Reducing Inter-process Communication Overhead in Parallel Sparse Matrix-Matrix Multiplication

Md Salman Ahmed*, Jennifer Houser†, Mohammad Asadul Hoque*, Phil Pfeiffer*
*Department of Computing
†Department of Mathematics and Statistics
East Tennessee State University
ahmedm@goldmail.etsu.edu, houserjd@goldmail.etsu.edu, hoquem@etsu.edu, phil@etsu.edu

***Abstract*—Parallel sparse matrix-matrix multiplication algorithms (PSpGEMM) spend most of their running time on inter-process communication. In the case of distributed matrix-matrix multiplications, much of this time is spent on interchanging the partial results that are needed to calculate the final product matrix. This overhead can be reduced with a one dimensional distributed algorithm for parallel sparse matrix-matrix multiplication that uses a novel accumulation pattern based on logarithmic complexity of the number of processors (i.e., $O(log(p))$ where p is the number of processors). This algorithm's MPI communication overhead and execution time were evaluated on an HPC cluster, using randomly generated sparse matrices with dimensions up to one million by one million. The results showed a reduction of inter-process communication overhead for matrices with larger dimensions compared to another one dimensional parallel algorithm that takes $O(p)$ run-time complexity for accumulating the results.**

***Keywords*—MPI communication pattern, overhead communication, parallel computing, performance analysis, scalability, sparse matrix-matrix multiplication, V2V communication**

I. Introduction

The widespread use and importance of matrix applications has created a compelling need for efficient algorithms for matrix-matrix multiplication. Matrix representations of real-world phenomena have numerous applications in science and technology, in fields that include electrical engineering, medical science, physics, quantum chemistry [1], mathematics, and computer science. Matrix-matrix multiplication is indispensable for almost every research field that involves scientific computation and numerical methods like optimization, linear algebra, algebraic multigrid [2], finite element analysis, and tensor contraction [3]. In computer science, areas such as graphics, networking, wireless communication, video and audio analysis, image processing, graph theory [4], big data analysis [5] and language processing [6] use matrix-matrix multiplication to a great extent. Networks, for example, are commonly modeled with adjacency matrices: two-dimensional matrices whose elements represent connections and weights between a network's nodes. Repetitive multiplication of adjacency matrices can determine multi-hop reachability, transitive closure and dynamic partitioning within a mobile ad hoc network.

Researchers have worked for several decades to devise matrix-matrix multiplication algorithms that outperform the traditional, $O(n^3)$ algorithm. The need for such algorithms is driven by the processing of very large matrices, often with trillions of elements. Currently the fastest matrix-matrix multiplication algorithm, the Coppersmith-Winograd algorithm, has a run time complexity of $O(n^{2.375477})$ [7]. In computations involving matrices of larger dimensions, the main challenge for the matrix multiplication algorithm is a scarcity of computational resources. Increasingly, parallel processing is being used to address this challenge.

In one important special case, the nature of the data being processed creates particular opportunities for fast multiplication. Sparse matrices, or matrices whose elements consist largely of zeros, are commonly used to model real-world phenomena. Algorithms for sparse matrix-matrix multiplication improve on classic algorithms by focusing solely on products of nonzero elements. These algorithms' performance depends on factors that include the number and distribution of nonzero elements in the matrices to multiply, the structures used to store the matrices, the number of processors allocated to a computation, and the efficiency of inter-processor coordination. In particular, the use efficient communication models and data structures can greatly speed up parallel multiplication.

Over the past few decades, researchers have extensively studied the Parallel Sparse Generalized Matrix-Matrix multiplication problem, hereafter referred to as PSpGEMM [8]. Numerous algorithms ([8]–[21]) have been designed that apply a variety of distribution models, storage mechanisms, and communication models to PSpGEMM. These approaches have been incorporated into standard libraries and tools such as BLAS. Despite of all these efforts, however, the impact of inter-process communication cost on the overall speed up and scalability has received relatively little attention. The scalability of any PSpGEMM algorithm depends largely on its strategy for inter-process communication, due to the amount of communication needed to exchange partial results between processors during the compilation of the final product matrix.

