Low-overhead Load-balanced Scheduling for Sparse Tensor Computations

Muthu Baskaran, Benoît Meister, Richard Lethin
Reservoir Labs Inc.
New York, NY 10012
Email: {baskaran,meister,lethin}@reservoir.com

***Abstract*—Irregular computations over large-scale sparse data are prevalent in critical data applications and they have significant room for improvement on modern computer systems from the aspects of parallelism and data locality. We introduce new techniques to efficiently map large irregular computations onto modern multi-core systems with non-uniform memory access (NUMA) behavior. Our techniques are broadly applicable for irregular computations with multi-dimensional sparse arrays (or *sparse tensors*). We implement a static-cum-dynamic task scheduling scheme with low overhead for effective parallelization of sparse computations. We introduce locality-aware optimizations to the task scheduling mechanism that are driven by the sparse input data pattern. We evaluate our techniques using two popular sparse tensor decomposition methods that have wide applications in data mining, graph analysis, signal processing, and elsewhere. Our techniques not only improve parallel performance but also result in improved performance scalability with increasing number of cores. We achieve around 4-5x improvement in performance over existing parallel approaches and observe "scalable" parallel performance on modern multi-core systems with up to 32 processor cores. We take real sparse data sets as input to the sparse tensor computations and demonstrate the achieved improvements.**

I. Introduction

Tensors or multi-dimensional arrays are a natural way of representing data with multiple aspects and dimensionality. Multi-linear algebraic computations such as tensor decompositions are becoming increasingly important in real-world applications for extracting and explaining the properties of such data. Tensor decompositions have applications in a range of domains such as signal processing, data mining, computer vision, numerical linear algebra, numerical analysis, graph analysis [1]. Kolda et al. have published a survey on various tensor decompositions and their applications [2]. Two of the prominent tensor decompositions are CANDECOMP/PARAFAC (CP) and Tucker decompositions.

The tensors representing real data are usually very large and sparse making the computations performed over them irregular. These large sparse tensors pose non-trivial challenges to optimize computations performed with them. The irregular codes are, in general, harder to parallelize as the amount of parallelism is dependent on the input data that is known only at runtime. Further, the irregular codes are usually memory-bound and spend lot of time in memory accesses. Furthermore, the data locality optimizations may depend on the input data complicating the design of optimizations to improve the performance of irregular codes. Additionally, the volume of data processed in critical data applications are growing in size, adding to the complexity.

978-1-4799-6233-4/14/$31.00

In this work, we study an important problem, namely, optimizing and parallelizing sparse tensor computations on modern multi-core systems. We develop techniques to parallelize the computations by scheduling them across processor cores with a low-overhead scheme that achieves good load-balance across the cores. We introduce a locality-aware factor in the task scheduling decision mechanism to improve data locality that is driven by the sparse data pattern of the input tensor.

We evaluate our techniques using two popular sparse tensor decomposition methods that are widely used in various real-world data analysis applications such as cybersecurity, environmental sensor analysis, and the like. We take real-world sparse data sets as input to our evaluation of the sparse tensor computations and demonstrate the achieved improvements. Our techniques not only improve parallel performance but also result in improved performance scalability with increasing number of cores. We achieve around 4-5x improvement in performance over existing parallel approaches and observe scalable parallel performance on modern multi-core systems with up to 32 cores.

The rest of the paper is organized as follows. We give a brief background on the tensor decomposition methods that we use to evaluate our techniques in Section II. We present our techniques to achieve an improved low-overhead load-balanced scheduling of large sparse tensor computations on modern multi-core systems in Section III. We demonstrate the efficiency of our techniques in Section IV. We conclude with a summary and a foreword to our future work in Section VI.

II. Background

In this Section, we give some background information that will help to understand our techniques and experimental study in this work. We discuss the basic definitions and algorithms for tensor decompositions.

