Parallel sparse matrix-matrix multiplication: A scalable solution with 1D algorithm

Article *in* International Journal of Computational Science and Engineering · January 2015

DOI: 10.1504/IJCSE.2015.073498

5 authors, including:

Mohammad Asadul Hoque
East Tennessee State University
27 PUBLICATIONS **129** CITATIONS
SEE PROFILE

Daniel Vrinceanu
Texas Southern University
120 PUBLICATIONS **966** CITATIONS
SEE PROFILE

Parallel Sparse Matrix-Matrix Multiplication: A Scalable Solution with 1-D Algorithm

Mohammad Hoque[1], Md. Rezaul Raju[2], Christopher Tymczak[3], Daniel Vrinceanu[4], Kiran Chilakamarri[5]
Department of Computing, East Tennessee State University[1]
Center for Research on Complex Networks, Texas Southern University[2,3,4,5]
hoquem@etsu.edu[1]; m.raju7957@student.tsu.edu[2]; {tymczakcj[3], vrinceanud[4], chilakamarrikb[5]}@tsu.edu

***Abstract*— This paper presents a novel implementation of parallel sparse matrix-matrix multiplication using distributed memory systems on heterogeneous hardware architecture. The proposed algorithm is expected to be linearly scalable up to several thousands of processors for matrices with dimensions over 10^6 (million). Our approach of parallelism is based on 1D decomposition and can work for both structured and unstructured sparse matrices. The storage mechanism is based on distributed hash lists, which reduces the latency for accessing and modifying an element of the product matrix, while reducing the overall merging time of the partial results computed by the processors. Theoretically, the time and space complexity of our algorithm is linearly proportionate to the total number of non-zero elements in the product matrix C. The results of the performance evaluation show that the algorithm scales much better for sparse matrices with bigger dimensions. The speedup achieved using our algorithm is much better that other existing 1D algorithm. We have been able to achieve about 500 times speedup with only 672 processors. We also identified the impact of hardware architecture on scalability. (*Abstract*)**

Keywords—MPI, Scalable, Sparse Matrix, Parallel Algorithm, Distributed Computing. (*key words****)***

I. INTRODUCTION

Numerical solutions of many critical problems reduce to various forms of matrix operations, in part or in full. Matrix-matrix multiplication has been deemed as the fundamental building block for solving many problems in almost every area of science and engineering. Many of these problems involve large eigenvalue problems or extremely sparse systems of linear equations. Hence, determining the product of two sparse matrices is a basic problem in combinatorial and scientific computing.

Applications of sparse matrix-matrix multiplication are scattered in various dimensions of research like vehicular ad hoc networks, multi-grid methods, clustering algorithms, delay tolerant networks, quantum mechanics, context-free languages and computational fluid dynamics. The parallel and distributed computing paradigm has minimized the bottleneck of large-scale data manipulation and computing needs. Therefore, the computational scientists heavily rely on the efficiency of parallel algorithms on linear algebra operations. Existing libraries for parallel dense matrix multiplication are proven to perform close to optimal efficiency. However, parallel sparse matrix-matrix multiplication algorithms are still considered as a research problem for both distributed and shared memory environment. It has been shown that current parallel algorithms for multiplying sparse matrices does not scale well for higher number of processors even though they perform reasonably well for limited number of processors [1,2]. In most cases, the scaling is deteriorated by the increased process-to-process communication costs. On the other hand the parallel execution time is also dependent on the implementation of data structures and storage mechanism. In addition, heterogeneous hardware architecture is a major concern for load balancing. Though heterogeneous systems i.e., general purpose x86 CPUs coupled with GPUs are gaining interests in the high performance community for accelerating some specific workload, incorporation of heterogeneous systems drive to additional difficulties in terms of programming interfaces, data distribution and communication among the heterogeneous processing elements [9]. Further, the complexity of sparse algorithms is highly dependent on the sparsity and distribution pattern of nonzero elements of the input matrices [2,3]. Hence, due to all these factors, achieving scalability for parallel sparse matrix multiplication algorithms is a very challenging problem.

In this paper we describe a novel implementation of parallel sparse matrix-matrix multiplication that makes use of distributed hash lists ensuring an efficient storage mechanism. This implementation is tested on the Linux clusters of the Texas Southern University HPC Center (TSU-HPCC) [8] using inputs of randomly generated but structured square matrices up to a maximum dimension of 10^6 (1 Million). Our algorithm has shown a trend for linear scaling up to the maximum limit of our homogenous computing resources, which is a total of 672 cores of processor. The maximum speedup attained using our resources was about 500 which is so far the best performance for any sparse matrix multiplication algorithm given the maximum number of cores. Also the average MPI communication overhead per process decreases with the increase of processors, which is the underlying factor for achieving the scalability with our algorithm. The scalability tends to be better for larger matrices. Additionally, the algorithm can perform equally for both structured and non-structured matrices.