This paper describes a comparison of two one-dimensionally distributed PSpGEMM algorithms in terms of the impact of inter-process communication cost. The first one, developed previously by Hoque et. al. [22], uses an algorithm with $O(p)$ run-time complexity to accumulate partial results. It is hereafter referred to as the Original version of PSpGEMM, the other uses a novel $O(log(p))$ algorithm to accumulate results. This comparison focuses on how communication overhead, particularly MPI overhead, impacts these algorithms' performance, relative to randomly generated sparse matrices with dimensions up to one million by one million. These preliminary results indicate a significant reduction of

inter-process communication overhead for matrices with larger dimensions compared to the Original PSpGEMM algorithm [22]. Section II reviews published communication models for PSpGEMM. Section III presents the algorithms' method of matrix-matrix multiplication. Section IV presents the details of the two algorithms (Original and Logarithmic) in terms of the communication patterns. Section V presents the results of performance analysis. Section VI concludes by summarizing these findings and discussing avenues for future work.

II. Related Work

The scalability and performance of parallel SpGEMM algorithms is highly dependent on inter-process communication, where most of these algorithms' execution time is spent. Most algorithmic designs, however, focus more on computation techniques rather than optimizing communications. A very few classical algorithms describe the communication cost of sparse matrix-matrix multiplication. A unified communication analysis of existing and new algorithms as well as an optimal lower bound for communication cost of two new parallel algorithms are given in [9]. In this paper, optimal communication costs of three 1D algorithms such as Naïve Block Row [8], Improved Block Row [23] and Outer Product [24] are outlined in terms of bandwidth costs and latency costs.

In [10] Ballard et al. describe CAPS, a parallel, communication-optimal algorithm for matrix multiplication. Their algorithm seeks to efficiently balance the load among participating processors while minimizing inter-processor communication. It recasts Strassen's sequential algorithm as a recursive tree, dividing the multiplication algorithm into 7 sub problems, based on whether the dimensions of the matrices to multiply are large (unlimited memory scheme with BFS traversal) or small (limited memory scheme with DFS traversal).

In [11] Ballard et al. describe a hypergraph partitioning approach for parallel sparse matrix-matrix multiplication. It models SpGEMM using a hypergraph and reduces the communication cost by communicating between processors along a critical path of the multiplication algorithm.

In [12] Utrera et al. discuss SpGEMM-related communication imbalances caused by the communication library and the interconnection network. The authors characterize this imbalance as a major source of performance degradation for sparse matrix-vector multiplication. They analyze their characterization using the fork-join and task based implementations and MPI protocols.

Most PSpGEMM algorithms assume that an efficient communication model is a natural consequence of an effective computation model. Only a very few papers describe the specific overhead due to the distribution and accumulation of partial results between processors: the source of most communication overhead. In what follows, we attempt to address the need for a better understanding of these overheads by providing a theoretical framework for an efficient partial results accumulation pattern; an implementation the pattern; and an analysis of the implementation's efficiency.

III. Outer Product Matrix Multiplication

Both of the algorithms studied use outer product matrix multiplication to solve $AB = C$, where A and B are sparse matrices of size $N \times N$. We assume that both A and B are symmetric matrices.

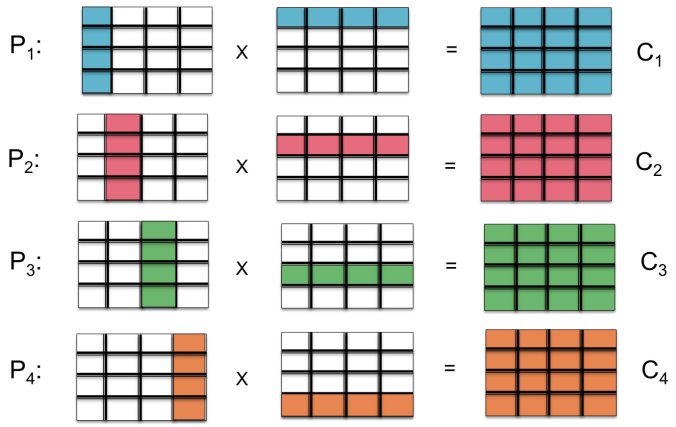

Fig. 1: Distribution of an input matrix using outer product multiplication on four processors.

