Lockless Stochastic Gradient Descent.

/SEOUL AI/

1. SDG and, more generally, the problem class considered
2. Lockless...
3. (Selected elements of) Convergence proof
4. A practical, code, mini, example in C++
5. Going further...

The problem class considered

$$\min_{w} \frac{1}{n} \sum_{i=1}^{n} f_i(w)$$

n is 'large'

Assumptions on f_i.

f_i is convex, L-smooth:

$$\exists L, \ \|\nabla f_i(a) - \nabla f_i(b)\| \leq L\|a - b\|$$

f_i is μ -strongly convex:

$$\exists \mu, \ \|\nabla f_i(a) - \nabla f_i(b)\| \geq \mu\|a - b\|$$

Lockless...
Several threads access a same state u.

- Atomic read u_m.
- Compute update u_{m+1}.
- Write update atomically. *This update might be overwritten.*
- Parallel batches interpersed by 'synchronized' common blocks.

Later summarized by a diagonal matrix B that holds 0 on data overwritten, and 1 otherwise.

Convergence proof

1. Algorithm:AsySVRG
   - Convergence speed $O(1/\tilde{T}), \tilde{T}$ a measure of total work, as opposed to $O(1/\sqrt{\tilde{T}})$ for Hogwild!
   - Key difference lies in using the *full* gradient.
2. Key convergence analysis steps

Algorithm:AsySVRG

ZHAO & LI 2016

$$
\begin{aligned}
& Initialization: \ p\, threads, \ initialize\, w_0, \ \eta; \\
& for\, t = 0, 1, 2, \ldots, T-1\, do \\
& \quad u_0 = w_t; \\
& \quad All\, threads\, compute\, the\, full\, gradient \\
& \quad \quad \nabla f(u_0) = \frac{1}{n} \Sigma_{i=1}^{n} \nabla f_i(u_0)\ in\, parallel; \\
& \quad u = w_t; \\
& \quad For\, each\, thread, \ do: \\
& \quad for\, j = 0\, to\, M-1\, do \\
& \quad \quad atomically: \ \hat{u} = u \\
& \quad \quad pickup\, i\, randomly\, in\, \{1, \ldots, n\} \\
& \quad \quad \hat{v} = \nabla f_i(\hat{u}) - \nabla f_i(u_0) + \nabla f(u_0) \\
& \quad \quad atomically: \ u \leftarrow u - \eta \hat{v} \\
& \quad end\ for \\
& \quad atomically: \ w_{t+1} = u \\
& end\ for
\end{aligned}
$$

CONVERGENCE

Let us write an equivalent write sequence $\{u_{t,m}\}$ for the t^{th} outer loop.

$$\begin{aligned} u_{t,0} &= w_t \\ u_{t,m+1} &= u_{t,m} - \eta B_{t,m} \hat{v}_{t,m} \end{aligned}$$

with

$$\hat{v}_{t,m} = \nabla f_{i_{t,m}}(\hat{u}_{t,m}) - \nabla f_{i_{t,m}}(u_{t,0}) + \nabla f(u_{t,0})$$

$B_{t,m}$ diagonal with entries 0 or 1.

CONVERGENCE [2]

$\hat{u}_{t,m}$ is read by the thread that computes $\hat{v}_{t,m}$. It is represented as:

$$\hat{u}_{t,m} = u_{t,a(m)} - \eta \sum_{j=a(m)}^{m-1} P^{(t)}_{m,j-a(m)} \hat{v}_{t,j}$$

With matrix $P^{(t)}_{m,j-a(m)}$ diagonal with entries 0 or 1. $a(m)$ a timing such as $0 \leq m - a(m) \leq \tau$.

KEY LEMMA

Let $p_i(x) = \nabla f_i(x) - \nabla f_i(u_{t,0}) + \nabla f(u_{t,0})$
And $q(x) = \frac{1}{n} \sum_{i=1}^{n} \|p_i(x)\|^2$

In AsySVRG, we have
$E[q(\hat{u}_{t,m})] < \rho E[q(\hat{u}_{t,m+1})]$
if we choose ρ and η so that (constraint on learning rate)

$$\frac{1}{1 - \eta - \frac{9\eta(\tau+1)L^2(\rho^{\tau+1}-1)}{\rho-1}} \leq \rho$$

