Scalability of Hybrid Sparse Matrix Dense Vector (SpMV) Multiplication

Brian A. Page, Peter M. Kogge
Dept. of Computer Science and Engineering
Univ. of Notre Dame, Notre Dame, IN 46556, USA
Email: (bpage1,kogge)@nd.edu

Abstract—SpMV, the product of a sparse matrix and a dense vector, is emblematic of a new class of applications that are memory bandwidth and communication, not flop, driven. Sparsity and randomness in such computations play havoc with conventional implementations, especially when strong, instead of weak, scaling is attempted. This paper studies improved hybrid SpMV codes that have better performance, especially for the sparsest of such problems. Issues with both data placement and remote reductions are modeled over a range of matrix characteristics. Those factors that limit strong scalability are quantified.

Keywords—Scalability, Hybrid SpMV, Communication Overhead, HPC;

I. INTRODUCTION

The product of a sparse matrix and a dense vector (SpMV) is a key part of many codes from disparate areas. Numerically, it makes up the bulk of the High Performance Conjugate (HPCG) [1] code that has become an alternative to LINPACK for rating supercomputers. It can also be used in machine learning apps such as SVM computations via gradient descent. When the matrix operations are changed from floating multiply and add to other non-numeric functions that form semirings, it becomes an essential part of many graph kernels [2], and is a key function in the recently released GRAPHBLAS spec [3]. There has even been a novel prototype hardware system built around such sparse operations [4].

Motivating the work presented here was an earlier study, [5], that looked at strong scaling of SpMV in a Message Passing Interface (MPI) environment for a variety of matrices of varying sparsity from a well-known repository. What caught our interest was an observed significant dip, not increase, in speedup versus number of processes (see Fig. 1 where the numbers on each line is the average non-zeros per row) that approached an order of magnitude for the sparsest cases, and from which recovery was slow as the compute resources increased. We have observed similar dips in other kernels and wanted to explore this phenomena more closely. Our specific goal was thus to explore the cause of the dip as a precursor for developing codes that scale better for these very sparse cases.

In organization, Section II describes the generic hybrid algorithm. Section III describes the spectrum of sparse matrices considered. Section IV provides necessary background. Section V discusses the code developed here. Section VI evaluates the results. Section VIII concludes.

II. SpMV HYBRID ALGORITHM OVERVIEW

Algorithm 1 shows the simplest sequential implementation of SpMV, consisting of two nested loops, the innermost of which processes one non-zero at a time from a row in A.

Algorithm 1 Hybrid SpMV: \( c = Ax \)

\[ A = mxm \] sparse matrix
\[ m = \text{number of rows to work over} \]
\[ c = \text{result vector} \]
\[ x = \text{dense input vector} \]
\[ a = \text{vector of non-zero } A \text{ element data values} \]
\[ \text{index} = \text{non-zero column id's} \]
\[ nnz_{\text{row}} = \text{number of non-zero elements in the } i^{th} \text{ row} \]

Fig. 1: Speedup from seen for 4 Sparse Matrices. This chart was generated using data seen in Table IV in [5].
about the system architecture. Algorithm 1 performs 2 floating point operations for every non-zero element in the sparse matrix, along with three memory read accesses: one into the index vector that identifies which column each non-zero belongs in, one into the matrix, and one into the dense vector \( x \). This assumes each \( c[i] \) is kept in a register until the row is complete. For most implementations there are very little cache hits. If each index entry can be held in 4 bytes, the memory traffic for each non-zero is approximately \( 8 + 4 + 8 = 20 \) bytes for each two flops per non-zero. An analysis of SpMV within the HPCG code [6] demonstrates that this ratio is correct, and thus the core SpMV computations are totally memory bandwidth bound, regardless of core flop capabilities.

If the arrays in Algorithm 1 are in shared memory, then a multi-threaded implementation can compute separate rows concurrently, allowing the memory system to be fully utilized, and maximizing performance to the 2 flops per 20 bytes limit.

