Parallel and Distributed Sparse Tensor Decomposition

Ye Yuan, Tengjiao Chen
School of Computer Science
Carnegie Mellon University
{yey1, tengjiac}@andrew.cmu.edu

Abstract

Tensors are increasingly used to analyze very large and sparse multi-way datasets in many areas such as social networks. In this project, we implement a parallel and distributed sparse tensor decomposition toolbox via OpenMP and MPI. By exploring the properties of Khatri-Rao product, we show how we can reduce the memory footprint of this algorithm. We propose two methods to effectively parallelize the Alternating Least Squares (ALS) algorithm in shared address model via OpenMP and message passing model via MPI. Our single machine OpenMP code achieves up to 6x speedup over sequential MATLAB tensor toolbox[6]. Our MPI code is able to decompose a large tensor with 10^8 non-zeros using multiple machines.

1 Introduction

Tensor is high dimensional array in Computer Science. We call 0-dimension tensor "scalar"; 1-dimension tensor "vector"; 2-dimension tensor "matrix". For higher dimension, there is no specific terminology given. As to 3 dimension, we call it 3-way (3-order, or 3-mode) tensor. For example, a tele communication company might record the information of who-call-whom everyday. Within one day, the information can be represented as a matrix. If we want to represent the data for the whole time line, then we will pile up all the matrix of who-call-whom from day 1 to day N (shown in Figure 1). If we look into the frontal slice of this tensor, it is exactly the matrix data between caller and callee on a specific day. Those tensor data are very useful for data mining purpose. For an intuitive example, we know that for matrix we have SVD method to make sense of the data. By SVD, we can know the rank (concept) of a matrix [8]. For three dimension tensor, we do similar analysis like SVD and get useful information like concept from those tensor [7]. The only difference is that tensor is on higher dimension. Table 1 lists all the notions we use in this report.

1.1 Problem: CP Decomposition and ALS method

In order to mine useful information out of tensor, we need to decompose the tensor into many small rank-one tensors. This is the idea of CP decomposition proposed by Hitchcock [3][4]. The CP decomposition can be represented mathematically as below.

$$\mathcal{X} \approx \|\lambda; A, B, C\| = \sum_r \lambda_r \circ a_r \circ b_r \circ c_r \tag{1}$$

The algorithm for CP decomposition we use in this paper is called Alternating Least Squares (ALS). For explanation purpose, I describe the algorithm specifically on three-way tensor. The algorithm is a converging algorithm. First, we initialize the factor matrix **A**, **B**, and **C** with random values. Then we

15618 Course Project (2016).

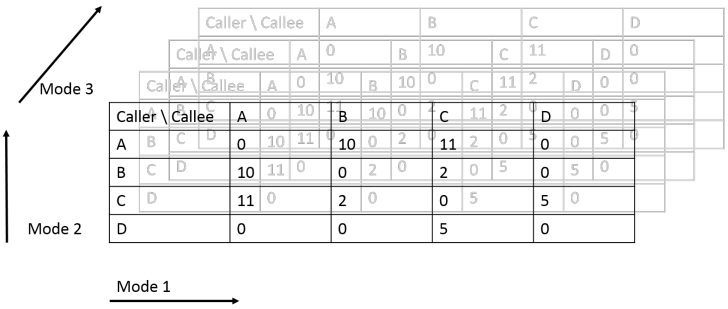

Figure 1: A Tensor Data Example.

Table 1: Tensor Notions

SYMBOL	DESCRIPTION
$\mathcal{X}$	a tensor (assume $\mathcal{X} \in \mathbb{R}^{I \times J \times K}$)
$\mathbf{A}$	a matrix (uppercase, bold font)
$\mathbf{a}$	a vector (lowercase, bold font)
$\mathcal{X}_{(n)}$	mode-n unfolding of tensor $\mathcal{X}$
R	the rank of a tensor $\mathcal{X}$
I, J, K	dimension size of mode-1,2,3 in $\mathcal{X}$
A, B, C	mode-1,2,3 factor matrix of $\mathcal{X}$
$\mathbf{a}_r, \mathbf{b}_r, \mathbf{c}_r$	the rth vector in mode-1,2,3
$\odot$	Khatri-Rao product
$\otimes$	Kronecker product
$*$	Hadamard product
$\mathbf{A}^T$	Transpose of matrix A
$\mathbf{A}^\dagger$	Pseudo inverse of matrix A
$\times_n$	mode-n product

fix **B** and **C** to compute **A**. Then we use updated **A** and **B** to compute **C**. Finally, we use updated **A** and updated **C** to compute **B**. An intuitive way to understand this algorithm is that we can consider **A**, **B**, and **C** as the patterns on three dimensions of this tensor. When we fix **A** and **B**, we are trying to find best **C** to fit into patterns of **A** and **B**. After each iteration, we will find a better fit for each other among **A**, **B**, and **C** until the algorithm converges.

