Scalability of Hybrid Sparse Matrix Dense Vector (SpMV) Multiplication

Brian Page
Univ. of Notre Dame
Notre Dame, Indiana
bpage1@nd.edu

Peter Kogge
Univ. of Notre Dame
Notre Dame, Indiana
kogge@nd.edu

ABSTRACT

The product of a sparse matrix and a dense vector (**SpMV**) is a key part of many codes, and a variety of studies have shown that the major driver to performance is memory bandwidth, and not peak floating point potential. One recent study in particular looked at strong scaling of a variety of matrices of varying sparsity, with performance that often declines with increasing parallelism. This paper reports both a more detailed analytic model and a carefully instrumented hybrid implementation using both OpenMP and MPI that explore these odd behaviors and identify the causes. Together, this should set the stage for better scaling algorithms on more advanced platforms.

CCS CONCEPTS

• **Computer systems organization** → **Embedded systems**; *Redundancy*; Robotics; • **Networks** → Network reliability;

KEYWORDS

multi-threading, parallel systems, mobile threads, memory architectures, performance, PGAS

ACM Reference Format:
Brian Page and Peter Kogge. 2017. Scalability of Hybrid Sparse Matrix Dense Vector (SpMV) Multiplication. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 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.[1] When the matrix operations are changed from product and add to a variety of other non-numeric functions that still form semi-rings, 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].

[1]http://www.hpcg-benchmark.org/

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM.
https://doi.org/10.1145/nnnnnnn.nnnnnnn

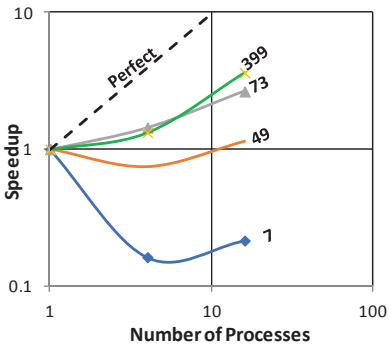

Figure 1: Speedup from [5] for 4 Sparse Matrices

Motivating the work presented here was an earlier study, [5], that looked at scaling of SpMV for a variety of matrices of varying sparsity from a well-known repository.[2] What caught our interest was an observed significant dip in speedup (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 available 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 duplicate the results, explore the cause of the dip, and extend the range of scaling, all as a precursor for developing better codes for these very sparse cases that would scale well to very large systems. However, in the process of performing this study, we found that the matrices used (all symmetric) were filed in the repository in a compressed lower form with only the lower diagonal half of non-zeros stored, and that even though the matrices were discussed in [5] in terms of their actual uncompressed sparsity, the computational results seemed to have used only this lower half, which approximately halved the effective sparsity, and distorted the distribution of non-zeros The work here has used the full version of these matrices.

In organization, Section 2 discusses the generic algorithm and then builds a simple model for estimating performance, as well as prior work. Section 3 discusses the code implemented here.Section 4 evaluates the results. Section 6 concludes.

2 AN ANALYTIC MODEL

This section discusses an analytic model for the SpMV algorithm run here that is modeled after the algorithm in [5]. It assumes we compute the product $y = Ax$ where the N x N matrix A has a

[2]https://www.cise.ufl.edu/research/sparse/matrices/

total of nnz non-zero elements, and where the system doing the computation is a $P = p^2$ array of MPI processes interconnected in a 2D array.

In terms of what to expect, the performance of the HPCG benchmark [1] is heavily dependent on SpMV-like calculations, and work on highly accurate modeling of its performance [6] led to the conclusion that HPCG's performance in terms of flops/sec was independent of the peak flop potential of the system running it, and primarily dependent on the available memory bandwidth. Fig. 2 demonstrates the correctness of this conclusion by diagramming real data taken from recent HPCG reports[3]. The x-axis is the peak flops of the reported system; the y-axis is the ratio of the sustained HPCG flops to the aggregate peak bandwidth of the system's memory (derived by determining the processing chips used and looking up their memory channel characteristics). The color and shape refer to different types of chips and systems, with the red squares representing system built from server-class chips, the blue representing Blue Gene, and the purple representing heterogeneous systems using GPUs. As can be seen, this ratio is independent of the peak system flops capability. In fact there is a nearly flat bounding line of a little less than 0.2 flops per byte of memory bandwidth for heavyweight server class processor chips, and somewhat less than 0.1 for GPUs and other architectures.

The model built here thus focuses on the relationship between memory bandwidth and performance in SpMV.

2.1 Algorithm Overview

The algorithm assumes that A has been divided into p^2 sub-matrices of equal size $(N/p)x(N/p)$, and on average has nnz/N non-zeros per row. The N-element column vector x is assumed striped across the columns, again in equal-sized N/p pieces so that all p processes in a column have copies of the same segment of x. The column vector y is assumed stored in the first column of processes, again in stripes of N/p elements per process. Each process then has all, and only, the non-zeros for the rows of A covered by the row the process is in, and all, and only, the non-zeros for the columns of A covered by the column the process is in. This is on average nnz/Np non-zeros per row in each process. They are stored in a CSR (compressed sparse row) format.

