Scalability of Hybrid Sparse Matrix Dense Vector (SpMV) Multiplication

Brian A. Page
CSE Dept.
Univ. of Notre Dame
Notre Dame, IN USA
bpage1@nd.edu

***Abstract*—TBD**

***Keywords*-Scalability, Hybrid SpMV;**

I. Introduction

-scaling concerns -its not just adds and multiplies -number of memory accesses and moving data around has a great impact on performance!

II. Prior Work and Analytic Models

The HPCG benchmark [1] is one that is dominated time-wise by SpMV and similar kernels. Fig. 1 diagrams data taken from recent HPCG reports[1]. The x-axis is the peak flops of the reported system; the y-axis is the ratio of the sustained HPCG flops to the peak bandwidth of the systems's memory (derived by determining the processing chips used and looking up their characteristics). The color and shape refer to different types of chips and systems, with the red squares representing system built from server-class chips, and the purple representing system using GPUs.

As can be seen, this ratio is independent of the peak system flops capability. In fact it is relatively flat at about 0.1 flops per byte of memory bandwidth for heavyweight server class processor chips, and somewhat less than 0.1 for GPUs and other architectures. Since SpMV is the bulk of HPCG, this is an indication that SpMV is relatively independent of core floating point capability, and instead highly dependent on chip memory bandwidth.

A recent complexity analysis of the HPCG benchmark [2] dove into HPCG performance as a function of system parameter on a kernel-by-kernel basis. The particular implementation of HPCG that was studied assumed that a sub-matrix of the total matrix was processed in each MPI rank as executed by a single core. The study rolled these numbers up into total execution time for the whole benchmark as a function of just memory bandwidth and a few network parameters. The model was extremely accurate when compared to measured HPCG data on several benchmarks.

The analysis of just the SpMV kernel within HPCG focused on just the in-core time, and computed that each sub-row as executed by a single thread on a single core required a net of the following bytes fetched from memory[2], where nnz_{row} is the average number of non-zeros per row in the row as processed by each core:

$$20 \ + \ 20 * nnz_{row} \tag{1}$$

Since each non-zero represents two flops (an add and a multiply), dividing this into $2 * nnz_{row}$ yields an estimate of the bytes of bandwidth needed from memory for each flop:

$$2 * nnz_{row}/(20 \ + \ 20 * nnz_{row}) \ = \ 1/(10 \ + \ 10/nnz_{row}) \tag{2}$$

For a nnz_{row} of 27 this is about 0.096 flops per byte of bandwidth. This correlates well with HPCG, as the non-SpMV parts of HPCG require slightly more bytes per flop. Approximately 10 bytes must be accessed from memory for each flops executed.

Multiplying this by the actual sustainable memory bandwidth of a node should then estimate the sustainable flops per second for SpMV running in all the cores in that node. [2] uses in its projections the bandwidth number returned by using the Triad STREAM benchmark The first three rows of Table I summarize the characteristics of the three chips used in systems modelled by [2], including the ration of the reported STREAM bandwidth to the maximum memory bandwidth as projected by the chip's characteristics.

Due to the computation impact that SpMV operations have on a many scientific applications there has been an effort to analyze its performance and scalablity characteristics. Bylina, Bylina, Stpiczunski, and Szalkowski [3] introduced and evaluated the performance of both multicore and multinodal implementations of SpMV on various chip architectures. A modified version of the SpMV algorithm found in the SPARSKIT Fortran library [4] for the multicore implementation. Using matrices from the University of Florida Sparse Matrix Collection (UFSMC), they found that for their multicore algoritm, similar performance was experienced accross all matrices tested when the number of threads remained low. Alternatively as architectures allow

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

[2] The paper computed a value of 27 for the average number of non-zeros per row partition, and each non-zero required two 8-byte fetches of floating point data and one 4-byte index reference, with another 20 bytes for starting the computation of a new row.