In contrast, the problems studied in this paper assume that SpMV is applied over large sparse matrices that must span multiple processes in a large distributed system. The matrix must be partitioned and distributed, and the pieces put back together to get a complete answer. A hybrid algorithm would then combine a distributed implementation with multi-threading within each process. This is the focus of this paper.

III. MATRIX BENCHMARK SUITE

We have chosen 25 matrices from the SuiteSparse Matrix Collection\(^1\) [7]. Table I shows the dimensions, number of non-zeros, sparsity, and average number of non-zeros per row for the chosen matrices. The matrices were selected based on their average \( \text{nnz}_{\text{row}} \), in addition to overall non-zero structure. At least two matrices from each \( \text{nnz} \) range were chosen with similar size or \( \text{nnz} \) per row but different structure.

Fig. 2 highlights the structure disparity between two of these matrices: atmosmodd and parabolic_fem. The non-zeros in atmosmodd cluster along the main diagonal, whereas parabolic_fem has a wider dispersion. We focused on structural differences to evaluate the impact of matrix structure on communication volume and message size across different load balancing methods. Additionally we chose matrices a quasi logarithmic fashion to ensure a wide spectrum of sparsity.

IV. BACKGROUND

In this section we provide a brief overview on related work. IV-A discusses matrix \( \text{nnz} \) structure and its effects on SpMV performance. The importance of optimal load balancing along with some current methods is elaborated on in IV-B. Finally, in IV-C we look at some of the underlying implementation of MPI and its potential effects on communication overhead.

A. Matrix Structure

As input matrix size increases, the desire to leverage larger and more diverse systems facilitates the leap to distributed and hybrid parallel implementations. In such cases, the distribution of non-zero elements in the matrix has significant impact on SpMV performance, and subsequently affects a myriad of design decisions [8], [9], [10], [11], [12].

While the structure of a sparse matrix will undoubtedly influence the efficiency of the storage format chosen, matrix structure exhibits greater influence over communication [13], [5], [14], [15], [16], [17], [18], [19]. Hybrid implementations possess improved data locality, thread scalability, and data reusability [15]. Elevated performance is not guaranteed however, as memory contention can occur with greater frequency as thread counts increase [14]. In spite of this, the potential for significant global communication reduction, by converting much of it to local communication, has ever greater effects on performance as system size is scaled up.

B. Work Distribution

An irregular memory access pattern is the root cause of SpMV performance degradation in distributed memory systems [13]. Therefore as process and thread count increase, given a matrix of arbitrary structure, the disparity in the volume of work as well as the data locality of the individual \( \text{nnz} \) elements being computed locally can change drastically.

\(^1\)Currently hosted at https://sparse.tamu.edu/
Many studies focusing on optimal work load balancing for distributed SpMV have been performed [17], [20], [18], [21], [22], [19] in an effort to obtain consistency in performance gains.

It should be intuitive to say that reducing communication improves overall SpMV run time by limiting latency, memory registration, and buffer copies to memory along with other factors. Several studies have shown that the key to avoiding excessive communication overhead is to determine the optimal work load distribution for a given matrix [17], [20], [18], [21], [22], [19]. These studies found that balancing work by accounting for \( nnz \) location within the input matrix allowed for reduced communication and message size.

Communication avoidance is a complex issue to resolve as it is rooted in the system hardware, interconnection hardware, as well as communication and user level protocols being used.

Furthermore, the optimal work distribution algorithm is likely to be optimal for only a subset of the matrices being evaluated. It then becomes necessary to find the appropriate level of optimality and efficiency.

C. MPI Overhead

MPI overhead is a well researched area and continues to be of importance as we look towards future HPC systems. As hybrid and or heterogeneous implementations of SpMV require communication at scale, message passing as well as interconnect topologies and their user level libraries play a vital role in their overall performance.

Message Volume, Size, and Latency. Message latency when using MPI is dependant on many factors such as message size, interconnect type, protocol selection, etc [23], [24]. For small messages, the Eager protocol utilizes pre-allocated buffer locations to perform communication between the two processes [24]. There are no acknowledgements or negotiation required by the Eager protocol, however it suffers from relatively low bandwidth utilization [23], [24]. If message size is greater than the threshold value in use, the Rendezvous protocol is normally utilized. The Rendezvous protocol has additional overhead costs due requiring allocation of appropriate buffer memory to receive the message, as well as perform a negotiation phase between the sender/receiver [24].

