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
2. An architecture-aware technique for optimizing sparse matrix-vector multiplication on GPUs by Maggioni and Berger-Wolf
3. A model-driven blocking strategy for load balanced sparse matrix–vector multiplication on GPUs

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 the following table:

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

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.csv("data/Example_WearableComputing_weight_lifting_exercises_biceps_curl_variations.cs
X <- apply(wearables[,-1], 2, as.numeric)
y <- as.numeric(wearables[,1])

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

```
##    user  system elapsed
##   0.074   0.002   0.077
```

But now note what occurs when the number of observations is a replicated to be 50 times larger.

```
biggerX <- do.call("rbind", replicate(50, X, simplify = FALSE))
biggery <- rep(y, times = 50)

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

```
##    user  system elapsed
##   3.864   0.125   4.131
```

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.

Results

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 1: Overview of expected coverage for k-peptide libraries of different sizes N with the different encoding schemes (NNN, NNB, NNK/S, and trimer). An additional line for maximum possible coverage is shown.

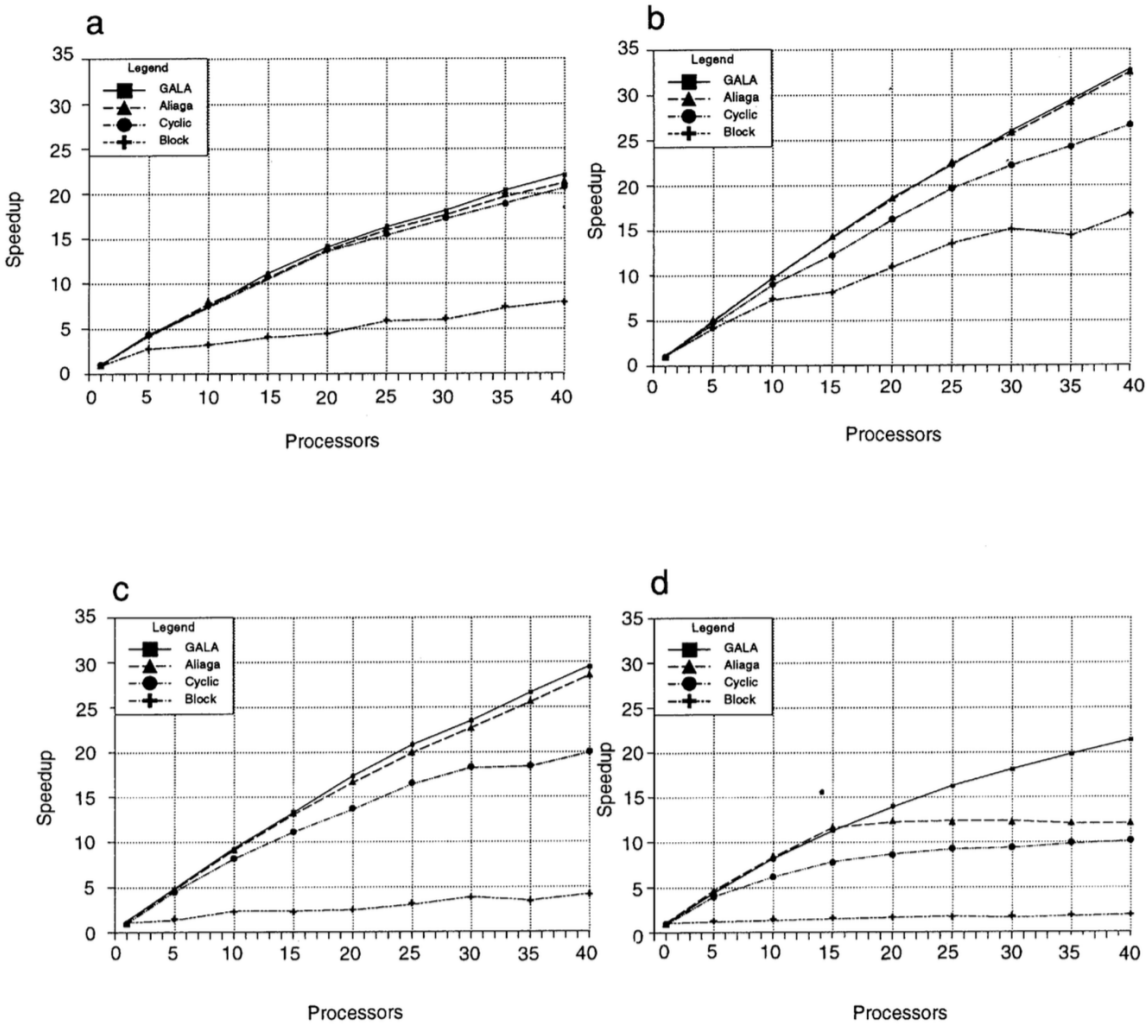

FIG. 8. Comparison of sparse matrix-vector multiplication speedup.

Figure 2: Overview of expected coverage for k-peptide libraries of different sizes N with the different encoding schemes (NNN, NNB, NNK/S, and trimer). An additional line for maximum possible coverage is shown.

Second Paper

Third Paper

Conclusion