In execution, each process concurrently performs its portion of the matrix-vector product, namely a $(N/p)x(N/p)$ by N/p product. The resulting N/p element column is sent to the process on column 0 of the process's row, and atomically added into the y vector segment stored there. Each of the processes has on average nnz/p^2 non-zeros in its sub-matrix, $nnz/(Np)$ of them in each of its N/p row segments. Each of these non-zeros requires 2 flops (a multiply and an add), for a total of $2nnz/P$ flops performed per process. The atomic adds into the final y vector are not counted.

After the reduction, a barrier across all column 0 processes signals completion.

2.2 Single Process

We first look at the computation within a single MPI process using as many threads as reasonable to perform the multiply. As discussed in [6] for the SpMV kernel, the processing of each row in the local matrix partition requires three memory references for each non-zero: one each for the next non-zero from A and the column index for that non-zero, and an access into the x vector to get the matching terms for the product. [6] modeled a 4-byte index, and determined that on average about 20 bytes needed to be accessed from memory to start/finish each row, regardless of the number of non-zeros. This gave a total of $20 + 20nnz/Np$ bytes needed to be accessed for each row of N/p elements in length. Assuming enough threads (each handling different rows) to keep the local memory busy, dividing this into 2 flops, and multiplying by the sustainable bandwidth from memory gives an initial flops/s estimate. For the HPCG modeled by [6], this seemed very accurate.

[6] assumed that all these bytes needed to be physically fetched from memory, i.e. no benefits from cache reuse. This is certainly true of the non-zero and index read from memory - each such term is read exactly once during the multiply and then never reused. The access of the x element needs more discussion. Given modern caches where 20MB or more may be available on-chip, depending on the number of rows of A, it may be possible to eventually capture all of x (for example an x segment of a million elements requires only 8MB of cache). Thus, if there are enough non-zeros in the partition of A, then the later ones are liable to find their x values cached, and not require memory bandwidth.

Assuming a cache line of B bytes (holding $B/8$ double floats from x) then a total of $(N/p)/(B/8)$ compulsory misses would be enough to read into cache the entire $8N/p$-byte segment of x. The rest of the nnz/p^2 accesses into x might then be cache hits and not require physical reads from memory. Thus the total memory traffic for just the x values is $min(BN/p/(B/8),\ B * nnx/p^2)$ bytes from memory, (with the second term in the min for very sparse cases where there are very few non-zeros and each non-zero causes an access). The total data needed from memory for the entire multiply done by one process is thus:

$$(N/p)(20 + 12nnz/Np) + min(8N/p,\ B * nnz/P) \quad (1)$$

$$= 20N/p + nnz/P + min(8N/p,\ B * nnz/P) \quad (2)$$

Dividing this into $2 * nnz/P$ (the number of flops per process) yields the number of flops performed per byte of memory bandwidth. Fig. 3 diagrams how this relative performance changes for the same matrices from Fig. 1 as a function of the number of processes into which the matrix is partitioned. The numbers on each line represent the average number of non-zeros on each row of the full matrix.

Assuming that all processes have the same available bandwidth from memory, then multiplying the point on the curve by the number of processes gives the system-wide flops/s per byte/s of memory bandwidth per process. Dividing this into the flops needed for the matrix gives something proportional to execution time. Fig. 3 normalizes this to that for one process, and plots a "speedup" number. The "perfect curve" on this chart corresponds to a perfect strong scaling reference. It is clear that scaling is sensitive to sparsity, but we don't yet see a dip as in Fig. 1.

As a reference, HPCG uses SpMV extensively, but is arranged as a weak scaling problem with a near constant 27 non-zeros per row per process, regardless of the number of processes. Using the above equations, this equates to 0.156 flops per byte of bandwidth,