INTRODUCING $r > 0$

Let x, y

$$
\begin{aligned}
& \|\nabla f_i(x)\|^2 - \|\nabla f_i(y)\|^2 \\
\leq \quad & 2\nabla f_i(x)^T \left(\nabla f_i(x) - \nabla f_i(y)\right) \\
& \qquad (compare\, \nabla f_i(x)^2 \; and \, \nabla f_i(x)^2 + (\nabla f_i(x) - \nabla f_i(y))^2) \\
\leq \quad & \tfrac{1}{r}\|\nabla f_i(x)\|^2 + r\|\nabla f_i(x) - \nabla f_i(y)\|^2 \\
\leq \quad & \tfrac{1}{r}\|\nabla f_i(x)\|^2 + rL^2\|x - y\|^2
\end{aligned}
$$

NEXT STEP...

$$\begin{aligned} & \|\nabla f_i(\hat{w}_t)\|^2 - \|\nabla f_i(\hat{w}_{t+1})\|^2 \\ \leq \quad & \frac{1}{r}\|\nabla f_i(\hat{w}_t)\|^2 + rL^2\|\hat{w}_t - \hat{w}_{t+1}\|^2 \end{aligned}$$

USING THE DEFINITION OF $\hat{w}_t$

$$\|\hat{w}_t - \hat{w}_{t+1}\| \leq 3\eta \sum_{j=t-\tau}^{t} \|\nabla f_{i_j} \hat{w}_j\|$$

f ONLY

$$\|\nabla f_i(\hat{w}_t)\|^2 - \|\nabla f_i(\hat{w}_{t+1})\|^2$$

$$\leq \frac{1}{r}\|\nabla f_i(\hat{w}_t)\|^2 + 9r(\tau+1)L^2\eta^2 \sum_{j=t-\tau}^{t} \|\nabla f_{i_j}(\hat{w}_j)$$

FIX i, AVERAGE OVER (RANDOM INDEX) i_j

$$\|\nabla f_i(\hat{w}_t)\|^2 - \|\nabla f_i(\hat{w}_{t+1})\|^2$$

$$\leq \quad \frac{1}{r}\|\nabla f_i(\hat{w}_t)\|^2 + 9r(\tau+1)L^2\eta^2 \sum_{j=t-\tau}^{t} q(\hat{w}_j)$$

SUM UP FROM 1 TO n

$$Eq(\hat{w}_t) - Eq(\hat{w}_{t+1})$$

$$\leq \quad \frac{1}{r}Eq(\hat{w}_t) + 9r(\tau+1)L^2\eta^2 \sum_{j=t-\tau}^{t} q(\hat{w}_j)$$

USE INDUCTION, TAKE $r = \frac{1}{\eta}$, INTRODUCE ρ

$$Eq(\hat{w}_t) \leq \frac{1}{1 - \eta - \frac{9\eta(\tau+1)L^2(\rho^{\tau+1}-1)}{\rho-1}} Eq(\hat{w}_j) \leq \rho Eq($$

Dummy problem considered

$$\underset{x}{argmin} \sum_i (a_i \,.\, x - b_i)^2$$

$$a_i[j] = \frac{i+j}{(j_{max}+1).(i_{max}+1)}$$

$$b_i = 1 + i * 0.01$$

Recipe... Data structure used.

```
struct LocklessSGD {
std::atomic < double * >  last_state;
std::atomic < int > version;
  ...

  struct Thread {
    unsigned vect_size;
    double* vect;
    double* read;
    LocklessSGD* root;
    ...
  }
};
```

Recipe... Read vector 'atomic but can fail', one possible implementation.

```
inline void AtomicButCanFailRead(double **v) {
  auto ve = root->version.load();
  double *tgt = root->last_state.load();
  for (unsigned i = 0; i < vect_size; ++i) { read[i] = tgt[i]; }
  if (ve == root->version.load()) {
    if (tgt == root->last_state.load()) {
      auto t = *v;
      *v = read;
      read = t;
    }
  }
}
```

(n.b. real atomic would require anti ABA pattern)

Recipe... Add vector 'atomic but can fail', one possible implementation.

```
inline void AtomicButCanFailAdd(const double * v, double lr) {
  auto ve = root->version.load();
  double *tgt = root->last_state.load();
  for (unsigned i = 0; i < vect_size; ++i) {
    vect[i] = tgt[i] + v[i] * lr;
  }
  if (ve == root->version.load()) {
    if (std::atomic_compare_exchange_strong(& root->last_state,
                                             & tgt, vect)) {
      vect = tgt;
      ++root->version; //atomic
    }
  }
}
```