A tensor is a multi-dimensional array and the *order* of a tensor is the number of dimensions, also called as *modes*, of the tensor. Two popular and prominent tensor decompositions are CANDECOMP/PARAFAC (CP) and Tucker decompositions. We will focus our discussion to CP decomposition and two widely used algorithms to determine CP tensor decomposition that we use for our further study and experimental evaluation in this paper.

a) CP Decomposition: The CP tensor factorization decomposes a tensor into a sum of component rank-one tensors (A N-way tensor is called a rank-one tensor if it can be expressed as an outer product of N vectors). The CP decomposition that factorizes an input tensor $\mathcal{X}$ of size $I_1 \times \cdots \times I_N$ into R components (with factor matrices $\mathbf{A}^{(1)} \ldots \mathbf{A}^{(N)}$ and weight vector λ) is of the form:

$$\mathcal{X} = \sum_{r=1}^{R} \lambda_r \mathbf{a}_r^{(1)} \circ \cdots \circ \mathbf{a}_r^{(N)}$$

where $\mathbf{a}_r^{(n)}$ represents the r^{th} column of the factor matrix $\mathbf{A}^{(n)}$ of size $I_n \times R$.

b) CP-ALS Algorithm: The widely used workhorse algorithm for CP decomposition is the alternating least squares (ALS) method. The CP-ALS algorithm is presented in Algorithm 1 [2].

Algorithm 1 CP-ALS Algorithm

initialize $\mathbf{A}^{(1)} \ldots \mathbf{A}^{(N)}$
repeat
  for $n = 1 \ldots N$ **do**
    $\mathbf{V} = \mathbf{A}^{(1)T}\mathbf{A}^{(1)} * \cdots * \mathbf{A}^{(n-1)T}\mathbf{A}^{(n-1)} * \mathbf{A}^{(n+1)T}\mathbf{A}^{(n+1)} * \cdots * \mathbf{A}^{(N)T}\mathbf{A}^{(N)}$
    $\mathbf{U} = \mathbf{X}_{(n)}(\mathbf{A}^{(N)} \odot \cdots \odot \mathbf{A}^{(n+1)} \odot \mathbf{A}^{(n-1)} \odot \ldots \mathbf{A}^{(1)})$
    $\mathbf{A}^{(n)} = \mathbf{U}\mathbf{V}^{\dagger}$
  end for
until convergence

c) CP-APR Algorithm: Chi and Kolda [3] have developed an Alternate Poisson Regression (APR) fitting algorithm for the non-negative CP tensor decomposition which is based on a majorization-minimization approach. The CP-APR algorithm described in [3] is presented in Algorithm 2.

Algorithm 2 CP-APR Algorithm

initialize $\mathbf{A}^{(1)} \ldots \mathbf{A}^{(N)}$
repeat
  for $n = 1 \ldots N$ **do**
    $\mathbf{B} = \mathbf{A}^{(n)}\Lambda$
    $\Pi = (\mathbf{A}^{(N)} \odot \cdots \odot \mathbf{A}^{(n+1)} \odot \mathbf{A}^{(n-1)} \odot \ldots \mathbf{A}^{(1)})^T$
    repeat
      $\Phi = (\mathbf{X}_{(n)} \oslash (\mathbf{B}\Pi))\Pi^T$
      $\mathbf{B} = \mathbf{B} * \Phi$
    until convergence
    $\lambda = e^T\mathbf{B}$
    $\mathbf{A}^{(n)} = \mathbf{B}\Lambda^{-1}$
  end for
until convergence

The main computations involved in the above algorithms are:

- Tensor mode-n matricization or representing the N-dimensional tensor as a two dimensional matrix, $\mathcal{X} \rightarrow \mathbf{X}_{(n)}$
- Matrix transpose
- Inverse of a diagonal matrix
- Pseudo-inverse of a matrix, $\mathbf{V}^{\dagger}$
- Matrix matrix multiplication, $\mathbf{C} = \mathbf{A}\mathbf{B}$
- Matrix Khatri-Rao product, $\mathbf{C} = \mathbf{A} \odot \mathbf{B}$
- Matrix element-wise division, $\mathbf{C} = \mathbf{A} \oslash \mathbf{B}$
- Matrix element-wise product (Hadamard product), $\mathbf{C} = \mathbf{A} * \mathbf{B}$

When the input tensor is dense, the above-mentioned computations are all dense and regular. However when the input tensor is sparse, as is the case in our study, the computations involving (and affected by the) sparsity of the input tensor are also sparse and irregular. In our study, we focus on the sparse or irregular computations in these algorithms.

