Distributed Algorithms for Sparse Matrix-Vector Multiplication

Eric Hare

April 28, 2015

Introduction

Sparse Matrix-Vector Multiplication (SpMV) is a widely used and widely explored problem. It is intrinsically parallelizable, but naive algorithms typically have very poor performance as the size of the matrix, or its sparsity drastically increases. This is because threads which handle the rows containing all or mostly all zeroes have far less work than the threads handling the dense portions of the matrix.

In this paper, I survey work that has been done to optimize the performance of these calculations. I've selected three papers which detail distributed algorithms. The three papers are:

1. Load-Balanced Sparse Matrix–Vector Multiplication on Parallel Computers by Nastea, Frieder, and El-Ghazawi (Gettys, Karlton, and McGregor 1997)
2. An architecture-aware technique for optimizing sparse matrix-vector multiplication on GPUs by Maggioni and Berger-Wolf (Maggioni and Berger-Wolf 2013)
3. Scalable Parallel Matrix Multiplication on Distributed Memory Parallel Computers by Li (Li 2001)

This work is relevant to my research in statistics, so I will begin by highlighting a situation in which such calculations are used in a statistical framework, before proceeding to highlight the novel contributions of each paper.

Background

Linear models in statistics are used when we wish to use several predictor variables in order to highlight a relationship with some response variable. A simple linear model may have only a single predictor variable, but in more complex applications, there may be thousands. The predictor variables are typically represented as a design matrix X, where the rows correspond to the number of observations in the data, and the columns correspond to the different features. In matrix/vector notation, we can define a typical linear model in statistics as:

$y = X\beta + \epsilon$

Where:

- y is an $n \times 1$ response vector
- X is an $n \times k$ design matrix
- β is a $k \times 1$ parameter vector
- ϵ is an $n \times 1$ error vector (that is, $\epsilon \sim MVN(0, \Sigma)$)

We wish to select estimates for β that minimize the squared error of this function, and in doing so, obtain the result:

$b = (X^T X)^{-1} X^T y$

Suppose we have the data given in Table 1:

	cty	hwy
1	18	29
2	21	29
3	20	31
4	21	30
5	16	26

Table 1: An example of five rows from a dataset containing city and highway miles per gallon.

We wish to use the highway mpg to predict the city mpg. Note that typically in linear models we also include an intercept term, or a column of 1s in the design matrix X. Then we have:

$$y = \begin{bmatrix} 18 & 21 & 20 & 21 & 16 \end{bmatrix}^T$$

$$X = \begin{bmatrix} 1 & 29 \\ 1 & 29 \\ 1 & 31 \\ 1 & 30 \\ 1 & 26 \end{bmatrix}$$

$$\beta = \begin{bmatrix} \beta_0 \\ \beta_1 \end{bmatrix}$$

$$\epsilon = \begin{bmatrix} \epsilon_1 & \epsilon_2 & \epsilon_3 & \epsilon_4 & \epsilon_5 \end{bmatrix}^T$$

$$(X^T X)^{-1} = \begin{bmatrix} 60.2714 & -2.0714 \\ -2.0714 & 0.0714 \end{bmatrix}$$

$$X^T y = \begin{bmatrix} 96 \\ 2797 \end{bmatrix}$$

$$b = (X^T X)^{-1} X^T y = \begin{bmatrix} -7.7286 \\ 0.9286 \end{bmatrix}$$

Hence, the equation for the line of best fit is given by $y = -7.7286 + 0.9286x$. Note that to derive the least squares estimators of the parameter vector β, we needed to compute the quantity $X^T y$. In this simple example, such a computation is trivial. But in a typical setting, there may be thousands of columns of X corresponding to thousands of different parameters in the parameter vector. It is not uncommon that X might be very sparse. For example, in feature selection applications, there may be features corresponding to a couple of observations, when the total size of the dataset is many orders of magnitude larger. The design matrix would contain 0s for each feature not present at a particular observation.