Recipe... CPU affinity

```
inline void SetAffinityMask(int core) {
	HANDLE process;
	DWORD_PTR processAffinityMask;
	for (int i = 0; i< ncores; i++) processAffinityMask |= 1 <<
	process = GetCurrentProcess();
	SetProcessAffinityMask(process, processAffinityMask);
	HANDLE thread = GetCurrentThread();
	DWORD_PTR threadAffinityMask = 1 << (2 * core);
	SetThreadAffinityMask(thread, threadAffinityMask);
}
```

3 variants considered.

1. Vector of atomic doubles
2. Vector of non-atomic doubles
3. 'Atomic can fail' global gradient update: 'my trick'

Some results.
/DUMMY EXAMPLE/
Dimension = 1000, sum of 1000 terms,
'by 100 outer loops (syn'd),
'by group of 10',
600,000 iterations...
AMD Ryzen 1700 (8 cores, 16 threads)
'my trick'

#threads	time	loss	'efficiency'	n.b.
1	18.2261s	1.70219e-13	1	
2	9.8457s	1.35281e-25	0.925586	
8	3.61774s	2.70749e-16	0.629746	goes much higher with long running batches
16	4.69994s	8.25322e-05	0.242371	Hyperthreading!

Some results.
/DUMMY EXAMPLE/
Dimension = 1000, sum of 1000 terms,
'by 100 outer loops (syn'd),
'by group of 10',
600,000 iterations...
AMD Ryzen 1700 (8 cores, 16 threads)
'real async: element by element'

#threads	**time**	**loss**	**'efficiency'**	**n.b.**
1	24.9629s	1.70219e-13	1	
2	13.2756s	5.99513e-25	0.940182	
8	12.2971s	2.70749e-16	0.253748	too many memory barriers
16	5.06642s	2.45501e-09	0.307946	Hyperthreading!

More Examples /DUMMY EXAMPLE/
Dimension = 1000, sum of 100 terms,
'by 100 outer loops (syn'd),
'by group of 1000',
600,000 iterations...
AMD Ryzen 1700 (8 cores, 16 threads)
'no sync: element by element, non-atomic'

#threads	time	loss	'efficiency'	n.b.
1	199.26s	1.60923e-27	1	
2	101.867s	1.08698e-05	0.978042	bad convergence!
8	35.2271s	6.05344e-14	0.707056	
16	28.7725	2.87305e-06	0.432836	

More Examples /DUMMY EXAMPLE/
Dimension = 1000, sum of 100 terms,
'by 100 outer loops (syn'd),
'by group of 1000',
600,000 iterations...
AMD Ryzen 1700 (8 cores, 16 threads)
'no sync: element by element, /atomic/'

#threads	time	loss	'efficiency'	n.b.
1	198.817s	1.60923e-27	1	
2	101.867s	0.000106731	0.967493	bad convergence!
8	26.6244s	5.8378e-14	0.933434	surprisingly: faster than non-atomic
16	23.0428s	2.83166e-06	0.539262	

More Examples /DUMMY EXAMPLE/
Dimension = 1000, sum of 100 terms,
'by 100 outer loops (syn'd),
'by group of 1000',
600,000 iterations...
AMD Ryzen 1700 (8 cores, 16 threads)
'my trick'

#threads	time	loss	'efficiency'	n.b.
1	198.483s	1.60923e-27	1	
2	102.069s	2.45525e-20	0.972301	good convergence!
8	27.4468s	5.8378e-14	0.903945	
16	23.0428s	3.80972e-06	0.51793	

Technical conclusion.

- Impact of the size of the gradient vector (of course!) and update method: what about recursive schemes / localised sub-vector updates?
- 'Real' async not always ideal: need to auto-tune key parameters - otherwise fragile convergence, fragile performances
- Learning rate is critical: ... need to auto-tune key parameters
- Batch size...

Possible next steps.

- Zhang 2017: YellowFin tuner, async creates momentum, thus a need to auto-tune learning rate /AND/ momentum while controling the amount of asynchronicity.
  Possibly the subject of a future presentation ...
- '1bit' updates a la CNTK
- ...

Stay tuned!