Both algorithms parallelize a serial method for matrix multiplication that begins by computing the outer product of A and B. This method takes the i^{th} column of matrix A and multiplies it by the j^{th} row of matrix B to produce a sub matrix C_i of dimension $N \times N$. This is continued such that each column of A and each row of B is multiplied together, which produces a total of N sub matrices: $C_1, \ldots, C_N$. The resulting sub matrices are summed element-wise to produce the final result, matrix C, as shown in equation 1:

$$\sum_{i=1}^{N} C_i = C. \tag{1}$$

In the following description of this algorithm's parallel implementations, we let p denote the total number of processors, N/p the number of rows or columns of the input matrix sent to each processor P_i and α the average number of non-zero elements in each row/column of an input matrix. Initially, the algorithms divide input matrices A and B into blocks of size N/p, distributing them over p processors. Each processor computes the outer product on its part of the matrix by multiplying each column in the block with each row in the block to produce a sub matrix C_i. The average number of non-zero elements in each row/column of a sub matrix C_i is at most α^2. Figure 1 illustrates the distribution of a matrix over four processors to produce four sub matrices.

Once each processor computes the sub-matrix that contains its portion of the results, the partial results are merged through the sending and receiving of data to corresponding processors. This merging is done based on the patterns outlined in the next section. Because of the resulting matrix's size (on the order of 10^{12} elements for the largest input size 10^6), the final matrix C is left distributed over the processes.

IV. Implementation of PSpGEMM Algorithm

We present two versions of parallel sparse matrix-matrix multiplication algorithms with distinct merging schema to

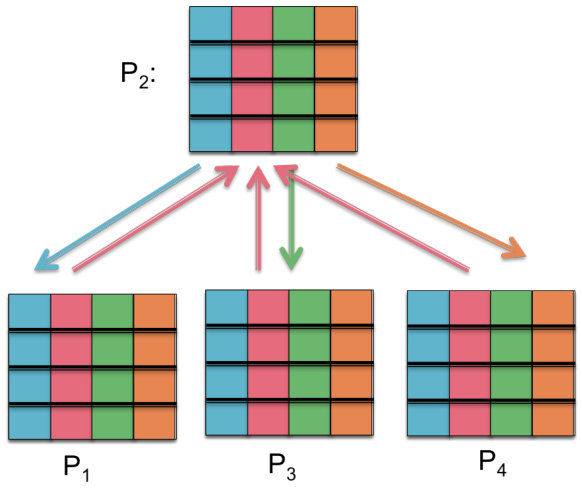

Fig. 2: Merging results onto process two using four processes in total.

illustrate a reduction in complexity created by a reduction in communication overhead. Both versions use the same storage mechanism and hashing techniques as described in [22]. The algorithms differ only in number of times data is sent and received between nodes during the merging of partial results that follows the computation of the sub matrices. We also present the mandatory and auxiliary storage mechanism for the two algorithms to exchange data.

A. Original Merging Pattern

The first merging pattern accumulates its partial results as follows. After each sub matrix is calculated, it is repartitioned into p column-wise blocks and then redistributed. Each process sends the i^{th} block of its sub matrix to the corresponding i^{th} processor to be merged with the partial results being received from the other processes. Figure 2 illustrates processor P_2 merging its results with the remaining three processors: processors P_1, P_3, and P_4 send partial results from their second block to P_2, and processor P_2 sends the partial results in the first, third, and fourth block to P_1, P_3, and P_4, respectively.

Based on the distribution process described in section II, if each processor receives $\lceil \frac{N}{p} \rceil$ columns upon the distribution of the input matrices, the total number of non-zero elements each process contains after computing its sub matrix C_i is equal to $\alpha^2 \lceil \frac{N}{p} \rceil$. Because each process exchanges data with $p-1$ processes, every process communicates an average of $\frac{p-1}{p} \alpha^2 \lceil \frac{N}{p} \rceil$ elements. Accordingly, the amount of data that a process transfers to other processes using this communication pattern has complexity of $O(\frac{\alpha^2 N}{p})$ [22].

The total communication overhead is determined by the number of processes that send and receive data, the amount of data transferred, and delays created by the irregular distribution of non-zero elements throughout the input matrices and the resulting variation in the number of computations each process needs to calculate its portion of the partial result. Let the largest of these delays, the synchronization delay, be denoted by δ. The total communication overhead is then given as $(p-1)(\lceil \frac{N}{p} \rceil + \delta)$.

