An Investigation into Hogwild! [NRRW11]

Abhijit Chowdhary
ac6361@nyu.edu

New York University — May 14, 2020

Contents

1 Introduction

Despite being the subject of much modern excitement, the ideas of gradient descent date all the way back to Cauchy in 1847. And although it's main application has been in the solution of recent big-data optimization problems, stochastic gradient has been around since the 1940s, formally by Robbins and Monro in 1951. In the last two decades, however, modern hardware has begun to see a tapering off of Moore's law, and has begun to expand out in a distributed fashion with multicore processors and GPUs; naturally the question becomes: In order to take advantage of the strengths of modern hardware, how can we parallelize a stochastic gradient method?

2 Hogwild!

Prior to 2011, parallel stochastic gradient methods had been introduced, but most suffered from poor scaling due to the necessity of locks. A naive implementation could look like:

Algorithm 1 Very Naive Parallel Stochastic Gradient

```
Require: Number of data points $N$, seperable loss function $f = \sum_{e \in E} f_e(x_e)$, Initial $x$.
 1: for epoch = 0 → MAX_EPOCHS do
 2:     #pragma omp parallel for
 3:     for $k = 0 \to N$ do
 4:         Choose $i$ uniformly from $\{1, \dots, |E|\}$.
 5:         #pragma omp critical
 6:             Read current parameters $x$.
 7:             Compute $\nabla f_i(x)$.
 8:             $x \leftarrow x - \eta \nabla f_i(x)$.
 9:     end for
10: end for
```

But it's clear here that such an algorithm would only effectively be parallelizing the unform sample of i in $\{1, \dots, |E|\}$. You can improve the above by selectively locking the values in x which actually change in the stochastic gradient step (say if $\nabla f_i(x)$ was sparse), but because the process of acquiring locks is much more expensive than floating point arithmetic, this helps little.

However, in 2011, the article "Hogwild!: A Lock-Free Approach to Parallelizing Stochastic Gradient Descent" proposed a very simple solution to this problem. Remove the locks!

Algorithm 2 Very Naive Parallel Stochastic Gradient

```
Require: Number of data points $N$, seperable loss function $f = \sum_{e \in E} f_e(x_e)$, Initial $x$.
 1: for epoch = 0 → MAX_EPOCHS do
 2:     #pragma omp parallel for
 3:     for $k = 0 \to N$ do
 4:         Choose $i$ uniformly from $\{1, \dots, |E|\}$.
 5:         Read current parameters $x$.
 6:         Compute $\nabla f_i(x)$.
```

```
7:         $x \leftarrow x - \eta \nabla f_i(x)$.
8:     end for
9: end for
```

3 Implementation

Here we discuss two main implementations of the Parareal algorithm using OpenMP.

3.1 Naive OpenMP

From the basic description of Parareal, we see that the fine propagator is able to be computed in parallel, so the main idea here is to parallelize the for loop corresponding to the fine propagator. See algorithm 3 for the pseudocode. A quick description is as follows:

(1) Lines $(1-2)$ describe the initial coarse approximation of the system. It's important that these solutions are loaded into both y_c and y, since they're needed for the next bit to satisfy the first same as last property.

(2) Lines $(4-8)$ are the computation of the fine approximation, and the construction of the corrector term δy. There are a few implicit assumptions here:

- We assume that $y(n)$ hold the previous λ_n^k.
- We assume that $y_c(n)$ already holds the value $\mathcal{G}(t_{n+1}, t_n, \lambda_n^k)$.

Both of these assumptions are true atleast for the first iteration due to our considerations in the first part. Furthermore, this part is completely dependent on previously computed information, and has no previous dependence on previous iterates, and therefore is embarrassingly parallel.

(3) Lines $(9-12)$ actually do the parareal iteration, we predict via line 10, and we correct via our previously computed $\delta y(n)$.

Algorithm 3 Naive Parallel Parareal Algorithm

```
Require: $y_0$ and coarse and fine solvers $\mathcal{G}$, $\mathcal{F}$.
1: $y_c \leftarrow \mathcal{G}(t_f, t_0, y_0)$.                    ▷ Coarsely approximate solution
2: $y \leftarrow y_c$.
3: while iter < max_iter && not converged do
4:     #pragma omp parallel for
5:     for $n = 0 \to P$ do
6:         $y_f(n) = \mathcal{F}(t_{n+1}, t_n, y(n))$.
7:         $\delta y(n) = y_f(n) - y_c(n)$.                    ▷ corrector term. FSAL
8:     end for
9:     for $n = 0 \to P$ do
10:        $y_c(n) = \mathcal{G}(t_{n+1}, t_n, y(n))$.                    ▷ Predict.
11:        $y(n) = y_c(n) + \delta y(n)$.                    ▷ Correct.
12:    end for
13: end while
```

