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

II. Related Work

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 [1] 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 [2] 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 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 in 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 distribution method chosen the density in addition to distribution of non-zeros within the matrix had the greatest impact on the scalability of their multinodal algorithm.

-discuss memory bandwidth as it impacts SpMV

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 Hybrid SpMV

```
 1: procedure OpenMP SpMV
 2:     Input: csrSpMV nodeCSR, int rowsPerThread
 3:     $threadId \leftarrow omp\_get\_thread\_num()$
 4:     $rStart \leftarrow threadId * rowsPerThread$
 5:     if $threadId == threadCount - 1$ then
 6:         $rEnd \leftarrow nodeCSR.Rows.size()$
 7:     else
 8:         $rEnd \leftarrow (threadId + 1) * rowsPerThread$
 9:     for $i \leftarrow rStart - rEnd$ do
10:         $dStart \leftarrow nodeCSR.Rows[i]$
11:         if $i == rEnd - 1$ then
12:             $dEnd \leftarrow nodeCSR.Data.size()$
13:         else
14:             $dEnd \leftarrow nodeCSR.Rows[i + 1]$
15:         for $j \leftarrow dStart - dEnd$ do
16:             result[i] += nodeCSR.Data[j] * nodeCSR.denseVec[i]
```

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 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

- 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

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

VI. Conclusions

Acknowledgment

References

[1] B. Bylina, J. Bylina, P. Stpiczyski, and D. Szakowski, “Performance analysis of multicore and multinodal implementation of spmv operation,” *Proceedings of the 2014 Federated Conference on Computer Science and Information Systems*, vol. 2, pp. 569–576, Oct 2014.

[2] 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.