This research if funded by the NSF CREST project. (Grant No. 1137732)

The following sections are organized as follows: section II provides an overview of the problem definition, notations and terminology used in this paper; section III describes the previous research within this and related fields; section IV describes implementation details and the algorithm, followed by the performance evaluation in section V. Finally we conclude in section VI indicating our future extensions of this algorithm and open issues for further investigation.

II. Problem Definition, Terminology and Notations

We consider quite similar terminology adopted by Buluc *et. al.* [4]. Our goal is to compute $\boldsymbol{C = AB}$, where $\boldsymbol{A}$ and $\boldsymbol{B}$ are the two input sparse matrices. The dimension of $\boldsymbol{A}$ and $\boldsymbol{B}$ are $M \times K$ and $K \times N$ respectively. For simplicity, we will be using square matrices (M = K = N) for analysis purpose, but the proposed algorithms can easily operate on rectangular matrices. Let,

p	= total number of processors
γ	= cost of one floating-point operation (nanoseconds)
α	= cost of insertion into a vector
β	= inverse bandwidth (nanoseconds per word)
δ	= cost of insertion into hash list
$nnz(A)$	= number of nonzero elements in $\boldsymbol{A}$
$A(:, i)$	= i^{th} column of $\boldsymbol{A}$
$A(i, :)$	= i^{th} row of $\boldsymbol{A}$
$A(i, j)$	= the element at the (i, j)th position of $\boldsymbol{A}$
$flops$	= total number of operations to compute $\boldsymbol{AB}$.
c	= avg. number of nonzero elements per row/column
N_i	= Starting column/row index for process p_i

Our algorithm assumes that the sparsities/densities of the input matrices known a priori. To make the analysis simple, we assume that both the input matrices $\boldsymbol{A}$ and $\boldsymbol{B}$ have equal sparsity. Hence, $nnz(A) = nnz(B) = cN$. For our input matrices, we considered c=75 for square matrices with a dimension of N=10^6 (1 Million) and c=152 for N=10^5.

Our matrix multiplication algorithm is based on the Outer Product calculation approach. Fig.1 shows a simple example of distributed matrix multiplication using outer product with 4 parallel processors.

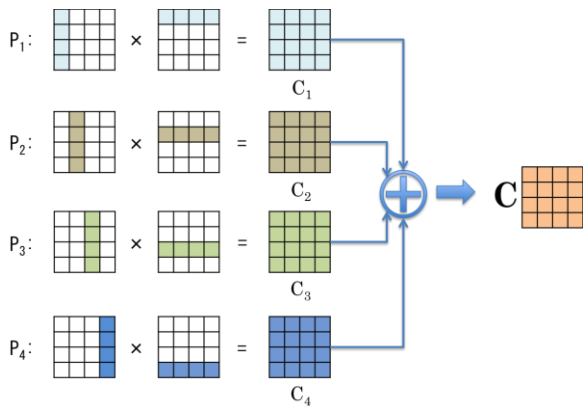

Fig. 1: Distributed multiplication using Outer Product

As shown in the figure, each of the processors P_i is assigned with block of columns, N_i to $N_{i+1} - 1$ and compute C_i in parallel, where,

$$C_i = \sum_{k=N_i}^{N_{i+1}-1} A(:,k) \times B(k,:) \quad (1)$$

After calculating the results of individual portion, the final result is merged by combining all the partial results from the parallel processors. Hence, the final product matrix,

$$C = \sum_{i=1}^{p} C_i = \sum_{i=1}^{p} \sum_{k=N_i}^{N_{i+1}-1} A(:,k) \times B(k,:) \quad (2)$$

III. Related Work

The Generalized Sparse Matrix-Matrix Multiplication problem (SpGEMM) has been extensively investigated by Gustavson *et al.* [5] which described the first classical serial algorithm for SpGEMM in 1978. This algorithm stores the elements of the matrices in Compressed Sparse Column (CSC) format and has a complexity of $O(flops + nnz + n)$, which is proven to be optimal for $flops \ max\{nnz,n\}$. Later, it was used by many mathematical software and libraries like MATLAB [6], CSparse [7] etc. The lower bound of time complexity for any sequential SpGEMM is which is met by Gustavson's algorithm [1]. Hence we consider the sequential work (W) to be equal to for our speedup analysis.