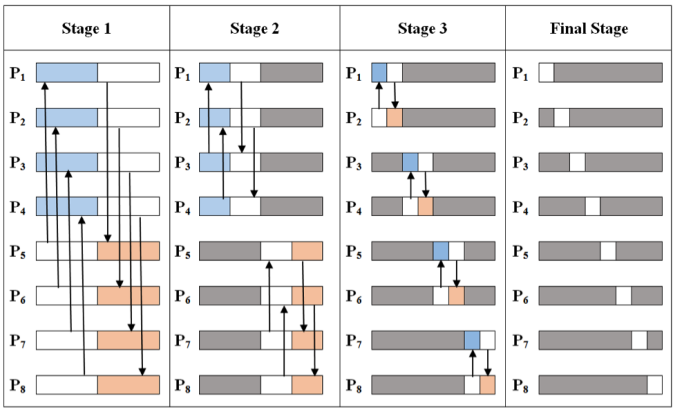

Fig. 3: Logarithmic communication between processes.

B. Logarithmic Merging Pattern

In the proposed Logarithmic merging pattern, each process P_i sends its partial results to another process in $log(p)$ number of stages where p is the total number of processes involved in calculating the partial results. In each of these stages, the process P_i divides its total partial result matrix into two bins. The first bin contains the elements of the partial matrix whose column indexes are less than a mid-value. The second contains the elements whose column indexes are greater or equal to this mid-value. The mid-value is calculated in each stage for a particular computing process from the number of column-wise blocks per process. This calculation also determines a low index (l) and a high index (h), based on the number of processes (p) and a process's rank: a unique number assigned to each processor. These indices determine which bin to send and which to receive.

After dividing the partial result matrices into two bins, process P_i calculates the rank (r) of another process P_j with which to interchange bins. P_i then exchanges half of its partial results with P_j by sending one of the two bins and receiving the other.

Figure 3 illustrates the merging pattern for 8 processes where each process communicates with other processes in 3 (i.e., $log_2(8)$) stages. In each stage, a processor P_i determines another processor P_j to send to, along with the bin to send. For example, in the first stage P_1 sends its second bin to P_5, while P_5 sends its first bin to P_1. Each process P_i distributes half of the partial results to P_j and discards the contents of the bin that was sent while appending the contents that it receives to its other bin. For example, P_1 appends the contents received from P_5 to its first bin and removes the contents from its second bin. Similarly, P_5 appends the contents received from P_1 to its second bin and removes the contents from its first bin. The gray areas in figure 3 indicate the removed contents.

Since each process divides its partial results into two bins at each stage, a process creates a total of p bins after completing the $log(p)$ number of stages. In the final stage, each process contains partial results from each of the p processes including itself. For example,

- In stage 1, results are exchanged between process pairs P_1 and P_5; P_2 and P_6; P_3 and P_7; and P_4 and P_8.

```
calculate_lhbr ( p, rank, s)
{
        x =  p / pow (2, s)
        if rank ≤ x then
                l = 0
                h = x
        else
                l = x * (round_to_next_integer (rank / x) – 1)
                h = x * round_to_next_integer (rank / x)
        endif

        mid = (l + h) / 2
        if rank ≤ mid then
                r = mid + rank – l
                b = 1
        else
                r = rank – mid + l
                b = 0
        endif
}
```

Fig. 4: Calculation of l, h, r and b values.

```
calculate_mid_value (l, h, rank, bpp)
{
        mid = (l + h) / 2
        if rank ≤ mid then
                half = ceil (((l + mid) * bpp) / 2)
        else
                half = ceil (((mid + h) * bpp) / 2)
        endif
}

logarithmic_communication ( p, bpp)
{
        stages = log2( p)
        rank = get_current_process_rank()

        for each s in stages do
                calculate_lhbr ( p, rank, s)
                calculate_mid_value (l, h, rank, bpp)
                divide the partial results into two bins using the mid-value half
                send bin b to process r
                append the received bin from process r
        end for
}
```

Fig. 5: Logarithmic merging algorithm.

In this exchange, each process acquires one additional set of partial results, generated by the other. Following stage 1, processes pairs P_1 and P_5; P_2 and P_6; P_3 and P_7; and P_4 and P_8 share each others' results.

- In stage 2, results are exchanged between P_1 and P_3; P_2 and P_4; P_5 and P_7; and P_6 and P_8. In this exchange, each process acquires two additional sets of partial results: one set generated by the exchange's other process and a second this other process acquired during stage 1. Following stage 2, processes P_1, P_3, P_5, and P_7 share results, as do processes P_2, P_4, P_6, and P_8.
- In stage 3, results are exchanged between P_1 and P_2; P_3 and P_4; P_5 and P_6; and P_7 and P_8. In this exchange, each process acquires the remaining four sets of partial results. Following stage 3, all processes have one another's partial results.

