CS6240: Project Final Report

Zachary Hillman, Zoheb Nawaz, Jackson Neal

December 12, 2021

Team Members: Zachary Hillman, Zoheb Nawaz, Jackson Neal
GitHub Spark Repo: `https://github.com/jacksonneal/sparse-matrix-product`
GitHub MR Repo: `https://github.com/CS6240/sparse-matrix-product-mr`

Project Overview:

Our topic is the matrix product of two sparse matrices using block partitioning. The multiplication of sparse matrices is applicable to several scientific computations, including, but not limited to, graph algorithms, graphics processing, and conjugate gradients. Sparse matrix product algorithms take advantage of the low density of the input and improve upon standard methods for finding dense matrix products. In this project, we will implement a sparse matrix product algorithm capable of scaling across clusters of varying sizes. Our algorithm will assume random sparseness of input and be *sparsity-independent* - making no inquiry about the structure of the sparseness. Our algorithm will use a block partitioning strategy to optimize runtime while minimizing data transfer. We will test our algorithm efficiency against a benchmark from a vertical-horizontal partitioning algorithm and a MapReduce block partitioning algorithm.

Input Data:

We generate our own sparse matrices as input data to our algorithm. We accept two matrices as input for the computation of $C = A \cdot B$. The matrix generation function accepts arguments for matrix dimension, n, density, d, and max value. Each matrix - A, B, C - is of dimension $n \times n$. From the density and dimension values, the independent probability that any (i, j) value in the matrix will be nonzero is d/n. Therefore, in A and B, we expect dn nonzero values. The max value parameter indicates the range $[0, max]$ that will be used to randomly generate nonzero values. The generated matrix will be in coordinate matrix form, a list of (i, j, v) tuples representing the location and value of nonzero entries in the matrix. Our input data will consist of sparse matrices generated from configured parameters in the ranges $100 \leq n \leq 10000$, $0 \leq (d/n) \leq 0.5$, and $max = 100$.

The following block shows an example of input in coordinate matrix form. The MR algorithm requires a $MatrixId$ entry in the tuple: $(MatrixId, i, j, v)$.

```
(L,0,3,49)
(L,2,3,32)
(R,3,3,98)
```

Full input data sets can be found in the following shared drive: `SparseMatrixProductInputDrive`.

Sparse matrix product H-V, V-H, or B-B (MapReduce or Spark)

Overview:

We are implementing Sparse matrix product using block partitioning in both MapReduce and Spark. The "Communication Optimal Parallel Multiplication of Sparse Random Matrices" provided many interesting approaches for block partitioning the sparse matrix product [1]. We implemented four Spark Sparse Matrix B-B algorithms, derived from 1-D and 2-D partitioning strategies. These algorithms are meant to optimize the communication phases, as these are the dominating operations when dealing with sparse matrices. We also implemented Spark Sparse Matrix V-H and MR Sparse Matrix B-B algorithms for comparison.

Sparse Matrix Product B-B Pseudo Code:

To save space, we only show pseudo code for a few of the implemented algorithms.

Spark V-H Algorithm The vertical-horizontal partitioning algorithm is straightforward, joining A column values with equal B row values, then reducing by corresponding output entry in C.

```
def sparseProduct(a: SparseRDD, b: SparseRDD): SparseRDD = {
  val a_col = a.map {
    case (i, j, v) => (j, (i, v))
  }
  val b_row = b.map {
    case (i, j, v) => (i, (j, v))
  }
  a_col.join(b_row).map {
    case (_, ((i, v), (j, w))) => ((i, j), v * w)
  }.reduceByKey(_ + _).map {
    case ((i, j), v) => (i, j, v)
  }
}
```