1.2 Challenge: Intermediate Data Explosion

The detailed implementation of ALS algorithm will be given in section 3. Before we see the exact algorithm, we want to present a potential challenge of this algorithm called "Intermediate data explosion".

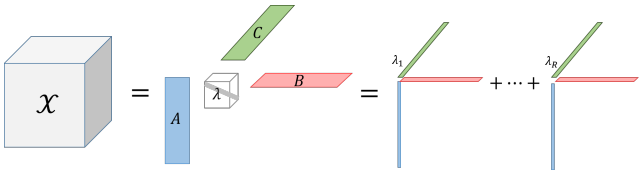

Figure 2: CP Decomposition

Algorithm 1 CPD-ALS

Input : tensor $\mathcal{X} \in \mathbb{R}^{I \times J \times K}$, rank R, iterations T
Output : $\lambda \in \mathbb{R}^R$,
$\mathbf{A} \in \mathbb{R}^{I \times R}$, $\mathbf{B} \in \mathbb{R}^{J \times R}$, $\mathbf{C} \in \mathbb{R}^{K \times R}$
Init **A**, **B**, and **C** with random values;
for t = 1,2...T **do**
  $\mathbf{A} \leftarrow \mathcal{X}_{(1)} (\mathbf{C} \odot \mathbf{B}) (\mathbf{C}^T\mathbf{C}) * (\mathbf{B}^T\mathbf{B})^\dagger$;
  Normalize **A** (storing norms in vector λ);
  $\mathbf{B} \leftarrow \mathcal{X}_{(2)} (\mathbf{C} \odot \mathbf{A}) (\mathbf{C}^T\mathbf{C}) * (\mathbf{A}^T\mathbf{A})^\dagger$;
  Normalize **B** (storing norms in vector λ);
  $\mathbf{C} \leftarrow \mathcal{X}_{(3)} (\mathbf{B} \odot \mathbf{A}) (\mathbf{B}^T\mathbf{B}) * (\mathbf{A}^T\mathbf{A})^\dagger$;
  Normalize **C** (storing norms in vector λ);
  if already converged
  then break;
end for
return λ, **A**, **B**, and **C**

The algorithm looks quite simple at the first glance. Notice that computation of **B** and **C** is symmetric to **A**. The only step we need to implement is

$$\mathbf{A} \leftarrow \mathcal{X}_{(1)}(\mathbf{C} \odot \mathbf{B})(\mathbf{C}^T\mathbf{C}) * (\mathbf{B}^T\mathbf{B})^\dagger \tag{2}$$

However, it is not possible to materialize $(\mathbf{C} \odot \mathbf{B})$. $\odot$ is called "Khatri-Rao product". We also denote $\mathbf{M} = \mathcal{X}_{(1)}\mathbf{C} \odot \mathbf{B}$ as MTTKRP (matricized tensor times Khatri-Rao product). The Khatri-Rao product between matrix **C** and **B** can be expressed as following:

$$\mathbf{C} = \begin{bmatrix} c_{11} & c_{12} & c_{13} & \dots & c_{1r} \\ c_{21} & c_{22} & c_{23} & \dots & c_{2r} \\ \dots & \dots & \dots & \dots & \dots \\ c_{k1} & c_{k2} & c_{k3} & \dots & c_{kr} \end{bmatrix}_{K \times R}$$

$$\mathbf{B} = \begin{bmatrix} b_{11} & b_{12} & b_{13} & \dots & b_{1r} \\ b_{21} & b_{22} & b_{23} & \dots & b_{2r} \\ \dots & \dots & \dots & \dots & \dots \\ b_{j1} & b_{j2} & b_{j3} & \dots & b_{jr} \end{bmatrix}_{J \times R}$$