At each stage, each process must determine a low value, a high value, the rank of another process with which to exchange data, and the bin (one of two) to send to the other process. Let

$$\begin{aligned} rank &= \text{the computing process's rank} \\ s &= \text{the current stage} \\ bpp &= \text{number of column-wise blocks per process} \\ half &= \text{the mid-value for dividing the partial results} \end{aligned}$$

Each process then uses the algorithm from figure 4 to calculate l, the process's low value for this stage; h, the process's high value for this stage; b, the index of the bin to send; and r, the other process's rank.

Figure 5 shows the Logarithmic algorithm's procedure for managing overall inter-process communication. In this algorithm the mid-value is calculated for diving the partial results into two bins.

Because the Original and Logarithmic algorithms implement identical methods for computing each matrix's partial results, each process's computations on each of its sub matrices will average $\alpha^2\lceil\frac{N}{p}\rceil$ operations resulting in $O(\alpha^2\lceil\frac{N}{p}\rceil)$ complexity. Based on the merging schema in the proposed communication pattern, the partial results are accumulated in $log_2(p)$ stages where p is the number of processes. On each stage, any one process of the p processes transfers on average $\frac{1}{p}th$ of the total data, i.e., on average the amount is $\alpha^2\lceil\frac{N}{p}\rceil$. Since the accumulation of partial results is done in $log_2(p)$ stages, the total amount of data transferred between processes is $log_2(p)\alpha^2\lceil\frac{N}{p}\rceil$, which results in a complexity of $O(log_2(p)\alpha^2\lceil\frac{N}{p}\rceil)$. Similarly to the delay in communication caused by varying computation times between nodes, the inclusion of the synchronization delay between nodes causes the total overhead communication to have complexity of $O(log_2(p)\alpha^2\lceil\frac{N}{p}\rceil + \delta)$.

C. Data Structures

Storing just the non-zero data elements of a sparse matrix greatly reduces the amount of space that such matrices consume. The two algorithms use lists (e.g., vectors) to store a matrix's data elements. This list pairs each data element with its row and column index.

The matrices generated by the outer product computations are stored in a hash table. Each element's hash key is generated from its row and column indices as its hash key. Hash keys are uniform over the size of the hash table. Collisions resulting from the hashing of multiple elements to the same key are managed using external hashing: i.e., with a key-indexed linked list. Each hash table stores partial results as well as a portion of the final result in the end.

In order to exchange a block of data with other processors, partial results must be copied from the hash table to a

Fig. 6: ETSU HPC Clusters

contiguous chunk of sequential memory (e.g., an array).

V. Performance Analysis

The performance of the two PSpGEMM algorithms was analyzed on Knightrider, one of two high-performance computing clusters at East Tennessee State University's High Performance Computing Center (figure 6). Knightrider, which the university obtained in 2011, consists of 48 HP Proliant BL280c G6 compute nodes and 1 HP DL380 G7 master node. The cluster totals 588 processors with 2.3 terabytes of memory, where each node contains a dual Xeon X5650 2.66 GHz processor, 12 cores, and 48 gigabytes of memory. The nodes are connected using a 4x QDR Infiniband interconnect and Voltaire 36-port Infiniband switches. The cluster hosts a total of 30 terabytes of central storage on its hard drives and 160 gigabytes of local storage on its compute nodes [25].

Each of the PSpGEMM algorithms was evaluated in terms of its total execution time, total distributed computing time, average computation time per process, total MPI communication overhead, and average communication overhead per process. The experimental parameters that were varied include the input matrix's dimension (up to one million) and the number of computing processes (up to 256). The total number of processes excludes a separate, master process, which both algorithms use to load the input file into memory: only the computation nodes are included in the calculations.