Message size varies with \( P \) (number of processes), changing how messages are sent. Optimizations such as the use of RDMA, buffer reuse, communication overlapping, and speculation all have a point at which their performance enhancement declines drastically if not disappears [23], [24].

Remote Direct Memory Access RDMA. RDMA permits zero-copy transfers of data between MPI processes by performing either an MPI_Put or MPI_Get operation [23], [24]. RDMA eliminates the need for buffers and their costly allocations, as well as the need to copy buffered data into its final memory location after transmission completion [24].

Communication Overlapping, Offloading/Onloading. Overlapping as well as offloading are designed to allow for communication during a subsequent computation phase, and are aided by the use of RDMA for increased bandwidth utilization with decreased latency. In the case of Overlapping, non-blocking MPI_Isend or MPI_Irecv are used to begin the communication phase before proceeding onto computation.

Offloading and Onloading work similarly. In Offloading communication is pushed onto the interconnect hardware for completion thereby allowing the process to continue. Conversely with onloading; communication overhead is handled by a selected number of threads, cores, or sockets in an effort to alleviate the impact of high bandwidth communication on the NIC or interconnect interface [24]. Onloading is becoming more common as many-core architectures such as Intel Xeon Phi (formerly Knights Landing) see greater use [25].

Collectives. MPI Collectives allows multiple processes to partake in a single communication event such as a reduction or gather. Several studies [26], [27], [28] have evaluated the performance and behaviour of MPI collectives in addition to implementing optimization methods. Collectives dramatically reduce communication volume and latency along with increase bandwidth utilization [26], [27], [28]. Enhanced RDMA has proven useful for such operations.

V. EXPERIMENTAL IMPLEMENTATION

To evaluate MPI overhead for our hybrid SpMV solver we implemented two different work distribution schemes. The first method distributes work to each process based on the unmodified \( nnz \) positioning of \( A \), and is hereafter known as the "naive" method. In contrast the "balanced" method creates a near uniform work load amongst all MPI processes. Section V-A elaborates upon the sparse matrix storage format we have to chosen utilize. Section V-B briefly discusses OpenMP and MPI as they pertain to our implementations. Subsequently sections V-C and V-D elaborate upon the naive and balanced work distribution methods respectively.

A. SpMV Storage Format

While many storage methods for sparse matrices have been devised, each has drawbacks, whether it be increased memory footprint, communication requirements, or implementation complexity [8], [29], [30], [9], [10], [11], [12]. Compressed Sparse Row (CSR) format is a commonly used method in which the non zeros within a given matrix are kept in three one-dimensional vectors or arrays: row, column, and data. CSR features a relatively simple structure and is therefore easily implemented and adapted for hybrid applications utilizing both MPI and OpenMP. Furthermore, nearly all other formats compare themselves to CSR [8], [9], [10], [11], [12]. While some formats may enable greater memory efficiency or communication reduction, the added implementation and algorithmic complexities are drawbacks that should not be overlooked. Therefore CSR was chosen here; however we acknowledge that some performance impact might be gained via the use of different storage formats.

B. Hybrid Design

Two widely known parallel methodologies, distributed memory and shared memory, offer the potential for increased
performance at the expense of elevated code complexity. MPI makes use of the distributed memory model by dispersing application functionality across different processes, thereby allowing for strong scaling through increases in system size. Increased system size comes at the expense of added communication requirements due to added intra/inter-process message count. The shared memory model has potential for localized performance enhancements from increased cache utilization and performance. Hybrid codes, typically MPI and OpenMP, take advantage of both models by implementing the shared memory model within each distributed memory element.