[3] http://www.hpcg-benchmark.org/

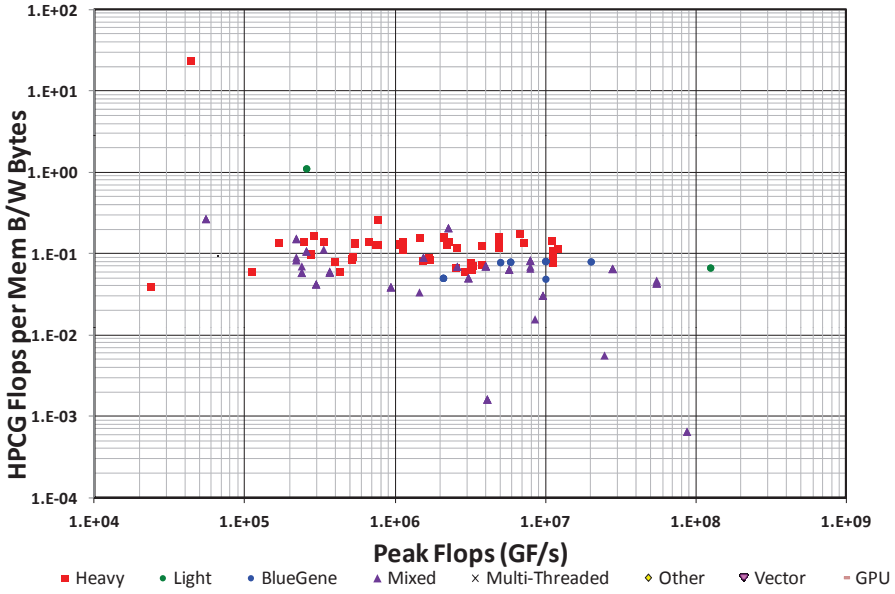

Figure 2: HPCG Flops per Byte of Memory Bandwidth vs. Peak flops.

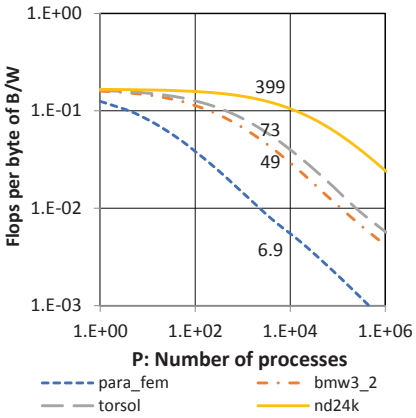

Figure 3: Scaling in Flops per byte of BW.

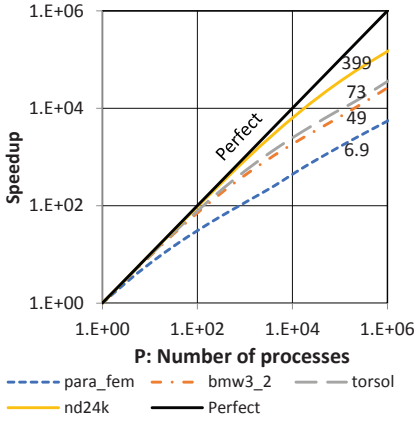

Figure 4: Single Process Speedup.

regardless of P. This is in good agreement with the observations from Fig. 2.

2.3 Inter-process Effects

The above analysis leaves out one important consideration - the effects of required inter-process communication when more than one process is involved in the computation. We ignore here any initial load and distribution of both A and x, but not the formation of the final y . Computing the overall y requires accumulating all the segments computed by each process. This is done independently over each row of processes, with each process in a row sending its N/p values via an *MPI_reduce* into an accumulation vector in the column 0 process in that row. This is then followed by an *MPI_barrier* to detect completion. Since the length of each segment of $8N/p$ bytes can be quite large (especially for small p), many MPI implementations will break the transmission of the full vector into smaller packets, and thus overlap each packet with the reduce computations in the target process.

We consider the *MPI_reduce* first. A "PLogP" pipelined model developed in [7] (that matched experimental data relatively well) estimates the time for such a reduce as:

$$T_{reduce} = (P-1) * (L + max(g(m_s), o_r(m_s) + \gamma m_s)) \quad (3)$$

$$+ (n_s - 1)(max(g(m_s), o_s(m_s) + \gamma m_s)) + o_s(m_s)) \quad (4)$$

where:

- L = the latency of an MPI message between two nodes

- m_s is the size in bytes of a segment into which MPI divides the input vector
- n_s is the number of segments into which MPI divides the input vector
- $o_s(m_s)$ and $o_r(m_s)$ are overheads for send and receive
- γ is the time "per byte" to do the reduction at the target
- $g(m_s)$ is the minimum gap between two messages on the same link

A recent presentation [8] reported an *MPI_reduce* latency on a QDR Inifiniband link (approximating what was in the system we used) as about $140\mu s$ for message segment sizes of 8 bytes and above. Another reference [9] provides some measurements on the MPI send and receive overheads of about $1-2\mu s$. If we want to convert these times into an equivalent memory bandwidth demand, we could multiply by the total memory bandwidth available to a process. For a modern socket with 4 memory channels of 1.866 GB/s each, this converts $1\mu s$ into about $4 * 1866 = 7464$ bytes of bandwidth.

