Available online at www.prace-ri.eu

Partnership for Advanced Computing in Europe

MapReduce-based Parallelization of Sparse Matrix Kernels for Large-scale Scientific Applications

Gunduz Vehbi Demirci[a], Ata Turk[a], R. Oguz Selvitopi[a], Kadir Akbudak[a],

Cevdet Aykanat[a*]

[a]*Bilkent University, Computer Engineering Department, 06800 Ankara, TURKEY*

Abstract

This whitepaper addresses applicability of the MapReduce paradigm for scientific computing by realizing it on the widely used sparse matrix-vector multiplication (SpMV) operation with a recent library developed for this purpose. Scaling SpMV operations proves vital as it is a kernel that finds its applications in many scientific problems from different domains. Generally, the scalability improvement of these operations is negatively affected by high communication requirements of the multiplication, especially at large processor counts in the case of strong scaling. We propose two partitioning-based methods to reduce these requirements and allow SpMV operations to be performed more efficiently. We demonstrate how to parallelize SpMV operations using MR-MPI, an efficient and portable library that aims at enabling usage of MapReduce paradigm in scientific computing. We test our methods extensively with different matrices. The obtained results show that utilization of communication-efficient methods and constructs are required on the road to Exascale.

1. Introduction

MapReduce [1] is a programming model that enables processing of large data in a parallel and distributed fashion on clusters that generally consist of commodity machines. The two basic steps of this model are the map, and the reduce phases. In the map phase, the data is distributed among worker nodes where each worker node contains a user-defined function that takes an input key/value pair (k_1, v_1) and generates a new list of intermediate key/value pairs, $\text{list}(k_2, v_2)$. The map phase is then followed by a reduce phase, which again contains a user-defined function that takes the intermediate key/value pair(s) $\left(k_2, \text{list}(\text{v}_2)\right)$ generated by the map phase as input. The input of the reduce phase may contain multiple values for the same key, which are reduced by the reducers to obtain a list values $\text{list}(\text{v}_2)$. In MapReduce programming model, these two computational phases operate on local data and there exists an intermediate data shuffling operation which require inter-processor communication. The MapReduce paradigm has proved its success by being realized and efficiently utilized on many large-scale projects [2] [3] [4] [5].

Recently, the scientific community tried to exploit the benefits of the MapReduce programming model [6] [7] [8] [9] [10] [11]. Most notably in [10], the authors developed a lightweight C++ library called MR-MPI that uses MPI primitives for inter-processor communication and provides basic functionality of the MapReduce programming paradigm. By building a MapReduce model on top of MPI, MR-MPI enables users to utilize native MPI functions, departing from conventional MapReduce implementations. MR-MPI supports different modes of

* Corresponding author. *E-mail address:* aykanat@cs.bilkent.edu.tr

operations (in-core and out-of-core) which allow processors to utilize disk in cases where data is too big to fit in memory of processors as most of the MapReduce implementations. It is small, portable and flexible; depending only a few well-known and widely used libraries, we did have no problems porting our code from one machine to another. On the downside, however, it does not provide any fault tolerance or data redundancy, which are actually not of prime concern in scientific computing community. In this project, we utilize MR-MPI library to develop a communication-efficient sparse matrix-vector multiplication (SpMV) kernel.

