Totally Balanced Matrices

Akhilesh Godi (CS10B037) & Abhiram R (CS10B060)

May 8, 2013

Abstract

The aim of the project was to explore the various characterizations of Totally Balanced Matrices and to exploit the different possible parallelizations and analyze their performance with the sequential counterparts. We shall attempt to first define in detail what each of these matrices mean and later on explain the various characterizations and the refinements made to these characterizations so that we could accommodate parallelism into our code, such that it scales. We will also present with appropriate figures and graphs show how our attempts at parallelizing have succeeded or failed accordingly, citing possible reasons for the same.

The choice of the problem was made after discussion with Krithika (Research Scholar with the Theory Group at IIT Madras) drawing inspiration from the properties of Totally Unimodular Matrices that we came across with in our Operations Research Course. Totally Balanced Matrices have interesting properties that we shall discuss further and finds application in Polyhedral Combinatorics. [1]

Balanced Matrices

Definition

A (0, 1)-matrix is balanced if it does not contain an (edge-vertex) incidence matrix of an odd cycle as a submatrix. [Reference to Anstee]

Characterization

The incidence matrix of a graph is said to be balanced if and only if the graph is bipartite, i.e it does not have an odd cycle as a sub graph.

Totally Balanced Matrices

Definition

A (0, 1)-matrix is totally balanced if it does not contain an incidence matrix of any cycle, of length at least 3, as a submatrix. [Reference to Anstee]

[1] Whenever we refer to a graph, we shall always consider the vertex-edge graph to be represented as a matrix with vertices as rows and edges as columns.

Characterization

1. The incidence matrix of a graph is said to be totally balanced if and only if the graph is a forest.
2. A totally balanced matrix and strongly chordal graph are both equivalent.
3. Several other characterizations of Totally Balanced Matrices shall be discussed below along with the algorithms and the parallelizations.

The problem of detecting *Totally Balanced Matrices* is particularly interesting because :

1. $x \in R^n$; $Mx \geq 1$; $x \geq 0$
2. $x \in R^n$; $Mx \leq 1$; $x \geq 0$
3. $x \in R^n$; $Mx = 1$; $x \geq 0$

all have integral solutions.

Exploring other characterizations

Doubly Lexical Orderings and the Γ Matrix

A doubly lexical ordering of a matrix is an ordering of the rows and of colums such that the rows as vectors are lexically increasing and the columns as vectors are lexically increasing. Also, every real valued matrix and hence any binary matrix will also have a doubly lexical ordering.

Let y_1, $y_2 \in R^p$ then the lexicographic ordering is defined as $y_1 \geq^L y_2$ if either $y_1(k^*) > y_2(k^*)$ with $k^* = \min \{ k : y_1(k^*) \neq y_2(k^*)\}$ or $y_1 = y_2$

Instead of finding the doubly lexical ordering of the matrix, we find the Totally Reverse Lexicographic ordering of the matrix.

A *(0, 1)* matrix is called totally reverse lexicographic (*TRL*), if

1. $a^{i+1} \geq^L a^i$ for i = 1, 2, 3, ... m-1
2. $a^{j+1} \geq^L a^j$ for j = 1, 2, 3, ... n-1

To find the doubly lexical ordering of the matrix we shall take the image of the obtained Totally reverse lexicographic ordering.

Algorithm to find TRL ordering of a matrix

The following is the algorithm that we came up with to find the TRL Ordering

1. We first find the column sums of the given matrix, and denote it by $\overrightarrow{d}$.
2. We then exchange the column that is represented by the maximum value in $\overrightarrow{d}$ with the last column.

3. Now, we rearrange the rows such that the last column becomes lexicographically ordered (0's before the 1's).

4. We now fix the last column, and repeat the above procedure till all the columns are fixed.

The output of this algorithm is a *Totally reverse lexicographic* matrix. Its two-way mirror image gives the *Doubly Lexical Ordering* required.

Avoiding a certain submatrix

Also, a matrix is said to be **Totally Balanced** if and only if it does not contain the matrix $\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}$ as a submatrix

But as we can see, permuting across all possible submatrices takes exponential time. Therefore, we make use of the Totally Reverse Lexicographic ordering of the matrix that we have computed. It is enough if we now just check if the transformed Totally Reverse Lexicographically ordered matrix contains the above mentioned matrix, which we shall hence forth refer to as the gamma matrix.