To help illustrate the increasing complexity, consider the following block of code, which reads a dataset consisting of fitness data derived from wearable computers. The dataset consists of 4024 observations of 159 variables.

```
wearables <- read.table("data/Example_WearableComputing.csv", sep = ";", header = TRUE)
X <- apply(wearables[,-1], 2, as.numeric)
y <- as.numeric(wearables[,1])
```

```
system.time(t(X) %*% y)
```

```
##    user  system elapsed
##       0       0       0
```

But now note what occurs when the number of observations is a replicated to be 500 times larger.

```
biggerX <- do.call("rbind", replicate(500, X, simplify = FALSE))
biggery <- rep(y, times = 500)

system.time(t(biggerX) %*% biggery)
```

```
##    user  system elapsed
##   0.017   0.000   0.018
```

There are several limitations of how this calculation is implemented in R. First, there is no explicit parallelism, only thread-level parallelism implemented by the OpenMP libraries. Second, although R supports parallel processing defined explicitly, even if such an algorithm was constructed, there would be no load balancing or inter-processor communication, which makes such an algorithm far less useful for the sparse matrix case. Third, the parallel framework within R allows a choice between a multi-core (single processor) or a multi-machine (single-core) parallelism, and doesn't exploit both. Finally, because of the sparsity, there is poor cache locality related to the accesses of the elements.

Load Balanced Sparse Matrix-Multiplication

Overview of Algorithm

I elected to begin my survey of the literature by discussing one of the earlier proposed solutions to this problem. Load-Balanced Sparse Matrix–Vector Multiplication on Parallel Computers by Nastea, Frieder, and El-Ghazawi explores a greedy allocation algorithm for sparse pattern matrices. The paper begins by laying out a set of assumptions for the calculations:

- $Y_i = AX_i$ where A is the sparse matrix, and X_i is a sequence of dense vectors
- The size of the sequence of X_i vectors is very large and not a priori known
- The resulting Y_i vectors are generated and transmitted on an individual basis

The fundamental aspect to the algorithm is to average the load distributed to each processor. In particular, define the following quantities:

$F = max_i\{\sum_{j=1}^{M}(nZ_{i_j})\}$

$i = 1, 2, ..., P$ represents the processors.

$i_j = \{i_1, i_2, ..., i_M\}$ represents indices of rows assigned to processor i

nZ_{i_j} is the number of non-zero elements in these rows.

Minimizing the function F amounts to minimizing the maximum number of non-zero elements assigned to a processor, and hence amounts to minimizing the largest computation time. Minimizing this function averages

out the load distributed to each processor. Note that this assumes that the algorithm knows a priori the number of non-zero elements in each row. As discussed in the paper, the overhead necessary to compute this is minimal relative to the gains the the algorithm provides.

One further optimization discussed is the idea of row-splitting. If a matrix is highly skewed as well as highly sparse (that is, the non-zero elements tend to occur in blocks rather than in an even distribution throughout the matrix), significant gains can be realized by splitting the row and assigning each split to a different processor. Note that this incurs some additional overhead as a new set of indices must be kept track of. Because of this overhead, the authors recommend that this only be done in the most extreme cases of sparsity and skewness.

The full pseudo-code of the algorithm is reproduced in Figure 1.

```
Procedure 1: GALA
INPUT:
row_index[N] - row indices sorted in decreasing order w.r.t. the number of non-zero elements;
row_size[N] - contains row sizes (ordered in the same way);

OUTPUT:
alloc[P,N] - a P x N array that contains the index of rows allocated to each  node, where P is the number of
processors;
rows_allocated[P] - stores the number of rows allocated to each processor;

ALGORITHM:
bucket_size[P] - stores the size of each processor bucket.
for (i=0, P-1)  /* initialization */
     bucket_size[i]=row_size[i]; /*  initialize the bucket of each processor with one row */
     rows_allocated[i]=1;
    alloc[i][0]=row_index[i];
endfor
for (i=P, N-1)
    size=bucket_size[0];
    greedy_proc=0;
    for (j=1, P-1) /* find  the "greediest" processor */
        if  (bucket_size[j] < size)
            size=bucket_size[j];
            greedy_proc=j;
        endif
    endfor
    bucket_size[greedy_proc]+=row_size[i]; /* adjust the "greediest" processor bucket size */
    k=rows_allocated[greedy_proc];
    alloc[greedy_proc][ k]=row_index[i];  /* add  row index in the list of  row indexes */
     rows_allocated[greedy_proc]++;   /* adjust pointer to array alloc[ ] */
endfor
```

FIG. 3. Greedy allocation-based load-balancing algorithm (GALA).