MR-MPI improves the performance of the MapReduce paradigm by introducing new data types and efficient communication routines that are centered around using MPI. In addition to basic key/value (KV) pairs, MR-MPI provides a new data type called key/multivalue (KMV) pair where the values related to single key are stored contiguously in a native C++ data type. Unlike a conventional MapReduce program, an MR-MPI program must make at least three function calls instead of two. The first of these is the *map* function where each processor generates KV pairs and stores them in its memory. The second is the *collate* function that corresponds to data shuffling operation. Collate is a two-stage operation in which at first the unique keys are identified and collected to form KMV pairs at each processor. This stage requires communication since any key may be stored by any processor at the end of the map operation. Typically, a hash function is used to determine the distribution of keys. MR-MPI allows users to integrate their own hash functions, which is a key point utilized in our work to reduce communication requirements of SpMV kernel. The second stage of the collate operation is a computational phase that consists of processors forming KMV pairs from the received KV pairs in the first stage. The communication in the collate phase can be carried out using either MPI_AlltoAll collective or point-to-point MPI_Send/MPI_IRecv constructs. The choice the selected communication type may affect the performance of the application. For example, in our case for SpMV, since processors generally communicate with a few number of neighboring processors, using the latter one shall avoid unnecessary performance losses due to communication. The final function that should be invoked by all processors is the *reduce* function. This is similar to the conventional MapReduce paradigm where each processor forms KV pairs using KMVs they own with a provided reducer operation. Among the mentioned phases, *map* and *reduce* do not necessitate any communication, whereas *collate* requires communication to accumulate KV pairs at corresponding processors.

In addition to these basic operations, MR-MPI provides several other operations that perform various tasks. Since these are central to our work, we review them here briefly:

- *map*: Generates KV pairs by calling a user program. This is serial and requires no communication between processors. When called with appropriate arguments, it can break large files into chunks and process them chunk-by-chunk. MR-MPI controls the mapping of tasks to processors. In default settings, each processor is assigned equal number of mappers.

- *aggregate*: Aggregates pairs onto processors by reorganizing KV objects into new KV objects. Before the aggregate operation, duplicates of a key may be stored across multiple processors. At the end of the aggregate operation, the duplicates of a single key are gathered at the corresponding processor, where the ownership of a key is typically determined by a hash function. This operation takes a user-defined hash function as input and the used hash function is of prime importance to obtain a good load balance between computational tasks. Requires communication for the reorganization of keys.

- *add:* Adds KV pairs from one object to another. Requires no communication and is performed serially. This is a typical util function that comes in handy when concatenating two already created MapReduce objects.

- *convert:* Converts KV pairs into KMV pairs. Prior calling the convert function, the KV pairs may contain duplicate keys and their related values. After calling convert, the values of the same key are concatenated to be a single KMV pair. MR-MPI uses a hash function to be able to find and concatenate values corresponding to duplicate keys. This operation does not require communication.

- *collate:* Aggregates KV pairs across processors and then converts them into KMV pairs. This operation requires communication and is actually equivalent to an aggregate operation followed by a convert operation.

- *reduce:* Calls back the user program to process KMV pairs. This operation requires no communication and processes a KMV pair to generate KV pair. The KMVs owned by processors are guaranteed to be unique. At the end of the reduce operation, each processor will own a list of unique KV pairs.

- *clone:* This is another util function that makes a KMV pair out of each KV pair provided. This comes in handy to pass an object with KV pairs to reduce function since its input should be KMV pairs. This operation does not handle duplicate KV pairs by concatenating them into a single KMV pair. Requires no communication.

- *kv_stats, kmv_stats:* Useful to obtain information about objects that store KV or KMV pairs.

There also exist other operations we do not describe here. Besides these, MR-MPI has several library variables that can help to tune or debug applications. For example, the *all2all* parameter can be used to determine the type of communication that will be performed during collate or aggregate operations (it can be either MPI_Alltoallv or irregular communication). Memory size of pages can also be adjusted using *memsize* parameter. Another important parameter is the *outofcore* setting that adjusts writing of memory pages to disk. For more information about all operations and settings used by MR-MPI, the interested reader should refer to MR-MPI manual [12].

2. SpMV with MR-MPI

In this section, we describe the implementation-level details of developing an SpMV operation in MR-MPI. For $Ax = b$, where $A = [a_{ij}]$ is an $m \times n$ sparse matrix and x is the solution vector, we designate the multiplication of a single column of A_j with x_j as an atomic task, i.e., a mapper task.

We evenly distribute the columns of A among processors using the built-in hash function of MR-MPI without considering any data locality. Prior to running SpMV, we execute two mapper functions to partition columns of A. The KV pairs generated by these functions are as follows:

- The first mapper function is for loading matrix data and KV pairs generated in this stage has the form K = (i, j) and V = (a_{ij}). In other words, for each nonzero in the sparse matrix, a KV pair is generated. The key corresponds to indices of the matrix while the value corresponds to the nonzero associated with that index.