3.2 Pipelined OpenMP

There's some clear problems with the naive implementation of Parareal. Primarily, let's examine the coarse computation portions, lines 1 and 10 of algorithm 3. Both of these portions iteratively compute the coarse approximation to the solution, but they lock up all processors while they do so. In fact, once we've computed the approximation for the nth node, we can immediately continue, instead of waiting for the rest of the serial computation to finish. This is referred to in literature (Ruprecht [?]) as *pipelining*. In fact, this would happen naturally in the MPI implementation, but since I had decided to use OpenMP, we have to tackle this problem.

The solution here is actually to mimic an MPI enviornment by wrapping the entire parareal algorithm in a parallel region. The idea is that thread p has the job of processing the fine and coarse operators, and later solution at position t_{p+1}. While we don't have the cost of MPI communication across processes, we still need to appropriately place locks around our reads and writes to prevent race conditions. A brief description of this new algorithm, whose pseudocode is provided in algorithm 4, follows:

(1) Line 1 encases the whole algorithm in a parallel region, essentially mimicing a MPI centric algorithm, except in shared memory.

(2) Lines $(2-8)$ are the initial coarse approximation of the solution. Note, that for the pth process, we actually integrate from $t_0 \to t_p$, in order to compute the coarse solution at t_{p+1}. We could instead have process 0 do the whole integration, and then communicate those values to all processes, but it was observed then that this would be no better than the serial computation in the naive algorithm. Note that the nowait statement allows to continue immediately, which is the *pipelining* that we desire.

(3) Lines $(12-16)$ describe the parallel computation of $\mathcal{F}$ and δy. Note, it's important to have locks around the points from which we read and write, since across this algorithm thread p reads from position p and writes into position $p+1$. A point of note is that if we could refine this algorithm so that it read and wrote into only position p, save for one location, it would vastly improve.

(4) Lines $(17-22)$ describe the predictor and corrector step of parareal, accompanied with the necessary locks. Note that this is surrounded in a ordered directive, which is necessary since to compute the predictor step for $p+1$, we need $y(p)$. Therefore, we have no choice but to do this iteratively. However, because of the nowait clause on line 10, we aren't forced to wait for the other threads to finish before continuing on.

Algorithm 4 Pipelined Parallel Parareal Algorithm (source Ruprecht [?])

```
Require: $y_0$ and coarse and fine solvers $\mathcal{G}$, $\mathcal{F}$.
1: #pragma omp parallel                          ▷ Enclose whole algorithm in parallel region
2: #pragma omp for nowait
3: for $p = 0 \to P$ do
4:     if $p \neq 0$ then
5:         $y(p) = \mathcal{G}(t_p, t_0, y_0)$.    ▷ Repeated work across threads, but $\mathcal{G}$ is cheap.
6:     end if
```

```
 7:     $y_c(p+1) = \mathcal{G}(t_{p+1}, t_p, y(p))$.
 8: end for
 9: while iter < max_iter && not converged do
10:     #pragma omp for ordered nowait  ▷ nowait critical to avoid waiting for threads $> p$
11:     for $p = 0 \to P$ do
12:         omp_set_lock($p$)                                              ▷ Read lock.
13:         temp = $\mathcal{F}(t_{p+1}, t_p, y(p))$.
14:         omp_unset_lock($p$) and then omp_set_lock($p+1$)                ▷ Write lock.
15:         $y_f(p+1) =$ temp, $\delta y(p+1) = y_f(p+1) - y_c(p+1)$.
16:         omp_unset_lock($p+1$)
17:         #pragma omp ordered            ▷ ordered because $y(p)$ depends on $y(p-1)$.
18:             omp_set_lock($p$)                                          ▷ Read lock.
19:             temp = $\mathcal{G}(t_{p+1}, t_p, y(p))$.
20:             omp_unset_lock($p$) and then omp_set_lock($p+1$)            ▷ Write lock.
21:             $y_c(p+1) =$ temp, $y(p+1) = y_c(p+1) + \delta y(p+1)$.
22:             omp_unset_lock($p+1$)
23:     end for
24: end while
```

References

[NRRW11] Feng Niu, Benjamin Recht, Christopher Re, and Stephen J. Wright. Hogwild!: A lock-free approach to parallelizing stochastic gradient descent, 2011.