To check if a matrix contains a submatrix, we shall use the naive approach to check for containment. This is done, as there is no better known algorithm for computing a submatrix detection containment.

Parallelization Strategies

Firstly we will discuss the parallelizations adopted in finding the Totally Reverse Lexicogaphic ordering and then we shall discuss gamma-free detection.

Parallelizing TLRO

For the *Totally Reverse Lexicographic Ordering*, we first make sure we exploit the inherent parallelisms in the problem by computing the sum of the elements in columns in parallel. Also, for row swaps, we do the copying of elements by using pointers instead of doing an element by element copy. Also, column swaps are done in parallel by copying elements across columns parallely.

Another alternate method was to use bit wise operations i.e swap using XOR which fared better than naive swapping with a temporary variable in the case of large inputs.

Some other ideas

It could be clearly seen in the algorithm presented for the *Totally Reverse Lexicographic ordering* that, at every iteration, we first fix the column that we are going to swap and move it to the end of the submatrix that we are considering and after that perform a series of row

operations. The bottleneck here, if carefully observed is the inability to do the column swaps and row swaps both in constant time by doing a pointer swap. In order to get over this, the following were the strategies that we could come up with :

Idea 1

Since the issue is inability to do column swaps, a naive approach would be to transpose the matrix, do the swap and transpose it back. This clearly could not yield good results, as transposing a matrix takes O(n*m) time sequentially and can't get any better than O(n*m/p) where p is the number of processors which in our case is very small, where as column swap by just copying and swapping the elements would take O(n) time. Hence this idea was not beneficient.

Idea 2

Use of buckets i.e 2 way pointers The idea was to maintain row buckets and column buckets, such that instead of rows containing elements, we shall have elements belonging to a row bucket and a column bucket. These were maintained in the form of lists of elements for rows and columns. In case of row swaps all we needed to do was to change the bucket numbers, and so is the case for the column swaps. But this did not improve the performance. Infact the performance worsened, this can be attributed to the extra data structure maintenance. More importantly this was not a good idea, as in each step of the iteration, we fix a column and then apply transformations on the remining submatrix (not including this column) for the future iterations. Therefore for this to work, we needed to copy the elements of the fixed columns into another matrix, such that our row, column pointer operations could be done in O(1). This also has a problem, since in the case we want to access an element a_{ij} we would need to traverse all the lists to figure out where this element is present, since we do not have a back pointer for every element. Therefore the idea was dropped.

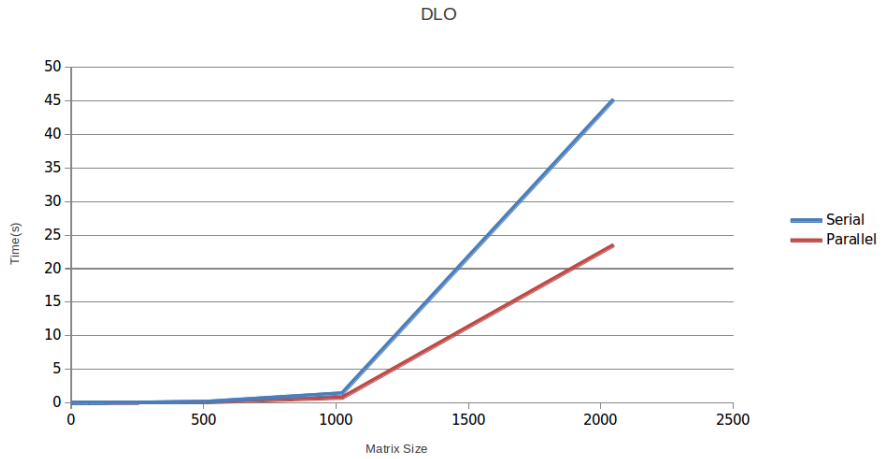
Parallelizing gamma-free detection

This is comparatively easy to do, since it is embarrassingly parallel. Greedy spawning of threads works, and there would be no concurrency issues as such with the problem as we would just be reading the elements to find occurrences of gamma in the whole matrix.