Figure 1: Pseudo-code for the greedy algorithm of the first paper. The algorithm performs operations on a major node (a leader). The leader uses the GALA routine to assign the most dense remaining rows to the processors with the minimum current workload.

Results

The performance improvements were evaluated using four different matrices of slightly different characteristics, and performing the simulation on an Intel Paragon supercomputer. Figure 2 gives a table describing the different properties of the test matrices. Note the sparsity value is given as the proportion of non-zero elements. Hence, all of the matrices test are quite sparse, containing at most 5.55% non-zero elements.

TABLE I
Statistical Data on Sparse Matrices

					Statistical data		
Name	Type	Order	Nonzero's	Sparsity	Average	Std. dev.	C.O.V.
ORANI 678	Unsymmetric	2529	90158	0.014	35.650	87.962	2.4674
PSMIGR 1	Unsymmetric mostly block-diagonal	3140	543162	0.055	172.981	235.575	1.362
BCSSTK28	Symmetric	4410	219024	0.011	49.665	9.682	0.195
zipf0.1 (synthetic)	Unsymmetric skewed	3500	49000	0.004	14.0	94.594	7.757

Figure 2: Matrices used for the evaluation of the algorithm. In particular, note that the last matrix is a synthetically generated matrix from a Zipf distribution. This distribution simulates sparsity and skewness characteristics that GALA is most effective at handling.

Four different algorithms were compared. The four algorithms are:

1. **Block** - In this allocation, no attempt at any load balancing is done. Each processor is given a contiguous block of rows for processing. In the paper, this is called the unbalanced case. Although it is balanced in terms of number of rows, it is unbalanced in terms of workload given any sort of sparsity or uneven distribution of non-zero elements.
2. **Cyclic** - Each row i is allocated to processor i mod P, where P is the number of processors. This is considered by the authors to be an example of naive load balancing, because it rests on the assumption that the distribution of non-zero elements maintains a continuous pattern throughout the matrix.
3. **Aliaga** - An iterative load balancing algorithm that generates swaps of matrix rows among processors to gradually smooth the maxima and minima of load. This algorithm has a time complexity that depends on the distribution of the data.
4. **GALA** - The algorithm presented in this paper.

The speedup as a function of the number of processors is presented in Figure 3. The top left box corresponds to the first matrix tested. Note that all but the block algorithm perform well in handling a sparse unsymmetric matrix, assuming that the distribution is not too skewed. The unsymmetric mostly block diagonal matrix has the best speedup as a function of the number of processors for both the Gala and the Aliaga algorithms, likely due to its relative density compared to the other three. The best results for the Gala algorithm come in the scenario of the fourth matrix, in which the Aliaga algorithm asymptotes at about a 12x speedup regardless of whether 20 or 40 processors are used. The Gala algorithm, meanwhile, has about a 22x speedup at 40 processors. This illustrates the success of the algorithm in handling skewness in the data distribution.