Our code utilizes MVAPICH2 with Infiniband support and OpenMP 4.5. Our cluster has 72 nodes, each with two 8-core Xeon E5-2650v2 @ 2.6GHz, and connected via Mellanox FDR Infiniband in non-blocking configuration. Each node is assigned two MPI processes, one per socket, with each MPI process creating one OpenMP thread per core. We insured process and thread affinity by binding processes to their assigned socket, and OpenMP threads to their respective cores.

Fig. 3: Work Distribution Communication Pattern: (a) Work for each process column is sent to that columns process master, (b) each column master then sends each process in its column their own work portion. (c) The Naive method, described in Section V-C and [5], utilizes MPI_Reduce across each process row to compile calculated row values at the row master process. (d) In contrast, the Balanced method, described in Section V-D, uses a variable gather compile 

C. Work Load Distribution: Naive

Our first work load distribution method, based on [5], splits the matrix $A$ into $P$ sub-matrices $A_{ij}$, each of which is assigned to a corresponding process $P_{ij}$ in the process matrix. The master process, $P_{00}$, sends the column master process in each process column the data for all processes within that process column. Column masters, $P_{0j}$, distribute row, col, and data vector information to each process they govern.

Splitting up $A$ in this manner is not only conceptually simplistic, but also limits the $nnz_{row}$ per process that any process can receive. Any process in the process matrix will receive at most $\frac{P}{P}$ rows, with at most $\frac{P}{P}$ columns per row. Within each $P_{ij}$, multiple OpenMP threads are initialized and work over the process’s work allotment concurrently. Once each process has completed its work allotment, all processes participate in an MPI_Reduce with MPI_Sum. This is possible since each process in a given process row computes a sub sum of the entire row in $A$. The row masters, $P_{i0}$, now have the final result for their respective row.

We now look at the quality of work load distribution generated by this method. As mentioned previously the naive method is tied directly to the structure of $A$. It is thus possible for a process, or multiple processes, to be assigned no work to perform during the SpMV computation phase. To evaluate the quality of work load distribution between the naive and balanced methods, several metrics were employed. It is well known that the standard deviation can experience issues in illuminating the true nature of a distribution if it is not Gaussian. The balanced distribution method is designed with the intent of generating a near uniform distribution, as will be discussed in greater detail in the following section, and is therefore not Gaussian. Instead the Mean Absolute Deviation (MAD) [31] was chosen to indicate the distribution quality.

A perfectly uniform distribution would have a MAD of 0.0, meaning that the average deviation from the mean is 0.0. A higher MAD indicates a distribution with much greater variation from the mean and indicate process load imbalance. MAD values were normalized and represent % of total number of $nnz$ for each $A$. The min, max, and median MADs observed for the selected matrices were 6.0E-03, 3.2E-01, 7.1E-02 and respectively. These values while appearing small are in fact relatively large, indicating significant imbalance.

D. Work Load Distribution: Balanced

The goal of the balanced method is to create a uniform work load distribution across all $P$ MPI processes, thereby minimizing communication and message size. The optimal amount of $nnz$ per process is calculated as being $\omega = \frac{nnz}{m}$, where $m$ is the dimension of $A$. It is worth noting that $\omega$ may be less than ideal for computation due to cache performance. However it does allows for a quasi-uniform work distribution based on a matrices’ internal structure. Where possible entire contiguous rows are assigned to a single process so that cache misses may be reduced where possible. For very large matrices
it is possible that the number of $nnz$ in a given row may exceed that of $\omega$, therefore $\omega$ is used as the break point for maximum row size. Should $nnz_{row_i} > \omega$, row $i$ is split and a new row containing all $nnz$ of row $i$ after the $\omega^\text{th} nnz$, is created with the same row id.

Once we insure that there are no rows exceeding $\omega$, all rows were then sorted by length. An efficient greedy bin packing algorithm assigns the sorted rows to MPI processes from largest to smallest, based on a process’s current $nnz$ assignment. Due to the desire to maintain rows in their largest possible contiguous form we allowed for the possibility of over/under assignment of work to any given process. Clearly this can generate sub-optimal work distributions. While optimal bin packing would be ideal, it is known to be an NP-complete problem [32], [33], therefore efficiency and ease of implementation took precedence in our study. The quality of distribution produced by our method is discussed later.