$$\mathbf{C} \odot \mathbf{B} = \begin{bmatrix} c_{11} \cdot b_{11} & c_{12} \cdot b_{12} & \dots & c_{1r} \cdot b_{1r} \\ c_{11} \cdot b_{21} & c_{12} \cdot b_{22} & \dots & c_{1r} \cdot b_{2r} \\ \dots & \dots & \dots & \dots \\ c_{11} \cdot b_{j1} & c_{12} \cdot b_{j2} & \dots & c_{1r} \cdot b_{jr} \\ c_{21} \cdot b_{11} & c_{22} \cdot b_{12} & \dots & c_{2r} \cdot b_{1r} \\ c_{21} \cdot b_{21} & c_{22} \cdot b_{22} & \dots & c_{2r} \cdot b_{2r} \\ \dots & \dots & \dots & \dots \\ c_{21} \cdot b_{j1} & c_{22} \cdot b_{j2} & \dots & c_{2r} \cdot b_{jr} \\ \dots & \dots & \dots & \dots \\ c_{k1} \cdot b_{11} & c_{k2} \cdot b_{12} & \dots & c_{kr} \cdot b_{1r} \\ c_{k1} \cdot b_{21} & c_{k2} \cdot b_{22} & \dots & c_{kr} \cdot b_{2r} \\ \dots & \dots & \dots & \dots \\ c_{k1} \cdot b_{j1} & c_{k2} \cdot b_{j2} & \dots & c_{kr} \cdot b_{jr} \end{bmatrix}_{(KJ) \times R}$$

As we know $\mathbf{B} \in \mathbb{R}^{J \times R}$ and $\mathbf{C} \in \mathbb{R}^{K \times R}$, their Khatri-Rao product should be $(\mathbf{C} \odot \mathbf{B}) \in \mathbb{R}^{(JK) \times R}$. For a medium tensor, usually J and K are around 10^6. So JK will be 10^{12}. Assume we store each element as double with 8 byte. Then we will need 8×10^{12} bytes (i.e. 8 TB) space to store those intermediate results. This is of course not something desirable. So we need to avoid those intermediate data explosion. We will show how to solve this problem in later sections.

1.3 Contributions

In our project, our main contribution is a parallel and distributed sparse tensor decomposition toolbox. By exploring the properties of Khatri-Rao product, we show how we can reduce the memory footprint of this algorithm. We also propose two methods to effectively parallelize the Alternating Least Squares (ALS) algorithm in shared address model via OpenMP and message passing model via MPI. The advantages of our methods are:

Performance: It is fast. Our single machine OpenMP code achieves up to 6x speedup over sequential MATLAB tensor toolbox[6].

Scalability: It scales well with the number of thread or processes and is able to handle larger-than-memory tensors. For OpenMP code, we achieved 6x speedup with 12 threads. For MPI code, we achieved 5x speedup up with 32 processes. We are able to decompose a tensor with 10^8 entries within several minutes.

2 Related work

Giga Tensor[6] is a MATLAB toolbox for tensor decomposition. As we know, MATLAB toolbox can only support in-memory computation. However, as the tensor data is getting larger and no longer fit fully into memory, there is a need for a technique to deal with larger-than-memory tensors. This paper proposed a Map-Reduce framework with much better scalability than MATLAB toolbox.

HaTen2 [5] is similar to Giga Tensor. It is during the time that tensor is becoming increasingly larger and cannot possibly fit into memory. It uses Hadoop to implement tensor decomposition in a lamda fashion. This paper is a further optimization on top of Giga Tensor. By running the Haten2 binary we find that the performance is way too slow for any practical usage and it also suffers from the garbage collection problem in runtime especially when we are decomposing large tensors (with more than 10^8 entries).

DFacTo [2] proposed a system for distributed tensor factorization. It introduced a way to avoid data explosion by exploring the properties of the Khatri-Rao product. The algorithm only requires two sparse matrix-vector products and is easy to parallelize. This is the main paper we referenced. We adopt DFacTo's idea to partition the tensor by rows. However, we dive deeper into how to achieve a balanced workload partition and we further parallelize more steps such as factor normalization.

3 Methods

In this section, we will discuss how to effectively solve the data explosion problem and how to parallelize the algorithm in both shared address and message passing models.

3.1 Handle Data Explosion

A. Derivation

Remember that the MTTKRP is the bottelneck of computing CP decomposition of tensor: $\mathbf{M} = \mathcal{X}_{(1)}\mathbf{C} \odot \mathbf{B}$. Assume $\mathcal{X}$ is a dense tensor, we can write the each row of $\mathbf{M}$ in this way:

$$\mathbf{M}(i,:) = \sum_{k=0}^{K}\sum_{j=0}^{J}\mathcal{X}(i,j,k)(\mathbf{B}(j,:) * \mathbf{C}(k,:)) \tag{3}$$

From this equation, we can find that for any row of $\mathbf{M}$, we actually only need to sum up all $\mathcal{X}(i,j,k)(\mathbf{B}(j,:) * \mathbf{C}(k,:))$ with non-zero of $\mathcal{X}$(i,j,k). Based on this observation, we can give an updated version of MTTKRP process without explicitly computing Khatri-Rao product in Algorithm 2. In this way, we can enumerate all non-zero elements in a slice of $\mathcal{X}$ and get the updated row of matrix $\mathbf{M}$.