Source Code Link: `src/main/scala/product/VerticalHorizontal.scala`

Spark 2-D Broadcast Algorithm The 2-D broadcast algorithm first partitions the responsibility of computing C. Using the number of partitions p, we can break C into a square grid of dimension $\sqrt{p}$. Using the same blocking strategy for A and B we can then send each block of the input matrices to the partitions that require them - A blocks will be broadcast over rows of the grid, B blocks will be broadcast over columns of the grid.

```
type SparseRDD = RDD[(Int, Int, Long)] // (i, j, v)
val P // # of partitions
val HP = new HashPartitioner(P)
```

```
// Access the list of partitions to broadcast A entry to, given row coordinate
def getAPartitions(i: Int, n: Int): List[Int] = {
  val p_dim = math.sqrt(P).toInt

  // determine row in p_dim X p_dim partition grid
  val row = (i / (n.toDouble / p_dim)).toInt

  // determine all other partitions in same row of grid
  List.range(0, p_dim).map(col => row * p_dim + col)
}

// Access the list of partitions to broadcast B entry to, given col coordinate
def getBPartitions(k: Int, n: Long): List[Int] = {
  val p_dim = math.sqrt(P).toInt

  // determine col in p_dim X p_dim partition grid
  val col = (k / (n.toDouble / p_dim)).toInt

  // determine all other partitions in same col of grid
  List.range(0, p_dim).map(row => row * p_dim + col)
}

def sparseProduct(a: SparseRDD, b: SparseRDD, n: Int, sc: SparkContext): SparseRDD = {

  // Broadcast A entries to partitions in same row, group by partition
  val a_par = a.flatMap {
    case (i, j, v) =>
      improvedSummaAPartitions(i, j, n).map(p => (p, (i, j, v)))
  }.groupByKey(HP)

  // Broadcast B entries to partitions in same col, group by partition
  val b_par = b.flatMap {
    case (j, k, v) =>
      improvedSummaBPartitions(j, k, n).map(p => (p, (j, k, v)))
  }.groupByKey(HP).mapValues {
    b_itr =>
      // Use map of B(j, [(k, v)]) for quick lookup later
      val b = new mutable.HashMap[Int, Seq[(Int, Long)]]()
      for ((j, k, v) <- b_itr) {
        b(j) = b.getOrElse(j, Seq.empty[(Int, Long)]) :+ (k, v)
      }
      b
  }

  // We can join on key of partition.
```

```
    // Each partition has all necessary information to compute its partition of C.
    val c_par = a_par.join(b_par).mapValues {
      case (a, b) =>
        val c = new mutable.HashMap[(Int, Int), Long]()
        for ((i, j, v) <- a) {
          for ((k, w) <- b.getOrElse(j, Seq.empty[(Int, Long)])) {
            c((i, k)) = c.getOrElse((i, k), 0L) + v * w
          }
        }
        c
    }

    // Final formatting to SparseRDD
    c_par.flatMap {
      case (_, c) => c.map {
        case ((i, k), v) =>
          (i, k, v)
      }
    }
  }
```

Source Code Link: `src/main/scala/product/ImprovedSUMMA.scala`

MapReduce B-B Partitioning Algorithm The MapReduce pipeline to compute the matrix product consists of two jobs: the first one computes the intermediate products and the second one combines the intermediate results associated with the different blocks. The partitioning strategy involves partitioning the left matrix into the $totalLeftHP$ horizontal partitions and $totalLeftVP$ vertical partitions, while the right matrix is only partitioned vertically into $totalRightVP$.

```
    PartialProductMapper {
        setup() {
            // read totalLeftHP, totalLeftVP and totalRightVP from global config
        }

        map(matrixId, row, col, val) {
            if (matrixId == 'L') {
                i = row % totalLeftHP
                j = col % totalLeftVP
                for (k = 0 to totalRightVP - 1) {
                    emit((i, j, k), (matrixId, row, col, val))
                }
            } else {
                j = row % totalLeftVP
                k = col % totalRighttVP
                for (i = 0 to totalLeftHP - 1) {
                    emit((i, j, k), (matrixId, row, col, val))
```

```
            }
        }
    }
}

PartialProductReducer {
    reduce(key, [(matrixId, row, col, val), ...]) {
        leftMap = {}
        rightMap = {}

        for each (matrixId, row, col, val) in values
            if (matrixId == 'L') {
                leftMap.put(col, (row, val))
            } else {
                rightMap.put(col, (row, val))
            }

        for each entry1 in leftMap
            for each entry2 in rightMap
                sum = 0
                for each entry3 in entry2.values
                    // compute product and update sum
                    sum += entry1.value.get(entry3.key, 0)
                            * entry2.value.get(entry3.key)

                emit(null, (entry1.key, entry2.key, sum))
    }
}

FinalProductMapper {
    map(row, col, val)
        emit (row, (col, val))
}

FinalProductReducer {
    reduce(row, [(col1, val1), ...]) {
        map = {}
        for each (col, val) in values
            // update product associated with a specific column
            map.put(col, map.get(col, 0) + val)

        for each (col, val) in map
            emit(null, (row, col, val))
    }
}
```

Source Code Link: `src/main/java/edu/neu/SparseMatrixProductBB.java`

Algorithm and Program Analysis:

Our algorithms' complexities are driven by the parameters n, d, and p. In analysis, we explore the theoretical effect of these parameters on the various algorithms we implemented.