Similar to the naive method, once work load distribution has been determined, the master process, $P_{00}$, sends all work for a given process column to that column’s column master $P_{0j}$ (Figure 3a). Each $P_{0j}$ then distributes the work assigned to each individual process within in its process column (Fig. 3b). Computation is then performed within all OpenMP threads of $P_{ij}$ like the naive method, yet now each process may have differing number of rows assigned to them. MPI_Reduce requires all participating processes to a send buffer of equal size potentially requiring superfluous message padding.

In the balanced distribution method it is not guaranteed and highly unlikely that all processes in each process row would posses portions of the same rows from the sparse matrix. Therefore in an effort to reduce message size we decided to implement a variable gather in the balanced method (Fig. 3d), as it would allow $P_{00}$ to receive the exact number of row solutions assigned to a given process. This eliminates the padding necessary for implementing MPI_Reduce. MPI_Gatherv requires a mapping of sorts so that MPI knows the incoming buffer size for each process. This mapping was generated during work load determination and kept for use during the gather phase. Our implementation of MPI_Gatherv was such that all results were gathered at the master and a post gather insertion into the result vector was performed.

Given that the exact algorithm utilized by MVAMPLICH2 for a collective is dependent on message size and other factors, we assume the variable gather implemented has a complexity of $O(log_2p)$, similar to a gather utilizing the binomial tree algorithm.

As mentioned previously it is possible for the final work distribution to be less than perfectly uniform thanks to our greedy implementation. Inspection of the MADs produced by our balanced distribution show a min, max, and median of 2.7E-08, 6.7E-03, and 4.74E-06 respectively. This indicates that we achieve a very high level of uniformity. Therefore our balanced work method can be utilized to create message sizes of greater uniformity for all processes in $P$ as intended.

Fig. 4: Observed SpMV computation time for single MPI process with varying OpenMP thread counts. Each line represents a different benchmark matrix from the chosen suite discussed in Section III. Single node multithreaded tests include no communication since OpenMP uses a shared memory model and only one node is in use.

VI. EVALUATION

A. Multithreaded SpMV Performance

To show the impact that communication from a distributed memory model has on Hybrid SpMV, we first analyzed 1 MPI process running on a single node, with increasing numbers of OpenMP threads. Fig. 4 shows that as thread count increases SpMV computation time decreases, up to a point. Each additional thread makes more memory references needed to perform additional useful computation. However, for most matrices evaluated, computation time reduction slows or stagnates completely after some value of $P$. This occurs approximately when the number of threads exceeds those supported by a single socket, and is likely due to cross-socket coherency traffic. Given relative best performance seen at 1 thread per core, subsequent experiments used a single MPI process per socket, and utilized all cores on that socket for computation.

B. MPI Communication Overhead

MPI communication, whether it be intra-node or inter-node, can dramatically outweigh the time requirement for useful computation. Fig. 5 shows MPI communication time as well as SpMV time for each matrix and $P$ examined. Note that for 1 process there is no communication time. After the utilization of 4 processes MPI_Reduce time is nearly always larger than SpMV computation time, and often several times as long.

Fig. 4 clearly indicates that distributed communication, even in a hybrid implementation, is extremely high for SpMV applications. In general SpMV computation time decreases with strong scaling, and as $P$ increases, the number of elements to be communicated by each process decreases as $\frac{m}{P}$. Fig. 6 shows the ratio of communication time to computation time from Fig. 5. On average, for a given matrix, as the number of processes increases, the ratio of comm-to-comp time increases
Fig. 5: Observed MPIReduce times using the naive distribution method. Communication time sits atop spmv computation time for comparison with differing MPI process counts. Benchmark matrices are shown in ascending order by avg \(nnz\) per row.