The results for each of the sub-parts are presented.

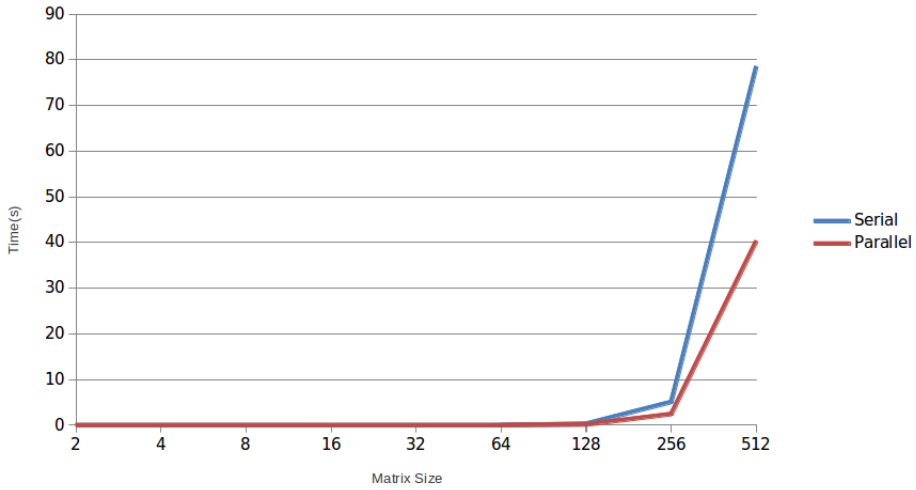
Doubly Lexical Ordering

	2	4	8	16	32	64	128	256	512	1024	2048
S	0.000232	2.39E-05	6.69E-05	0.000178	0.000488	0.001585	0.005016	0.016599	0.143157	1.40517	45.173
P	0.000454	5.41E-05	0.000213	0.000399	0.000994	0.003016	0.003934	0.015149	0.088083	0.760492	23.4753



Gamma Matrix Detection

	2	4	8	16	32	64	128	256	512
S	0.000386	4.50E-05	5.40E-05	0.000177	0.00132	0.019043	0.3025	5.08159	78.522
P	0.000638	6.11E-05	6.39E-05	0.000162	0.000758	0.010352	0.156352	2.43397	40.3066



The graphs show a near linear speedup across processors, also the speed-up is more visible

in the case of large matrices.

Simple Rows Characterization

Definition

Row M_i of the matrix M is a simple row in M if $m_{ij} = m_{ik} = 1$ implies that either $M_j < M_k$ or $M_k < M_j$.

Denote M* as $\begin{pmatrix} I & M \\ M' & J \end{pmatrix}$

where I is the Identity Matrix, and J is the matrix with 1 as the diagonal elements and 0 elsewhere.

With this definition, we find that the procedure to determine if M is totally balanced, can be reduced to searching for a simple row of M*, deleting that row and the corresponding column, and then continuing in this fashion until no simple rows can be found. Given the structure of M*, it is easy to see that if M* contains a simple row then it contains a simple row in the submatrix [I M]. (Otherwise, the set of columns of M is totally ordered by " < ".) Moreover, such a row is simple in M * if and only if the corresponding row of M is simple in M. Thus, the procedure to determine if M is totally balanced can be reduced to deleting simple rows one at a time and stopping when no simple rows can be found.

From *Anstee and Farber's* paper on Totally Balanced matrices, we have the following ***Theorem*** :

A matrix is totally balanced if and only if every submatrix with at least two rows has at least two simple rows.

We use the above theorem's result for our algorithm.

Algorithm

The following algorithm has been implemented

Step 1: Compute the column sums of A : s_1, s_2,. . . ,s_m. For each i, j, compute the number, r_{ij}, of l's in the ith row of the submatrix formed by those columns with a 1 in the jth row. For j = 1,2,..., n we compute the conjugate of the sequence (r_{1j}, r_{2j}, . . . , r_{nj}).

Step 2 : If $n \leq 2$ then output "A is totally balanced," and STOP.

Step 3 : Set j = 1.

Step 4 : Check if row j is simple by checking whether the column sums of the columns with a 1 in row j match the conjugate sequence associated with row j. If they do not, then row j is not simple, go to *Step 6*.