The sparse computations that we focus are:

- `'sparse' matricized tensor times Khatri-Rao product (mttkrp) computation` in CP-ALS algorithm:

$$\mathbf{X}_{(n)}(\mathbf{A}^{(N)} \odot \cdots \odot \mathbf{A}^{(n+1)} \odot \mathbf{A}^{(n-1)} \odot \ldots \mathbf{A}^{(1)})$$

- Π `computation` in CP-APR algorithm:

$$\Pi = (\mathbf{A}^{(N)} \odot \cdots \odot \mathbf{A}^{(n+1)} \odot \mathbf{A}^{(n-1)} \odot \ldots \mathbf{A}^{(1)})^T$$

- Φ `computation` in CP-APR algorithm:

$$\Phi = (\mathbf{X}_{(n)} \oslash (\mathbf{B}\Pi))\Pi^T$$

III. Techniques for Improving Sparse Tensor Computations

Improving the parallel performance of sparse tensor computations is a non-trivial challenge due to the following reasons: 1) the amount of parallelism is dependent on the non-zero pattern of the input sparse tensor data that is known only at runtime, and 2) tensor computations have "mode-specific operations" - operations that access data with respect to the orientation of a mode or dimension of the multi-dimensional data - that make optimizations specific to one mode ineffective for other modes. Furthermore, most of the tensor computation methods involve iterative repetition of mode-specific operations that makes the strategy of applying optimizations specific to a mode ineffective, as the liveness of the optimizations becomes limited. The techniques that we developed to address the challenges in optimizing sparse tensor computations center around the following aspects: 1) identifying and extracting more concurrency, 2) reducing barrier synchronizations or thread waits, 3) improving data locality, 4) improving load balance among processors, and 5) reducing parallelization overheads such as the task scheduling overhead.

A. Identifying and Extracting Parallelism

The tensor computations such as the tensor decomposition methods involve complex data dependencies within the algorithm that make the task of exploiting concurrency a challenging one. Among the various possible concurrency or parallelism patterns, some of the most prevalent ones in sparse tensor computations are: 1) presence of parallelism to compute independent partial results from non-zeros of input tensor data and a reduction of the partial results after an all-to-all barrier synchronization across all processors, 2) presence

of parallelism to compute independent partial results and using a critical section or lock mechanism to commit partial results to the final output data structure, and 3) presence of synchronization-free parallelism to compute results from non-zeros of input tensor data. The first two patterns require more synchronizations across processors. However partitioning and balancing the workload among processors in such computational patterns may not be challenging. The computational load can be statically divided and balanced among processors (and need not be dynamically divided based on the sparse pattern of the input tensor). Each processor can independently work on a balanced computational load and produce independent results on local variables that are merged through a reduction phase or produce independent results that are committed atomically using a critical section or lock.

In the case of computational pattern with synchronization-free parallelism, there is no synchronization involved. However it is non-trivial to efficiently (with low overhead and good load-balance) partition the computations across the processors while exploiting synchronization-free or synchronization-minimal parallelism. If there is load imbalance, it can negate performance benefits achieved from reduced synchronizations. The load imbalance arises when the workload is divided based on static information rather than dynamic information induced by the sparsity of the data. For example, in the sparse tensor decomposition methods, the size of the output matrices having the decomposition results, is known statically. However the number of non-zeros of the sparse data contributing to each element of the output matrices is dependent on the sparse data pattern that is only known at runtime. So, in the case of sparse tensor decompositions, if the workload is divided among processors based on the number of output elements they compute, it may result in load imbalance. This is because the amount of computations done to produce an output element varies across elements as it is dependent on the number of non-zeros contributing to the element. Good load balance is possible only when the sparse data pattern is used to drive the computation partitioning. Hence there is a need for dynamic partitioning of workload at runtime.