γ for this reduction should reflect an atomic double float to memory from the incoming data. This is notionally 5 memory accesses of 8 bytes each per 8 bytes of message: a write into the message buffer followed by a read and a write into a user space buffer, a read from that buffer, a read from the target y location, and a final write to the target location.[4] This assumes that no extra memory locations are needed to manage the atomicity of the update. Thus γ is the equivalent time to about $5 * 8/8 = 5$ bytes of memory bandwidth per byte of input. Since it is likely that at least page-sized pieces of the y in a process is in a single memory channel, it is probably appropriate to multiply this by the number of memory channels to convert it into a realistic number.

It is likely that the gap between messages is smaller than the processing term, so $g(m_s)$ can be ignored.

For our problem $n_s = (8N/p)/m_s$, and we approximate $o_s(m_s) = o_r(m_s) = o$. Also, if we define M as the memory bandwidth available to a process in B/s, then we can convert Eqtn. 3 from time into an equivalent memory load in bytes of:

$$M(P-1)(L+o+\gamma m_s) + (8NM/pm_s - 1)(2o+\gamma m_s) \tag{5}$$

One consideration for the above equation is the $(P-1)$ term which says that each of the $P-1$ processes in the row other than the column 0 process add a separate latency of L to the total. This may not be rational, as its probable that the end-to-end latency from each process is highly overlapped in some way as they all start at roughly the same time. Thus we may want to consider replacing $(P-1)L$ by simply L, but leave the $(P-1)$ in front of the other terms, as they are largely serialized on the target process. However, when P is large, this latter term may get excessive, and it may be worth considering an *MPI_reduce* implementation that performs a log reduction. In this case, assuming a base Q reduction, each level would require a separate L, but only $(Q-1)$ repetitions of the latter terms. This leads to the following cases for Eqtn 5:

[4]It is possible that the latter two are satisfied by cache. With N/p y values this is thus between $8N/p$ and $24N/p$. It is also possible that all three accesses may even be overlapped.

$$No\_Overlap: \; M(P-1)(L+o+\gamma m_s) \tag{6}$$
$$+(8NM/pm_s - 1)(2o+\gamma m_s) \tag{7}$$
$$Overlap: \; ML + M(P-1)(o+\gamma m_s) \tag{8}$$
$$+(8NM/pm_s - 1)(2o+\gamma m_s) \tag{9}$$
$$Log_Q: \; M * log_Q(P)(L+o+\gamma m_s) \tag{10}$$
$$+log_Q(P) * Q(8NM/pm_s - 1)(2o+\gamma m_s) \tag{11}$$
$$\tag{12}$$

Finally, the second collective is a barrier at the end between all p processes in column 0. The same "PLogP" model from [7] estimates that such time is again a function of L, with an extra factor of 2 for the return from the barrier. This would add a term $2ML$ to the above.

2.4 A Common Model

For an estimate of the memory load on a process in column zero in terms of memory load, we can add Eqtns. 1 and one of 6, along with a term $2ML$. Fig. 5 diagrams these three models as speedup numbers, where the time for the reference single process is in fact that for just the local computations, with no reduce or barrier.

For the Non-Overlapped and Overlapped cases, the speedups for the denser cases are fairly good up to a certain point (about $6x6$ for the Non-Overlapped and $10x10$ for the Overlapped), after which speedups actually decline. This is due to the sparsity per local row in each process dropping below a critical point where overhead now dominates, as the number of processes that have to be serially summed together into the column 0 becomes large. The sparsest case is even worse, and in fact gains very little until a moderate amount of parallelism is present.

The Log case is most inserting in that it matches the shapes of Fig. 1 very well, at least up to the limited process array sizes in Fig. 1. Thus we suspect that the MPI implementation used in [5] was probably an "optimizing" version that performed a logarithmic approach to multiple reductions. In particular, this set of curves includes the pronounce dip in performance for the sparsest case at small numbers of concurrency. Further, this case does not experience the actual declines in speedup at larger systems, but essentially goes flat.

3 EXPERIMENTAL IMPLEMENTATION

3.1 Work Distribution and MPI

Communication and computational behavior of a multi-nodal solver developed for a previous study [5] was used as the basis for our implementation Unlike [5], we chose to forgo the use of proprietary libraries such as BLACS and MKL deciding instead to write explicit MPI and OpenMP directives to control distributed and shared memory behavior across the cluster environment. It was felt that exclusion of these libraries provided, greater control over program operation throughout our evaluation. Special care was paid to insure that the communication pattern matched that of the 2D cluster methods in the BLACS library which were used in the prior work.