- Using KV pairs generated in the first stage, we define another mapper task to group the loaded nonzero elements in a column-wise fashion. This is achieved by converting K = (i, j) and V = (a_{ij}) pair to K = (j) and V = (i, a_{ij}). By doing so, we can now use these keys (which correspond to columns) and assign columns to processors using a simple hash function that uses modulus whose operator is number of processors involved in computation. Here, we also need row index i in value which will be used in determining the indices of the output vector b.

In a similar manner, the x vector is also formed as K = (j) and V = (x_j). Notice that the keys of the matrix and the x vector are determined by column indices. In MR-MPI context, matrix A, vectors x and b all correspond to MapReduce objects.

The rest of the operations correspond to performing an SpMV and are all performed on b vector MapReduce object as follows:

↓ Add x to b (using MR-MPI function *add*). MapReduce object b now contains KV pairs $(j), (x_j)$.

↓ Add A to b (using MR-MPI function *add*). MapReduce object b now also contains KV pairs $(j), (a_{ij})$ as well as $(j), (x_j)$.

↓ Using *convert* on b, convert KV pairs to KMV pairs. This operation will construct a list of values from from a_{ij} and x_j elements that correspond to the same key. At the end of this convert operation, b will contain KMV pairs of the form $(j), (a_{*j} \dots x_j)$, containing possibly multiple nonzero elements and a single x vector element.

↓ Perform column-wise sparse matrix vector multiplication using *reduce* operation of KMV lists of columns. The atomic reducer task in this stage is the multiplication of column j with x vector entry x_j.

Each processor generates a new list of KV pairs that correspond to elements of the output vector. The input to this reduce stage is the $(j), (a_{*j} \ldots x_j)$ and the elements of the output vector is formed by $a_{*j} * x_j$ for each nonzero a_{ij} in column j. Consequently, the output of this stage is $(i), (b_i)$ KV pairs. Note that processors may generate a set of KV pairs for the same key; there may be duplicate $(i), (b_i)$ pairs at a single processor as well as these keys might be distributed among processors.

↓ Prior to the collate stage, the communication volume that will be incurred in the collate stage can be reduced by forming a single KV pair from KV pairs that correspond to the same key by summing their values. In this way, sending of a key (i) multiple times by a single processor can be avoided. In addition a single summed value is formed and communicated instead of multiple values. This is achieved by first forming a KMV $(i)(b_i^1) \ldots (i)(b_i^n)$ pair from duplicate keys using *convert* operation. This KMV is then used to *reduce* with a simple summation operator to form the KV pair $(i)(b_i = b_i^1 + \ldots + b_i^n)$. Hence the communication volume can be drastically reduced by sending only a single *i, (bi)* KV pair instead of $(i)(b_i^1) \ldots (i)(b_i^n)$ KV pairs. Doing so requires a convert and a reduce operation which will cause extra overhead. This trade-off between communication volume and computation is investigated in our experimental analysis.

↓ The *collate* operation is run to form KMV pairs at processors. The values at different processors that correspond to the same key are aggregated using a hash function. Apart from the partitioning of columns of A initially, one can here provide a user-defined hash function to distribute the tasks of reducing $(i), (b_i)$ KV pairs. The used hash function may have an impact on the balance of volume of communication and the computational reduce stage afterwards. The output of this stage is the $(i), (\text{list}(b_i))$ KMV list. If a convert and reduce operation is performed prior to collate as mentioned in the above paragraph, $\text{list}(b_i)$ can contain at most a single value from each processor. If not, the processor responsible for a specific key can receive multiple values from each processor which correspond to that key.

↓ The final operation is the *reduce* operation that outputs KV pairs using KMV pairs generated after collate phase. The reduce operator is the summation of the values that correspond to key b_i (i). This stage completes forming of the solution vector which may conveniently be copied back to vector x in case of usage with iterative solvers.