Our approach to optimize sparse tensor computations tries to exploit, as much as possible, parallelism in computations with minimal synchronization, and tries to balance the workload among processors with low overhead. We develop a technique that identifies good partitioning of the sparse data computations across processors. The main "partitioning criterion" behind our technique is that the computations involving the non-zeros of input tensor are partitioned for each mode into groups based on the index of the mode along which computations are performed (called the candidate mode) such that computations involving any two non-zeros in the sparse input tensor with the same index along the candidate mode fall in the same partition group. The number of partition groups depend on the number of processors (in our case each processor is assigned a partition group). The partitioning is done dynamically while performing the computations. For each mode, the computations are performed in a fine-grained parallel manner and dynamically assigned to processors such that the partitioning criterion is satisfied. This dynamic scheduling of computations (or tasks) to processors and partitioning of the workload is done by a light-weight runtime layer. The runtime layer adds very little overhead in distributing the computations. Performing the partitioning along with the fine-grained parallel execution of the original computations helps in achieving a good balanced load distribution.

B. Improving Load Balancing with Reduced Overhead

While proper scheduling is needed to schedule the computations across the processors in a balanced manner, it is also equally very important to have a low overhead scheduling so that the benefits of load balancing are not offset by the scheduling overhead. It is well known that a parallelizing scheduler with a static task scheduling scheme has less scheduling overhead, but may not balance the load properly if the load balancing depends on dynamic runtime information (such as the sparse data pattern in sparse tensor computations). On the other hand, a parallelizing scheduler with a dynamic task scheduling scheme has more scheduling overhead than the static scheme but can do better load balancing based on runtime information. We come up with an approach that combines the benefits of static and dynamic task scheduling, namely, less scheduling overhead and better load balance.

Our approach works as follows. When there are multiple iterations of a computation block and if the computation block is to be distributed across processors and if the non-zero structure of the sparse tensor (or the data access pattern of the sparse tensor) does not change within the block, the first iteration of the block is peeled and executed with a dynamic task scheduling scheme. The runtime layer logs "state" information about the processor workload (such as which portions of the computation block (or tasks) get executed on which processor). In the subsequent iterations, the logged information about the processor workload is used to schedule tasks across processors, thereby avoiding the need for dynamic scheduling to balance load at runtime, and execute the computation block with low scheduling overhead. This static-cum-dynamic (or "hybrid") task scheduling approach greatly reduces the task scheduling overhead and also guarantees an improved load balance across processors.

In the case of sparse tensor computations, as mentioned earlier, there is usually a repetition of mode-specific operations and the iteration with dynamic scheduling is carefully chosen before the start of a computation block where the logged information can be reused. The logged information is sometimes stored for all modes, if the storage requirement does not blow up memory (depending on the number of tasks and the number of modes in the tensor).

C. Improving Performance through Better Data Locality

The following data locality optimizations are achieved directly as a result of exploiting a synchronization-free or synchronization-minimal parallelization pattern: 1) bringing the production and consumption of data values closer and local to a processor (within the computation region that lies in between processor synchronizations) and 2) keeping data values local to a processor (as much as possible). In addition

to the above optimizations that are obtained as a result of the parallelization, we also introduce techniques to improve data locality in modern multi-core systems with NUMA behavior. We make locality-aware decisions while scheduling tasks to processors. Whether we exploit synchronization-free parallelism or not, we give priority to schedule tasks that share data on to processors that are closer in the NUMA topology. However the information of what data is shared is also mostly known only at runtime and it depends on the pattern of the non-zeros of the input data.