Matrices, stored in Matrix Market Format (*.mtx), are read from file by the master MPI process and converted to Compressed Sparse Row (CSR) or Compressed Sparse Column (CSC) format, depending

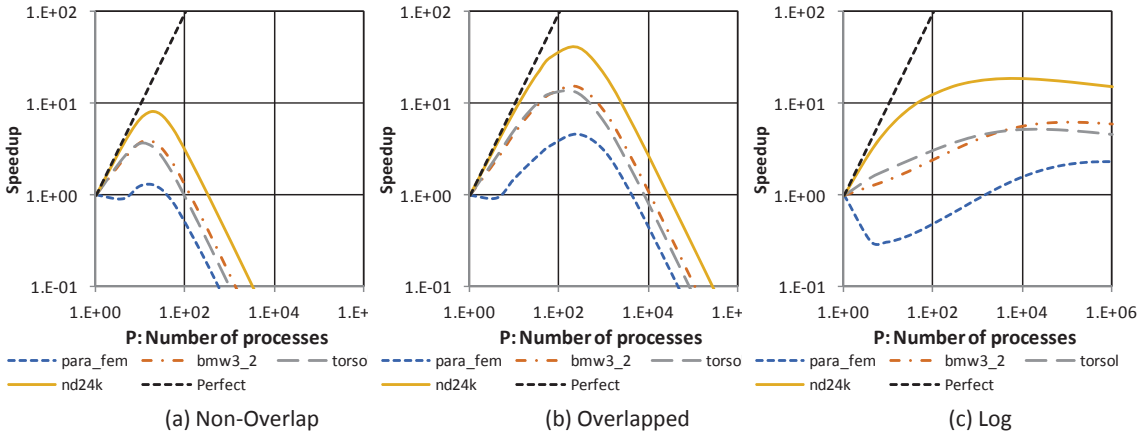

Figure 5: The Three Models.

on the input matrix. CSR and CSC formats provide a reduced memory requirements, thereby increasing performance while reducing data transfer in multi-node environments since only data about non-zero elements in the input matrix are kept. While the matrix is being read in, the distribution of work amongst the process matrix is being determined. The *sub matrix method* used in [5] breaks the input matrix A into p^2 sub matrices with nearly identical dimensions based on the number of rows/cols in teh process matrix and the row/col count of A. We chose to require that p be a non-negative square value, therein the size of each sub-matrix from A_p will be A_{rows}/p x A_{cols}/p.

The csrSpMV class was created to contain sub-matrix information in CSR/CSC format so that work allocation across MPI processes could be performed prior to MPI communication amongst the process matrix taking place. As each non-zero is read, its row and column are used to determine which A_p it will be assigned to, and is subsequently added to the csrSpMV object representing that sub-matrix

The first row of the process matrix containing processes with global MPI ranks 0 to $(p-1)$ are termed *column masters*, while the first column of processes with global rank such that $rank\ \%\ \sqrt{p}$ are *row masters*. The MPI master process sends every column master all data to be distributed amongst its column. Column masters will then transmit each process in its column their individual work allotment, wherein after receipt each process proceeds to computation.

It is important point out considerable work imbalances including process with no work can occur between processes in the *sub matrix* distribution method. As discussed in [5] such imbalances can lead to entire computing nodes sitting idle as they have no data to process, thereby potentially decreasing overall performance.

3.2 OpenMP SpMV

Algorithm 1 shows the procedure used to perform the multi-threaded SpMV computation within the OpenMP pragma. Each process performs this algorithm with the number of OpenMP threads established at runtime via command line parameters, along with several shared and private variables that can be accessed by an individual thread.

Algorithm 1 Hybrid SpMV

```
1:  procedure OpenMP SpMV
2:      Input: csrSpMV nodeCSR, int rowsPerThread
3:      threadId ← omp_get_thread_num()
4:      rStart ← threadId * rowsPerThread
5:      if threadId == threadCount − 1 then
6:          rEnd ← nodeCSR.Rows.size()
7:      else
8:          rEnd ← (threadId + 1) * rowsPerThread
9:      end if
10:     for i ← rStart − rEnd do
11:         dStart ← nodeCSR.Rows[i]
12:         if i == rEnd − 1 then
13:             dEnd ← nodeCSR.Data.size()
14:         else
15:             dEnd ← nodeCSR.Rows[i + 1]
16:         end if
17:         for j ← dStart − dEnd do
18:             result[i] += nodeCSR.Data[j] * nodeCSR.denseVec[i]
19:         end for
20:     end for
21: end procedure
```

The csrSpMV object containing that particular processes data, called *nodeCSR*, is shared amongst all threads as is the *result* vector. It is appropriate to share data in such a manner since data contained within *nodeCSR* is only read from, while individual threads access only those elements of *result* corresponding to the rows to which they have been assigned. All other variables explicitly listed as private are necessary to ensure each thread has a copy within its memory space without the possibility of being overwritten. Each process carries out the SpMV procedure once it has acquired all data from its column master and is ready to proceed with computation.

Table 1: Properties of Benchmark Matrices

Name	Rows/Cols	Non-Zeros	Density
parabolic_fem	525825	3674625	6.98
bmw3_2	227632	5757996	49.59
torso1	1116158	8516500	73.32
nd24k	72000	28715634	398.83

3.3 Reduction and Validation

Upon completion of computation all processes within a row perform an MPI reduction in which results are summed and stored within the row master's *result* vector. After having performed the reduction a gather is performed on the process column containing the global master process (a row master), as well as all other row masters. Finally the global master process has all results and can proceed with secondary computation if necessary.

