Sparse and Dense Data Representation for Matrix Multiplication using B-B Partitioning
Team: Mean Partitions
Project Proposal

Mustafa Kapadia, Sahil Gandhi, Sarthak Kothari, Siddhant Benadikar

---

1. Input Data

The input data used for performing the tasks look as follows:
2,3,6
4,5,7
Where the first integer is the row, the second integer is the column and the third is the value of the cell at the corresponding row and column id.

2. Task Details

2.1. Matrix Multiplication using B-B Partitioning

2.1.1. Overview

Matrix multiplication is widely used in a variety of applications and is often one of the core components of many scientific computations. Since dense matrix multiplication algorithm is highly computationally intensive, there has been a great deal of interest in developing parallel formulations of this algorithms and testing its performance on various parallel architecture [1].

One of the most commonly and widely used algorithms for parallel Matrix multiplication is Cannon's algorithm. It is extremely suitable for NxN matrix. The main advantage of this algorithm is that its storage requirements remain constant and are independent of the number of processors [2]. The two NxN matrices A and B are divided into square submatrices of size $\frac{a}{\sqrt{p}}$ and $\frac{b}{\sqrt{p}}$ among the p processors. The sub-blocks of A and B residing with the processor (i, j) are denoted by A^{ij} and B^{ij} respectively, where $0 \le i \le \sqrt{p}$ and $0 \le j \le \sqrt{p}$. In the first phase of the execution of the algorithm, the data in the two input matrices are aligned in such a manner that the corresponding square submatrices at each processor can be multiplied together locally. This is done by sending the block A^{ij} to the processor (i, (j + i) mod $\sqrt{p}$), and the block B^{ij} to the processor ((i + j) mod $\sqrt{p}$, j). The copied sub-blocks are then multiplied together. Now the A sub-blacks are rolled one step to the left and the B sub-blocks are rolled one step upward and the newly copied sub-blocks are multiplied and the results added to the partial results in the C sub-blocks. The multiplication of A and B is complete after $\sqrt{p}$ such steps.

2.1.2. Pseudo-Code

```
SparseEncoding (class) =  row, column, value, source;
CannonKey (class) = PartitionRow, PartitionColumn;
Mapper1(for every line which is a matrix cell)
{
        SparseEncoding ob = inputLine();
        // Get Regions
        If Input == A Matrix :
```

```
            Row   = i / (No. of Rows in A / √p ) ;
            Column   = j  / (No. of Cols  in A / √p ) ;
            //Rotate
            Column =  ((Column + Row) mod √p );
            // where i, j is the row and column for object ob
        Else if Input == B Matrix :
            Row   = i / (No. of Rows in B / √p ) ;
            Column   = j  / (No. of Cols  in B / √p ) ;
            //Rotate
            Row =  ((Column + Row) mod √p );
            // where i, j is the row and column for object ob
        Emit ( (Row, Column), Ob);
}
Mapper2(for every line which is a matrix cell)
{
        SparseEncoding ob = inputLine();
        // Get Regions
        If Input == A Matrix :
            Row   = i / (No. of Rows in A / √p ) ;
            Column   = j  / (No. of Cols  in A / √p ) ;
            // where i, j is the row and column for object ob
            //Rotate
            Column =  ((Column + Row) mod √p )
        Else if Input == B Matrix :
            Row   = i / (No. of Rows in B / √p ) ;
            Column   = j  / (No. of Cols  in B / √p ) ;
            // where i, j is the row and column for object ob
            // Rotate
            Row =  ((Column + Row) mod √p );
        Else if Input == C Matrix:
            Row   = i / (No. of Rows in A / √p ) ;
            Column   = j  / (No. of Cols  in B / √p ) ;
        Emit ( (Row, Column), Ob)
}
Reducer( (Row, Column), Ob)
{
        Add all A objects to A's Hashmap;
        Add all B objects to A's Hashmap;
        Add all C objects to A's Hashmap;

        For i in A's Row:
```

```
                For j in B's Column:
                        S = 0;
                        For k in range A's Column:
                                S+=  Multiply A[i,k] * B[k,j]
                        If C[i,j] exist:
                                C[i,j] += S;
                        Emit((i,j,S), null);

}
getPartition(key)
{
        PartitionID = (key.Row * √p ) + key.Column;
        Return PartitionID;
}

Driver()
{
        Iter = 0;
        Do{
                job1.setMapper(Mapper1);
                job1.setMultipleInput(A Matrix);
                job1.setMultipleInput(B Matrix);
                job1.setReducer(Reducer);
                job1.setOutput(Output+iter);

                job2.setMapper(Mapper2);
                job2.setMultipleInput(A Matrix);
                job2.setMultipleInput(B Matrix);
                job2.setMultipleInput(C Matrix); // Output of first Job
                job2.setReducer(Reducer);
        }while( iter < √p );
}
```

2.1.3. Algorithm Analysis

- In the initial alignment step, the maximum distance over which block shifts is $\sqrt{p} - 1$
  – The circular shift operations in a row and column directions take time: $t_{comm} = 2(t_s + \frac{t_\omega n^2}{p})$
- Each of the $\sqrt{p}$ single-step shifts in the compute-and-shift phase takes time: $t_s + \frac{t_\omega n^2}{p}$.
- Multiplying $\sqrt{p}$ submatrices of size $\frac{n}{\sqrt{p}} * \frac{n}{\sqrt{p}}$ takes time: $\frac{n^3}{\sqrt{p}}$
- Parallel Time = $\frac{n^3}{\sqrt{p}} + 2\sqrt{p}(t_s + \frac{t_\omega n^2}{p}) + 2(t_s + \frac{t_\omega n^2}{p})$

2.1.4. Experiments

Note: All experiments were run locally as our aim was to explore the size of the data output from the map and reduce phases.