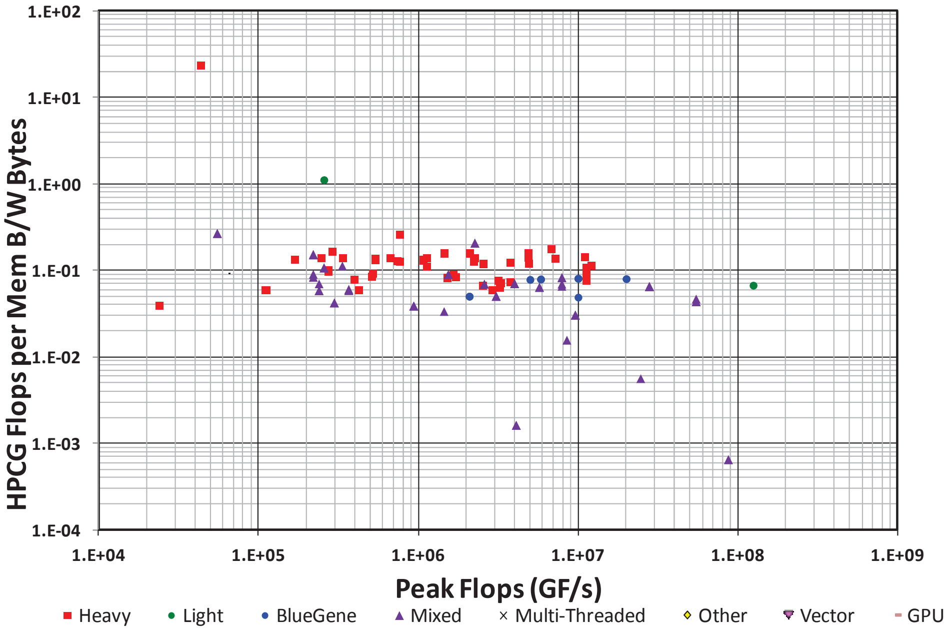

Figure 1. HPCG Flops per Byte of Memory Bandwidth vs. Peak flops.

	Chip Parameters				Node Parameters				SpMV Specific		
Chip Type	Cores	Total Memory Channels	Peak B/W (GB/s)	Peak Flops (GF/s)	Chips	Peak B/W (GB/s)	STREAM B/W (GB/s)	Ratio	nnz_{row}	Estimated SpMV (GF/s)	Measured SpMV (GF/a)
Chips used in Reference for HPCG Benchmark											
E5-2670	8	4	51.2	166.4	2	102.4	75.28	73.5%	27		
6276	16	4	51.2	147.2	2	102.4	54.4	53.1%	27		
X5560	4	3	32	44.8	2	64	27.44	42.9%	27		
Chips used in Reference for SpMV Benchmark											
X5650	6	3	32	63.84	2	64	N/A	N/A	6.98		1.9
E5-2660	8	4	51,2	140.8	2	102.4	N/A	N/A	6.98		5.3
Chips used in this paper.											
E5-2650v2	8	4	59.7	166.4							

Table I
SpMV Projection Based on System Parameters.

for increased thread count, higher performance can be obtained, and it was noted that the use of OpenMP allowed for performance comperable to that of the optimized Intel MKL version of SpMV cite Intel MKL ?.

Bylina et al's multinodal implementation distributed equal sized sub matrices of a given benchmark matrix to each MPI process, where each process would then work on the non-zeros contained within that submatrix via a multithreaded version of Intel's MKL SpMV routines. For the "submatrix" distribution method the density and distribution of non-zeros within a matrix has the greatest impact on the performance at scale of their multinodal algorithm.

Similar to Bylina et al Ariful Azad et al [5] explored the performance impact of multilevel parallelism of sparse matrix operations. While this particular work focused upon sparse matrix-matrix multiplication (SpGEMM) it did identify several characteristics inherent to 2D algorithms. Azad et al discussed the implementation of a 3D algorithm which utilized the concept of submatrix distribution, much like Bylina et al [3]. 2D decomposition is incorporated into their 3D decomposition method in an effort to further reduce data transfer and thereby increase performance by reducing overhead.

Algorithm design and matrix storage format have been at the heart of many research endeavors in an effort to find more optimal methods of performing sparse matrix operations [?], [6] couple morein this cite?. Aydin Buluc and John R. Gilbert took at look at SpMV with hyperspace matrices, that is matrices in which the number of non-zero elements was less then the number or rows in the matrix. The outcome was that storage formats such as CSR and CSC would be inefficient for such matrices due to the need to account for rows which dit not contain any non-zeros thereby generating overhead without adding to the floating point operations being performed during SpMV computation [6]. Much of this effort stems from the prevelance of multicore processors and the utilization of the submatrix distribution pattern performed after a 2D decomposition of the original matrix.