During development it was necessary to ensure that the hybrid portion of the program was computing the correct SpMV result for a given matrix A and a dense vector. In order to verify accuracy of the hybrid version, a sequential version of the SpMV algorithm was performed on the master process only, prior to the hybrid portion of the program being performed. The results from each method were then compared and any differences indicated an error in computation. This was performed with a series of matrices, increasing in size, until no discrepancies were found amongst the test matrices. With the validity of the hybrid algorithm's communication and computation tested/verified, the sequential computation was removed so that evaluation could be proceed.

4 EVALUATION

4.1 Cluster Environment and Methodology

For our tests we utilized a 64 node computing cluster utilizing IBM NeXtScale nx360 M4 servers each with dual 8-core Intel(R) Xeon(R) CPU E5-2650 v2 at 2.60GHz. The hardware specifications for each server provided two sockets and 16 cores per node, for a total of 1024 cores available within the cluster. Finally, cluster nodes were connected via Mellanox FDR non-blocking Infiniband. Our version of the sub matrix method was compiled on the test environment using the mpic++ compiler for OpenMPI 2.0.1, with pass through to gcc/g++ 6.2.0 and OpenMP version 4.5.

Time measurements taken during tests were gathered using the MPI_Wtime() method, with nanosecond clock precision, in all MPI process and OpenMP thread count per process variations examined. SpMV computation times were taken once each process had received its work allotment, and again after all nodes had completed their computation. The same method was used for acquiring reduction times, however with the addition of an *MPI_Barrier* prior to determining the find total reduction time. This was done to ensure time measurements incorporated potential workload imbalances generated by the sub matrix distribution method.

During our tests, we utilized the matrices as in [5]. Table 1 lists the properties of each matrix including the number of rows/cols, total number of non-zeros, as well as the matrix's density. Density for this purpose is calculated as $rows/non - zeros$. Additionally all matrices being evaluated are square nxn symmetric matrices. Each matrice's symmetric MMF file was expanded to obtain a coordinate MMF file containing an entry for every *nnz* in the full matrix.

4.2 Impact: Thread Count

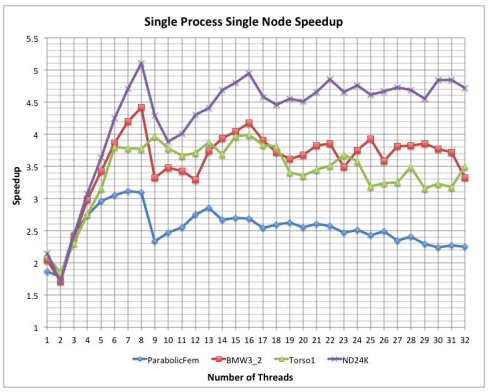

Figure 6: Single Node Speedup

Where computation is bound by memory references and cache hits capitalizing upon multi-core utilization can greatly increase performance [10]. In hybrid codes optimizing performance exhibited from a individual node is essential for overall execution. Figure 6 illustrates the impact of increasing thread count locally on a single node due to the increased memory access as the socket cores become saturated with work. A peak can be seen near 8 threads with another visible peak at 16.

Once all cores on a single processor are assigned a thread that chips memory bandwidth as been fully allocated thereby achieving the maximum on chip memory bandwidth. Beyond 8 threads an increase in overhead from the distribution of work amongst the disjoint memory profiles on each chip degrades performance until total memory bandwidth allocation overcomes this overhead. Over subscription of node sockets/cores produces stagnation and declines in speedup across all matrices examined. It is worth noting that the code generated for these experiments made no effort to control thread core or socket affinity, therefore it is highly likely that scheduling in an oversubscribed node is a cause for declining performance.

4.3 Impact: Process Count

Increasing the number of processes eventually requires the use of additional nodes, thus introducing communication overhead. We examined the performance impact of scaling up p such that a node had between 1 and 16 processes. Subsequently we scaled thread count so that all available processing hardware on a node was used during the study. The performance characteristics for 2 MPI processes per node (where applicable) with varying total process count and threads per process are visible in Fig. 7.