B. Data Structure for Tensor

To effectively calculate modified MTTKRP step in Algorithm 2, we store the tensor using a hierarchical list. Each mode is stored as a list of slices. For each slice, we use two lists to store the indices

Algorithm 2 MTTKRP

Input : Slice $\mathcal{X}$(i, :, :), **B**, **C**
Output : Row **M**(i, :)
Init **M**(i, :) $\leftarrow$ 0
for all non-zero elements **in** $\mathcal{X}$(i, :, :) **do**
  M(i, :) $\leftarrow$ **M**(i, :) + $\mathcal{X}$(i, j, k) (**B**(j, :) * **C**(k, :))
end for

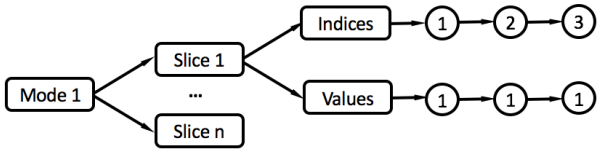

Figure 3: Data structure for tensor

and values of each non-zero element (shown in Figure 3). For indices, unlike the Compressed Row Storage (CSR) format, we only use one integer to represet the indices. For exaample, in node 1, we can use $z = k \times K + j$ to represent the index of a non-zero value. In MTTKRP, we can use $z\%J$ and z/J to recover index j and index k, which are used to retrieve corresponding rows from factor matrices **B** and **C**.

In this storage scheme, we will need 3 copies of the tensor. Assume we have m non-zero elements, we will need $3m$ integers to store indices, and $3m$ floating point numbers to store the tensor values. By storing the tensor in this way, it is convenient for us to parallelize the ALS algorithm by slices.

3.2 Shared Address Model

In this section, we will talk about how to parallelize the algorithm under a shared address model.

A. Task mapping

We have pointed out a way to compute each row of matrix **M** in Algorithm 2. We can also find that: each row of matrix **M** can be calculated in parallel. Thus, the parallelization for a shared address model is quite straight forward: each thread pick up a slice of tensor $\mathcal{X}(i, :, :)$ and calculate corresponding row $\mathbf{M}(i, :)$, which is shown in Figure 4. Since each thread will write to distinct rows of **M**, there is no lock required. Each thread may require R words to store the temperal reuslt of **M**. Since $R \ll nnz$, this storage is negligible.

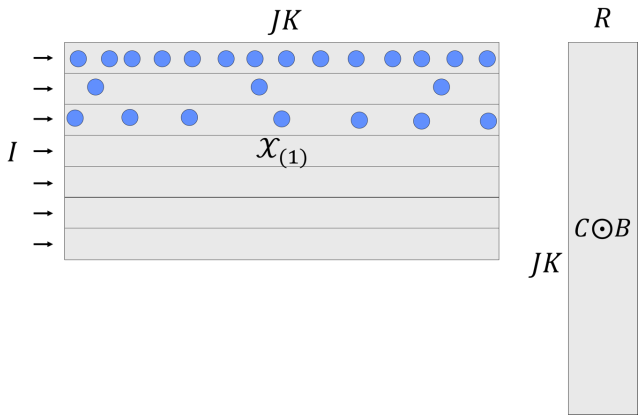

Figure 4: Task mapping for shared address model

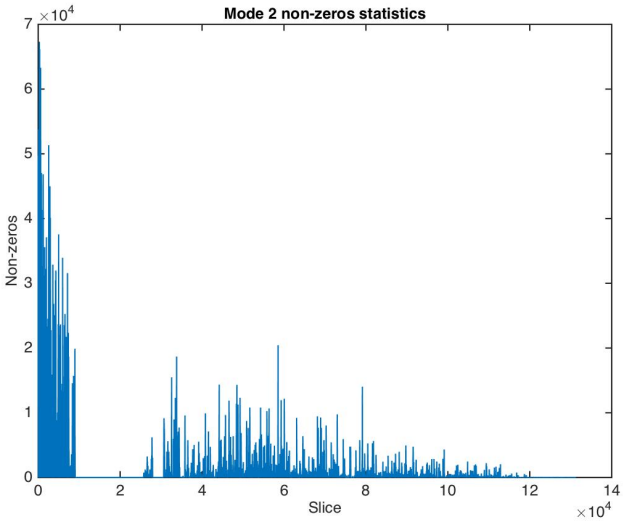

Figure 5: Mode 1 non-zero statistics for ML ratings dataset

B. Dynamic vs. Static Scheduling

Although the parallelization scheme is straight forward and easy to implement. In our initial attempt, we have very little performance gain with more threads. Later we find that this is due to the extremely imbalanced distribution of non-zeros among slices. In Figure 5, we count the number of non-zeros in each slice of dataset ML ratings. We can find that the number of non-zeros across the slices in ML ratings can vary by several orders of magnitude. Some slices have almost 10,000 non-zeros, some slices only have several non-zeros. Since the computation load is proportional to non-zeros, this unstructured pattern of tensor leads to very severe load imbalance.