Having examined storage formats and decompositons strategies, the 2D decomposition and communcation pattern implemented by Bylina et al via BLACS and MKL was chosen for our analysis. In order to further analyze the impact on performance generated by the characteristics of a matrix, such as sparsity, and non-zero distribution, the matrices analyzed by Bylina et al will serve as the proof of correctness as we increase scale.

need to tie these papers into why you are choosing to use the Bylina paper as the method to reproduce and why it makes sense to do so.

tie in memory access stuff into these papers and the bylina paper, otherwise this whole section is disjoint peices!

III. Implementation

A. Work Distribution and MPI

The application that was written to emulate the behavior of that used in the study performed by Bylina et al was written using the C++ programming language. 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. 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. It was felt that by not including these packages greater control over communication and memory access parameters could be achieved, even though the result may not be as highly optimized for particular applications, architectures, or compilers.

Benchmark matrices in the Matrix Market Format were chosen from the University of Florida Sparse Matrix Collection. The characteristics of the matrices chosen is discussed in greater detail in section 4a ”Benchmarks”. Matrices are read from file by the master MPI process and converted to Compressed Sparse Row (CSR) or Compressed Sparse Column (CSC) format, depending on the individual matrix being input. Both CSR and CSC formats provide a reduction on memory requirements thereby increasing performance while reducing data transfer in multinode environments since only data about non-zero elements in the input matrix are kept this needs a cite. While the matrix is being read in, the distribution of work amongst the MPI processes is being determined. The distribution pattern used by Bylina et al and which we have emulated, we call the *sub matrix method*, splits the input matrix A into p^2 sub matrices in which each piece has nearly identical dimensions based on the number of processes p and the row or column count of A. Therein the size of each sub matrix from A_p will be $A_{rows}/\sqrt{p}$ x $A_{cols}/\sqrt{p}$. We chose to require that p be a non-negative square value.

The csrSpMV class was created to contain submatrix information in CSR/CSC format so that work allocation accross MPI processes could be performed prior to MPI communication amongst those processes taking place, and stores information about each non-zero in three vectors. As each new non-zero is read the sub matrix it is to be assigned to is easily determined from its row and column, and is subsequently added to the csrSpMV object representing that submtrix.

Given that there are the same number of processes as submatrices, p, we can view the processes as being laid out in a matrix P in which process P_{ij} will receive data corresponding to submatrix A_{ij}. 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 ranks such that $rank\%\sqrt{p}$ are *row masters*. The MPI master process sends every column master the data contained within the csrSpMV object containg all data to be destributed amongts its column. Column masters will then send information about non-zeros to each process in its column. Each process receives its work allotment, if any, and proceeds to computation.

It is important to note that given the *sub matrix* distribution method considerable work imbalances including process with no work can occur between processes. As discussed in Bylina et al such imbalances can lead to entire computing nodes sitting idle as they have no data to process, thereby potentially decreasing overall performance.

B. OpenMP SpMV

Algorithm 1 shows the procedure used to perform the multithreaded SpMV computation within the OpenMP pragma section of the program. Each process performs this algorithim using the number of OpenMP threads set by the administrator at runtime via command line parameters. The OpenMP pragma establishes several shared and private variables that can be accessed by an individual thread.

```
#pragma omp parallel num_threads(control.ompThreads) shared(nodeCSR, result) private(ompThreadId, start, end, i, j, rowsPerThread)
```

As seen in the pragma above, the number of threads created is a value set at runtime and stored in the ompThreads variable with the control structure which contains other paramaters needed for distribution and control of the application. The csrSpMV object containing that particular node’s data, called $nodeCSR$, is shared amongst all threads as is the $result$ vector. We can share these items since the data