Using above stages, an SpMV operation can be performed in parallel. In case of usage with an iterative solver, the MapReduce object that represents the matrix should be added to the MapReduce object b only once. However, it is necessary to update the b object at the end of an iteration where x is updated. There exists a single communication phase which is realized in collate stage. Careful usage of part vectors as hash functions allows one to reduce communication requirements of the SpMV operation by minimizing and/or balancing the volume of communication.

3. Communication-efficient SpMV

By utilizing user-defined hash functions, we can distribute mapper and reducer tasks and achieve data locality to further reduce communication requirements of parallel SpMV multiplies. This can particularly be achieved using part vectors. In this section, we describe two strategies to achieve simultaneous mapper and reducer task assignment based on graph- and hypergraph-based partitioning models. This is a pre-processing stage performed on the matrix used in SpMV operations. The output part vectors obtained at the end of the pre-processing phase are utilized as hash functions in MR-MPI for distributing mapper and reducer tasks among processors.

3.1 Communication volume reduction and balancing with graph model

The first method is based on graph partitioning and it correctly encapsulates the communication volume incurred in the SpMV operations performed *without* extra convert and reduce operations prior to collate stage. By using a multi-constraint formulation, it is also possible to obtain a balance on the communication volume incurred in the collate stage. The given matrix A is modeled with a bipartite graph $G = (V = V_R \cup V_C, E)$ where the vertex set V_R corresponds to rows of matrix A and V_C corresponds to columns of matrix A [13]. Nonzero elements of the matrix are represented by the edges of the bipartite graph. This model allows obtaining an unsymmetric permutation of the matrix and can be used to determine the assignment of mapper and reducer tasks in MR-MPI

by partitioning G into P parts where P is the number of processors in the system. To minimize and balance the volume of communication at the same time, a two-weight formulation should be adopted.

$P = 512$						
	with convert/reduce before collate			**without** convert/reduce before collate		
	RAND	**G/MC**	**H/MC**	**RAND**	**G/MC**	**H/MC**
dielFilterV2real	360.0	11.5	14.7	378.0	50.9	106.0
Serena	470.0	6.6	7.9	502.0	60.6	108.0
gsm_106857	162.0	5.6	7.4	170.0	23.3	47.6
gupta2	19.9	12.2	13.7	32.8	29.2	31.8
Emilia_923	299.0	5.5	7.0	319.0	50.4	96.4
Long_Coup_dt0	627.0	8.2	10.6	674.0	101.0	193.0
pkustk11	37.2	1.2	1.6	40.4	13.3	21.5
gearbox	64.9	1.8	2.4	70.3	19.6	36.8
$P = 1024$						
dielFilterV2real	367.0	16.0	21.1	379.0	67.8	142.0
Serena	481.0	8.7	11.5	502.0	78.0	149.0
gsm_106857	165.0	8.2	11.3	170.0	32.3	65.5
gupta2	22.1	13.8	15.7	32.9	29.9	32.1
Emilia_923	306.0	7.4	9.9	320.0	64.1	123.0
Long_Coup_dt0	645.0	11.2	14.8	675.0	130.0	248.0
pkustk11	38.5	1.9	2.4	40.4	17.2	26.2
gearbox	67.1	2.6	3.6	70.4	25.8	44.9

Table 1: Volume of communication incurred during SpMV operations for 512 and 1024 processors (volume of communication is in megabytes).

3.2 Communication volume reduction and balancing with hypergraph model

The second method is based on hypergraph partitioning and it correctly encapsulates the communication volume incurred in the SpMV operations performed *with* extra convert and reduce operations prior to collate stage. Again, by using a multi-constraint formulation, it is also possible to obtain a balance on the communication volume incurred in collate stage. The given matrix A is modeled with a hypergraph $H = (N,\ V = V_R \cup V_C)$. In this model, since column-wise partitioning of A is utilized, there exists a net $n_i \in N$ for each row of A. The columns, which correspond to computational tasks, are represented by the vertices in vertex set V_C. The connectivity set of the nets are determined by the nonzero elements in the matrix. To be able to obtain an unsymmetric permutation of matrix A and thus balance communication volume, we add a vertex $v_r \in V_R$ for each row of the matrix. Obtaining a partition of H with P parts, where P is the number of processors in the system, corresponds to assignment of mapper and reducer tasks in MR-MPI. As in the graph model, to minimize and balance the volume of communication at the same time, a two-weight formulation is adopted.