The default parallel for-loop of OpenMP uses a static decomposition. OpenMP runtime will divide the loop into equal-sized chunks or as equal as possible. This will assign hugely different amounts of work to each thread, which leads to severe load imbalance and harm scalability.

Fortunately, OpenMP also supports dynamic scheduling for parallel for-loop. For dynamic scheduling, OpenMP runtime will use the internal work queue to give a chunk-sized block of loop iterations to each thread. When a thread is finished, it retrieves the next block of loop iterations from the top of the work queue. However this is not the end of story. Keeping a work queue and fetching work from queue will involve extra overheads. We should be careful about the granularity of work. By default, the chunk size is 1. In experiments we find that it brings too much overheads to schedule so many tasks. And if the chunk size is too large, we can not effectively balance work load. Empirical results show that chunk size 16 is a good choice in our problem. We will discuss about is in section 4.

3.3 Message Passing Model

In this section, we will talk about how to decompose the task into different processes under a message passing model. Our main idea is to let each process hold a fraction of tensor $\mathcal{X}$. Each process will calculate its own partial results and communicate with each other. We will illustrate the whole process based on Figure 6.

A. Task decomposition

Notice that in the code we actually use "Allreduce" and "Allgatherv" to transfer data. For convenience, we choose to use a master-slave architecture to illustrate our method.

Step 0: In the beginning of computation, the master node will read the tensor and conduct workload partition. In this process, the target is to determine a balanced workload for each process, which is the key part of MPI code. We will talk about it in later section. Master will need to determine continuous rows that every process should hold. Then, master node will broadcast the partition result to each

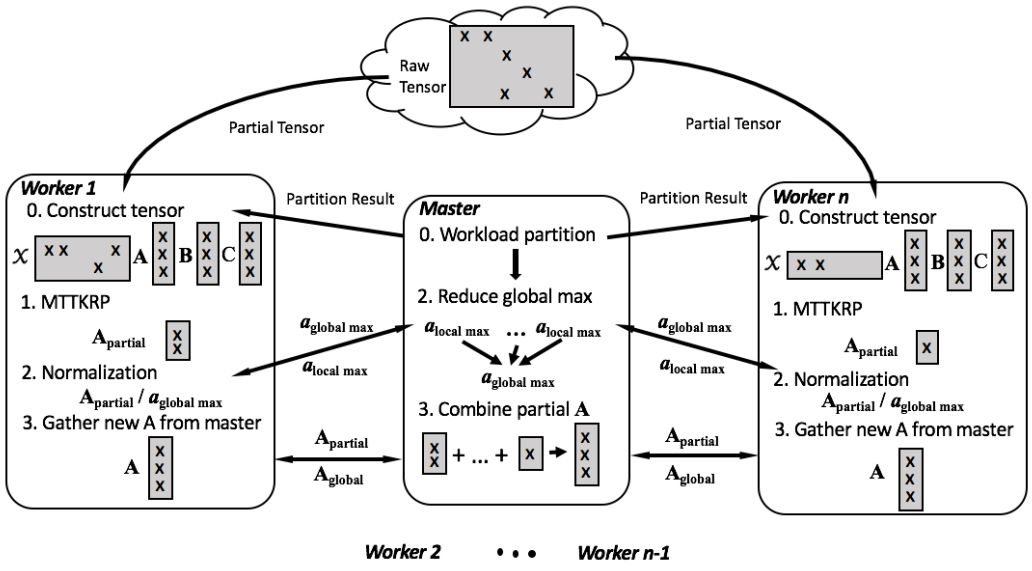

Figure 6: Task mapping architecture for message passing model

worker. After receiving this information, each worker will read the tensor file and construct its own partial tensor $\mathcal{X}_{partial}$ (step 0 in Figure 6). Also, we decide to let each worker has an identical copy of factor matrices **A**, **B**, **C**.