Spark V-H: The Spark V-H algorithm was implemented to give us a performance baseline. It is straightforward, joining A column values with B row values. This join will result in a full shuffle of all data. From our density definition, we expect each row to have d nonzero values. Therefore, the join will also result in data duplication by a scale of d. The final reduce operation will also result in a full shuffle. The second shuffle should dominate the complexity and we see an expected communication cost of $\mathcal{O}(d^2n)$. Memory cost is constant and negligible since all operations process streams.

Spark 1-D Iterative: The 1-D iterative algorithm is known as the Naive Block Row partitioning [1]. In this algorithm, we partition both A and B row-wise into p blocks. Each partition is assigned one block row of A. Over p iterations, we then send one block of B to each partition until each partition has seen each block of B. This algorithm has a communication cost of $\mathcal{O}(dn)$ that repeats over p iterations. In the implementation, we should also note that each partition is going to hold its entries in memory. This provides easy access but incurs a memory requirement of $\Omega(dn/p)$. This may be acceptable if density is sufficiently low. We also can increase p and reduce the memory cost.

Spark 1-D Broadcast: The 1-D broadcast algorithm is known as the Improved Block Row partitioning [1]. This algorithm improves on the iterative solution by immediately sending B blocks and only sending to partitions that require them. We know if partition "i owns the ith block row of A, A_i, and the jth subcolumn of A_i contains no nonzeros, then [partition] i does not need to access the jth row of B" [1]. The probability that a subcolumn of A_i has at least 1 nonzero is d/p [1]. From this, each partition will require dn/p rows of b, introducing a communication cost of $\mathcal{O}(d^2n)$ and a memory requirement of $\Omega(d^2n/p)$. We expect the number of nonzeros in C to be d^2/n [1]. This algorithm will only function if we have enough partitions to hold all the entries of C in memory at once.

Spark 2-D Iterative: The 2-D iterative algorithm is inspired by the Sparse SUMMA algorithm [1]. The algorithm block partitions both A and B into grids. In a cyclical manner, the blocks of A are passed row-wise and the blocks of B are passed column-wise every iteration. At completion, each partition will have seen the blocks of the A row and the B column that contribute to that partition's output block of C. This algorithm has a communication cost of $\mathcal{O}(dn)$ that repeats over $\sqrt{p}$ iterations. Each iteration results in a shuffle as all blocks are assigned a new partition. The memory requirements are $\Omega(dn/p)$ as no duplication occurs, but we are still holding entries in memory.

Spark 2-D Broadcast: The 2-D broadcast algorithm is inspired by the Improved Sparse SUMMA algorithm [1]. The details are provided above with the pseudo code. Every block of A and B is duplicated along a dimension of the partition grid. This results in $\sqrt{p}$ data duplication. From this, we have a communication cost of $\mathcal{O}(dn\sqrt{p})$ and a memory requirement of $\Omega(dn/\sqrt{p})$.

MR B-B: As described above, the MR job works with a 2-D partition on the left matrix and a

1-D partition on the right matrix. The algorithm sends all input data to all partitions, resulting in duplication of scale p and a communication cost of $\mathcal{O}(dnp)$. The MR job holds the matrix entries in memory for easy access, incurring a memory requirement of $\Omega(dn)$ for each partition.

Complexity			
	Communication	Iterations	Memory
V-H	$\mathcal{O}(d^2n)$	1	1
1-D Iter	$\mathcal{O}(dn)$	p	$\Omega(dn/p)$
1-D Broadcast	$\mathcal{O}(d^2n)$	1	$\Omega(d^2n/p)$
2-D Iter	$\mathcal{O}(dn)$	$\sqrt{p}$	$\Omega(dn/p)$
2-D Broadcast	$\mathcal{O}(dn\sqrt{p})$	1	$\Omega(dn/\sqrt{p})$
MR B-B	$\mathcal{O}(dnp)$	1	$\Omega(dn)$

Looking at the complexity table, we can derive a few insights about theoretical performance before looking at our experiments:

(i) The V-H and 1-D broadcast algorithms may suffer from communication cost if d is too large.

(ii) Among the Spark algorithms, if $d \ll p$, 2-D broadcast will have the highest memory requirement. If not, 1-D broadcast will have the highest memory requirement. The MR algorithm has the highest memory requirement overall.

(iii) If $d \ll p$, $d \ll \sqrt{n}$, making communication cost dominate the runtime complexity, we can expect 2-D broadcast to be the most efficient algorithm regarding time complexity.

(iv) Although the MR algorithm has one "iteration" it is composed of two jobs and the added complexity of HDFS interaction is not represented in the table.

Experiments:

A full display of our experiments can be found through the following link: `experiments.csv`
Here, we explore the highlights of those experiments.