Fig. 6: Communication-to-Computation Time Ratio: This chart shows the ratio of communication time to SpMV computation time for the evaluated process counts. Matrices are listed in order ascending according to \(nnz\) row. Matrices with the lowest \(nnz\) row experience the worst \(\frac{comm}{comp}\) ratio, while higher \(nnz\) row matrices see considerably less.

as well. Additionally we notice that the matrices with fewer \(nnz\) per row experienced some of the highest ratios, while matrices with higher avg \(nnz\) per row experienced some of the lowest. Furthermore, in an effort to eliminate the impact of cluster node selection on communication paths all tests were run 50 times with the averaged results used in analysis.

While Fig. 5 and 6 prove that communication time is high, these charts only include data for our tests over Mellanox Infiniband interconnects. Infiniband’s performance increases over Ethernet are well documented and beyond the scope of this paper. We did however examine the communication time for the entirety of the benchmark suite using both distribution methods, and observed comm-to-comp ratios of 10x or more.

The balanced method is designed to reduce communication by providing a balanced workload. Fig. 7 shows the comm-to-comp ratios observed using the Infiniband interconnect. It illustrates the same pattern in which matrices with low average \(nnz\) per row experience higher comm-to-comp ratios while the most dense matrices received the lowest ratios. In Fig. 7(b) at \(P = 81\) there is a considerable jump in communication overhead somewhere in the range of 10x compared to \(P = 64\). Our system consists of 72 dual socket nodes, connected to two Infiniband switches. The jump from 64 to 81 processes corresponds with a jump in total node utilization as well. At 81 processes our system size now spans two switches meaning that messages across the additional hardware will have increased latency. This is the reason for the observed increase, as after this initial threshold communication times increase at a \(log_2(p)\) rate similar to \(P < 81\).

It is clear that as \(P\) increases MPI communication becomes the dominant factor in overall runtime, even for the densest matrices in the benchmark suite.

C. Communication Overhead Impact

To compare communication time against theoretical expectations Fig. 8 plots the normalized observed time against a \(log_2(p)\) curve. The solid line beginning at 1 represents \(log_2(p)\), the simplified time complexity for MPI_Gather [26], while the second solid line beginning at 81 represents \(\gamma log_2(p)\)
in which $\gamma$ is the observed scaling factor caused by our network hardware layout. Some variance was expected when producing the fit shown in Fig. 8 as a single latency and effective bandwidth value was calculated for the two regions, $P = 0 - 64$ and $P = 81 - 121$. A tighter fit is possible if latency and effective bandwidth are accounted for each value of $P$ when normalizing and plotting.

What can be seen is that for most $P$, the observed MPI communication time is in the range of the expected MPI_Gather time complexity. However for 36-49 processes there are significant dips in the communication time for nearly all matrices. In fact when each matrices values were plotted and fit individually, we noticed a similar dip for all matrices.

The message being transmitted is the computed value for a particular row, or multiple rows depending on the amount assigned to any arbitrary process. As $P$ increases $\frac{m}{P}$ decreases, meaning the number of rows that must be sent per process decreases correspondingly. MPI implements a message size threshold with which it determines when to utilize the Eager or Rendezvous protocol for sending messages. Once message size drops below this threshold MPI will select the Eager protocol, which experiences very low latency due to the elimination of: memory buffer allocation, negotiation between processes, and message receipt acknowledgements.

While there are other causes for this communication performance increase, such as cache effects or better work balancing, we believe that the transition between protocols provides the greatest impact at these values of $P$. Shortly after these local minima, we saw an increase in communication time as $P$ increased. This was expected due to the nature of the default communication algorithm for MPI_Gather. Similar results were found when we analyzed the naive distribution method, which also has time complexity of $O(\log_2(p))$.

After evaluating communication time for larger process counts, we then sought to evaluate the overall performance by combining SpMV computation time with its corresponding communication time. As can be seen in Fig. 9 the speedup seen for all matrices can be split up into three general performance bands. Obtaining the best performance are those matrices with the highest $nnz$ per row, receiving an increase in performance.
Fig. 8: Observed MPI Collective Times: shows the normalized time measurements for MPI_Gatherv used within the balanced method plotted along the expected $O(\log_2(p))$ curves for both small and large values of $P$. Each circle represents a different benchmark matrix evaluated at the given process count.