Algorithm 1 Hybrid SpMV

```
procedure OpenMP SpMV
    Input: csrSpMV nodeCSR, int rowsPerThread
    $threadId \leftarrow omp\_get\_thread\_num()$
    $rStart \leftarrow threadId * rowsPerThread$
    if $threadId == threadCount - 1$ then
        $rEnd \leftarrow nodeCSR.Rows.size()$
    else
        $rEnd \leftarrow (threadId + 1) * rowsPerThread$
    for $i \leftarrow rStart - rEnd$ do
        $dStart \leftarrow nodeCSR.Rows[i]$
        if $i == rEnd - 1$ then
            $dEnd \leftarrow nodeCSR.Data.size()$
        else
            $dEnd \leftarrow nodeCSR.Rows[i + 1]$
        for $j \leftarrow dStart - dEnd$ do
            result[i] += nodeCSR.Data[j] * nodeCSR.denseVec[i]
```

contained with $nodeCSR$ will only read from, and each individual thread will only acess the elements of $result$ corresponding to the rows which it has been assigned to work on. All other variables explicitly listed as private are necessary to insure each thread has a copy within its memory space without the possibility of being overwritten. Each process carries out the SpMV procedure once it is acquired all data from its column master and is ready to proceed with computation.

C. Reduction and Validation

Upon completion of computation all nodes within a row perform an MPI recution where each processes results are summed and stored within the row master's *result* vector. This is possible as each processes within the same row of the process matrix are working on the same rows from A, but only on those non-zero elements contained within their assigned submatrix from A. After having performed the reduction a gather is performed on the process column contaning the global master process (also a row master), and all other row masters. At this point the global master process now has all results and can proceed with secondary computation if necessary.

During development it was necessary to insure 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 descrepencies were found amongst the test matrices. With the validity of the hybrid algorithms communication and computation tested and verified, the sequential master only computation was removed so that benchmarking tests could be performed.

IV. Evaluation

-our program behaves the ”same” as the Bylina et al paper.

- cluster architecture, network connectivity, number of nodes (max used), etc. - how timings where taken and GFlops calculated based on these time measurements. - note clock precision (nanoseconds) - number of tests run per test permuation - best, worst, and averages were taken for all times recorded and GFlops calculated

- then tables and pretty graphs to talk about

- As we add threads, we are adding memory access (bandwidth) due to the usage or more cores on a chip

- as we increase MPI processes we increase the number of processors, and threfore the potential for more threads.

-performance as a function of the number of non-zeros per row. What I mean by this is that as you increase the number of processes, you split up rows into smaller sections, thereby decreasing the of non-zero elements in a given row for each submatrix. eventually there will be so few non-zeros that you will have nothing to do and only overhead to keep you busy (performance will drop).

Trade off space: - # of processes - threads per process - # of processes per socket - # of nodes utilized - matrix size - # of non-zeros

A. Benchmarks

why Parabolic_Fem, bmw3_2, torso1, and nd24k were chosen. -sparsity/density -size -symmetry

- how the GFlops will be calculated for symmetric and non-symmectric matrices (they are the same, however the Bylinia paper did some weird shit with their numbers!)

B. Impact: Process Count

C. Impact: Thread Count

asdfsadf

D. Impact: Sparsity

V. Future Work

- run computation on larger number of nodes/cores - wider range of sparsitys and non-zero distribution within matrices - explore performance impact of different distribution/decomposition methods - possible explore the impact of different storage formats within the different distribution methods - gpus/knights landing

VI. Conclusions

- careful hybrid coding can give you better performance than MPI or OpenMP alone.

Acknowledgment

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] 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.

[3] 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.

[4] Y. Saad, “Sparskit: A basic tool kit for sparse matrix computations,” *Proceedings of the 2014 Federated Conference on Computer Science and Information Systems*, vol. 2, May 1990.

[5] A. Azad, G. Ballard, A. Buluç, J. Demmel, L. Grigori, O. Schwartz, S. Toledo, and S. Williams, “Exploiting multiple levels of parallelism in sparse matrix-matrix multiplication,” *SIAM Journal on Scientific Computing*, vol. 38, no. 6, pp. C624–C651, 2016.

[6] A. Bulu and J. R. Gilbert, “On the representation and multiplication of hypersparse matrices,” *2008 IEEE International Symposium on Parallel and Distributed Processing*, pp. 1–11, 2008.