On the other hand, the parallel version of SpGEMM (known as PSpGEMM) has not been developed to the efficiency of its dense matrix counterpart. While there have been significant progress in development of parallel dense linear algebra libraries like PBLAS [18], ScaLAPACK [20] etc. within the last couple of decades, but the sparse libraries haven't reached up to that level of enrichment mostly due to the lack of scalability. Most of the current sparse algorithms are implemented on a distributed memory environment using MPI communication. As of today, to the best of our knowledge, there hasn't been any PSpGEMM algorithm implemented on hybrid architecture that combines the benefits of both distributed as well as shared memory environment using MPI and OpenMP in concert. Other variants of programming tools in shared memory architecture like ccNUMA, PGAS are not yet considered for sparse matrix-matrix multiplication. However, several hybrid parallel algorithms have been developed for sparse matrix-vector multiplication [15,17] that is basically a special case of sparse matrix-matrix multiplication and has fewer constraints. In these cases, the algorithm needs to focus primarily only on the single sparse matrix on the left while the vector is usually dense and the multiplication operation generates a single vector as output making the storage management with the data structures greatly simplified.

Siegel *et al.* [9] described a co-design approach for implementing PSpGEMM efficiently on a heterogeneous cluster using both multicore CPUs and GPUs. Their algorithm was based on the original Gustavson's algorithm [5]. Their work dealt with dynamically optimizing the load balancing for GPUs. Unfortunately, they have also shown that no speedups can be achieved with their implementation for hyper-sparse matrices where the density of *nonzeros* are less than 0.01%.

In the domain of distributed memory environment, Buluc *et al.* [1,2,3,4] has contributed significantly for examining the scalability of different implementations of PSpGEMM with 1D, 2D and 3D decomposition. They are the first to develop any 2D or 3D algorithm for PSpGEMM, while as of today, no other algorithm has been implemented using 2D or 3D decomposition. All of their algorithms are based on inner product and use a novel data-structure called Doubly Compressed Sparse Column (DCSC), which ensures a storage complexity of $O(nnz)$. Unfortunately, their 1D algorithms are not scalable beyond a certain number of processors, while 2D algorithms are [1,3]. The main reason for 1D algorithms not being scalable is the increased MPI communication overhead [2]. One of the problems with their 1D algorithm is that, it requires each of the parallel processors to broadcast an entire row of B matrix during every iteration. This causes a storm of communication that eventually makes the algorithm un-scalable. However, their 2D algorithms (Sparse SUMMA and Sparse CANNON) do show a better scaling performance in terms of communication overhead and overall execution time. In contrast to their 1D approach, our algorithm uses an outer product method where each processor conducts point-to-point communication with others for merging the intermediate results. We will now describe the computational resources for which we implemented our novel algorithm.

IV. Algorithm Description

A. Data-Structure and Storage Mechanism

1) Distributed Hashlist

We use distributed hash lists for storing the matrices. Open hashing technique is considered with an initial table size varying from N to 10N. The hash key is composed of the row and column indexes of the non-zero elements of the matrices. Each element is stored as a *Node*-type data structure containing <row, column, value> tuple. The reason for not choosing closed hashing or open addressing type of hash list is to avoid frequent resizing of the product matrix. Even though the sparsity of the input matrices are known a priori, the product matrix belonging to each processor may have a varying number of non-zero elements ranging from N/p to c^2N/p. Each incidence of table resizing leads to redistribution of the elements of the entire list along the modified address space. This is a costly operation having $O(hashsize)$ complexity which can occur more than once for each processor. That is the reason we considered open hashing to eliminate the cost of frequent resize.

2) Auxiliary Storage

MPI communication does not support non-sequential memory data transfer. As open hashing mechanism involves separate chains to resolve collisions, we had to define auxiliary storage for MPI communication. Once every processor completes the computation of individual portion of the result, it then transfers the non-contiguous hash-list of C matrix into contiguous chunks of memory. For that we utilize the benefits of 'vector' type data structures that implicitly support dynamic contiguous memory allocation. The total hash-list is divided into different queues, Q_j based on the column index. Each processor needs to perform this step only once after finishing the computation. Hence, the total storage required for storing and exchanging the C matrix by every processor is $(2c^2N)/p$ that meets the upper bound of optimal storage requirement $(\frac{c^2N}{p})$.

B. Details Steps of Algorithm

The input matrices A and B are initially stored as compressed sparse columns (CSC) and compressed sparse rows (CSR) respectively. These data structures are implemented as arrays of vectors. Each processor has $\lceil N/p \rceil$ vectors from both A and B that corresponds to $\lceil N/p \rceil$ columns of A and $\lceil N/p \rceil$ rows of B. The entire multiplication algorithm has three major steps to compute the final product matrix *C*, where the result matrix *C* is left distributed across the processors.

