Improving the Performance of sparse LU Decomposition in GEMPACK

Florian Schiffmann

March 28, 2019

Abstract

In GEMPACK the computational general equilibrium (CGE) model is solved in its linearized form. In this form, the solution to the CGE model is computed by solving a sparse non symmetric linear system of equations(LSE). In most computer programs, the solution to a LSE is computed either by using matrix decomposition methods (direct), iterative methods or subspace methods. GEMPACK employs a sparse LU decomposition with iterative refinement to solve the LSE. Until GEMPACK 12 these solvers purely relied on algorithms from the Harwell Subroutine Library (HSL) with MA48 as the default solver. While computationally efficient, computing the LU decomposition using these off-the-shelf routines remained the time limiting step for many CGE models. In this paper, we will present a new algorithm for the analysis phase and other improvements to the original HSL routines which greatly improve the solution time for most CGE problems.

The preparation stage for the analysis phase involves the reorganization of the matrix into strong components. By replacing the original modified Hopcroft-Karp algorithm with a tree grafting algorithm to determine a suitable bipartite matching the relative time spent can be reduced by a large factor. The most time consuming part in the analysis phase is the determination of a suitable pivot structure. We developed series of buffered sorted data structures, lower bound estimates and an on-the-fly partitioning of the matrix into sparse and full processed parts which efficiently handles this problem. The new algorithm is able to find a suitable pivot structure with lower number of fill-ins at a significantly reduced cost than the original MA48 solver. For the most common CGE problems we see reductions in solution time by a factor 5 to 10.

1 Introduction

Finding the solution to a linear system of equations(LSE) is a problem that arises in many different fields. While the text book solution required the computation of the matrix inverse, in pratical applications iterative methods, or decomposition methods are employed depending on the details of the problem. In cases where a good precondtioner can easily be constructed, iterative methods show superior performance over direct method, otherwise it is more favorable to use direct solvers. With growing problem size N, the use of dense linear algebra methods quickly becomes prohibitiv as memory requirements grow with $O(N^2)$ and computational complexity with $O(N^3)$. For problems in which a lot of entries in the matrix are 0 or close enough to 0 to be neglected, sparse linear algebra methods are preferable due to their better scaling. Our focus in this paper will be on sparse direct unsymmetric solvers as these are required for CGE problems.

The developemnt of improved sparse linear algebra methods to solve LSE has been an ongoing effort for decades. Thei main application is related to physics applications such as finite element methods and computational chemistry. In these pysically motivated problems, we often find a certain near sightedness. This means that the direct interactions of a specific part of the problem is short ranged. Taking advandtage of this, the problem can be expressed in sparse matrix form which furthermore retains a certain degree of structure imposed by the underlying problem. As these problems present the majority of cases where sparse LU solvers are employed, a certain bias in the algorithic development towards certain matrix structures can be expected. However, the structure and connectivity of CGE matrices is quite different to these matrices. Therefore, it is expected that applying existing algorithms to CGE problems can help to expose inefficiencies in existing algorithms and inspire new ideas to increase the computational efficiency of the algorithm.

In this paper we focus on the algorithm proposed by Duff. This algorithm divides the LU factorization into three parts: analyse, factorize and solve. Here, we will mainly focus on improvements to the analyse phase, but several references to the factorize stage are necessary to understand the overall perfermance of the new algorithm. In the first part of this paper we will discuss the general ideas and data structures employed in the original algorithm. The next three sections will deal with changes to the algorithms, new data structures and exploiting bound estimates during the algorithm respectively. Next, we will discuss the potential of using openMP parellism for additional time savings. The last part of the paper discusses the handling of numerical singularities.

2 Overview of MA48 LU

The analyse phase of MA48 starts by transorming performing Dulmage-Mendelsohn decomposition of the matrix graph. In the MA48 algorithm this involves a modified form of the Hopecroft-Karp algorithm (cite DUFF MC21C), to ensure the diagonal of the matrix is fully occupied. The diagonally occupied matrix is then decomposed into strong components using Tarjans algorithm. The result of these transformation is then used to permute the matrix into block triangular from. With this structure, only the diagonal blocks need to decomposed. The second step of the analysis phase is the determination of suitable pivots for the factorization of the matrix. In MA48, a partial right looking LU decomposition is performed using either Zlatev or Markowitz strategy for the pivot selection. Naturally, the matrix densifies during this process, i.e. the pivot selection algorithm becomes slower the more pivots are chose. MA48, therefore provides the option to switch to dense matrix processing once the sparsity of the reduced matrix reaches a user defined threshold. At this point, the analysis of the matrix is aborted and the remainder of the matrix will be factorized as a dense block. The data structures employed in the original implementation are vectorized versions of the CSR format without any constraint on the ordering of the elements in the sparse rows. Furthermore, the employed data structures are static in size, which requires seeral reordering steps during the decomposition.