Step 1: Then let's see how we execute one iteration of ALS algorithm. Figure 4 only shows the update process of **A**. In each iteration, each process calculate its own partial MTTKRP result $\mathbf{A}_{p_i}$. Since $\mathbf{N} = (\mathbf{C}^T\mathbf{C}) * (\mathbf{B}^T\mathbf{B})^\dagger$ is very fast, we decide to let each process redundantly compute **N**. By multiplying **N**, each process has its own unnormalized partial $\mathbf{A}_{p_i}$. The final complete **A** will be:

$$\mathbf{A} = \mathbf{MN} = [\mathbf{M}_{p_1}\mathbf{N} \quad \mathbf{M}_{p_2}\mathbf{N} \quad ... \quad \mathbf{M}_{p_n}\mathbf{N}] \tag{4}$$

Step 2: Before gathering the complete **A**, we need to normalize each column of factor **A**. In this step, each element in A should divide the maximum value in its own column. However, each process only holds a fraction of complete **A**. So after each worker comes up with it's local maximum value, the master node will reduce the global maximum value of each column of **A**. Each process will then pull this global maximum value and finish normalization.

Step 3: After normalization, the master node will gather all partial $\mathbf{A}_{p_i}$ and format the complete **A** and sent it back to each worker. Now we have updated the factor **A**, which is needed for the update of factor **B** in next iteration.

We only take the update of **A** as an example in Figure 6. In real implementation, we will repeat step 1 to step 3 for updates of each factor matrix **A**, **B** and **C** until convergence.

B. Communication volume analysis

So far, we have described the task mapping scheme for our message passing model. There are two forms of overheads in message passing model: data communication and load imbalance among processes. In Figure 6, we have 2 main data communication steps. In step 2, the amount of data we need to transfer is $O(IR)$, which is actually independent with number of processes. In step 3, the amount of data we communicate is $O(nR)$, where n is the number of processes. Since $R \ll I$, communication volume of this step is negligible.

C. Balanced workload partition

As we have seen in shared address model, the workload in each slice can vary a lot. Master node is responsible for finding an appropriate workload decomposition for workers.

Table 2: Tensor Data

Name	Shape	Non-zeros
FreeBase-1	23M $\times$ 23M $\times$ 166	9.9×10^7
FreeBase-2	38M $\times$ 38M $\times$ 532	1.39×10^8
NELL	26M $\times$ 26M $\times$ 48M	1.44×10^8

Our first idea is to partition rows based on number of non-zeros because computational load is proportional to the number of non-zeros assigned to a process. This is actually a chains-on-chains partition problem [9]. The objective of the CCP problem is to find a sequence of n-1 separator indices to divide a chain of tasks with associated computational weights into consecutive parts such that the bottleneck value—the maximum workload of a processor—is minimized. [9] introduced some optimal and fast solutions to find such a partition.

However, in practice we find that a greedy partition works well enough: we start from the first row of tensor, traverse each row and greedily assign a slice to a process until a partition has reached a target work load $\mathcal{X}(nnz)/n$, which is an ideal balanced work load. Since slices can vary in non-zeros by several orders of magnitude, adding a dense slice may push a partition significantly over the target size. When we find that adding a slice will push a partition over target size, we will compare which one is closer to average work load and decide whether adding this slice or not. We will repeat this process for each mode and record the start and end rows of each process, which are broadcasted to workers.

In experiments, we find that decomposing the work by non-zeros is not enough. The reason is that: only the work load of MTTKRP is related to number of non-zeros. The work load of factor update and normalization is actually proportional to number of rows.

4 Experiments

A. Experimental Setup

We implement our code in pure C++ with OpenMP and MPI. Indices of tensor are stored using 64-bit unsigned integers. Values of tensor are stored in double precision floating pointer numbers. We conduct all our experiments in SCS rocks cluster. The single machine we use has 20 Intel(R) Xeon(R) X5355 processors running at 2.66 GHz. We compiler our code using g++ (GCC) 4.9.2 and optimization level O3. And we set tensor rank to 10 in all experiments.

B. Datatset

We evaluated the performance of our code from data sets in Table 2. Free Base Music 1(FB1), and Free Base Music 2(FB2) are from real data. FB1 records the information of what user listens to which music for 166 days. FB2 records the similar information for 532 days. NELL is from Never Ending Language Learning project [1]. Since we don't have access to processed tensor data, we choose to random generate a NELL tensor according to its shape and density.

C. OpenMP

First we want to show the speedup we can get against MATLAB tensor toolbox. MATLAB uses in-memory method with several optimizations. However, it is a single thread solution and doesn't scale with more threads. We tested our method with 12-thread dynamic setting and compared our performance with MATLAB. Figure 7 shows our results.

We tested our performance on two data sets: FB1 and FB2. We can find that our method is about 6$\times$ faster than MATLAB. The experimental results prove that by exploring the parallelism of the algorithm, we can accelerate tensor decomposition drastically.