1) Partial Result Calculation

Let N_i be the starting column index for the block of columns handled by processor P_i . During this step each of the processors, P_i iterates over *k* for a total of $\lceil N/p \rceil$ iterations and computes a Cartesian product of the column vector $A(:,k)$ with the row vector$B(k,:)$, generating a sub-matrix ofC_i. Each processor contains the chunk of data comprising of $\lceil N/p \rceil$ columns of *A* and $\lceil N/p \rceil$ rows of *B*. If there are c_1 *non-zero* elements in $A(:,k)$ and c_2 *non-zero* elements in $B(k,;)$ then the number of elements added to the C_i sub-matrix will bec_1c_2. For simplicity of analysis, we assume $c_1 = c_2 = c$. As this operation is done for $\lceil N/p \rceil$ times, the total number of floating point operations in this step will be $\lceil N/p \rceil c_1c_2$, which yields a time-complexity of $O\left(c^2N/p\right)$.

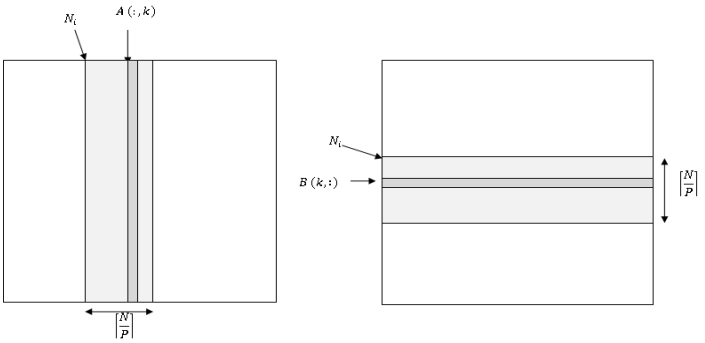

Fig. 2: Multiplication handled by P_i

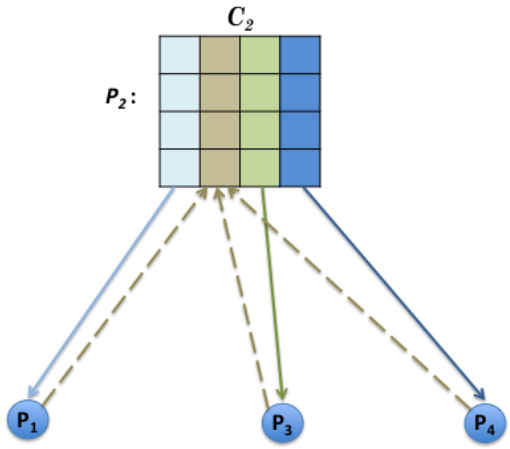

Fig. 3: Inter-process communication pattern (p = 4)

Algorithm

for each processor P_i do in parallel
{
    Calculate_Partial_Result (P_i);
    Exchange_Result (P_i);
    Merge_Result (P_i);
}

Calculate_Partial_Result (P_i)
{
    for $(k = N_i;\ k \le N_i + \lceil N/p \rceil; k++)$
        for each non-zero element $A(x,k)$ in column $A(:,k)$
            for each non-zero element $B(k,y)$ in row $B(k,:)$
                $C_i(x,y) = C_i(x,y) + A(x,k) \times B(k,y)$;
}

Exchange_Result (P_i)
{
    $N\_per\_proc = \lceil N/p \rceil$;
    for each node T in Hash List C_i
        $j = \lfloor \frac{T.column}{N\_per\_proc} \rfloor$;
        if $(j \neq i)$
            Insert T into Q_j
    for each processor $P_j \neq P_i$
        Exchange Q_j with P_j
        $C_i^j \leftarrow Q_j$
}

Merge_Result (P_i)
{
    for each processor $P_j \neq P_i$
        for each element $(x,y) \in C_i^j$
            $C_i(x,y) = C_i(x,y) + C_i^j(x,y)$
}

Fig. 4: Proposed Algorithm

2) Exchange of Partail Results

In this step every processor distributes a portion of the local result C_i to all other processors. This is demonstrated in Fig. 3 where one of the p processors, P_2 sends the partial results stored in hash list C_2 to P_1, P_3 and P_4. The total size of hash list C_2 is equal to $c^2\left\lceil \frac{N}{p} \right\rceil$ which is further distributed into p column-wise blocks. Every processor P_i only accumulates the block of columns in C_i that includes columns indexing from N_i to $\left(N_i + \left\lceil \frac{N}{p} \right\rceil\right)$. The total amount of data sent by each processor is equal to $(p-1) \times \frac{c^2\lceil \frac{N}{p} \rceil}{p}$ which is $O(\frac{c^2 N}{p})$. Hence, the complexity of this step is also equal to $O\left(c^2 N/p\right)$.

As mentioned previously, the exchange of results imposes some extra overhead of queuing the hash list into $(p-1)$ vectors of consecutive memory data structure. While accessing each element of C_i, the column index is divided by $\left\lceil \frac{N}{p} \right\rceil$ to get the appropriate index j of the sub-hash list C_i^j that is inserted into the vector Q_j. This Q_j vector is communicated over MPI and sent to P_j.