4. Results

We present our results on a variety of matrices selected from University of Florida Sparse Matrix Collection [16] for number of processors $P = 64, 128, 256, 512, 1024$. The mapper and reducer task assignments are achieved by using user-defined hash functions as part vectors. We compare three alternatives that are used for assignment of mapper and reducer tasks in MR-MPI:

- RAND: The assignment of mapper and reducer tasks is performed randomly.
- G/MC: The multi-constraint graph model explained in Section 3.1.
- H/MC: The multi-constraint hypergraph model explained in Section 3.2.

The partitioning of the graph constructed with G/MC and the hypergraph constructed with H/MC are partitioned with MeTiS [14] and PaToH [15] respectively.

4.1 Volume of Communication

In Table 1, we present the volume of communication obtained at 512 and 1024 processors for eight matrices. We tested the effect of using a convert and reduce operation prior to collate phase. As expected, this reduced volume of communication drastically in the collate phase, especially for G/MC and H/MC. For instance, at $P = 1024$ processors for dielFilterV2real matrix, performing a convert and reduce operation before collate reduced the volume of communication from 67.8 MB to 16.0 for G/MC (76% volume reduction) and from 142.0 MB to 21.1 MB for H/MC (85% volume reduction). Similar findings are observed for all matrices in Table 1.

4.2 Runtime Results

This section presents the obtained runtime results for four matrices. We use the notation "-WC"or "-WOC" to indicate if a convert/reduce operation is performed prior to collate stage where inter-processor communication is performed. As seen from the results, using a pre-processing step to reduce communication requirements of the SpMV operations is critical to obtaining better scalability. In all figures, especially at large number of processors, the benefits of volume reduction methods are validated by attaining better runtime. In all figures, H/MC-WOC and G/MC-WOC achieve best results. This shows that performing a convert/reduce operation before inter-processor communication may not pay off. The reason behind this is that these two operations are expensive especially since the convert operation uses a hash table to efficiently handle duplicate KV pairs [10]. It can be said that reducing volume benefits the SpMV operations typically showing off better scalability (RAND vs G/MC or H/MC). However, further reduction of volume of communication at the expense of more computation may not pay off as the obtained results illustrate (G/MC-WC vs G/MC-WOC or H/MC-WC vs H/MC-WOC). As seen from the figures, using a random hash function leads to poor scalability, especially when processor counts increase, which is the case for RAND-WC and RAND-WOC.