Step 5 : Otherwise, row j is simple. Delete row j from A, update column sums and conjugate sequences, set $n \leq n - 1$ and go to Step 2.

Step 6 : If j = n then output "A is not totally balanced," and STOP. Otherwise, set j = j + 1 and go to Step 4.

In the Step 1 we use the conjugate of a sequence which is defined as follows :

Consider the sequence (r_1, r_2, ... r_n) of integers, then we define the conjugate as the sequence (s_1, s_2, s_3,..... s_m) where $s_i = |\{ j : r_j \geq i \}|$ and $m = \max \{r_1, r_2, ... r_n\}$

Ryser's paper relates conjugate sequence to (0, 1) matrices with no submatrices :
$\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$ and $\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$

And from Anstee's paper, A row of the (0, 1) matrix is simple if and only if the submatrix B of A formed by those columns with a 1 in the given row has no submatrices :
$\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$ and $\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$

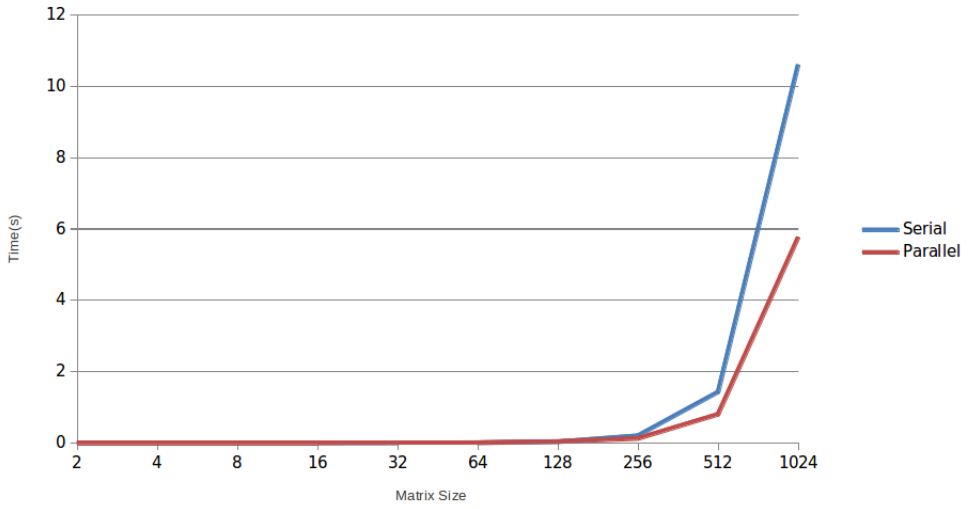
Thus the step 1 of the algorithm is testing to see if a given row is simple can be accomplished by comparing the sequence of column sums with the conjugate of the sequence of row sums.

Parallelization

1. We make the computation of column sums parallelizable.
2. Also finding the conjugate sequence at every stage is parallelizable
3. The matrix elements $\{r_{ij}\}$ are found in parallel.

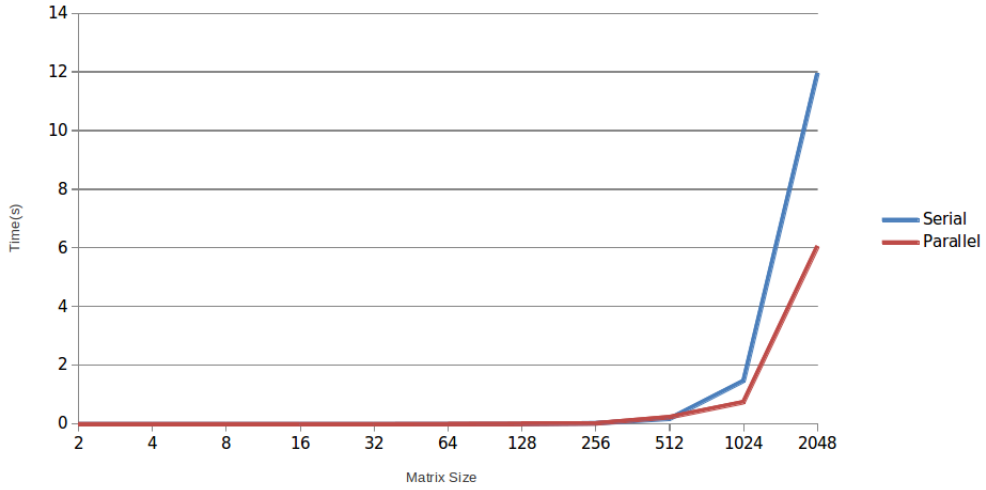
The results and the speedup obtained from the implementations are summarized in the graphs shown.