For all but the least sparse matrix performance accelerates quickly until 9-16 processes, followed by a sharp decline as overhead from the non-logarithmic *MPI_Reduce* outpaced the benefit of additional processes. These results agree with the analytical model from

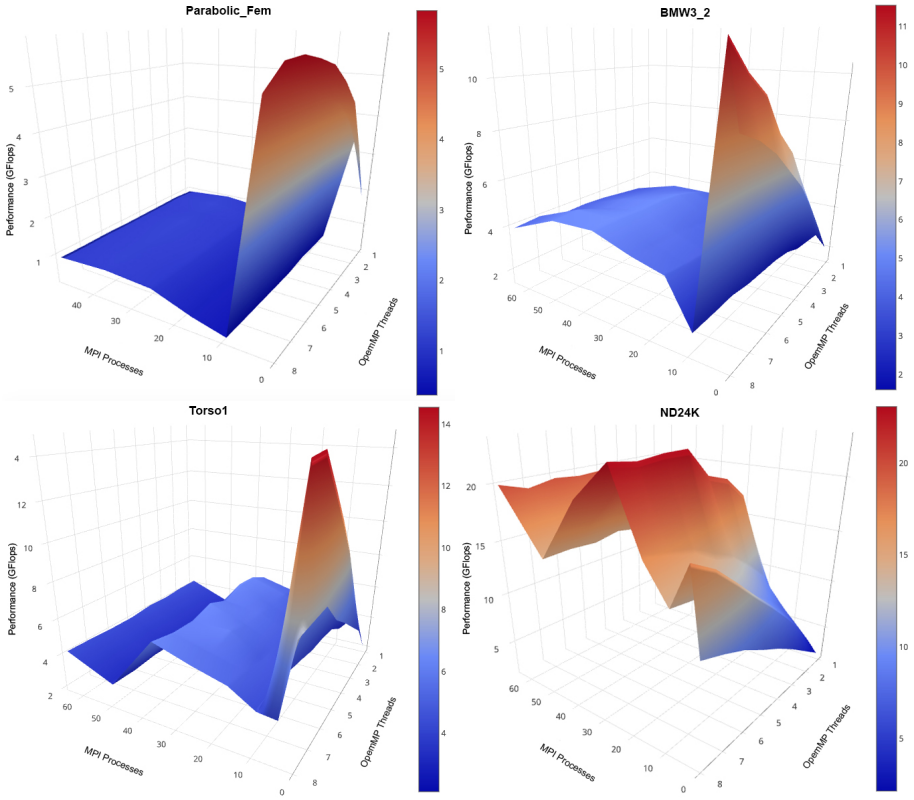

Figure 7: Hybrid SpMV Performance for Matrices with varying Sparsity

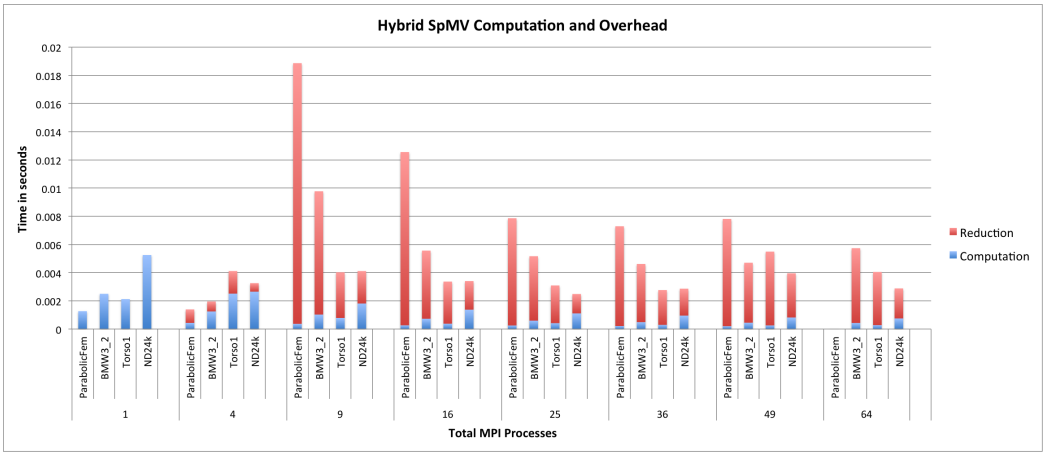

Figure 8: Computational Time and Reduce + Barrier Overhead

?? where in such a decline is exhibited in Fig. 5(a) for the Non-Overlapping case. Due to an inability to increase p beyond 64 total processes (8x8 process matrix), our results did not indicate if nd24k would experience similar behavior, however, due to the impact sparsity has on performance at scale, it is likely that the trend exhibited in 5(a) would hold true.

4.4 Impact: Sparsity

The extent to which multi-core, mutli-node, or hybrid strategies are capable of producing increased performance is widely dependent on quantity and distribution of *nnz* within a matrix. In the single process single node case, Fig. 6, only one the sub-matrix is created where $A_{ij} = A$. Given that the Algorithm 1 requires entire rows

within a sub-matrix assigned to a process to be operated on by a single OpenMP thread, there is no decline in the *nnz* per row.