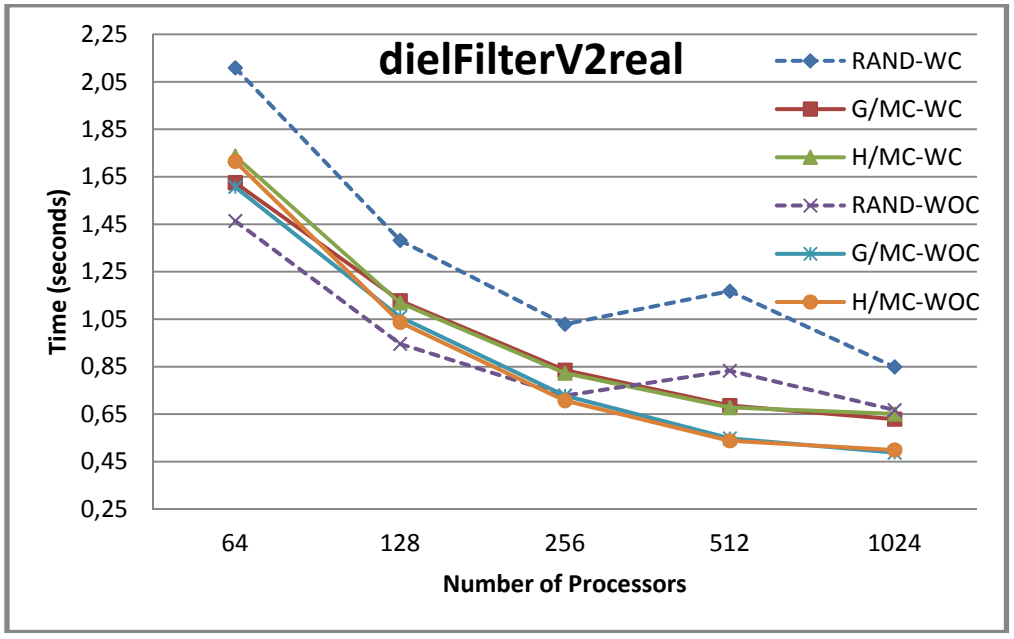

Figure 1: Comparison of different mapper/reducer task assignment strategies on SpMV for dielFilterV2real matrix.

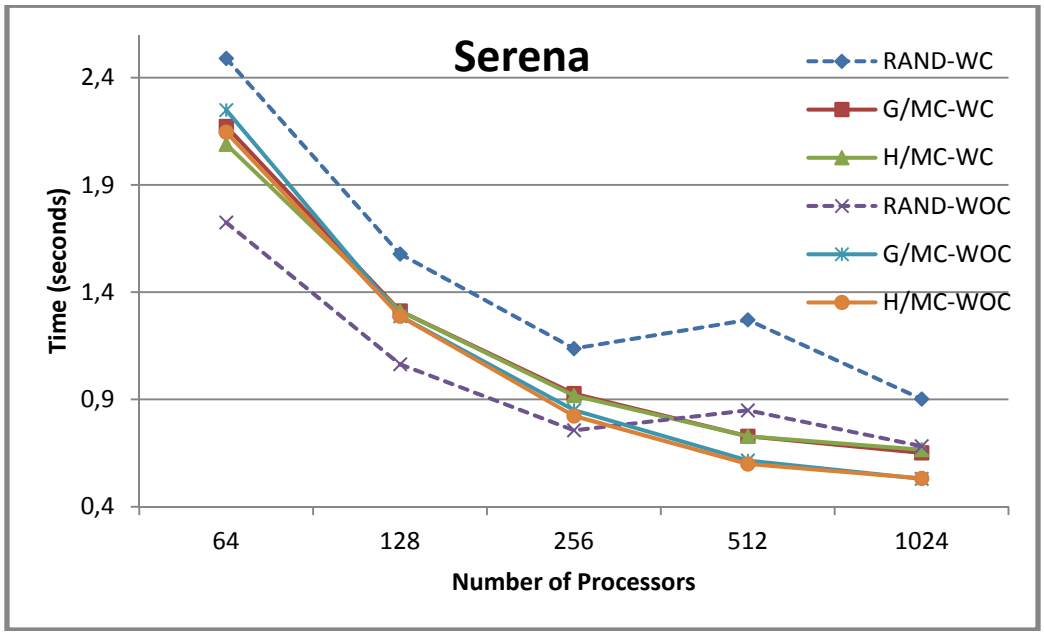

Figure 2: Comparison of different mapper/reducer task assignment strategies on SpMV for Serena matrix.