3) Merging results

Once each processor P_i receives the partial result (C_i^j) from each of the other processors (P_j), it merges with its own hash list of C_i. Each partial result (C_i^j) contains $\frac{c^2\lceil \frac{N}{p} \rceil}{p}$ elements on an average. Hence, for *p-1* processors, the total merging from this step also requires a total of $O\left(c^2 N/p\right)$ operations. The result matrix is kept distributed over *p* processes. Fig. 4 describes the complete step-by-step description of the algorithm.

V. Performance Evaluation

We have evaluated the performance of our algorithms on TSU-HPCC clusters. Our High Performance Computing center has medium scale computational resources, which includes two Linux clusters: ***Ares*** and ***Hades.*** Fig. 5 below shows the topology of the two clusters and Fig. 6 shows the actual racks where the cluster is hosted.

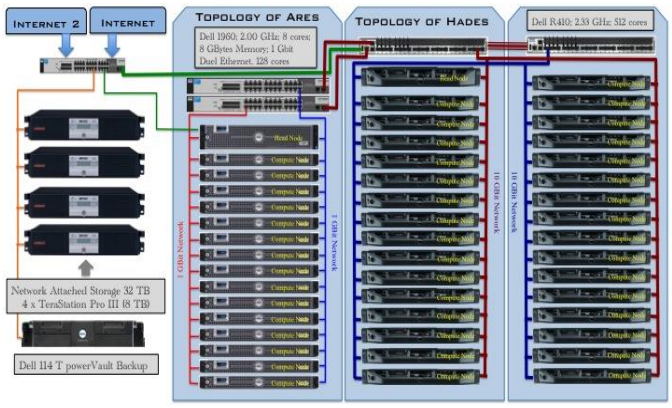

Fig. 5 Topology of ARES and HADES

Fig. 6 TSU HPCC clusters

Ares has 16 dual-slot quad-core nodes with Intel Xenon 5350 2.0 GHz processors with 8 Gigabytes of memory per node. ***Hades*** has i) 8 dual slot hyper threaded quad core nodes with the Intel E5520 2.33 GHz Xeon processor with 12 gigabytes of memory; ii) 28 dual slot hyper threaded 6 core nodes with the Intel EE5645 2.40GHz Xeon Processor with 24 gigabytes of memory.

The nodes are connected via two different switches, one is a standard 1Gb Ethernet switch and the other through 10 Gigabit Ethernet using an ultra low latency Force10 switch. The full parallel cluster has a total of 944 virtual cores and a total memory of 996 Gigabytes, with a theoretical peak speed of 5.0 Teraflops. Due to the heterogeneity of the nodes and processors we mainly used the homogenous nodes for scalability evaluation, which comprised of the 28 EE5645 nodes of HADES, having a total of 672 cores. For some of the cases we have also included the 8 E5520 nodes that made a total of 800 processors to be used in our evaluation.

A. Scalability

Our algorithm shows linear scalability up to the maximum number of homogeneous processors available in our HPCC. In our experiment, the speedup is calculated based on multiprocessor execution time compared with single processor execution. Figure 7 shows the scaling performances for matrix dimension of 1M x 1M. The left vertical axes are in logarithmic scale. It is evident from the results that for larger matrices the algorithm performs much better, as it can reach up to ~500 times speedup with 672 processors for 1M matrix. The speedup also increases linearly with the increase of number of processors. The trend shown by the results give us an impression that the algorithm might prove to be linearly scalable up to at least few thousand processors. Fig. 8 shows a comparison of speedup for different matrix sizes. While the 100K matrix does scale linearly, the speedup, however, is much less compared to the 1M matrix. The main reason of better efficiency for larger matrices can be understood from the percentage of total execution time spent in communication, which is elaborated in the next subsection.