In contrast as the number of MPI processes is increased, *nnz* per row within a sub-matrix A_{ij} decreases proportionally to p. For example Parabolic_Fem's initial *nnz* per row of 7 when process count increases from 1 to 4, doubling the number of cols in the process matrix, and reduces *nnz* per row via $density_{A_{ij}}/p$. As can be seen in Fig. 7 matrices with fewer *nnz* per row, exhibit decline in multi-nodal performance more rapidly than those with greater initial starting *nnz* pder row. In the case of Parabolic_Fem, once $p > 6$ *nnz* per row is less than 1, leading to work imbalances across processes and lost performance as a result.

Subsequently while the average *nnz* per row decreases quickly, overhead associated with reduction accross the larger process count in addition to the *MPI_Barrier* halting overall progress, begins to outpaced computation speedup seen from acquiring additional processes or nodes. Figure 8 outlines that while computational time for SpMV may decrease, reduction and barrier weigh down overall run-times The gradual decrease in overhead seen is a symptom of decreased message size between each process and its *rowmaster*, however a process with nothing to do will continue to have more overhead than productive computation.

5 FUTURE WORK

In the future this study will benefit from analysis at greater scale, with a wider range of matrix sparsity and *nnz* distribution. Alternatively the impact of various decomposition methods beyond *submatrix*, including balancing and work stealing methods, will be explored. While new distribution methods are being used new matrix storage formats can be introduced in an effort to find optimal pairing amongst the two design decisions. Clearly, alternative ways for performing reductions is essential. Lastly this study was undertaken with the ultimate goal of evaluating these design techniques upon new architectures in use as well as entering the pipeline, such as GPUs and the Intel Phi (Knights Landing).

6 CONCLUSIONS

In this study, a analytical model for approximating hybrid SpMV performance and speedup was created and examined via the reproduction of a prior multi-core/multi-nodel implementation [5]. From the analysis performed, SpMV computations are predominantly bound by the memory characteristics of the architecture being utilized, with overall scalability affected strongly be sparsity and the resulting mal-distribution of work among processes, and by the costs of combining final sums via reductions. The *submatrix* distribution method while inherently simplistic to program, is inherently prone to imbalance, thereby leading to differences in performance on a per matrix basis. However, as mentioned, this requires additional study.

REFERENCES

[1] J. Dongarra and M. Heroux, "Toward a new metric for ranking high performance computing systems," Sandia National Labs, Sandia Report SAND2013 4744, June 2013. [Online]. Available: http://www.sandia.gov/~maherou/docs/HPCG-Benchmark.pdf

[2] J. Kepner and J. Gilbert, *Graph Algorithms in the Language of Linear Algebra*, J. Kepner and J. Gilbert, Eds. Society for Industrial and Applied Mathematics, 2011. [Online]. Available: http://epubs.siam.org/doi/abs/10.1137/1.9780898719918

[3] [Online]. Available: http://graphblas.org/index.php?title=Graph_BLAS_Forum

[4] W. S. Song, V. Gleyzer, A. Lomakin, and J. Kepner, "Novel graph processor architecture, prototype systems and results," *HPEC COnference*, 2016.

[5] B. Bylina, J. Bylina, P. Stpiczynski, and D. Szalkowski, "Performance analysis of multicore and multinodal implementation of spmv operation," in *Computer Science and Information Systems (FedCSIS), 2014 Federated Conference on*, Sept 2014, pp. 569–576.

[6] V. Marjanović, J. Gracia, and C. W. Glass, "Performance modeling of the hpcg benchmark," in *High Performance Computing Systems. Performance Modeling, Benchmarking, and Simulation.* Springer International Publishing, 2014, pp. 172–192.

[7] J. Pješivac-Grbović, T. Angskun, G. Bosilca, G. E. Fagg, E. Gabriel, and J. J. Dongarra, "Performance analysis of mpi collective operations," *Cluster Computing*, vol. 10, no. 2, pp. 127–143, Jun. 2007. [Online]. Available: http://dx.doi.org/10.1007/s10586-007-0012-0

[8] D. Panda, "The mvapich2 project: Pushing the frontier of infiniband and rdma networking technologies," 2015. [Online]. Available: http://hibd.cse.ohio-state.edu/static/media/talks/slide/osc_theatre_talk_mv2.pdf

[9] D. Doerfler and R. Brightwell, *Measuring MPI Send and Receive Overhead and Application Availability in High Performance Network Interfaces.* Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 331–338. [Online]. Available: https://doi.org/10.1007/11846802_46

[10] G. E. Blelloch, R. A. Chowdhury, P. B. Gibbons, V. Ramachandran, S. Chen, and M. Kozuch, "Provably good multicore cache performance for divide-and-conquer algorithms," in *Proceedings of the nineteenth annual ACM-SIAM symposium on Discrete algorithms.* Society for Industrial and Applied Mathematics, 2008, pp. 501–510.