As indicated by figures 7 and 8, the Logarithmic merging pattern reduces the average communication overhead and total communication overhead incurred by the Original merging pattern. Figure 9 shows that Original merging algorithm and the Logarithmic merging algorithm exhibit almost equal total overhead communication for input file $N = 100,000$. For the larger input sizes of $N = 500,000$ and N = $1,000,000$, the proposed merging algorithm exhibits lower total overhead communication. This may suggest that the greatest benefits

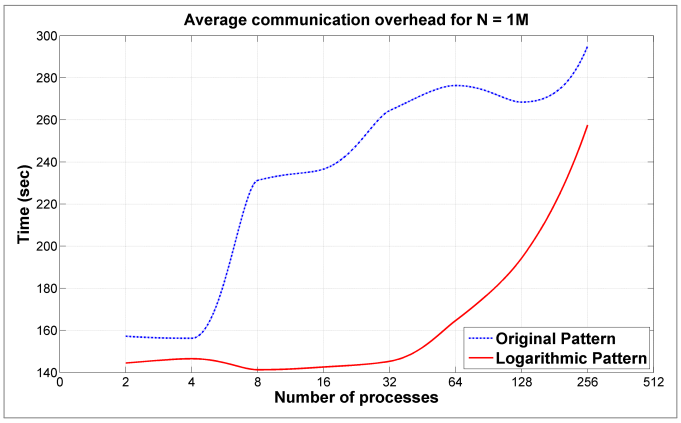

Fig. 7: Average communication overhead for $N = 1M$.

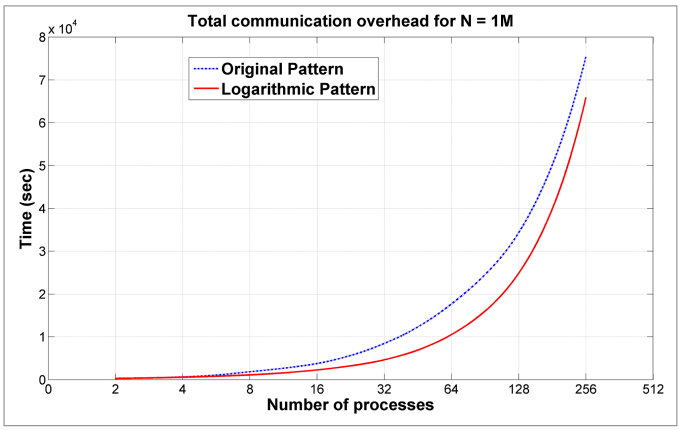

Fig. 8: Total communication overhead for $N = 1M$.

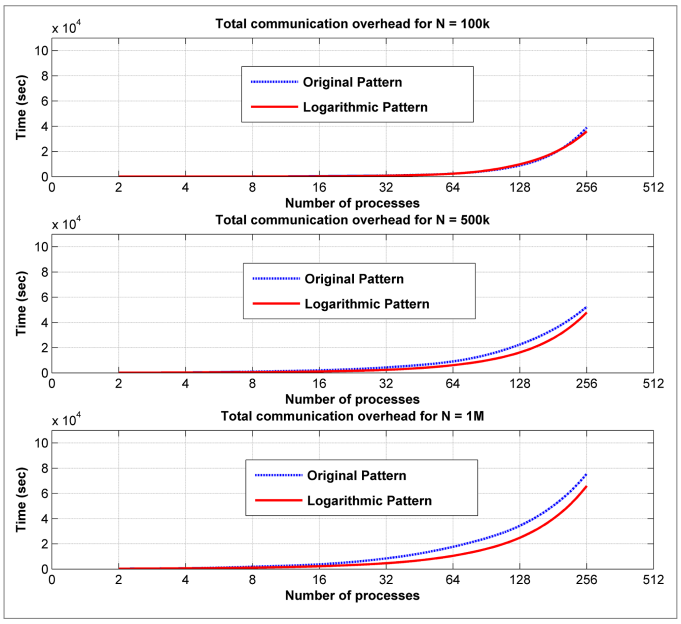

Fig. 9: Total overhead communication for $N = 100K$, $N = 500K$, and $N = 1M$.

