Improving Matrix-Matrix Multiplication in Algebraic Multigrid Context

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

This is abstract.

KEYWORDS

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

ACM Reference Format:
Majid Rasouli and Hari Sundar. 2019. Improving Matrix-Matrix Multiplication in Algebraic Multigrid Context. In *Proceedings of ACM Symposium on High-Performance Parallel and Distributed Computing (HPDC'19).* ACM, New York, NY, USA, 2 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

This is intro.

2 METHODS

To compute the coarse matrix Ac, a triple matrix multiplication should be done:

$$Ac = R \times A \times P \tag{1}$$

in which $R = P^T$. We do it in two parts, performing matrix-matrix multiplications (MATMULT) twice: first $A \times P$, then $R \times B$, in which $B = A \times P$.

The matrices are partitioned on multiple processors by row blocks (Figure 1). Matrices A and P have the same number of rows and consequently are partitioned the same way. R has less number of rows and has a different partition.

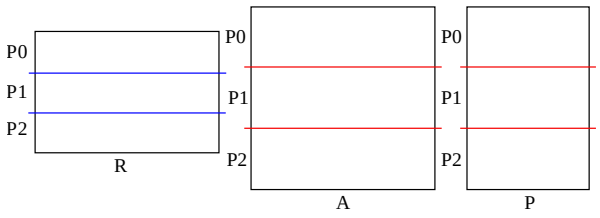

Figure 1

2.1 Part 1

In this part, we compute $B := A \times P$. We assume the same partition of rows of A on its columns. The same partition of rows of R is assumed on columns of P (Figure 2). Here we consider doing MATMULT on processor $P1$. To compute entry $B(i, j)$, we need to multiply row i of A with column j of P and add them together (since we are working with sparse matrices, we only consider the nonzeros).

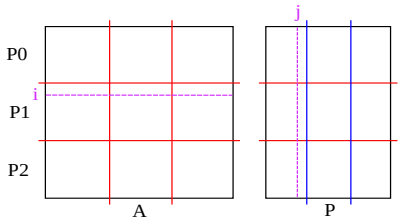

Figure 2

Column j is distributed between all the processors, so we will be able to perform summation $B_{ij} = \sum_k A_{ik} P_{kj}$, after communicating all nonzeros of column j and performing the multiplication. (explain this part clearer and show why this method is not good). To avoid that, we can use the fact that R is the transpose of P and we already have computed it. We note that column blocks of P (e.g. $r0$ in Figure 3) are actually row blocks of R transposed ($rt0$, which is transpose of $r0$).

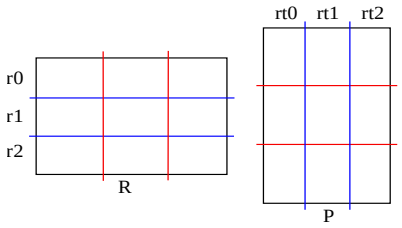

Figure 3

Algorithm 1 shows how we do it in an overlapped fashion using MPI. B_i is the row block of matrix B on processor i and B_{ik} is the sub-block result of multiplying A_i with rt_i.

HPDC'19, June 2019, Phoenix, Arizona USA

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

Algorithm 1 Part 1: $B_i = A_i \times P$

Input: A_i, R
Output: B_i (result of $A_i \times P$)
1: $R1 \leftarrow$ compute transpose of R_i (locally)
2: **for** $k = myrank : myrank + nprocs$ **do**
3: $R2 \leftarrow$ *Irecv transpose of R from right neighbor*
4: *Isend*$(R1)$ *to left neighbor*
5: $B_{ik} \leftarrow A_i \times R1$
6: *wait for Isend and Irecv to finish*
7: *swap*$(R1, R2)$
8: **end for**

2.2 Part 2

Now we explain how to do the second MATMULT: $R \times B$, in which $B = A \times P$. Figure 4 shows an example on processor $P1$. The blocks of R on processor $P1$ should be multiplied with the corresponding blocks of B with the same color. To do that we need to communicate whole B between all the processors.

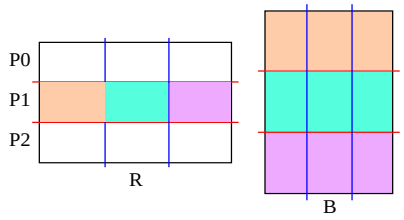

Figure 4

To avoid this expensive communication, we again can use the fact that we already have computed the transpose of R. We only perform the multplication of the green blocks on processor $P1$ in Figure 5, which can be done locally.

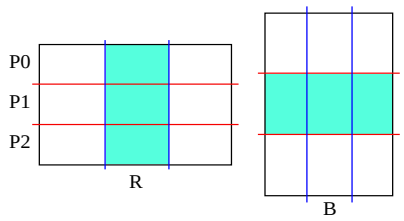

Figure 5

Using this method, we need to add duplicates twice (explain duplicates). Once, after doing the multplication; Second time, after sorting the result globally and putting the entries on the correct processors. Algorithm 2 shows how we have implemented it. Note that each sub-block of R_i should be multiplied by whole B_i, in contrary to the naive method, in which each sub-block of R_i is multiplied by only one other sub-block of B_i.

Algorithm 2 Part 2: $Ac = R \times B$

Input: P_i, B_i
Output: Ac_i
1: $PT_i \leftarrow$ compute transpose of P_i (locally)
2: **for** $k = 0 : nprocs$ **do**
3: $Ac_i \leftarrow PT_{ik} \times B_i$
4: **end for**
5: locally sort Ac_i and add duplicates
6: globallly sort Ac_i and add duplicates

3 NUMERICAL RESULTS

This is results.