The factorize stage of MA48 is performed on the reordered structure obtained by the analyse stage. Each diagonal block is decomposed using the Gilbert-Peierls algorithm using the pivots selected in the analysis stage. When all sparse pivots are used up, a modified version of the dense LU factorization is called for the reaminer of the matrix. This dense LU factorization uses level 1,2,3 blas calls and can be linked against optimized blas routines. The final step in MA48 would be the solution of the linear system. However, as this step is not performance critical in our applications, we omit the discussion of the algorithms employed.

3 Pivot Selection

The algorithm for selecting pivots for factorization is a basic right looking LU algorithm. Algorithm 1 illustrated the basic procedure which is employed. At every step in the Loop a pivot is chosen and the according column is eliminated from the matrix. These steps repeat these operations on the reduced matrix (result from the previous step) until all pivots are chosen. Generally, it is not efficient to perform the pivot search for all elements as the

reduced matrix becomes dense towards the end of the iterations. Therefore it is beneficial to terminate the loop early and use dense linear algebra methods during factorization for the remainder of the matrix.

procedure FINDPIVOTS(Matrix)
  $redMat \leftarrow Mat$
  for $i = 1; i++; i < N$ **do**
    $PivCol, PivRow \leftarrow bestPivot(reducedMatrix)$
    $redMat \leftarrow eliminateColumn(PivCol, PivRow, redMat)$
  end for
end procedure

Figure 1: Right looking LU analyse

In the next two sections, we will discuss the details of the selection and elimination algorithm. We will be starting with the elimination part, as this will explain the data structure for matrix storage, followed by the pivot selction algorithm.

3.1 Data Structures and algorithm for Elimination

During right looking LU factorization one of the computationally most expensive operations is the elimination of the pivot column from the reduced matrix. For dense vectors this is a simple scaled vector addition. In sparse data structures the problem is more complicated. The involved steps are:

1. Determination of distinct/common elements to both vectors
2. Scaled addup common elements
3. Calculating fill-ins (elements only in pivot vector)
4. Merging fill-ins and common elements into new data structure

Depending on the details of the implementation, some of the steps can be combined or performed in a different order. It is however useful to keep the steps seperate to understand the algorithmic complexity of this part of the algorithms.

Step 1) can be understood as computing the intersection of two lists of length N. The comlexity of this step is $O(N^2)$ for unordered lists, $O(Nlog(N))$ if a single vector is ordered and O(N+M) if both lists are ordered. It is obvious that ordered data structures would be preferable for this operation.

However, the maintance of common ordered data structures such as simple vectors or linked lists commonly exceeds the benefits if fill-ins occur. Therefore, MA48 and most other sparse matrix algorithms use unsorted data structures.

Here, we propose a sorted double buffered (SDB) data structure to store the row indicies which partially circumvents the maintanance cost problem. Initially the data is stored in ordered CSR form. We employ a vector of N row data pointers and associated values as the basic representation of out matrix. The row data structure itself contains a base vector and and a buffer vector. Both, base and buffer, are dynamically allocated vectors ordered row indices. The buffer is used to store fill ins and can be merged with the base vector if it grows too large. It is important to note, that deriving the fill ins from ordered data structures results in an ordered vector itself. Therefore, the initial assignment to the buffer vector is simply create as a pointer to the fill in vector. Adding entries to an existing buffer is a merge of two ordered vectors, which can be performed at a cost of $O(L_1+L_2)$ where L_1 and L_2 are the length of the first and the second vector respectively. For computational performance it is important to note that this is a cache friendly operation as it corresponds to an ordered traversal of two vectors. However, this operation has to be performed almost every time, a column takes part in the pivoting and becomes more and more expensive as the size of the buffer grows. It can easily be seen, that for very sparse metrices and many fill ins, the size of the buffer vector closely follows the number of non zeros in the respective column. In such a case, this data structure would not provide any benefits compared to storing the row indices in a single sorted vector. The solution to this problem is to occasionally merge the buffer and the base vector. As we are dealing again with sorted data structures, the cost of this operations is O(nnz) with nnz being the number of non zeros in the column. In this way we can limit the cost of the fill in merge operation to a fraction of the total merge. Chosing a reasonable upper bound for L_b, the cost of this operation becomes neglegible in the algorithm and only few base/buffer merges have to be performed.