On a small input, $n = 6000$, $(d/n) = 0.2$, all Spark block partitioning algorithms perform nearly equal, with run times around 8m. The V-H algorithm performs poorly, due to its high

communication cost and two shuffle stages, resulting in a runtime of 2.2h. The MR B-B algorithm performs well, lagging just behind the Spark programs with a runtime of 12m.

On a larger input, $n = 12000$, $(d/n) = 0.2$, we begin to differentiate between the algorithms. As expected, the 1-D broadcast algorithm cannot handle larger inputs due to its communication costs. The 1-D broadcast algorithm does not complete within 4h. The 1-D iterative, 2-D iterative, and 2-D broadcast algorithms all perform relatively equal when given this larger input, with run times of 1h, 49m, and 48m respectively. The similarity of the 2-D algorithms, and them both edging out 1-D iterative, is to be expected when looking at communication cost.

It is worth noting that we could have experimented with more variation in the d parameter. We were running higher d values than these algorithms were previously evaluated with [1]. Lower d values cause less multiplications and would have led to bigger differentiation in observed runtime as these algorithms are optimized for communication cost. Lower d values would allow algorithms like V-H and 1-D broadcast to become very competitive. These algorithms would also be competitive if we run into memory constraints, which was a non-issue during our experiments.

After determining the 2-D algorithms to be optimal, we focused our analysis and experimentation on the 2-D broadcast and MR B-B algorithms. We explore these in more detail below.

Speedup:

MR B-B:

Cluster	Instance Type	n	p	Run Time
2 workers	m3.xlarge	6000	12	33m
4 workers	m3.xlarge	6000	12	16m

As observed in the above table, the program shows good speedup = 33/16 = 2.0625, with the execution time dropping to less than half of the original time as the number of worker nodes doubles.

Spark 2-D Broadcast:

Cluster	Instance Type	n	p	Run Time
2 workers	m5.xlarge	6000	25	20m
4 workers	m5.xlarge	6000	25	7.8m

As observed in the above table, the program shows good speedup = 20/7.8 = 2.56, with the execution time dropping to less than half of the original time as the number of worker nodes doubles.

Scalability:

MR B-B:

Cluster	Instance Type	n	p	Run Time
5 workers	m5.xlarge	6000	12	12m
5 workers	m5.xlarge	12000	12	1.4h

As observed in the above table, as the input size increases, the execution time of the program increases significantly, due to the data duplication that occurs when partitioning the input into blocks.

MR B-B Effect of Partition Granularity:

Cluster	Instance Type	n	p	Run Time
5 workers	m5.xlarge	6000	12	12m
5 workers	m5.xlarge	6000	24	1.28h
5 workers	m5.xlarge	6000	60	1.5h

As observed in the above table, increased partitioning (fine granularity) increases execution time due to the increase in data duplication and higher data shuffle costs.

Spark 2-D Broadcast:

Cluster	Instance Type	n	p	Run Time
5 workers	m5.xlarge	6000	25	8m
5 workers	m5.xlarge	12000	25	48m

From the table above, the scalability of the 2-D broadcast algorithm is decent. We see a 4x increase in input size cause a 6x increase in running time.

Spark 2-D Broadcast Effect of Partition Granularity:

We held all other parameters constant to determine an optimal p value of 16. The running times for different p values were minimal so we do not display them here.

Results:

The correctness of each algorithm has been verified by testing each implementation against smaller matrices of the following dimensions: 4x4, 10x10 and 100x100.

The following is an example row from the output of executing the MR B-B algorithm. Each row in the output file represents a single cell in the resultant matrix in the i, j, v format.

```
1001,0,617823
```

Select result output and log samples are linked below. Further results can be provided on request.
Spark 2-D Broadcast $[n = 6k, (d/n) = 0.2, p = 25, cluster = 5 \cdot m5.xlarge]$: `stderr.txt`, `output`
MapReduce outputs: `MapReduceOutputDrive`, MapReduce logs: `MapReduceLogDrive`

Conclusions:

We have been able to research and implement multiple algorithms to solve the problem of sparse matrix product at scale using block partitioning. Sparse matrix product algorithms are typically optimized for multi-processor architectures. Our application of these algorithms to a distributed cluster is novel and provides functionality that Spark itself does not provide - multiplication of coordinate matrices. Our algorithms all out performed the naive vertical-horizontal partitioning approach. In a following study, we would repeat our experiments with lower density values, as to ensure that the communication cost dominates runtime complexity. Future works may also attempt to translate 3-D partitioning strategies into Spark and MapReduce [1].

References:

1. Ballard G, Buluc A, Demmel J, Grigori L, Lipshitz B, Schwartz O, Toledo S (2013) Communication optimal parallel multiplication of sparse random matrices. In: SPAA, pp 222–231