum_{i=1}^{N} \ell(x_i, f(x_i, \theta))
\]

\[
\nabla L(\theta) = \frac{1}{N} \sum_{i=1}^{N} \nabla_\theta \ell(x_i, f(x_i, \theta))
\]

- A serial algorithm:

\[
\text{while (!converged(}\theta\text{))}
\]
\[
\text{for (int } i=0; i<N; ++i)
\]
\[
\theta(t) = \theta(t-1) - 1/N * \nabla_\theta \ell(x_i, f(x_i, \theta(t-1)))
\]
PARALLELISM?

- A serial algorithm
  
  ```
  for (int i=0; i<N; ++i)
    for (int j=0; j<|\theta|; ++j)
      \theta_j = \theta_j^{old} - 1/N * \nabla_{\theta_j} \ell(x_i, f(x_i, \theta^{old}))
  ```

- One way to Parallelize

  ```
  #parallel across K threads:
  for (int i=k*N/K; i < (k+1)* N/K; ++i)
    for (int j=0; j<|\theta|; ++j)
      \theta_j = \theta_j^{old} - 1/N * \nabla_{\theta_j} \ell(x_i, f(x_i, \theta^{old}))
  ```
- Inner loop:
  
  \[
  \text{for (int } j=0; j<|\theta|; ++j) \]
  
  \[
  G(\theta_j^{old}) = \ldots \]
  
  \[
  \theta_j = \theta_j^{old} - G(\theta_j^{old}) \]
  
- With some transformation:
  
  \[
  \text{for (int } j=0; j<|\theta|; ++j) \]
  
  \[
  G(\theta_j^{old}) = \ldots \]
  
  \[
  \text{if } G(\theta_j^{old}) \neq 0 \]
  
  \[
  \theta_j^{new} = \theta_j^{old} - G(\theta_j^{old}) \]

For each sample:
- Only small number of parameters updated;
- These parameters rarely overlap.

Thread 1

Read: \(\theta_j^{(old)}\)

Write \(\theta_j^{(1)}\)

Thread 2

Read: \(\theta_j^{(old)}\)

Overwrite \(\theta_j^{(1)}\) with \(\theta_j^{(2)}\)
PARALLELISM?

- The version “RR” tries to improve on the locking cost by using a round-robin schedule of updates.
- The version “AIG” does a fine locking of the elements of $\theta$.
- Most of the time, the change will be for individual element of $\theta$, but even fine-grained locking is expensive.
KEY OBSERVATION: SPARSE SEPARABILITY

- The updates, even with the overwrite may give a good ‘delta’ direction.
- Potential threat: it may not give ‘strong enough’ direction indication.

- For many real-world problems, the model:
  - Usually has \textbf{large number} of parameters.
  - Only uses \textbf{a small fraction} of parameters to predict each data sample.
  - Parameters used for predicting different samples \textit{rarely overlap}.
  - Each parameter is \textit{not often} used.
EXAMPLES

- **Sparse SVM:**
  - Data vector $x_i$’s are sparse.

- **Matrix Completion:**
  - Learn large matrix $M$ as the product of $AB$, from few cells $M_{ij}$’s.

- **Graph Cuts**
  - Partition graph nodes according to sparse similarity matrix.
RESULT ALGORITHM

- Update without lock is totally practical!
- Hogwild algorithm:

```plaintext
Algorithm 1 HOGWILD! update for individual processors
1: loop
2: Sample e uniformly at random from E
3: Read current state $x_e$ and evaluate $G_e(x_e)$
4: for $v \in e$ do $x_v \leftarrow x_v - \gamma G_{ev}(x_e)$
5: end loop
```

- $e$ is data sample $x_v = \theta$, $G_e(x_e)$ is gradient.
- no lock on shared parameters $x_e$, totally asynchronous.
ASSUMPTIONS

We assume Lipschitz continuous differentiability of $f$ with Lipschitz constant $L$:

$$\|\nabla f(x') - \nabla f(x)\| \leq L\|x' - x\|, \quad \forall x', x \in X.$$  \hspace{1cm} (8)

We also assume $f$ is strongly convex with modulus $c$. By this we mean that

$$f(x') \geq f(x) + (x' - x)^T \nabla f(x) + \frac{c}{2}\|x' - x\|^2, \quad \text{for all } x', x \in X.$$  \hspace{1cm} (9)