Fig. 9: Combined performance for SpMV computation and Communication of result, in GFlops. Each line represents a different benchmark matrix for the given number of MPI Processes, using the balanced distribution method.

until 16 processes. Matrices with approximately 10 to 100 $nnz$ per row form the center band exhibiting mild speedup until 4 processes before flattening out. Here any additional performance gains seen as process count increases is likely offset by the accompanying increase in communication overhead. Finally the lower band consisting of matrices with fewer than 10 $nnz$ per row never experience an increase speedup.

Regardless of overall matrix sparsity or $nnz$ distribution pattern, all matrices tested exhibited similar behavior as process count increased. The dip in performance seen between 25 and 36 processes is likely due to crossing the switch capacity boundary for the Infiniband fat tree configuration. The peak at 49 processes corresponds to a switch between the Eager and Rendezvous message transfer protocols within MPI thus providing a boost in performance as message size decreased below the threshold. Degradation in performance after 49 processes is due to several factors including compounding switch latency, rapidly declining number of non-zeros per process, and increased message volume.

Another important observation is one of maximum peak performance. From Fig. 9 it is obvious that some matrices drastically outperformed others. In general, the more dense a matrix, the higher its maximum performance, while the matrices with higher sparsity experienced performance flattening for lower values of $P$. The results suggest an upper bound for performance based on communication overhead and its accompanying process count.

VII. Future Work

While our work has shown that performance degradation created by communication overhead exists for extremely sparse problems on conventional computing clusters, further exploration is needed. Sparse problems will be evaluated on additional architectures such as GPUs and Intel’s Xeon Phi, as will the communication impact these architectures experience in a multi-node environment. Similarly we would like to investigate resource over-subscription in an effort to further explain potential NUMA effects brought on by the cross socket coherency traffic complexities we believe were seen in Fig 4.

Our tests thus far have required that the entirety of a sparse matrix and its accompanying dense vector fit on a single node. The issues associated with larger data sets incapable of residing on individual nodes, as well as the ability for streaming updates to the sparse matrix and or the dense vector are also of interest. These concepts are worthy of further analysis as they will likely generate communication issues for work load distribution and computation.

Also, as mentioned in the Introduction, SpMV-like kernels have key use in non-numeric applications such as GraphBLAS. A valuable study would be to generalize the algorithm for non-floating point operation suites and data, such as might be found in calculations around adjacency matrices. In addition, new data compression methods will be evaluated in an attempt to further reduce communication between processes and nodes. Finally, it would be of significant value to explore variations in the $x$ vector, from being sparse itself, to multiple vectors, to matrices (both dense and sparse).

VIII. Conclusions

This study evaluated hybrid SpMV implementations with particular focus on strong scaling where the problem is fixed and additional computing resources are added. We demonstrate that dingle process performance does improve as thread counts increase, but flattens out once the memory system is saturated, with some degradation in performance once the process covers more than one socket. In addition, this paper demonstrates that overall scalability in a message passing environment is driven by its communication pattern. Our findings suggest SpMV possesses an inability to strong scale beyond some point, even with cutting edge network interconnects such as Infiniband.
and that this scaling limit is tightly tied to the level of sparsity in the matrix. Further, the current focus in the literature on reducing SpMV computation time (such as by smart partitioning) does not address this dominant communication overhead factor at scale. Manycore architectures are thought to be the answer to this issue, however our studies show that as $P \to \infty$ communication overhead is the dominant factor and cannot be discarded. For extremely sparse matrices we saw an impact due to inter-node communication with as few as 2 nodes (4 processes). This would suggest a need to shift efforts away from reducing SpMV computation time in favor of attempting to mitigate or eliminate conventional communication within such applications. Alternative schemes for remote aggregation will be central for new algorithms that perform better than the current paradigm.

IX. ACKNOWLEDGEMENTS

This work was supported in part by the Department of Energy, National Nuclear Security Administration, under the Award No. DE-NA0002377 as part of the Predictive Science Academic Alliance Program II, in part by NSF grant CCF-1642280, and in part by the University of Notre Dame.

REFERENCES