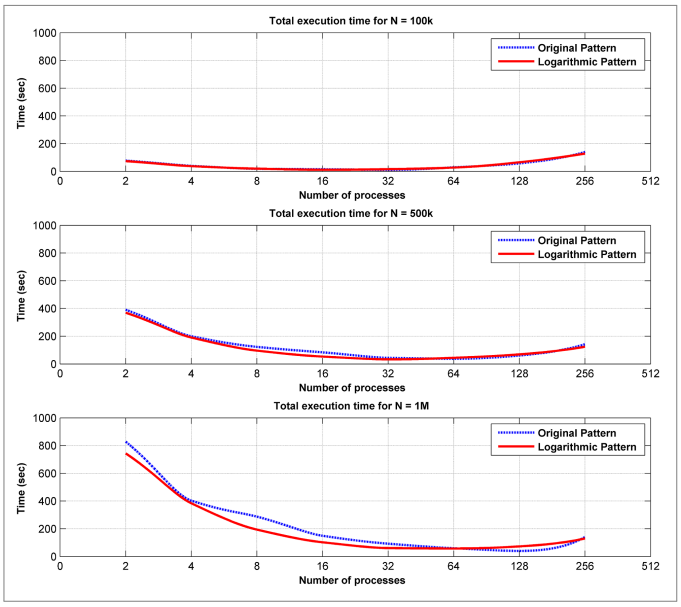

Fig. 10: Total execution time for $N = 100K$, $N = 500K$, and $N = 1M$.

from the Logarithmic algorithm occur for larger matrices, which is precisely what the algorithm is designed for. Likewise, for the smallest input size, the Original merging pattern and the Logarithmic pattern achieved almost equal total execution time (Figure 10).

VI. Conclusion and Future Work

In this paper, we have explored two merging patterns for accumulating the partial results of sparse matrix-matrix multiplication in parallel. A theoretical framework and supporting implementation have been developed for a merging pattern where each node sends and receives half of its data in $log_2(p)$ iterations, resulting in total communication overhead of $O(log_2(p)(z^2\lceil\frac{N}{p}\rceil + \delta))$. Based on the performance on the high-performance computing cluster Knightrider, the data collected for three input sizes (100K, 500K, 1M) shows that the proposed Logarithmic pattern, as predicted, incurs lower communication overhead, which in turn reduces the total execution time.

Several issues related to the algorithms' relative performance still need to be addressed. Currently, the Logarithmic merging algorithm assumes that the number of processors in use is an exact power of 2. This restriction will be removed in a forthcoming version of this algorithm, which will allow it to run on any number of processors. One particular issue of the Logarithmic merging pattern is its failure to yield as great of an improvement over the Original linear merging pattern as anticipated. Our analysis attributes this failure to the overhead incurred by copying data from a processor's hash table into a contiguous package for transmission. Our future study will focus more on the optimization of the data packaging overhead.

Another topic of particular interest is the Logarithmic algorithm's scalability. This can be assessed by running the algorithm at a more powerful facility like Oak Ridge National Lab [26] for larger number of processors. Exploring the performances based on different sizes and implementations of the hash table and varying the sparsity and distribution of non-zero elements in the input matrices can help obtain additional information concerning the scalability and characteristics of the Logarithmic merging algorithm.

References

[1] J. VandeVondele, U. Borstnik, and J. Hutter, "Linear scaling self-consistent field calculations with millions of atoms in the condensed phase," *Journal of chemical theory and computation*, vol. 8, no. 10, pp. 3565–3573, 2012.

[2] W. Briggs, "van e. henson, and s. mccormick," *A Multigrid Tutorial,*, 2000.

[3] J. R. Gilbert, V. B. Shah, and S. Reinhardt, "A unified framework for numerical and combinatorial computing," *Computing in Science & Engineering*, vol. 10, no. 2, pp. 20–25, 2008.

[4] S. Van Dongen, "Graph clustering via a discrete uncoupling process," *SIAM Journal on Matrix Analysis and Applications*, vol. 30, no. 1, pp. 121–141, 2008.

[5] A. Buluç and J. R. Gilbert, "Parallel sparse matrix-matrix multiplication and indexing: Implementation and experiments," *SIAM Journal on Scientific Computing*, vol. 34, no. 4, pp. C170–C191, 2012.

[6] G. Penn, "Efficient transitive closure of sparse matrices over closed semirings," *Theoretical Computer Science*, vol. 354, no. 1, pp. 72–81, 2006.

[7] V. V. Williams, "Multiplying matrices faster than coppersmith-winograd," in *Proceedings of the forty-fourth annual ACM symposium on Theory of computing*. ACM, 2012, pp. 887–898.

[8] A. Buluc and J. R. Gilbert, "Challenges and advances in parallel sparse matrix-matrix multiplication," in *Parallel Processing, 2008. ICPP'08. 37th International Conference on*. IEEE, 2008, pp. 503–510.