In our approach, processors dynamically acquire workload through the light-weight runtime layer, while doing the actual computations simultaneously. Depending on the location of the data needed for the computations, each processor may spend different duration of time in waiting to load/store data (due to cache misses and other NUMA effects). This may lead to an imbalance in the workload acquired by each processor. The runtime layer migrates workload (or equivalently tasks) from one processor to another so as to balance the load and the migration is governed by the NUMA topology. The processors are assumed to be arranged in a binary tree-like NUMA topology. Therefore a processor is assumed to be close to one processor at the innermost level, three processors at the next level and so on. From the binary representation of a processor id (an integer between 0 and p-1, where p is the number of processors), we identify the NUMA neighbors of the processor at each level. We now explain how the workload or task migration works.

Each processor has a local work queue and it dynamically acquires its workload, through the runtime layer, from a work pool of tasks and fills its local work queue. The runtime layer then accesses each processor's workload and characterizes each processor as 'overloaded' or 'underloaded' with respect to the average workload (average workload is simply the total amount of computations needed by the problem divided by the number of available processors). For each 'underloaded' processor, the runtime layer exclusively selects a topologically closer 'overloaded' processor to borrow workload from. The migration is done such that neither the 'borrower' processor is overloaded nor the 'lender' processor is underloaded as a result of migration. If load imbalance still exists, the runtime layer selects the next topologically closer 'overloaded' processor for the 'underloaded' processor, and repeats the migration attempt going along the topology till the balance is met. Since the workload or tasks are migrated between processors based on their NUMA topology, data sharing is facilitated between NUMA neighbors. This optimization is very vital in both performance improvement and performance scaling.

In our hybrid static-cum-dynamic task scheduling scheme, this NUMA-aware task migration is performed only during the dynamic phase of the scheduling scheme and the load balanced schedule, resulting after task migration, is logged and used during the static phases of the scheduling scheme.

IV. Experimental Results

We provide a detailed experimental study on the evaluation of our techniques to improve the parallel performance and scalability of sparse tensor computations on multi-core systems.

A. Data Sets

We used real sparse tensor data sets to evaluate the techniques described in our work. The real data sets used are: 1) Facebook social network data from [4], 2) Enron email data from [5], and 3) cyber data gathered from a network traffic sensor in our lab during a particular interval of time on a day.

The Facebook data set is a $63891 \times 63891 \times 1591$ tensor with 737934 non-zero entries, representing activities between 63891 Facebook users over 1591 days. The Enron email data set is a $105 \times 105 \times 27$ tensor with 5418 entries, representing emails exchanged between 105 users over a period of 27 months. The cyber data is a very large five-dimensional sparse tensor of size $14811811 \times 1899 \times 1899 \times 3 \times 6067$, with the modes being `timestamp`, `senderIP`, `receiverIP`, `protocol`, and `responseBytes`. It has 2085108 entries.

B. Measurements

The baseline sequential versions in our study include the sequential implementation of CP-ALS and CP-APR algorithms (Algorithms 1 and 2 in Section II). We measure time taken to execute the CP-ALS and CP-APR algorithms on the above mentioned data sets. Since our techniques focus on sparse tensor computations, for the purpose of clearly demonstrating the effectiveness of our techniques, we report only the timing of relevant portions of the algorithms involving computations related to the sparse input tensor. The sparse tensor computations account for a significant portion of the total execution time in these methods. In the CP-ALS method, the sparse computations account for around 65% of the total execution, and in the CP-APR method, they constitute around 85% of the total execution time.