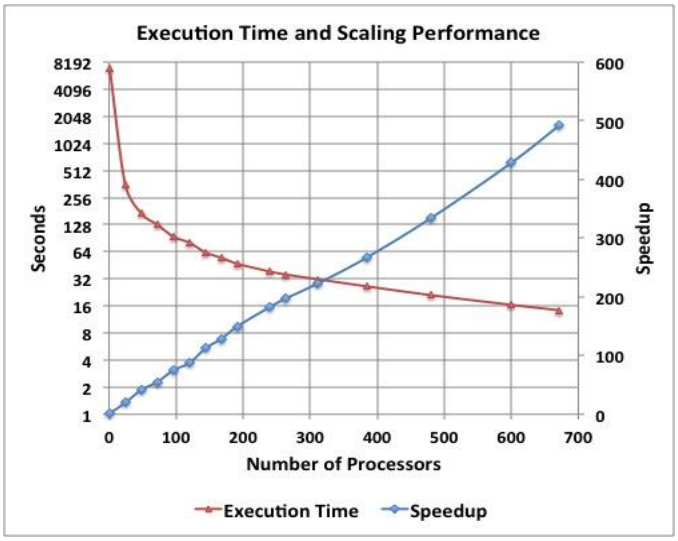

Fig. 7 Scalability of 1M Matrix

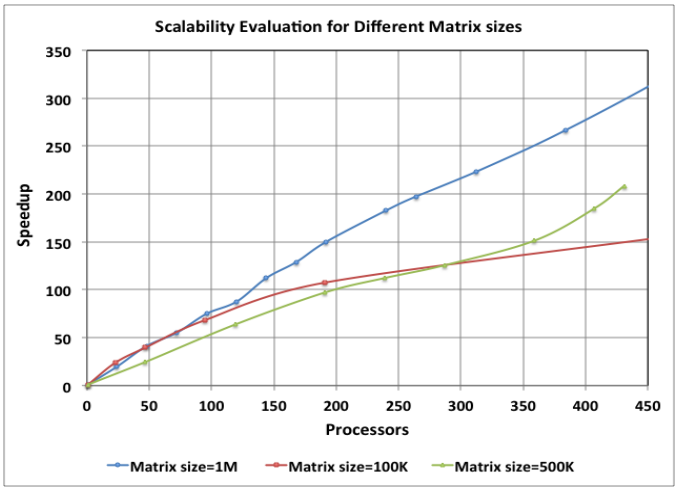

Fig. 8 Comparison of Speedup for different Matrix Size

B. Communication Overhead

We evaluated the MPI communication overheads for all the three different matrix sizes (1M, 500K, 100K). Fig. 9 shows the percentage of communication overhead for different matrix sizes. As single process yields serial execution, there is no communication overhead in this case. As applicable for any MPI based distributed 1D algorithm, the percentage of MPI communication overhead increases with the increment of number of processors. A closer look into Fig. 9 reveals that the percentage of communication time is less for bigger matrices. As for example, for 432 processors, the percentage of overhead is 50% for 1M matrix whereas it accounts for 70% in case of 100K. This communication overhead could be significantly reduced if we had InfiniBand network for inter-connection of nodes. We also measured the statistics of average communication delay per process for different matrix sizes (Fig. 10). It is evident that the increase of processors implies to a decrease of average communication delay, which is the reason why our algorithm scales better than other existing 1D algorithms.

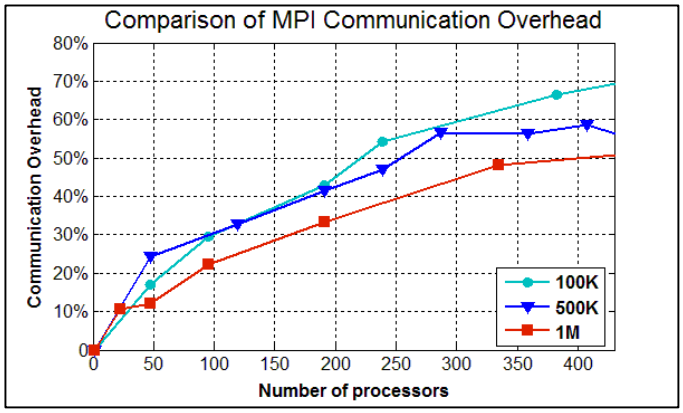

Fig. 9 Comparison of Communication Overhead

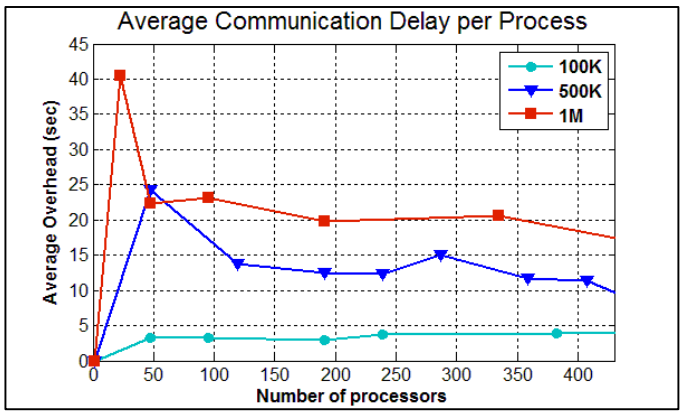

Fig. 10 Average Communication Delay

C. Computation vs. Communication Time

Fig 11 shows the ratio of computation and communication time for various matrix dimensions. It is observed that the portion of computation time reduces significantly with the increase of number of processors, while the total computation time does not reduce in that respect. Initially the portion of computation is much higher that that of computation. As the number of processor increase, computational cost tends to decrease fast. After a certain time, communication overhead dominates the computation time, which makes the algorithm scalable up to a certain number of processors beyond which the efficiency does not improve with respect to processors increase. In our case, this scalability might be bounded by a few thousand processors for larger matrices.