The dataset consists of two files containing a sparse representation of matrices that we want to multiply. Matrix 'A' and Matrix 'B' are both NxN in dimension. Matrix 'C' is the resultant matrix after multiplication, also of NxN dimension.

Our program currently works perfectly for 4 partition blocks; there is a bug for larger partition values which we are investigating and will fix before further analysis. The algorithm creates reduce tasks equal to the number of partitions (4 in our experiments). Since all experiments were ran locally in addition to the bug in the program, we cannot comment on the Scaleup of the program. However, as described in the Overview section, we are working towards obtaining the theoretical scaleup. Currently, we measure the performance by looking at the amount of data transferred between the map and reduce phases with different sizes of matrices. We ran 3 experiments, on varying sizes of matrices for both dense and sparse representations. We observed the difference in data transferred across all experiments and conclude that the algorithm does not duplicate data. There is a 1-1 mapping of data transfer to the the data required to generate the result. In the current implementation approach, the program behaves similarly to a Graph problem in MapReduce, wherein we also write-to-file the original A and B matrices with their updated coordinates. In future iterations, we wish to eliminate that, resulting in an approximate ⅔ reduction in the data generated by the reduce phase. The number of bytes transferred from the mapper to the reducer was found to be proportional to the data size. In the current algorithm, there is no way of reducing this value.

2.1.4.1. Scalability

Keeping the number of partitions constant to 4, we increased the dimension of the matrices from 100x100 to 1000x1000. The difference in time taken between the two experiments was as follows:

[A]: 99x99: 5 sec for sparse data
[**B]**: 99x99: 5 sec for dense data
[**C**]: 1000x1000: 17 mins for sparse data
[**D**]: 1000x1000: program didn't couldn't locally for dense data

We can see that there is a huge difference in the times taken for the above experiments. While running [**A]** each partition only multiplies a 25x25 matrix. But for [**C**] each partition multiplies a 200x200 matrix. We will make it scalable by making the program work with more number of partitions so we can divide the work between multiple workers.

2.1.4.2. Result-Sample

29,50,170360,C
28,63,56030,C
27,76,135704,C
26,89,95320,C
29,51,144636,C
28,64,155038,C
27,77,89239,C
26,90,120667,C
29,52,143700,C
28,65,113370,C

This is the sample output for a when multiplying a 99x99 matrix. The given format is a (rowId, columnId, value, matrixLabel)

2.2. Sparse representation for B-B

For the sparse matrix representation, we decided to encode the data as custom object. Each object holds the row index, column index and the value. Since our ultimate goal is to use the Twitter edges dataset; which is sparse in nature, we designed our algorithm to work on sparse data format instead of a dense format. This reduces the overall data transferred in the shuffle phase making the overall memory utilization and the I/O low. We are also able to successfully perform BB Partitioning and matrix multiplication using this representation.

2.2.1. Experiments

The experiments remain the same as that for the main task. However, here we compare the results between the sparse and the dense representations of the matrices. The total data shuffled after every iteration was 704952 bytes for the dense representation and 360822 bytes for the sparse representation for the same dataset. This indicates a 1.8 times reduction of data transfer for a 99x99 sparse matrix. One should note that the data used was fabricated, and generated dataset was not extremely sparse. For extremely sparse matrices, this approach will prove to be very efficient with the only downside of data transfer. Our research suggests that for extremely sparse matrices it is less efficient to shuffle data to multiple partitions, and instead one should take advantage of the sparsity and the closeness or locality of the data.

3. Risks

Cannon's algorithm works well with perfect square and it is a challenge to extend it matrices that cannot be partitioned into blocks of perfect squares. The number of partitions also have to be a perfect square which implies the number of workers need to be a perfect square or else a worker machine may be left idle or a worker machine may get twice as much work.

4. References

[1] Anshul Gupta, Vipin Kumar. Scalability of Parallel Algorithms for Matrix Multiplication https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=4134256

[2] Cannon's Algorithm Wikipedia. https://en.wikipedia.org/wiki/Cannon%27s_algorithm

[3] Ortega, Patricia. Parallel Algorithm for Dense Matrix Multiplication. CSE633 Parallel Algorithms Fall 2012. https://cse.buffalo.edu/faculty/miller/Courses/CSE633/Ortega-Fall-2012-CSE633.pdf

[4] Zhiliang Xu. Lecture 6: Parallel Matrix Algorithms (part 3) https://www3.nd.edu/~zxu2/acms60212-40212/Lec-07-3.pdf