C. Experimental System

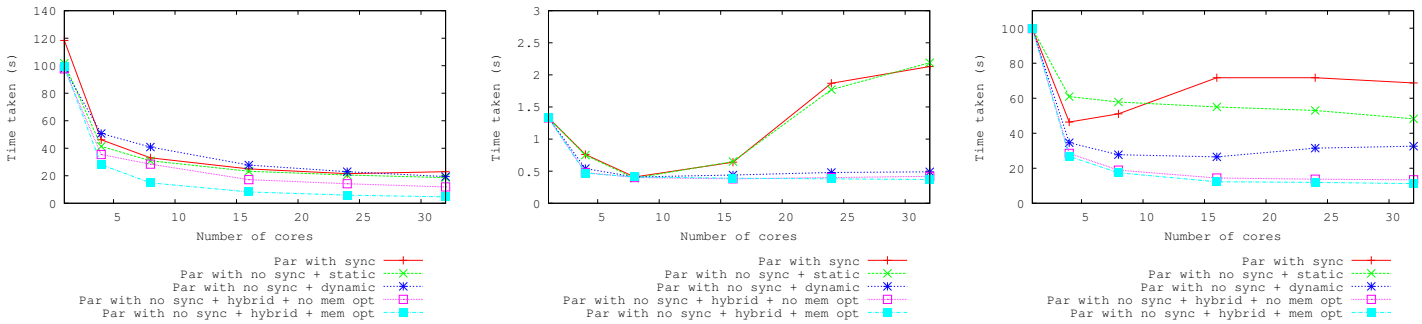
We use a modern multi-core system to evaluate our techniques. The system we use is a quad socket 8-core system with Intel Xeon E5-4620 2.2 GHz processors (Intel Sandy Bridge microarchitecture chips) and it supports 32 concurrent threads (64 threads if the Hyper-threading feature is used, which we do not use for our experiments). Each of the 8 cores in a socket has a 32 KB private L1 instruction cache, a 32 KB private L1 data cache, and a 256 KB private L2 cache. Each core also shares a 16 MB L3 cache. The system has 128 GB of DRAM.

We use the Intel icc compiler, version 13.1.1 for our experiments. We set KMP_AFFINITY=compact for our experiments. We use OpenMP library to parallelize the computations in our study.

D. Parallel Versions for Evaluation

The following parallel versions of the CP-APR and CP-ALS algorithm are used for our performance evaluation:

- `Par with sync` - This is the parallel version in which workload is partitioned statically and independent partial results are calculated by each processor in local variables and reduced after a barrier synchronization.



(a) Facebook Data (b) Enron Data (c) Cyber Data

Fig. 1. Parallel performance of CP-APR on different data sets on 32-core system

- `Par with no sync` - This is the synchronization-free parallel version in which workload is partitioned statically. It has serious load balancing concerns.
- `Par with no sync + dynamic` - This is the parallel version in which workload is partitioned dynamically based on the sparse pattern of the input data. This version needs no synchronization, has good load balance, but suffers from dynamic scheduling overhead.
- `Par with no sync + hybrid + no mem opt` - This is the parallel version that uses hybrid static-cum-dynamic task scheduling scheme for workload distribution described in Section III-B. This version needs no synchronization, has good load balance, and also has less dynamic scheduling overhead.
- `Par with no sync + hybrid + mem opt` - This is same as `Par with no sync + hybrid + no mem opt` version, additionally having the data locality optimization described in Section III-C. This version is the representative version that includes all optimizations discussed in this work.

E. Load Balancing

We first illustrate how efficiently load balancing is achieved in sparse tensor computations using our approach. For illustrative purpose we discuss our study performed using the $63891 \times 63891 \times 1591$ Facebook tensor with 737934 non-zeros.

The load on each processor in the parallelization of CP-APR method depends on the number of non-zeros that each processor needs for its computation. So the number of non-zeros processed by each processor is a good measure of load on each processor. Static scheduling that distributes workload based on the statically known information, namely, "number of output elements", leads to poor load balance. This is observed in our experiments and this is evident from Figure 2 that shows the number of non-zeros distributed in each of the 32 cores in the quad socket 8-core system. Dynamic scheduling based on the pattern of the sparse data performs very good load balance (as shown in Figure 2).

F. Parallel Performance and Scalability

We first discuss our experimental results on the performance of the CP-APR algorithm improved using our techniques.