D. Impact of Hardware Architecture

The results with our algorithm displayed here are mostly within homogeneous computing system architecture. Our cluster topology routes over 2 different speed Ethernet switches (1Gb and 10Gb) which impacts on communication latency and we strongly anticipate that this proposed algorithm will be further up-scaled in case of ultra low latency switches such as InfiniBand. Moreover, we experienced inconsistent results in execution time, speed up and communication latency while running on heterogeneous computational resources i.e., on differing processing speed, on node or off node process etc. Our algorithm performance on heterogeneous systems significantly reflects the impact of internode or intra-node memory bandwidth. However, ratio of communication and computational time is significant and our scalability does clearly improve with same number of process on larger matrix dimension, as matrix with dimension of 1M performs better than that of 100K and 500K.

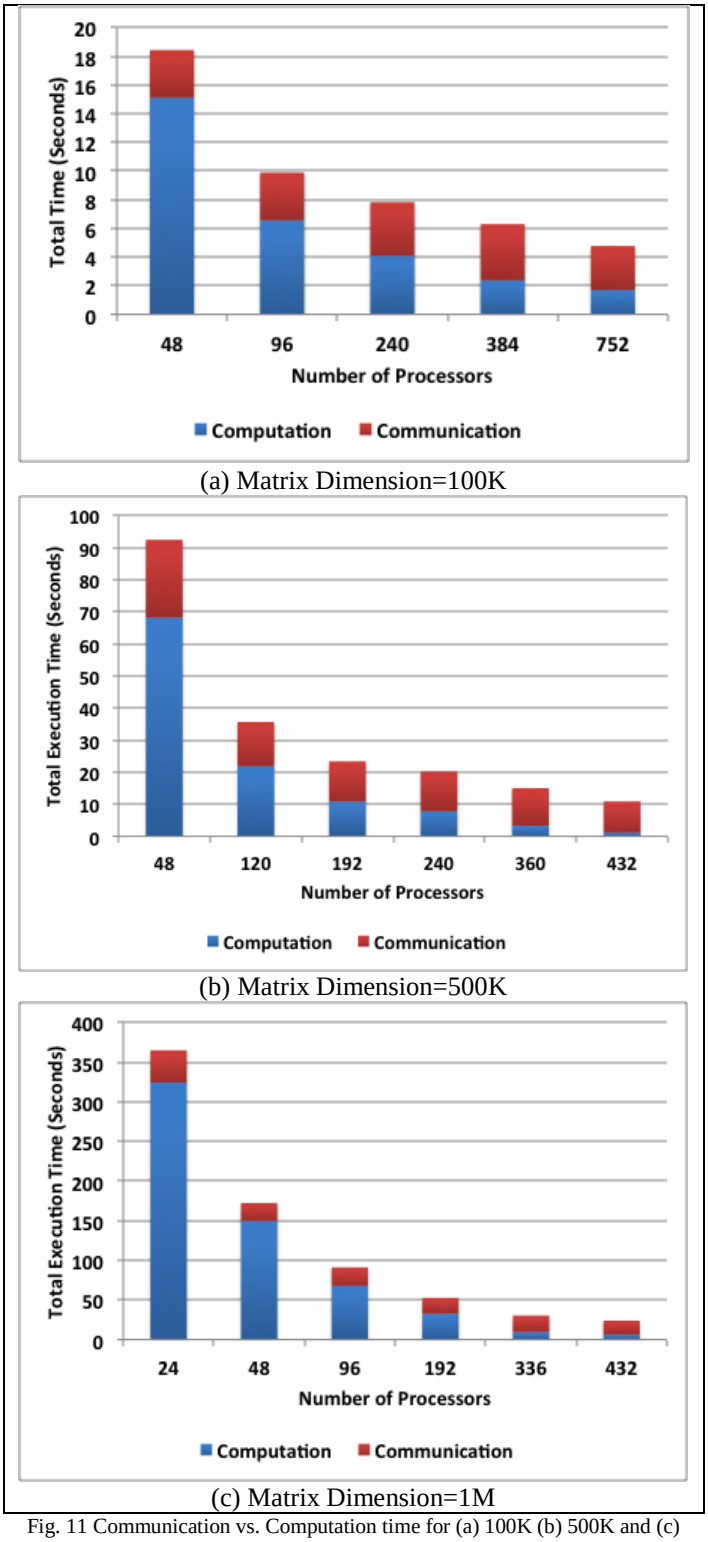

Fig. 11 Communication vs. Computation time for (a) 100K (b) 500K and (c) 1M Matrix

