Checkpoint Report: Parallel and Distributed Tensor Decomposition

Ye Yuan   **Tengjiao Chen**
{yey1,tengjiac}@andrew.cmu.edu

1. Progress

By the time we wrote this checkpoint report, we have finished a correct and sequential implementation of sparse tensor decomposition. We have also tried a straight forward way to parallelize the algorithm. In summary, we have achieved our mid-way goals. We will discuss about our current methods and show some preliminary results.

2. Current Methods

We use an algorithm called Alternating Least Square (ALS). In the following algorithm, calligraphic $\mathcal{X}$ represents a $I \times J \times K$ tensor, which we need to decompose. Our target is to get $A_{I\times R}, B_{J\times R}, C_{K\times R}, \lambda$ such that:

$$\mathcal{X}(i,j,k) \approx \sum_{r=1}^{R} \lambda_r A(i,r)B(j,r)C(k,r)$$

```
Algorithm 1: CP-ALS for the 3rd order tensors
Input  : $\mathfrak{X}$: A 3rd order tensor
         $R$: The rank of approximation
Output: CP decomposition $[\![\boldsymbol{\lambda}; \mathbf{A}, \mathbf{B}, \mathbf{C}]\!]$
repeat
    $\mathbf{A} \leftarrow \mathbf{X}_{(1)}(\mathbf{C} \odot \mathbf{B})(\mathbf{B}^T\mathbf{B} * \mathbf{C}^T\mathbf{C})^\dagger$
    Normalize columns of $\mathbf{A}$
    $\mathbf{B} \leftarrow \mathbf{X}_{(2)}(\mathbf{C} \odot \mathbf{A})(\mathbf{A}^T\mathbf{A} * \mathbf{C}^T\mathbf{C})^\dagger$
    Normalize columns of $\mathbf{B}$
    $\mathbf{C} \leftarrow \mathbf{X}_{(3)}(\mathbf{B} \odot \mathbf{A})(\mathbf{A}^T\mathbf{A} * \mathbf{B}^T\mathbf{B})^\dagger$
    Normalize columns of $\mathbf{C}$ and store the norms as $\boldsymbol{\lambda}$
until no improvement or maximum iterations reached
```

2.1 Optimize Memory Footprint

One big problem for this algorithm is called the intermediate data explosion. In a large sparse tensor, dimension I, J, K are usually in million-scale. The rank R is usually 2 to 10. For each iteration, e.g. $A = X_{(1)}(C \odot B)(B^TB * C^TC)^\dagger$. This equation can be viewed as two parts: $M_A = X_{(1)}(C \odot B)$, $V = (B^TB * C^TC)^\dagger$. Here V is a dense matrix of size $R \times R$ and is easy to compute. The issue is in M_A, for a naive matrix operation, the shape of $(C \odot B)$ is $JK \times R$. Remember that I, J, K could be in million-scale, a naive explicit matrix operation will lead to a data explosion.

We can find that the calculation of each row of M_A can actually be written in the following way:

$$M_A(i,:) = \sum_{z=1}^{JK} X_{(1)}(i,z)B(z\%J,:) * C(z/J,:)$$

Figure 1 shows a example of mode 1 tensor. So for each slice of $X_{(1)}$, only non-zeros contribute to computation. Based on this observation, to reduce memory footprint, we designed following data structure to store the sparse tensor: we store each mode of tensor (e.g. $X_{(1)}$) using a list a slices (shown in Figure 1). Slices are stored in a compressed sparse row (CSR) fashion. We maintained two arrays for each slice. One array stores the indices of non-zeros, another array stores the value of non-zeros in corresponding order. The memory required to store the tensor will be $O(6 * m)$, where m is the number of non-zeros.

Also notice that we store the indices for all 3 tensor modes. In this way, when we do computation for any mode, data of consecutive slices will sit in continuous memory, which is good for spacial locality.

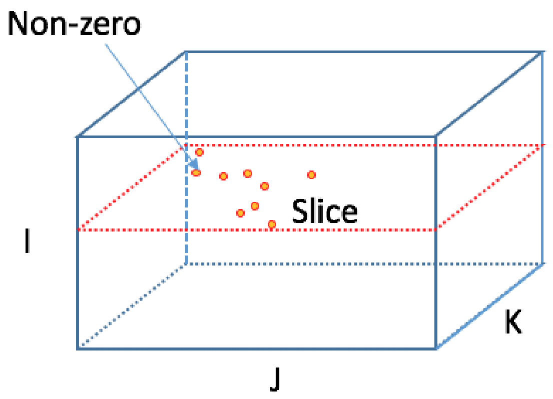

Figure 1: Mode 1 tensor illustration: the red dashed rectangle represents a data slice. Red dots are non-zero elements in tensor, . The distribution of non-zeros is usually very imbalanced in the tensor

2.2 Naive Task Mapping

There are quite a lot of parallelism we can explore in this algorithm. For example, the computation of each slice shown in Figure 1 could be done in parallel. Currently we use OpenMP to parallelize the computation of each row. However, the speedup is not very good, we will analyze it in next section.