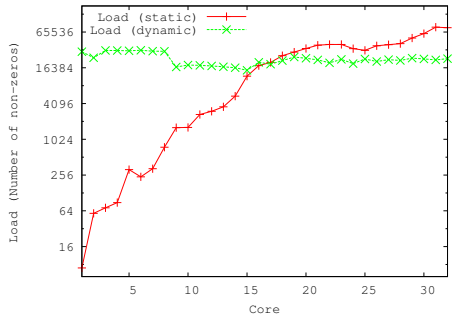
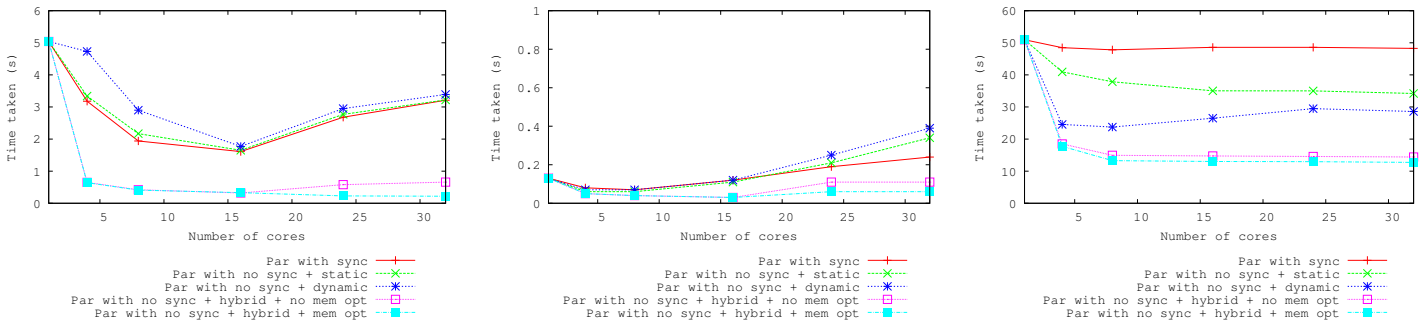


Fig. 2. Static and dynamic load distribution

Figure 1(a) shows the improvement in execution time due to our techniques in the CP-APR method operating on the Facebook tensor data. As it can be seen, the performance improvement due to our light-weight hybrid static-cum-dynamic task scheduling clearly performs better than the other versions with pure static or pure dynamic task scheduling. Also, it is evident that the parallel versions (even the ones with reduced synchronizations) perform poorly when there is more memory traffic across sockets (i.e. when processor cores in different sockets are concurrently executing and generating memory traffic and polluting different levels of cache). Our data locality optimizations play a vital role in performance improvement when cores across sockets are involved in execution. Even when all the cores are fully occupied and when there is more possibility of shared cache pollution, we are able to get good performance improvement, mainly because of the data locality optimizations respecting the NUMA model.

The execution time of sparse tensor computations in the CP-APR method operating on the smaller Enron data set is shown in Figure 1(b). Our techniques show improved performance compared to other versions. Also, it can be seen that the performance degrades at higher core counts without low-overhead load balancing and data locality optimizations. Even with our techniques, though the performance does not degrade, we do not see a good scalability. This is possibly because of the smaller size of the Enron data set that implies lesser parallelism available to be exploited.

Figure 1(c) shows the execution time of sparse tensor



(a) Facebook Data (b) Enron Data (c) Cyber Data

Fig. 3. Parallel performance of CP-ALS on different data sets on 32-core system

computations in the CP-APR method on the cyber data set. We make the following observations. When more than one socket is in use, the memory traffic by processor cores across sockets slows down the performance affecting the scalability. Without locality-aware low-overhead load-balanced scheduling, the performance drops with increasing number of processor cores.

From Figure 1, it is evident that locality-aware load balancing across parallel cores with less overhead is key for performance scalability. Also, we get around 4-5x performance improvement compared to the parallelization approach (`Par with sync` version) that currently exists in literature for sparse tensor computations.