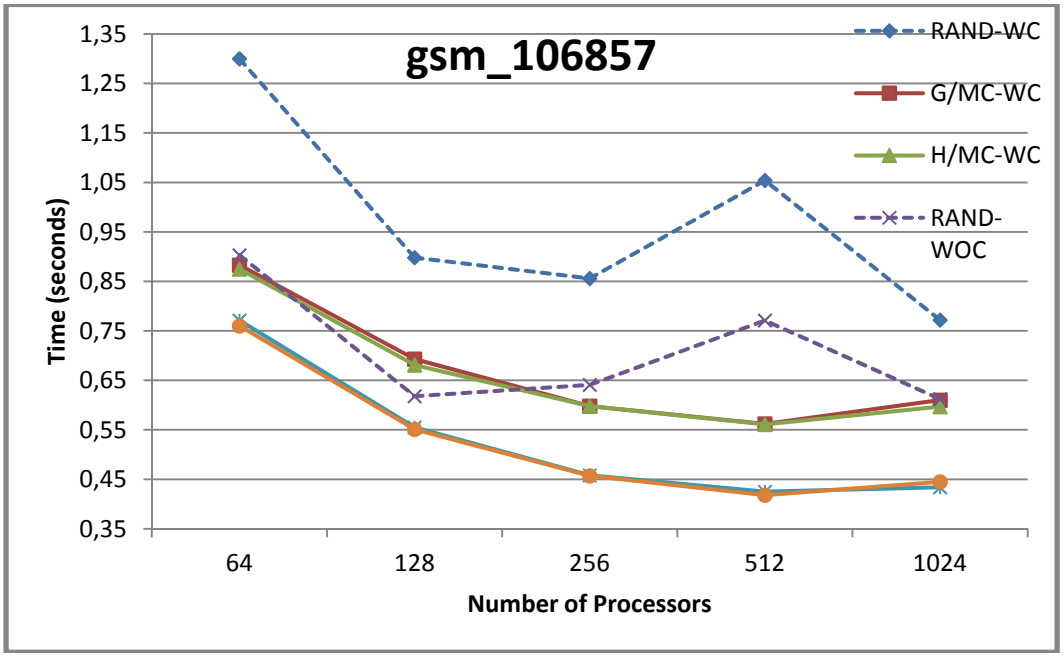

Figure 3: Comparison of different mapper/reducer task assignment strategies on SpMV for gsm_106857 matrix.

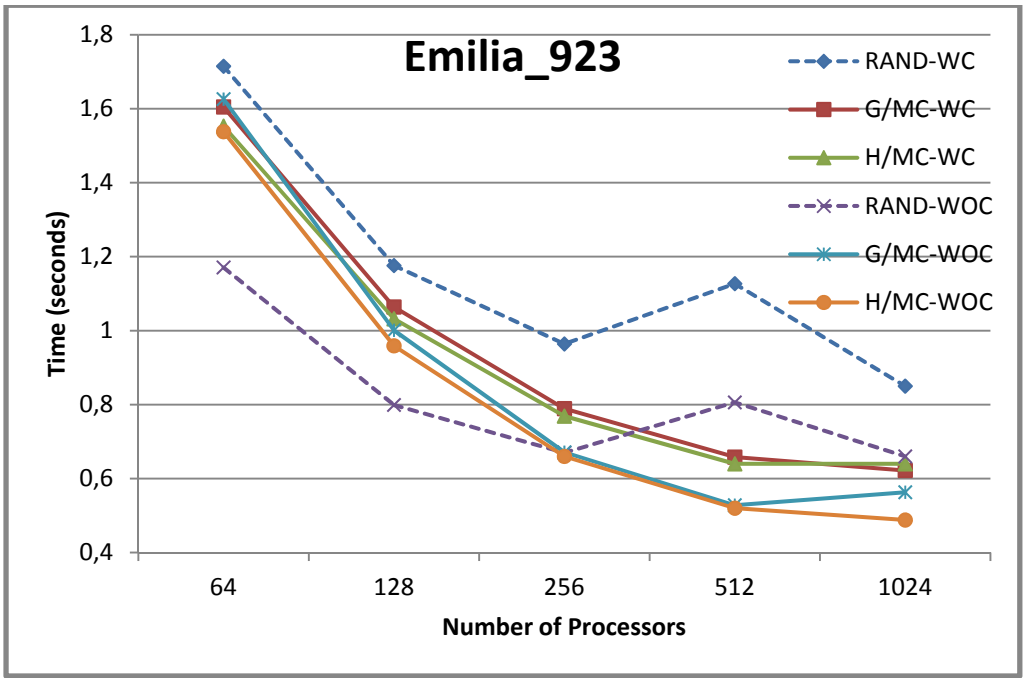

Figure 4: Comparison of different mapper/reducer task assignment strategies on SpMV for Emilia_923 matrix.