3.2 Pivot Selection

For sparse LU factorization the opimal choice of pivots retains a maximal sparsity but does not impede the numerical stablilty of the factorization. The optimal solution to this problem is NP hard, therefore approximations are neccessary. Generally, the stability of a pivot element i in a column J

can be estimated as

$$S = \frac{J_i}{||J||_\infty} \tag{1}$$

An upper bound for the number of fill ins (F) resulting from a pivot choice of row i and column j can easily be obtained as

$$C = (\mathbf{NZR}(i) - 1)\,(\mathbf{NZC}(j) - 1) \tag{2}$$

where **NZR** and **NZC** indiciate number of non zeros in the row and column respectively. This equation above simply states the assumptions that there are no common elements between the pivot column and the columns it applies to (non zeros in column j). In the original Harwell it is possible to choose between Zlatev or Markowitz strategy to serach for the optimal pivot. The Zlatev strategy traverses column in ascending order according to their number of entries until either the cost cannot be improved any further or a maximum search length is reached. Markowitz extends this search by interleaving the Zlatev serach with a search through rows in ascending order according to their number of entries. For each row, the minimum cost for all intersecting columns is computed. The advandtage of Markowitz strategy is that the 'optimal' pivot is found as soon as all row/columns with less than C^2 entries have been screened. It is important to note, that the results for Markowitz and Zlatev with infinite search depth are identical. Here we present an improved implementation of the Zlatev strategy allowing for high search depth. We achieve this by avoiding the recomputations of known quantities and the use of lower bound estimates to allow for an early escape within the column size based searches.

One quantity which can be reused is maxabs norm of the column vectors ($||J||_\infty$) in Equation 1. The cost of computing $||J||_\infty$ is proportional to the number of nonzeros in the respective column. Normally this computation would have to be performed everytime a column is reviewed as potential pivot. However, $||J||_\infty$ can only change if the J is changed. The only time this happens is, if J takes part in the column elimination process, i.e. J has a nonzero in pivot row. Therefore only few $||J||_\infty$ will have to be updated each step. We exploit this fact by storing $||J||_\infty$ for each column and update the required values after elimination.

Another useful quantity to store is the cost of the best pivot (CBP) for already analysed columns. While columns changed by the elimination in the previous step have to be fully analysed, for columns unaffected the CBP is a lower bound for the current cost. The reason for this is that the only elements ever removed from the matrix are the elements of a pivot row and column. Columns that share have a nonzero element in the pivot row it partake in

the elimination and its CBP is will be recomputed. All other columns can only be indirectly affected by fill ins in the elimination columns and thus the CBP either increases or remains identical.

4 Other Modification to MA48 Analysis

As discussed above MA48 uses a depth first version of the Hopcroft-Karp algorithm (MC31). In the original paper Duff and Reid discuss the potential benefit of a cheap heuristic pre assignment of the graph. However, no benefit has been seen and thus the idea has not been persued in MC31. We find, that this holds true for many small and medium sized CGE matrices but the time cost for MC31 can become significant in some badly structured problems or for large CGE matrices. For this reason we investigated alternative bipartite matching algorithms and pre matching heuristics. We find that significant gains can be obtained using a greedy initialization of the problem using the Karp-Sisper initialization of the problem. In most CGE matrices ¿99% of all matches are found by this strategy. Using this initialization in combination with MC31 reduces the time to solution in many cases by more than 50%. While this is a significant speed up, we find that completing the matching requires very deep search trees for MC31. Replacing MC31 with Azads MS-BFS graft algorithm overcomes this problem. MS-BFS graft partially reuses the contructed search trees and therefore is less sensitive to this specific problem.

For the grafting algorithm to work efficiently a different representation of the matrix graph is requried. For large matrices this transformation