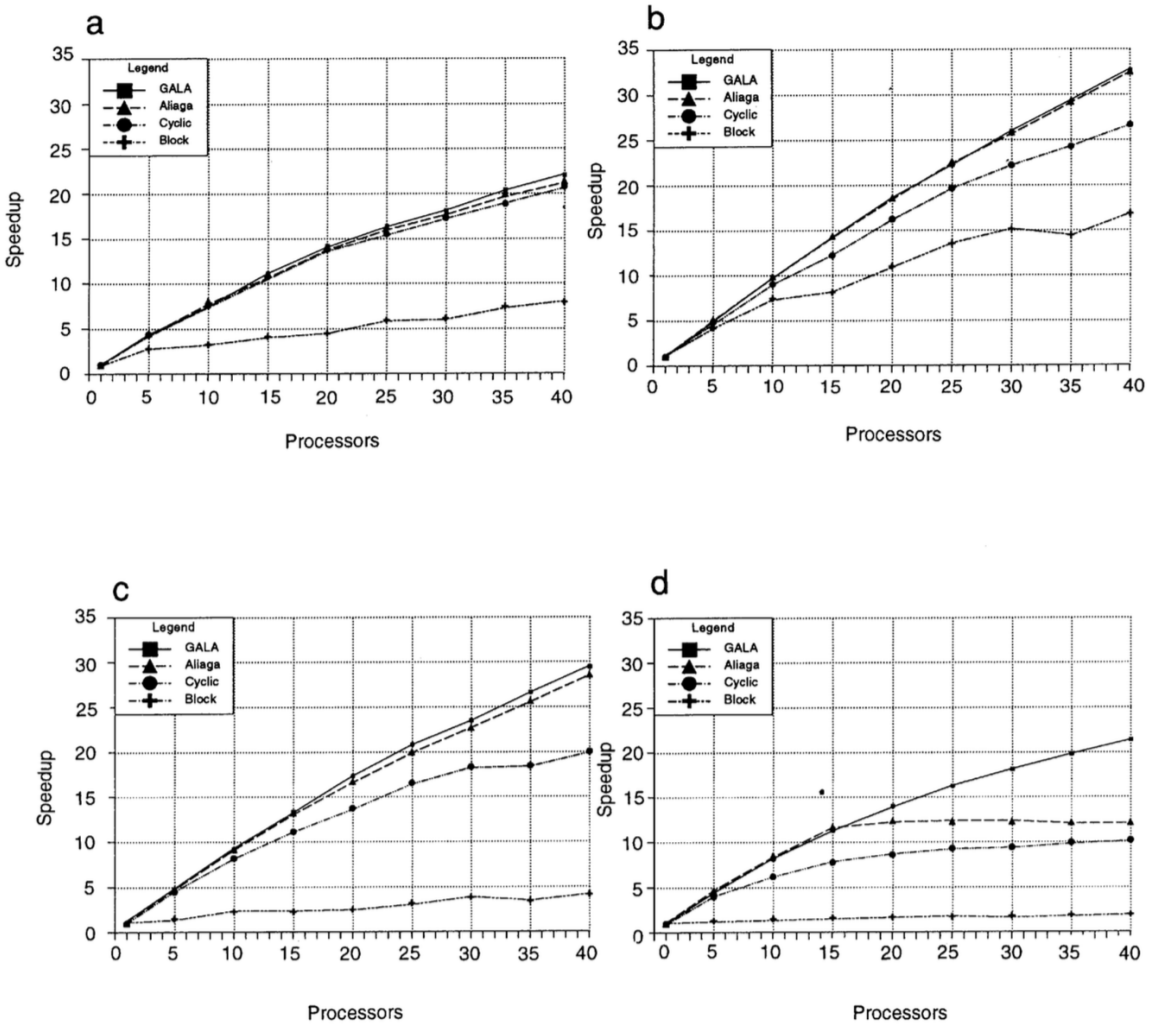


FIG. 8. Comparison of sparse matrix-vector multiplication speedup.

Figure 3: Graph of the speedup of the algorithm as a function of the number of processors. Note that GALA performs most strongly on the bottom right panel, which is the panel corresponding to the synthetically generated data.