In second part, we want to show the scalability of our algorithm. We ran our algorithm with 1, 2, 4, 8, 12 threads over two data sets (FB1 and FB2). We used two settings in our experiments: dynamic scheduling and static scheduling. Figure 8 shows the results.

In the graph, two solid lines represent dynamic scheduling solution. Two dash lines represent static scheduling solution. We can see that both scheduling scales with number of threads. Dynamic

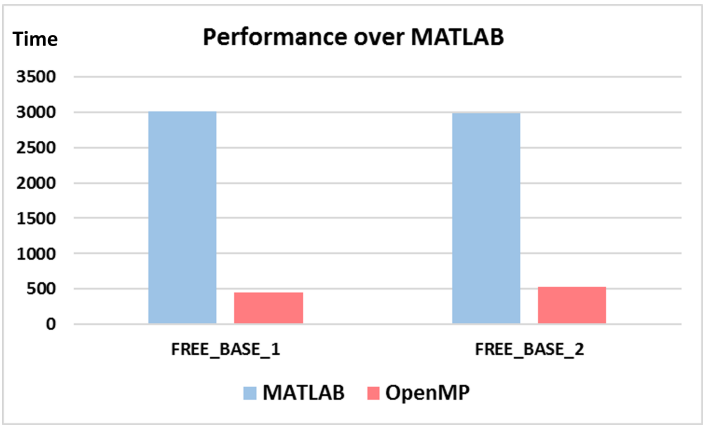

Figure 7: Performance over MATLAB.

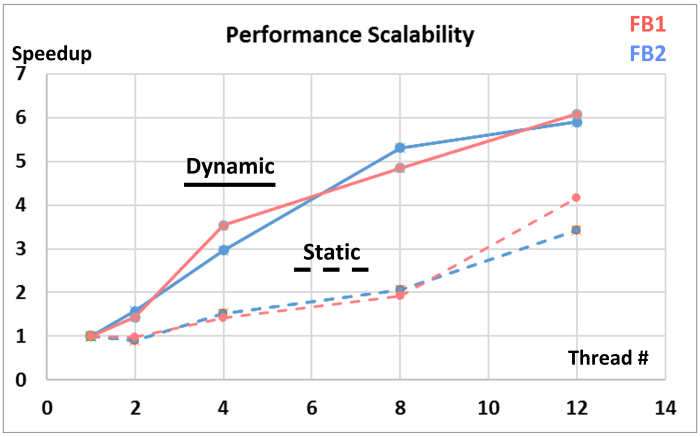

Figure 8: Performance over MATLAB.

scheduling scales much better than static scheduling. The reason is what we explained before: Tensor is very sparse, the distribution of non-zero entries scatter non-uniformly across different slices. Static scheduling will starve some of the threads, leads to unbalanced work distribution and thus has poor performance. Dynamic scheduling is more flexible and leads to a more balanced work distribution among different threads. However, due to the overhead involved in dynamic scheduling, we can not get perfect speedup.

D. MPI

We also tested our MPI code with larger tensors. Our initial plan is to compare with Haten2. But later we find that HaTen2 is way slower than our code. So we decide only report the scalability of our implementation. We tested our code in two settings: pure MPI and hybrid MPI + OpenMP.

Figure 9 shows results of FB1 tensor. For this dataset, we use 1 node and up to 16 processors. Since we have only 16 processors, we only spawn 2 threads in each process and use dynamic scheduling described in shared address model. We can find that with OpenMP, we can get further speedup. However, because we can not perfectly balance workload and the overheads brought by data communication, we can not achieve perfect scalability.

Figure 10 shows results of NELL tensor. For this dataset, we use 2 nodes and up to 32 processors. Since this data is uniformly generated, we don't have load imbalance problem. We can observe much better scalability. But due to data communication overheads, we still can not get perfect linear speedup.

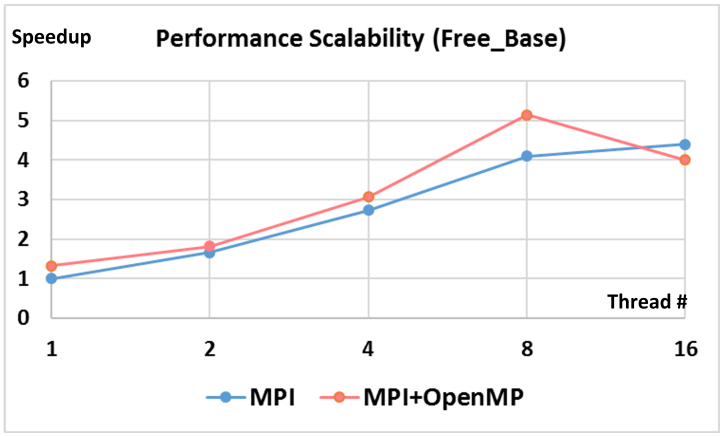

Figure 9: MPI scalability in FreeBase-1 dataset.