The improvements from our techniques are also evident from the experiments on CP-ALS algorithm using Facebook, Enron, and cyber data sets as shown in Figure 3. For CP-ALS on Facebook data, the performance of all versions except for the one with all our optimizations, deteriorates at core counts higher than 16. For CP-ALS on cyber data, the performance of all versions except for the one with all our optimizations does not improve or deteriorates as number of processor cores increases. For CP-ALS on Enron, though our techniques show higher performance compared to other versions on the same number of cores, our techniques did not scale as well for higher core counts. As explained in the case of CP-APR on Enron data set, the smaller problem size of the Enron data set is a possible reason for such a behavior at higher core counts as there is potentially less parallelism available to be exploited across more cores.

V. Related Work

There is a rich literature on techniques for optimizing graph computations (e.g. very recently [6]). There has been a lot of study on hypergraph partitioning techniques as well (e.g. [7]). However these techniques are only well suited for partitioning sparse matrix computations. They might introduce costly overhead for sparse tensor computations. Kang et al. [8] have presented techniques to handle computations with very large sparse tensors on a distributed memory environment. Many modern distributed memory clusters are clusters of multi-core systems and our solution is important to improve the performance on each multi-core system within a cluster and hence it can be coupled with the work of Kang et al. to improve the overall performance and scale in such systems.

VI. Conclusion

In this work, we have introduced new techniques to efficiently map large irregular computations with sparse tensors (sparse multi-dimensional arrays) onto modern multi-core systems. We have implemented a locality-aware low-overhead static-cum-dynamic task scheduling scheme for effective parallelization of sparse tensor computations. We have demonstrated an improvement of around 4-5x using our techniques on two widely sparse tensor decomposition algorithms and have shown scalable parallel performance on modern multi-core systems with up to 32 processor cores. In future, we would like to evaluate various emerging runtime system software, such as the Open Community Runtime [9], developed as a part of various ongoing high-performance computing research activities and compare/integrate our techniques with them.

References

[1] D. Dunlavy, T. Kolda, and W. P. Kegelmeyer, "Multilinear Algebra for Analyzing Data with Multiple Linkages," in *Graph Algorithms in the Language of Linear Algebra*, J. Kepner and J. Gilbert, Eds. SIAM, 2011.

[2] T. G. Kolda and B. W. Bader, "Tensor Decompositions and Applications," *SIAM Review*, vol. 51, no. 3, pp. 455–500, September 2009.

[3] E. C. Chi and T. G. Kolda, "On Tensors, Sparsity, and Nonnegative Factorizations," arXiv:1112.2414 [math.NA], December 2011. [Online]. Available: http://arxiv.org/abs/1112.2414

[4] B. Viswanath, A. Mislove, M. Cha, and K. P. Gummadi, "On the evolution of user interaction in facebook," in *Proceedings of the 2nd ACM SIGCOMM Workshop on Social Networks (WOSN'09)*, August 2009.

[5] J. Shetty and J. Adibi, "The Enron Email Dataset - Database Schema and Brief Statistical Report." [Online]. Available: http://www.isi.edu/ adibi/Enron/Enron.htm

[6] M. Frasca, K. Madduri, and P. Raghavan, "Numa-aware graph mining techniques for performance and energy efficiency," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, ser. SC '12. Los Alamitos, CA, USA: IEEE Computer Society Press, 2012, pp. 95:1–95:11.

[7] U. V. Catalyurek and C. Aykanat, "PaToH: A Multilevel Hypergraph Partitioning Tool, Version 3.0, Bilkent University, Department of Computer Engineering, Ankara, 06533 Turkey."

[8] U. Kang, E. Papalexakis, A. Harpale, and C. 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*, ser. KDD '12. New York, NY, USA: ACM, 2012, pp. 316–324.

[9] V. Sarkar, B. Chapman, W. Gropp, R. Knauerhase, T. Mattson, and W. Pinfold, "Open Community Runtime," https://01.org/projects/open-community-runtime.