Architecture Aware Technique

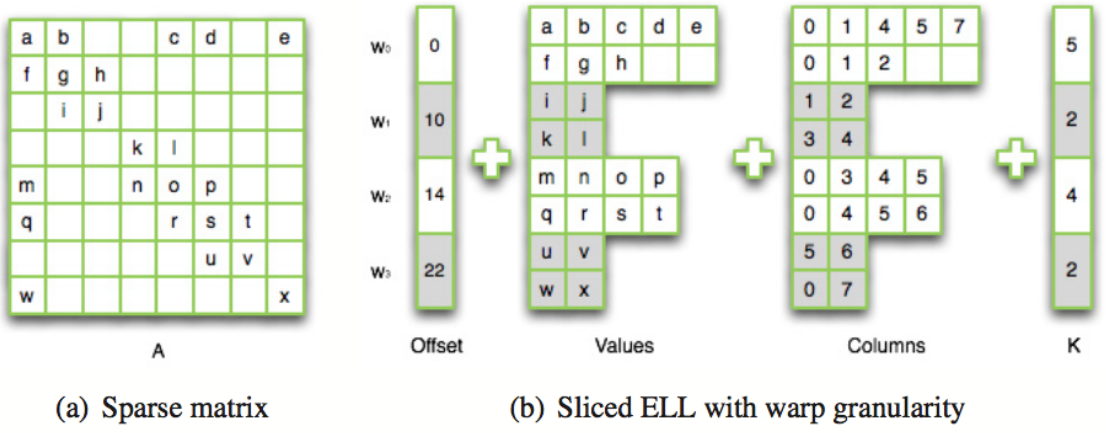
Overview of Algorithm

The next paper I explored was An architecture-aware technique for optimizing sparse matrix-vector multiplication on GPUs by Maggioni and Berger-Wolf. In this paper, a novel heuristic for reducing the number of cache accesses within hardware level thread blocks is given. They also present an improved variation of a sparse matrix data structure.

To understand the improvements, its important to note the fundamental improvements brought to the issue by the architecture of modern GPUs. A GPU is composed of several Streaming Multiprocessors (SMs), each one containing CUDA cores. In the Nvidia GTX 580, for example, there are a total of 512 cores, allowing for an optimal 512 operations per clock cycle. Each of these cores is connected to a random access memory through a cache hierarchy with two levels. Taking into account the number of threads that can theoretically

be ran by a single core, GPUs offer the potential for a massive speedup through significant parallelism. But, for many of the reasons previously mentioned, these gains are not often realized when performing linear algebra tasks involving sparse matrices.

The first optimization discussed is a compression format for sparse matrices. In ELL compression, an nxm matrix is stored in an nxk data structure, where k is the maximum number of non-zero entries in any particular row. There is a separate nxk data structure storing the column index of the particular non-zero entry. Figure 4 illustrates an example of using sliced ELL compression to store a sparse matrix. Sliced ELL compression is similar to regular ELL compression, except that the matrix is partitioned into different slices where there is a local value of k for each slice. This has the effect of reducing the amount of zero-padding needed in order to align the nonzero entries in the matrix.



(a) Sparse matrix (b) Sliced ELL with warp granularity

Fig. 1: Sparse matrix compression

Figure 4: Sliced ELL compression format for sparse matrices presented in this paper. In particular, since there is a local value for the number of columns (k) for each warp, the sliced ELL format allows smaller matrices to be divided up to each warp locally.

The key optimization discussed in this paper surrounds improving the performance of the cache by reducing cache misses and minimizing the number of transactions that must take place. The **CACHE TRANSACTION MINIMIZATION PROBLEM** is defined formually in the paper. Given the following:

- A warp consisting of w threads (or rows)
- A list of cache line mappings C_i, for each thread t_i, corresponding to k memory elements
- A wxk scheduling table S where each row i can be any permutation of the C_i.

The object is to minimize $z(S) = \sum_{j=0}^{k-1} | \cap_{i=0}^{w-1} S_{i,j}|$. In other words, the goal is to minimize the sum of cache line intersections in the scheduling table S (as these are redundant transactions). The algorithm presented does so by greedily choosing a row which maximizes the amount of overlap with rows that have already been placed, which in turn will minimize the number of cache transactions. The time complexity amoritized is approximately $O(n)$ for this algorithm (technically, it is $O(k^2 n)$, but because the sliced ELL format allows k to be significantly smaller in sparse matrices than n, this is approximated well by $O(n)$). In Figure 5, the results of the algorithm are given in a table. The authors note that the speedup is not significant, but that novel contributions in this arena have not performed significantly better.