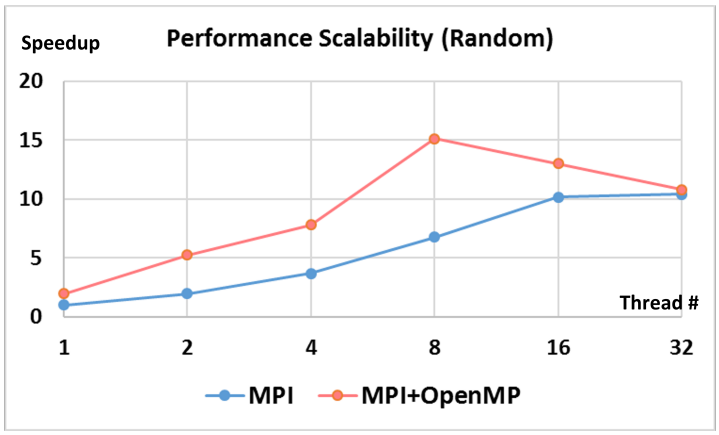

Figure 10: MPI scalability in FreeBase-1 dataset.

5 Conclusions

A. Lessons learned

In this project, we successfully implement fast and scalable sparse tensor decomposition toolbox. We have learned so much in this project:

1. Load balancing really matters in parallel computing. We tried many different ways to balance work load and gain some experiences. For static scheduling, the key challenge is to choose a precise measurement of workload. We still don't have an answer for how to choose such measurement. For dynamic scheduling, it's very important to choose an appropriate granularity of task.

2. Profiling is very important in performance optimization. If we want to optimize a program, the first thing we should do is always profiling. For example, in our first version of MPI code, normalization is not paralleliezd because we think this part "should be fast". After profiling we find that with more processes, the normalization will take similar amount of time with MTTKRP. Then we decide to further parallelize this part and achieved better scalability.

B. Future work

There are still a lot of work we can do for this topic. For example, in shared address model, we only tried OpenMP's dynamic scheduling feature. Cilk is famous for its dynamic scheduling capacity, we can also use Cilk to parallelize our algorithm and compare it with OpenMP. For message passing model, there are two forms of overheads: data communication and load imbalance among processes. In our project, we only optimized load imbalance. We choose to let each process keep a copy of

factor **A**, **B** and **C**. The amount of data communicated is proportional to the size of factor matrices. In fact, we may consider to further distribute factor matrices in processes and realize a more fine-grained task mapping.

C. Distribution of Total Credit

Equal work was performed by both project members.

References

[1] Andrew Carlson, Justin Betteridge, Bryan Kisiel, Burr Settles, Estevam R Hruschka Jr, and Tom M Mitchell. Toward an architecture for never-ending language learning. In *AAAI*, volume 5, page 3, 2010.

[2] Joon Hee Choi and S Vishwanathan. Dfacto: Distributed factorization of tensors. In *Advances in Neural Information Processing Systems*, pages 1296–1304, 2014.

[3] Frank L Hitchcock. The expression of a tensor or a polyadic as a sum of products. *Journal of Mathematics and Physics*, 6(1):164–189, 1927.

[4] Frank L Hitchcock. Multiple invariants and generalized rank of a p-way matrix or tensor. *Journal of Mathematics and Physics*, 7(1):39–79, 1928.

[5] Inah Jeon, Evangelos E Papalexakis, U Kang, and Christos Faloutsos. Haten2: Billion-scale tensor decompositions. In *2015 IEEE 31st International Conference on Data Engineering*, pages 1047–1058. IEEE, 2015.

[6] U Kang, Evangelos Papalexakis, Abhay Harpale, and Christos Faloutsos. Gigatensor: scaling tensor analysis up by 100 times-algorithms and discoveries. In *Proceedings of the 18th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 316–324. ACM, 2012.

[7] Tamara G Kolda and Brett W Bader. Tensor decompositions and applications. *SIAM review*, 51(3):455–500, 2009.

[8] Ch Aswani Kumar and S Srinivas. Concept lattice reduction using fuzzy k-means clustering. *Expert systems with applications*, 37(3):2696–2704, 2010.

[9] Ali Pınar and Cevdet Aykanat. Fast optimal load balancing algorithms for 1d partitioning. *Journal of Parallel and Distributed Computing*, 64(8):974–996, 2004.