Randomized Sparsification for Improving Performance and Scalability of Algebraic Multigrid

Majid Rasouli
School of Computing, University of Utah
Salt Lake City, Utah
rasouli@cs.utah.edu

Hari Sundar
School of Computing, University of Utah
Salt Lake City, Utah
hari@cs.utah.edu

ABSTRACT

Algebraic Multigrid (AMG), mostly because of its black-box nature, is being used widely as a solver and also preconditioner for large sparse linear systems. Its scalability and performance, however, suffers from loss of sparsity in coarser levels of multigrid. Because of filling-in, each level of multigrid has a matrix denser than its previous level, which causes higher cost for different operations in the solver, including matrix-vector product (MATVEC) and matrix-matrix multiplication (MATMAT). In a parallel setting, the communication is usually the bottleneck of the solver, and the denser the matrices, the more expensive the communication. In our method, we use randomized sparsification to increase sparsity of coarse matrices at different levels of the Multigrid hierarchy, but still keeping the important information of the matrices. The randomness feature of our method guarantees the whole range of information from the matrix being preserved, so the solver will not remove a specific range of entries and consequently losing parts of the matrix information. By using the correct degree of sparsification, our experiments show the performance boost that we achieve for specific operations such as MATVEC and MATMAT and also for our solver as a whole.

KEYWORDS

algebraic multigrid, AMG, sparse, sparsification, matvec, matrix-matrix product, parallel

ACM Reference Format:
Majid Rasouli and Hari Sundar. 2019. Randomized Sparsification for Improving Performance and Scalability of Algebraic Multigrid. In *Proceedings of International Conference on Supercomputing (ICS'19).* ACM, New York, NY, USA, 2 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

This is intro.

ICS'19, June 2019, Phoenix, Arizona, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

2 METHODS

2.1 Sparsification

Each coarse matrix is computed by the triple product $As[l+1] = R[l] \times As[l] \times P[l]$, then it is sparsified based on a randomized strategy. We want to retain the majority of entries with higher absolute values and also a few entries with lower absolute values. The element-wise matrix sparsification suggested in [] is not efficient in practice.

Algorithm 1 shows an improved version. Matrix A is saved as a vector of entries with an arbitrary order. The probability of selecting each entry A_{ij} is

$$p_{ij} = \frac{A_{ij}^2}{\sum_{lk} A_{lk}^2} \tag{1}$$

in which the summation goes through all entries of A (Frobenius norm of A).

Algorithm 1 $As = Sparsify(A, S, F)$

Input: A (original matrix), S (size of the sparsified matrix), F (Frobenius norm of A)
Output: As (sparsified matrix)

```
iter ← 0, i ← 0
while i < S do
    p_iter ← A[iter]^2 / F                ▷ probability of entry iter
    if rand() < p_iter then
        As ← A[iter]
        i++
    end if
    iter++
    if iter ≥ size(A) then
        iter −= size(A)
    end if
end while
```

The issue with Algorithm 1 is that for big matrices, the probability of choosing each entry becomes very low, because the denominator becomes very large, relative to the numerator (check first line in Figure 1).

To fix this issue, the probability of each entry is scaled to the whole $[0, 1]$ interval by multiplying all the probabilities by the highest probability (second and third lines in Figure 1). Algorithm 2 shows how the probabilities are scaled.

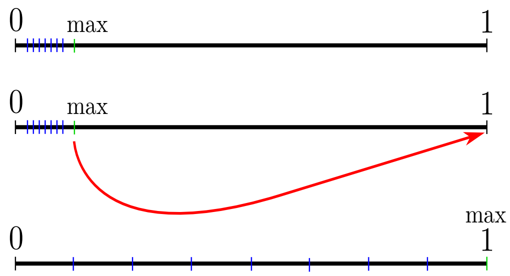

Figure 1: Scaling the probabilities of choosing entries to be retained, to the unit interval

Algorithm 2 $As = Sparsify(A, S, F, max)$

Input: A (original matrix), S (size of the sparsified matrix), F (Frobenius norm of A), max (maximum value of A)
Output: As (sparsified matrix)

$iter \leftarrow 0, i \leftarrow 0$
$IMP = \frac{F}{max^2}$ ▷ IMP: inverse of max probability
while $i < S$ **do**
    $p_{iter} \leftarrow \frac{IMP * A[iter]^2}{F}$ ▷ probability of entry $iter$
    if $rand() < p_{iter}$ **then**
        $As \leftarrow A[iter]$
        $i++$
    end if
    $iter++$
    if $iter \geq size(A)$ **then**
        $iter- = size(A)$
    end if
end while

This algorithm may take a long time, since the random value is generated in the unit interval $[0, 1]$, and some entries may still have very low absolute value and consequently very low probability. To fix that, after passing through all the entries of the matrix once, the random value will be generated in a tenth of the unit interval $[0, 0.1]$ for the next round and between 0 and 0.01 for the round after that and so on (Algorithm 3).

Algorithm 3 $As = Sparsify(A, S, F, max)$

Input: A (original matrix), S (size of the sparsified matrix), F (Frobenius norm of A), max (maximum value of A)
Output: As (sparsified matrix)

$iter \leftarrow 0, i \leftarrow 0, randFactor \leftarrow 1$ ▷ initialize $randFactor$ to 1
$IMP = \frac{F}{max^2}$
while $i < S$ **do**
    $p_{iter} \leftarrow \frac{IMP * A[iter]^2}{F}$
    if $rand()/randFactor < p_{iter}$ **then**
        $As \leftarrow A[iter]$
        $i++$
    end if
    $iter++$
    if $iter \geq size(A)$ **then**
        $iter- = size(A)$
        $randFactor* = 10$
    end if
end while

We want the matrix to stay symmetric after sparsification. Also, the diagonal entries should be kept (why? smoothers?). So we go through the lower triangle of the matrix. We add every diagonal entry. And if a non-diagonal entry is chosen to be added, its transpose entry will be added too. Also a boolean vector is used to avoid having duplicates. This way we will have a matrix of exactly the desired size and also there is no need to remove duplicates (Algorithm 4).

Algorithm 4 $As = Sparsify(A, S, F, max)$

Input: A (original matrix), S (size of the sparsified matrix), F (Frobenius norm of A), max (maximum value of A)
Output: As (sparsified matrix)

$i \leftarrow 0$
$chosen[size(A)] \leftarrow False$ ▷ initialize boolean vector to $False$
for $iter = 0$; $iter < nnz, i < S$; $iter++$ **do**
    if $A[iter].row == A[iter].col$ **then**
        $As \leftarrow A[iter]$
        $chosen[iter] \leftarrow True$
        $i++$
    end if
end for
$iter \leftarrow 0, randFactor \leftarrow 1$ ▷ initialize $randFactor$ to 1
$IMP = \frac{F}{max^2}$
while $i < S$ **do**
    if $!chosen[iter]$ and $A[iter].row < A[iter].col$ **then**
        $p_{iter} \leftarrow \frac{IMP * A[iter]^2}{F}$
        if $rand()/randFactor < p_{iter}$ **then**
            $As \leftarrow A[iter]$
            $As \leftarrow A[iter]^T$
            $i+ = 2$
            $chosen[iter] \leftarrow True$
        end if
    end if
    $iter++$
    if $iter \geq size(A)$ **then**
        $iter- = size(A)$
        $randFactor* = 10$
    end if
end while

3 NUMERICAL RESULTS

This is results.