Benchmark	Random [GFLOPS]	Column [GFLOPS]	Heuristic [GFLOPS]	Cache Reduction
cop20k_A	6.728	10.689	10.738	0.91
mc2depi	17.713	18.722	18.733	1.00
nd6k	8.698	12.339	12.757	0.48
raefsky3	18.021	20.654	20.860	0.82
poisson3Db	4.488	5.514	5.952	0.86
scircuit	3.577	3.986	4.012	0.88
cage14	7.276	19.971	19.765	0.93
mac_econ_fwd500	6.710	7.340	7.573	0.77
offshore	8.569	9.610	9.996	0.80
spal_004	2.788	4.555	10.193	0.35
filter3D	9.310	12.526	12.997	0.80
kkt_power	8.056	10.232	10.678	0.87
TSOPF_RS_b2383	13.703	23.864	23.966	0.97
CO	11.099	18.821	18.795	0.84
shipsec1	16.587	18.369	18.415	0.79
F1	5.777	12.606	12.697	0.87
thermomech_dK	4.878	9.231	9.390	0.94
pdb1HYS	10.434	12.897	13.200	0.63
Average 1.09x speedup achieved with Cache Heuristic				

Table 2: Cache heuristic applied to warp-grained ELL format

Figure 5: Table listing the reduction in the number of cache transactions for several different benchmarks. Overall, a 1.09x speedup is noted by using this algorithm.

Scalable Parallel Matrix Multiplication

A more generalized computation is a standard matrix multplication. This generalizes matrix-vector multiplication, and is also used in its own right in statistical computations relating to linear models. In the paper Scalable Parallel Matrix Multiplication on Distributed Memory Parallel Computers, Li presents a framework for scalable matrix multiplication that unifies theores of both sequential and parallel matrix multiplication algorithms.

The fundmental theorem shown is as follows. For any $O(N^\alpha)$ sequential matrix multiplication algorithm over an arbitrary ring with $2 < \alpha \leq 3$, there is a fully scalable parallel implementation on Distributed Memory Parallel Computers (DMPC). That is, for all $1 \leq p \leq N^\alpha/log(N)$, multiplying two $N \times N$ matrices can be performed by a DMPC with p processors in $O(N^\alpha/p)$ time, and linear speedup can be achieved in the range $[1..N^\alpha/log(N)]$. In particular, multiplying two $N \times N$ matrices can be performed in $O(log(N))$ time by a DMPC with $N^\alpha/log(N)$ processors. The authors note that this matches the performance of parallel random access memory (PRAM).

DMPC has the following characteristics. Each processor has local memory, but there is no global shared memory. All processor communication is done via message passing. In a clock cycle, a processor takes either a communication or a computation step where "idle" is a valid computation step. The time complxity of DMPC, therefore, is the total number of communication and computation steps. Figure 6 illustrates a quick diagram of a DMPC with P processors.

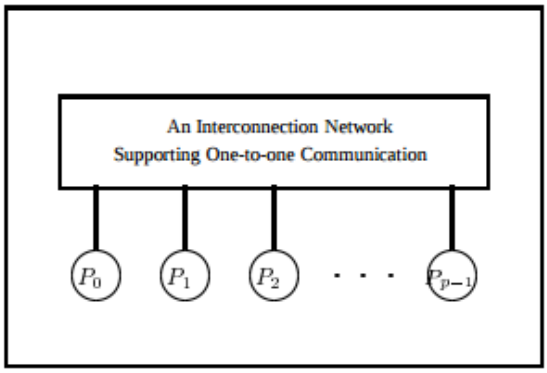
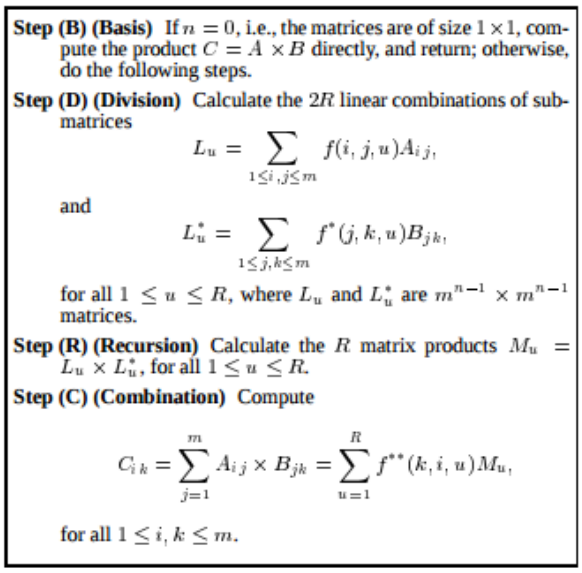


Figure 1. A distributed memory parallel computer (DMPC).