VI. CONCLUSION

In this paper we present novel 1D algorithm for parallel sparse matrix-matrix multiplication (PSpGEMM) using a distributed hash-list. Our algorithm scales linearly up to the maximum limit of our available computational resources. This algorithm performs equally for both structured and un-structured sparse matrices with a regular distribution of non-zero elements over the rows and columns. The speedup achieved using our algorithm is so far better that any other existing 1D algorithms. The computational delay can be further reduced with efficient hash function that can distribute the elements uniformly over the hash table. Also, the communication latency will be significantly reduced in InfiniBand network resulting higher efficiency and scalability. Since our final result matrix is stored distributed over the processes, we can iteratively multiply the result to compute chain multiplication of an input matrix. We are currently extending our algorithm to utilize the benefits of hybrid parallel environment with OpenMP.

REFERENCES

1. Buluc, A., & Gilbert, J. R. (2008). Challenges and advances in parallel sparse matrix-matrix multiplication. Parallel Processing, 2008. ICPP'08. 37th International Conference on, 503-510.
2. G. Ballard, A. Buluc, J. Demmel, L. Grigori, B. Lipshitz, O. Schwartz and S. Toledo, Communication Optimal Parallel Multiplication of Sparse Random Matrices, Proceedings of ACM Symposium on Parallelism in Algorithms and Architectures, SPAA 2013.
3. Buluc, A., & Gilbert, J. R. (2012). Parallel sparse matrix-matrix multiplication and indexing: Implementation and experiments. SIAM Journal on Scientific Computing, 34(4), C170-C191.
4. Buluç, A., & Gilbert, J. R. (2010). Highly parallel sparse matrix-matrix multiplication. UCSB Technical Report, June 2010.
5. Fred G. Gustavson. Two fast algorithms for sparse matrices: Multiplication and permuted transposition. ACM Transactions on Mathematical Software, 4(3):250{269, 1978.
6. John R. Gilbert, Cleve Moler, and Robert Schreiber. Sparse matrices in Matlab: Design and implementation. SIAM Journal of Matrix Analysis and Applications, 13(1):333{356, 1992.
7. Timothy A. Davis. Direct Methods for Sparse Linear Systems. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2006.
8. High Performance Computing Center at Texas Southern University, http://hpcc.tsu.edu
9. Siegel, J.; Villa, O.; Krishnamoorthy, S.; Tumeo, A.; Xiaoming Li, "Efficient sparse matrix-matrix multiplication on heterogeneous high performance systems," IEEE International Conference on Cluster Computing Workshops and Posters, vol., no., pp.1,8, 20-24 Sept. 2010
10. Choi, J. (1997). A new parallel matrix multiplication algorithm on distributed-memory concurrent computers. High Performance Computing on the Information Superhighway, 1997. HPC Asia'97, 224-229.
11. Chorley, M. J., & Walker, D. W. (2010). Performance analysis of a hybrid MPI/OpenMP application on multi-core clusters. Journal of Computational Science, 1(3), 168-174.
12. Elmroth, E., Gustavson, F., Jonsson, I., & Kågström, B. (2004). Recursive blocked algorithms and hybrid data structures for dense matrix library software. SIAM Review, 46(1), 3-45.
13. Kamal, H., & Wagner, A. (2012). Added concurrency to improve MPI performance on multicore. Parallel Processing (ICPP), 2012 41st International Conference on, 229-238.
14. Rabenseifner, R., Hager, G., & Jost, G. (2009). Hybrid MPI/OpenMP parallel programming on clusters of multi-core SMP nodes. Parallel, Distributed and Network-Based Processing, 2009 17th Euromicro International Conference on, 427-436.
15. Schubert, G., Fehske, H., Hager, G., & Wellein, G. (2011). Hybrid-parallel sparse matrix-vector multiplication with explicit communication overlap on current multicore-based systems. Parallel Processing Letters, 21(03), 339-358.
16. Yuster, R., & Zwick, U. (2005). Fast sparse matrix multiplication. ACM Transactions on Algorithms (TALG), 1(1), 2-13.
17. ZIAVRAS, S. G., & MANIKOPOULOS, C. N. Matrix multiplication on an experimental parallel system with hybrid architecture, in proceedings of 4th World CSCC 2000, Athens, Greece.
18. Almadena Chtchelkanova, John Gunnels, Greg Morrow, James Overfelt, and Robert A. van de Geijn. Parallel implementation of BLAS: General techniques for Level 3 BLAS. Concurrency: Practice and Experience, 9(9):837-857, 1997.
19. Edith Cohen. Structure prediction and computation of sparse matrix products. Journal of Combinatorial Optimization, 2(4):307-332, 1998.
20. ScaLAPACK, http://www.netlib.org/scalapack/