[9] G. Ballard, A. Buluc, J. Demmel, L. Grigori, B. Lipshitz, O. Schwartz, and S. Toledo, "Communication optimal parallel multiplication of sparse random matrices," in *Proceedings of the twenty-fifth annual ACM symposium on Parallelism in algorithms and architectures*. ACM, 2013, pp. 222–231.

[10] G. Ballard, J. Demmel, O. Holtz, B. Lipshitz, and O. Schwartz, "Communication-optimal parallel algorithm for strassen's matrix multiplication," in *Proceedings of the twenty-fourth annual ACM symposium on Parallelism in algorithms and architectures*. ACM, 2012, pp. 193–204.

[11] G. Ballard, A. Druinsky, N. Knight, and O. Schwartz, "Brief announcement: Hypergraph partitioning for parallel sparse matrix-matrix multiplication," in *Proceedings of the 27th ACM on Symposium on Parallelism in Algorithms and Architectures*. ACM, 2015, pp. 86–88.

[12] G. Utrera, M. Gil, and X. Martorell, "Evaluating the performance impact of communication imbalance in sparse matrix-vector multiplication," in *Parallel, Distributed and Network-Based Processing (PDP), 2015 23rd Euromicro International Conference on*. IEEE, 2015, pp. 321–328.

[13] J. Berntsen, "Communication efficient matrix multiplication on hypercubes," *Parallel computing*, vol. 12, no. 3, pp. 335–342, 1989.

[14] J. Choi, D. W. Walker, and J. J. Dongarra, "Pumma: Parallel universal matrix multiplication algorithms on distributed memory concurrent computers," *Concurrency: Practice and Experience*, vol. 6, no. 7, pp. 543–570, 1994.

[15] R. C. Agarwal, S. M. Balle, F. G. Gustavson, M. Joshi, and P. Palkar, "A three-dimensional approach to parallel matrix multiplication," *IBM Journal of Research and Development*, vol. 39, no. 5, pp. 575–582, 1995.

[16] Q. Luo and J. B. Drake, "A scalable parallel strassen's matrix multiplication algorithm for distributed-memory computers," in *Proceedings of the 1995 ACM symposium on Applied computing*. ACM, 1995, pp. 221–226.

[17] B. Grayson and R. Van De Geijn, "A high performance parallel strassen implementation," *Parallel Processing Letters*, vol. 6, no. 01, pp. 3–12, 1996.

[18] D. Irony, S. Toledo, and A. Tiskin, "Communication lower bounds for distributed-memory matrix multiplication," *Journal of Parallel and Distributed Computing*, vol. 64, no. 9, pp. 1017–1026, 2004.

[19] E. Solomonik and J. Demmel, "Communication-optimal parallel 2.5 d matrix multiplication and lu factorization algorithms," in *Euro-Par 2011 Parallel Processing*. Springer, 2011, pp. 90–109.

[20] G. Ballard, J. Demmel, O. Holtz, B. Lipshitz, and O. Schwartz, "Strong scaling of matrix multiplication algorithms and memory-independent communication lower bounds," *arXiv preprint arXiv:1202.3177*, 2012.

[21] G. Ballard, J. Demmel, O. Holtz, and O. Schwartz, "Graph expansion and communication costs of fast matrix multiplication," *Journal of the ACM (JACM)*, vol. 59, no. 6, p. 32, 2012.

[22] M. A. Hoque, M. R. K. Raju, C. J. Tymczak, D. Vrinceanu, and K. Chilakamarri, "Parallel sparse matrix-matrix multiplication: a scalable solution with 1d algorithm," *International Journal of Computational Science and Engineering*, vol. 11, no. 4, pp. 391–401, 2015.

[23] M. Challacombe, "A general parallel sparse-blocked matrix multiply for linear scaling scf theory," *Computer physics communications*, vol. 128, no. 1, pp. 93–107, 2000.

[24] C. P. Kruskal, L. Rudolph, and M. Snir, "Techniques for parallel manipulation of sparse matrices," *Theoretical Computer Science*, vol. 64, no. 2, pp. 135–157, 1989.

[25] H. P. C. Center. (2007) East tennessee state university. Retrieved: Feb 20, 2016. [Online]. Available: http://www.etsu.edu/hpcc/

[26] (2007) Oak ridge national laboratory ornl. Retrieved: Feb 20, 2016. [Online]. Available: https://www.ornl.gov/