Figure 6: A diagram of Distributed Memory Parallel Computers (DMPC). Note that this structure allows for one-to-one processor communication.

The authors show two lemmas which illustrate the communication capabilities. The first is that a processor group, a set of processors with consecutive indices, can send a matrix to a same sized set of processors in a single step. This of course follows from the one-to-one communications protocol of the DMPC. Lemma 2 follows naturally, which says that all processors can receive this matrix in at most log(R) + 1 steps.

The authors procede by unrolling the **recursive billinear** algorithm for performing the matrix multiplication. This algorithm is shown in Figure 7. Essentially, the algorithm performs a divide and conquer approach where the pieces of the matrix are divided recursively, and then combined into the final result. Step D (the divide step) and Step C (the combine step) in particular are highlighted as for how to perform the algorithm in parallel. The basic idea is that the equations given in the paper define a set of processors that need the results of particular computations. Using the communication system, the results are made available to the processors needing them, and each processor in parallel works on its piece of the computation.



Step (B) (Basis) If $n = 0$, i.e., the matrices are of size 1×1, compute the product $C = A \times B$ directly, and return; otherwise, do the following steps.

Step (D) (Division) Calculate the $2R$ linear combinations of submatrices

$$L_u = \sum_{1 \le i,j \le m} f(i,j,u) A_{ij},$$

and

$$L_u^* = \sum_{1 \le j,k \le m} f^*(j,k,u) B_{jk},$$

for all $1 \le u \le R$, where L_u and L_u^* are $m^{n-1} \times m^{n-1}$ matrices.

Step (R) (Recursion) Calculate the R matrix products $M_u = L_u \times L_u^*$, for all $1 \le u \le R$.

Step (C) (Combination) Compute

$$C_{ik} = \sum_{j=1}^{m} A_{ij} \times B_{jk} = \sum_{u=1}^{R} f^{**}(k,i,u) M_u,$$

for all $1 \le i,k \le m$.

Figure 2. The recursive bilinear algorithm.

Figure 7: The recursive billinear algorithm for matrix multiplication. The algorithm presented in the paper unrolls this algorithm into a parallel iterative algorithm.

Overall, while dense, this paper provides an example of optimal parallelizability. In $O(log(N))$ time, an $N \times N$ matrix can be multiplied given $N^{\alpha}/log(N)$ where $O(N^{\alpha})$ is the running time of the sequential algorithm (which, in this paper, was the recursive billinear algorithm).

Conclusion

Ultimately, I felt these three papers presented a broad but interesting overview of parallel and distributed algorithms relevant to my research in Statistics. The GALA procedure yielded solid speedups for SpMV, particularly in the midst of highly sparse and high skewed data distributions. The cache optimization in the second paper yielded a relatively modest improvement. Nonetheless, the sparse matrix storage format presented could be helpful for a wide range of these applications. Lastly, the theoretical results presented in the last paper provided a solid foundation for the possibilities of parallel matrix multiplication across a set of processors.

References

Gettys, Jim, Phil Karlton, and Scott McGregor. 1997. "Load-Balanced Sparse Matrix–Vector Multiplication on Parallel Computers." *Journal of Parallel and Distributed Computing.*

Li, Keqin. 2001. "Scalable Parallel Matrix Multiplication on Distributed Memory Parallel Computers." *Journal of Parallel and Distributed Computing.*

Maggioni, Marco, and Tanya Berger-Wolf. 2013. "An Architecture-Aware Technique for Optimizing Sparse Matrix-Vector Multiplication on GPUs." *Procedia Computer Science.*