5. Conclusions

We have implemented a communication-efficient SpMV operation using MR-MPI library which is designed to be able to use MapReduce programming model in scientific computing. Experimenting with different partitioning and implementation-based methods, we showed that achieving scalability improvements for SpMV is only viable by exploiting data locality. Our experiments for different number of processors with matrices from different domains show that naïve schemes do not scale well beyond a few hundreds of processors. To obtain a good scalability, various optimization techniques centered on communication requirements of the SpMV are necessary.

References

[1] Dean, J., & Ghemawat, S. (2008). MapReduce: simplified data processing on large clusters. *Communications of the ACM*, *51*(1), 107-113.

[2] White, T. (2009). Hadoop: The Definitive Guide: The Definitive Guide. O'Reilly Media.

[3] Olston, C., Reed, B., Srivastava, U., Kumar, R., & Tomkins, A. (2008, June). Pig latin: a not-so-foreign language for data processing. In Proceedings of the 2008 ACM SIGMOD international conference on Management of data (pp. 1099-1110). ACM.

[4] Thusoo, A., Sarma, J. S., Jain, N., Shao, Z., Chakka, P., Anthony, S., ... & Murthy, R. (2009). Hive: a warehousing solution over a map-reduce framework. Proceedings of the VLDB Endowment, 2(2), 1626-1629.

[5] Chang, F., Dean, J., Ghemawat, S., Hsieh, W. C., Wallach, D. A., Burrows, M., ... & Gruber, R. E. (2008). Bigtable: A distributed storage system for structured data. ACM Transactions on Computer Systems (TOCS), 26(2), 4.

[6] Cohen, J. (2009). Graph twiddling in a mapreduce world. Computing in Science & Engineering, 11(4), 29-41.

[7] Ekanayake, J., & Fox, G. (2010). High performance parallel computing with clouds and cloud technologies. In Cloud Computing (pp. 20-38). Springer Berlin Heidelberg.

[8] Kang, U., Tsourakakis, C. E., & Faloutsos, C. (2009, December). Pegasus: A peta-scale graph mining system implementation and observations. In Data Mining, 2009. ICDM'09. Ninth IEEE International Conference on (pp. 229-238). IEEE.

[9] Tu, T., Rendleman, C. A., Borhani, D. W., Dror, R. O., Gullingsrud, J., Jensen, M. O., ... & Shaw, D. E. (2008, November). A scalable parallel framework for analyzing terascale molecular dynamics simulation trajectories. In High Performance Computing, Networking, Storage and Analysis, 2008. SC 2008. International Conference for (pp. 1-12). IEEE.

[10] Plimpton, S. J., & Devine, K. D. (2011). MapReduce in MPI for large-scale graph algorithms. Parallel Computing, 37(9), 610-632.

[11] Hoefler, T., Lumsdaine, A., & Dongarra, J. (2009). Towards efficient mapreduce using mpi. In Recent Advances in Parallel Virtual Machine and Message Passing Interface (pp. 240-249). Springer Berlin Heidelberg.

[12] http://mapreduce.sandia.gov/doc/Manual.html, MapReduce-MPI (MR-MPI) Library Documentation

[13] Hendrickson, B., & Kolda, T. G. (2000). Graph partitioning models for parallel computing. Parallel computing, 26(12), 1519-1534.

[14] Karypis, G., & Kumar, V. (1998). A fast and high quality multilevel scheme for partitioning irregular graphs. SIAM Journal on scientific Computing, 20(1), 359-392.

[15] Catalyurek, U. V., & Aykanat, C. (1999). Hypergraph-partitioning-based decomposition for parallel sparse-matrix vector multiplication. Parallel and Distributed Systems, IEEE Transactions on, 10(7), 673-693.

[16] Davis, T. A., & Hu, Y. (2011). The University of Florida sparse matrix collection. ACM Transactions on Mathematical Software (TOMS), 38(1), 1.

Acknowledgements

This work was financially supported by the PRACE project funded in part by the EUs 7th Framework Programme (FP7/2007-2013) under grant agreement no. RI-312763.