When $f$ is strongly convex, there exists a unique minimizer $x_*$ and we denote $f_* = f(x_*)$. We additionally assume that there exists a constant $M$ such that

$$\|G_e(x_e)\|_2 \leq M \quad \text{almost surely for all } x \in X.$$ \hspace{1cm} (10)

We assume throughout that $\gamma c < 1$. (Indeed, when $\gamma c > 1$, even the ordinary gradient descent algorithms will diverge.) Our main results are summarized by the following
THEORETICAL GUARANTEE

- Condition:
  - Convex function;
  - Gradient magnitude is bounded;
  - Number of workers is less than $n^{1/4}$, $n$ is number of parameters;
  - Fine-tuned step size.

- After $k \geq \Theta \left( \frac{\log(1/\epsilon)}{\epsilon} \right)$ steps, $\mathbb{E}[f(x_k) - f_*] \leq \epsilon$.

- Serial SGD convergence rate: $\Theta(1/\epsilon)$.

- Hogwild can be further optimized to get the same rate.
EXPERIMENTS

Baseline approaches:

- RR: processors are ordered; each update the decision variable in order
- AIG: only lock particular parameters when updating ($\theta_i$’s with gradients)
- Hogwild: no locking

Three applications:

- SVM (Sparse SVM), MC (Matrix Completion), Cuts (Graph Cuts)
- **Speed**: Much faster than ordered locked update.
  - 9.5s vs 61.8s; 301.0s vs 2569.1s
- **Accuracy**: Almost the same training & test error.
Hogwild is much faster. Even only adding locks to all parameters, may significantly slow it down.

Figure 2: Total CPU time versus number of threads for (a) RCV1, (b) Abdomen, and (c) DBLife.
MATRIX COMPLETION PROBLEM

- Same trends for different datasets.
- Does not hurt accuracy.
- When gradient computation becomes slow, the gap shrinks.
GENERALIZATION & RECENT PROGRESS
The paper released in 2011, NN was not popular.
SGD is also popular for NN training
NN is non-convex, no theoretical guarantee.

Can Hogwild generalize to NN?
IN TENSORFLOW

- Originally designed to use Hogwild (named asynchronous parameter updates).
- Also supports synchronous and synchronous with backups.
- See Tensorflow paper OSDI 2016

- In 2016, “Revisiting Distributed Synchronous SGD” (ICLR 2016 Workshop) experimented with comparing the strategies.
- **Async**: similar to Hogwild
- **Sync**: lock and update; optimized

- In Hogwild, though each step may be faster, but more steps to converge.
- Slightly hurts accuracy, and takes more time to converge.

---

**Figure 8**: Convergence of Sync-Opt and Async-Opt on Inception model using varying number of machines. Sync-Opt with backup workers converge faster, with fewer epochs, to higher test accuracies.
RECENT APPROACH

- Synchronous with backup workers:
  - $n$ workers, but each step only requires $m < n$ workers’ result to update.
- Overcome stragglers.
- SGD samples training data randomly;
  - each worker processes different batch;
  - OK if ignored.
Backup workers accelerates synchronous up to roughly 10%.

For large NN, gradient computing is the bottleneck

- Hogwild is not fast that much.
LET’S THINK NOW ABOUT GENERAL PROGRAMS

- Removing synchronization and reading stale data
- Various techniques over the years:
  - Dropping tasks (Rinard 2006 ICS)
  - Removing barriers (Rinard 2007 OOPSLA)
  - Reading stale data (Thies et al. PLDI 2011)
  - Removing locks
  - Parallelizing with data races (Misailovic et al. 2012, 2013)
  - Breaking data dependencies
  - ...

Studying various iterative and non-iterative programs, typical speedup is around 20% to 30%
Kinds of Dependencies

- **Actual**: exist in the program
- **State**: exist in the program and can be satisfied with extra code to match the original result, but faster than conventional
- **Apparent**: do not exist, but the compiler/compiler cannot prove that they are unnecessary

Strict preservation of every actual dependencies may not necessary, Preservation on any apparent dependency is not necessary
Dependencies in Non-deterministic Codes?