3. Preliminary Results

We tested our algorithm in two datasets: Yelp and ML ratings. The statistics of these two datasets are shown in following table:

Dataset	Yelp Data	ML ratings
Shape	$552,339 \times 77,079 \times 5$	$138,493 \times 131,262 \times 10$
Non-zeros	2,225,213	20,000,263

The speedup for Yelp with multi-threading is shown in Figure 2. We can find that the speedup is really terrible. With 128 threads, we only get 2x speedup. However, after visualizing the data distribution in the tensor, this result is not strange. We find that the workload in each slice is super imbalanced (shown in Figure 3). Some slices have thousands of non-zeros, but some slices only have several non-zeros. The number of non-zeros varies by several orders of magnitude.

There are some drawbacks with our current data structures and we will try to design a new scheme in following weeks:
(1) The granularity of parallel task is too large. We currently try to parallelize each slice, but is seems that the work load of each slice is too imbalanced. We will try to store the tensor in fibers.

(2) We need to design a scheme to partition the tensor into a $p_1 \times p_2 \times p_3$ grid over $\mathcal{X}$. The target is to minimize the load imbalance by balancing the number of non-zeros for each thread.

4. Next Steps

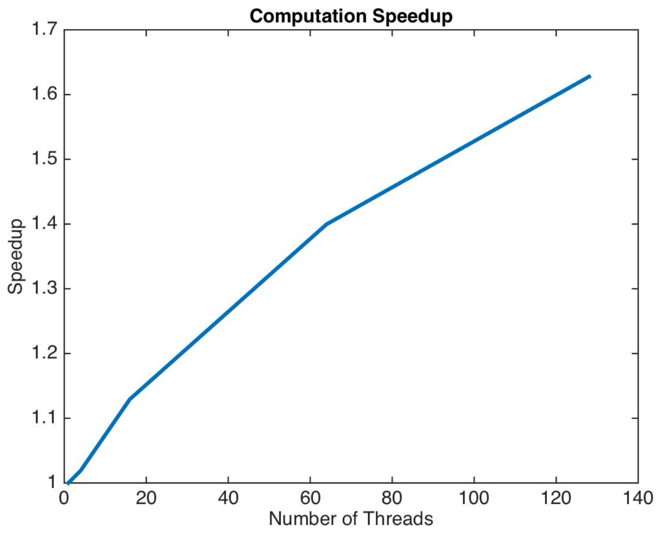

Figure 2: Speedup of naive task mapping

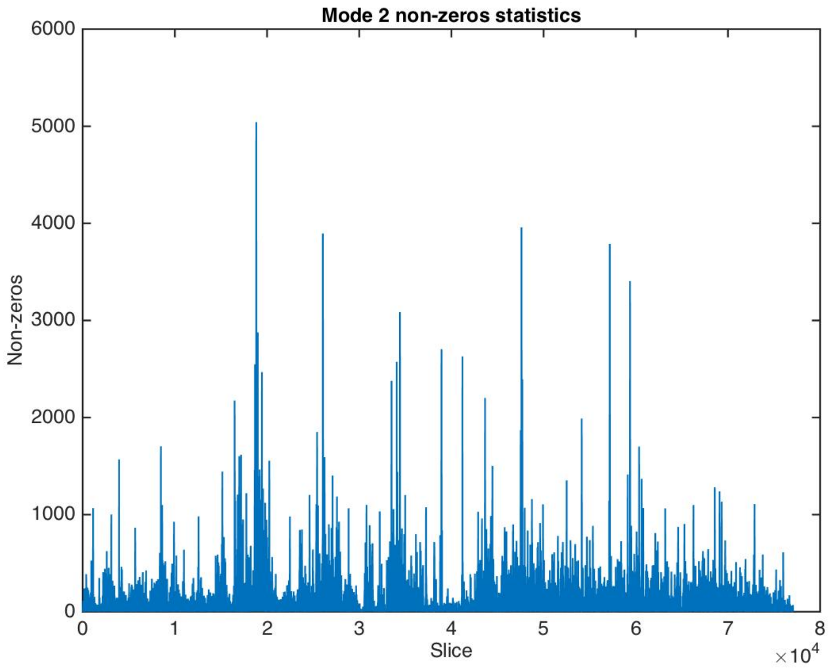

Figure 3: Number of non-zeros in each slice of $X_{(2)}$

Based on our current observations, we will make some slight changes on plans for next 3 weeks:

Week 4 (Nov. 21 - Nov. 27): Try dynamic job scheduling using Cilk and try to partition the workload in a more balanced way (Ye Yuan). Design a MPI version to decompose large tensors (Tengjiao Chen),

Week 5 (Nov. 28 - Dec. 4): Keep optimizing performance. Benchmark our implementation with Matlab and Haten2.

Week 6 (Dec. 5 - Dec. 11): Prepare for poster and finish the final report.