	2	4	8	16	32	64	128	256	512	1024
S	0.000278	3.08E-04	3.59E-04	0.000876	0.004507	0.00669	0.034111	0.202752	1.4254	10.6057
P	0.000314	7.66E-04	0.00073	0.0005	0.004142	0.00776	0.039397	0.123281	0.803413	5.7668



The above graph represents the results when totally balanced matrices are given as test cases.

	2	4	8	16	32	64	128	256	512	1024	2048
S	0.00028	2.94E-04	1.44E-04	0.000196	0.000325	0.002245	0.005925	0.02989	0.183636	1.46946	11.9427
P	0.000478	4.67E-04	0.000561	0.000467	0.001289	0.002919	0.00631	0.024834	0.238417	0.749495	6.05381



The above graph represents the results when non-totally balanced matrices are given as test cases.

Other Attempts

- We know that Totally Balanced Matrices are subsets of Balanced Matrices, so we spawn another thread that checks if the matrix is balanced concurrently. Checking if a matrix is balanced is simple if we consider the given matrix to be a vertex-edge incidence matrix of a given graph. In that case, this is just checking if the graph is bipartite i.e if the graph has a proper 2-coloring. This takes more time than the TBM detection as the input has to be converted from a matrix form to a graphical representation having a list of edges and nodes for computing whether the graph is 2-colorable.

- Another attempt was to check whether the matrix was balanced or not by checking if a matrix is Totally Unimodular first. If it is, then the matrix is balanced. We use the method of identifying a balanced matrix that is also a zero-one matrix is through the subsequence count, where the subsequence count SC of any row s of matrix A is

  $SC = |\{t \mid [a_{sj} = 1, a_{ij} = 0 \text{ for } s < i < t, a_{tj} = 1], j = 1, ..., n\}|$

  If a matrix A has $SC(s) \leq 1$ for all rows $s = 1, ..., m$, then A has a unique subsequence, is totally unimodular and therefore also balanced. We do so by spawning a thread at the beginning of the program before spawning threads for our algorithm to compute if the matrix is a TBM parallely. Note that the above was beneficient only in

cases where matrices were not Totally Balanced hence facilitating early exit by declaring that the matrix is not TBM.

- Here is one of the application problem that uses *TBM* that we have encountered.
  - TB matrices arise in the formulation of some location problems as set covering problems.
  - T = (V, E) : Tree with non-negative costs
  - d_{ij} : cost of the unique path joining i and j, with $\forall$ i, j $\in$ V
  - $r_j \geq 0$; $\forall$ j $\in$V : radius of node j
  - $T_j = (V_j, E_j)$, where $V_j = \{ i \in V : d_{ij} < r_j \}$, is a neighborhood subtree of T rooted at node j with cost c_j.
  - Finding a minimum cost set of neighborhood subtrees that covers V is a set covering problem min $\{ f(x) : Ax \geq 1; x \in B^n\}$, where $a_{ij} = 1$ if $i \in V_j$ and $a_{ij} = 0$ otherwise

Conclusion

The analysis of the results show scalability across processors for the algorithms presented. One must also notice that the algorithms present were inherently sequential in nature and are iterative. Hence it is difficult to achieve better parallelism with any of the characterizations studied.

References

[1] Wikipedia

[2] Anstee and Farber's paper - [AF84] R.P Anstee and M. Farber. Characterizations of totally balanced matrices. Journal of Algorithms, 5(2):215-230, 1984.

[3] Anna Lubiw's Paper - Doubly Lexical Orderings of Matrices. Anna Lubiw - SIAM J. Comput 01/1987; 16:854-879. pp.854-879

Acknowledgements

We would like to thank Krithika Ramaswamy for her efforts in helping us understand the various characterizations.