- For the same input, nondeterministic programs produce different results in each run.
- Use the error margins of the ordinary execution to find less important dependencies.
- Non-determinism masks broken (unsatisfied) dependencies.
- Use inexpensive checks to make sure the speculative execution matches those expected from the original program.
Figure 2. Output variability of nondeterministic PARSEC benchmarks. Several exhibit very high variability and are particularly amenable to STATS.
Opportunity State Dependency

- Thread level parallelism is constrained by a sequential chain of dependences
- Opportunity: break this dependence to increase parallelism
- Fix: do ‘speculation’, if the result is too different, drop those updates and reexecute
Approach

Break the dependency occasionally
• Run inexpensive transfer function
  Ensure that the impact is not large
  • If small, continue,
  • If large, reexecute (infrequently)
Code Modification

Bodytrack: Pose estimation program

```cpp
void estimateLocations() {
    vector<int> frameIds(numFrames);
    vector<Particle> model(numParticles);
    vector<BodyPart> positions;
    for(auto frameId : frameIds) {
        Frame f = getFrame(frameId);
        model = updateModel(numAnnealingLayers, model, f);
        positions = getPositions(model);
    }
}
```

Figure 7. Original code of bodytrack.

```cpp
class Input { int frameId; };
class Output { vector<BodyPart> positions; };
class State {
    vector<Particle> model;
    State& operator=(State&);
    bool doesSpecStateMatchAny(set<State*>);
};
Output* computeOutput(Input* i, State* s) {
    Frame f = getFrame(i->frameId);
    s->model = updateModel(T0_numAnnealingLayers, s->model, f);
    Output* o = new Output();
    o->positions = getPositions(s->model);
    return o;
}
void estimateLocations() {
    vector<Input*> i(numFrames);
    vector<Particle> model(numParticles);
    State s; s.model = model;
    StateDependence<Input, State, Output>
    stateDep(&i, &s, computeOutput);
    stateDep.start(); stateDep.join();
}
```

Figure 8. Use of SDI in bodytrack.
Figure 12. For most benchmarks, STATS generates a significant amount of extra parallelism that saturates the hardware resources of our platform. “Original” is the out-of-the-box benchmark that has been parallelized by traditional means. “Seq. STATS” (“Par. STATS”) is the binary generated by STATS starting from the sequential (multi-threaded) version of a benchmark. The bar graphs show maximum speedup.
Energy Consumption

Figure 15. The binaries generated by STATS use considerably less energy compared to the original benchmarks.
Accuracy Impact: Can run more

Figure 16. STATS can increase the original output quality by spending the saved time to iterate more over the same dataset.

Where is it good to use: Applications that analyze a long stream of data (e.g., bodytrack, facedet, streamcluster) where the information about inputs that is automatically computed (e.g., 3D location of bodies, 2D location of faces, centroids of multi-dimensional points) has the “short memory” dependence property.
Transient hardware errors are a rising concern. Traditional hardware redundancy is too expensive. Software-driven solutions are promising, but some errors escape as Silent Data Corruptions (SDCs).

As technology scales, hardware reliability is more important. Hardware is more susceptible to transient (soft) errors.

Many applications require very high reliability guarantees. Volkswagen reported ~20% disengagements due to software hang/crashes, WAYMO, CA DMV 2016 Dataset, DSN 2018.
Unreliable Hardware

Architects make great efforts to minimize errors

Some errors slip through the cracks – silently corrupt computation results

Image from “Inter-Agency Workshop on HPC Resilience at Extreme Scale”, DoD, ‘12
Erroneous executions (has soft errors)

Error-free execution

Masked

Detected

Silent Data Corruption (SDC)

Failure

No output

Output

Output

Output

Output

Graphic by Abdulrahman Mahmoud
How do We See at Software Level?

float x:

- Sign
- Exponent (8 bit)
- Mantissa (23 bits)

Often large impact

Often small impact
How do We See at Software Level?

Corrupted Bits

int x:

But also int* x... what happens then?
Challenges and Traditional Solutions

Detection:
• Run twice, compare the results
• Instruction Replication
• Algorithm-based fault tolerance

Recovery:
• Checkpoint-restart
• Run three times, do majority voting
Challenges and Approximate Solutions

Detection:
• Run twice, compare the results
• Instruction Replication
• Algorithm-based fault tolerance

Recovery:
• Checkpoint-restart
• Run three times, do majority voting

Run exact and approximate versions, ensure they don’t differ by too much
Challenges and Approximate Solutions

Detection:
- Run twice, compare the results
- Instruction Replication
- Algorithm-based fault tolerance

Recovery:
- Checkpoint-restart
- Run three times, do majority voting

Replicate only some instructions
For the others, either rely on the property of the computation or develop inexpensive checkers
Challenges and Approximate Solutions

Detection:
• Run twice, compare the results
• Instruction Replication
• Algorithm-based fault tolerance

Recovery:
• Checkpoint-restart
• Run three times, do majority voting

Make the algorithmic techniques aware of the approximation
Challenges and Approximate Solutions

Detection:
• Run twice, compare the results
• Instruction Replication
• Algorithm-based fault tolerance

Recovery:
• Checkpoint-restart
• Run three times, do majority voting

Checkpoint only a small part of the state
Restart only when necessary
Challenges and Approximate Solutions

Detection:
• Run twice, compare the results
• Instruction Replication
• Algorithm-based fault tolerance

Recovery:
• Checkpoint-restart
• Run three times, do majority voting

If we need to re-execute, run only approximate algorithm
Try to do ‘local repair’ on the output
Lightweight Check and Recover

\[ z = x \times y \]
\[ z' = x \times y \]
\[ z == z' ? \]

Code Re-Execution
(SWIFT, DRIFT, Shoestring)

\[ y = \text{foo}(x) \]
\[ \text{DNN}(x, y) ? \]

Anomaly Detection
(Topaz, Rumba)

\[ s = \text{SAT}(p) \]
\[ \text{verify}(s, p) ? \]

Verification
(for NP-Complete)

Slide by Keyur Joshi
Reliability is the probability of obtaining the *exact* answer.
The Try-Check-Recover Mechanism

Some research languages\textsuperscript{1,2} expose \textit{Try-Check-Recover mechanisms}.

\begin{verbatim}
try { solution = SATSolve(problem) }
check { satisfies(problem, solution) }
recover { solution = SATSolve(problem) }
\end{verbatim}

\textsuperscript{1}“Relax”, M. de Kruijf, S. Nomura, and K. Sankaralingam, ISCA ’10
\textsuperscript{2}“Topaz”, S. Achour and M. Rinard, OOPSLA ’15

Slide by Keyur Joshi
Code Re-Execution – SWIFT\(^1\)

// Instruction 1
try { z = x*y \([p\_try] \) \(\text{rnd}()\); } 
check { z == (x*y \([p\_try] \) \(\text{rnd}()\)) } 
recover { z = x*y \([p\_rec] \) \(\text{rnd}()\); }

// Instruction 2
try { w = x+y \([p\_try] \) \(\text{rnd}()\); } 
check { w == (x+y \([p\_try] \) \(\text{rnd}()\)) } 
recover { w = x+y \([p\_rec] \) \(\text{rnd}()\); }

\(^1\)G. A. Reis, J. Chang, N. Vachharajani, R. Rangan, and D. August, CGO ‘05
// Instruction 1 and 2
try {
    z = x*y [p_try] rnd();
    w = x+y [p_try] rnd();
}
check {
    z == (x*y [p_try] rnd()) && w == (x+y [p_try] rnd())
}
recover {
    z = x*y [p_rec] rnd();
    w = x+y [p_rec] rnd();
}

¹K. Mitropoulou, V. Porpodas, and M. Cintra, LCPC '13
Code Re-Execution – Shoestring

// Instruction 1
try { z = x*y [p_try] rnd(); }
check { z == (x*y [p_try] rnd()); }
recover { z = x*y [p_rec] rnd(); }
// Instruction 2 not considered critical
w = x+y [p_try] rnd();

1S. Feng, S. Gupta, A. Ansari, and S. Mahlke, ASPLOS ‘10
Anomaly Detection – Topaz

try {
    z = f(x,y) [p_try] rnd();
}
check {
    isUnusual(x,y,z)
}
recover {
    z = f(x,y) [p_rec] rnd();
}
Hardware Error Flag\textsuperscript{1,2}

```c
try {
    z = x*y [p_try] rnd();
}
check {
    !(read_hw_err_flag())
}
recover {
    z = x*y [p_rec] rnd();
}
```

\textsuperscript{1}“Relax”, M. de Kruijf et al., ISCA ’10 \quad \textsuperscript{2}“Replica”, V. Fernando et al., ASPLOS ’19