An Optimized Leveled Parallel RCM for Bandwidth Reduction of Sparse Symmetric Matrices

Thiago Nascimento Rodrigues
Maria Claudia Silva Boeres
Lucia Catabriga
Federal University of Espírito Santo
Av. Fernando Ferrari, 541, 29075-910, Vitória, ES, Brasil
nascimenthiago@gmail.com
{boeres,luciac}@inf.ufes.br

ABSTRACT

This work presents an implementation of the Leveled Parallel RCM algorithm as well as an improvement version of it based on some proposed enhancements. The use of the bucket array as the main data structure and the suppression of some steps performed by the original version of the algorithm led to outstanding reordering time results and significant bandwidth reductions. The OpenMP framework is used for supporting the parallelism and both versions of the algorithm are tested with large sparse and symmetric matrices.

KEYWORDS. Bandwidth Reduction, Leveled Parallel RCM, Sparse Matrices

Paper topics (Combinatorial Optimization)

1. Introduction

The resolution of large sparse linear systems $Ax = b$, in which A is a sparse matrix, is central in several simulations in science and engineering and is generally the part of the simulation that requires the highest computational cost. In the way to simplify the solution of this kind of system, the minimizing of the bandwidth and reducing of the envelope play an efficient role. These pre-processing methods consist of finding the permutation of rows and columns of the matrix which ensures that nonzero elements are located in as narrow a band as possible along the main diagonal. The sparsity of the matrix is not changed by permutations.

As [Papadimitriou 1976] proved the bandwidth minimization problem is NP-complete, many heuristic algorithms have been proposed for solving the problem. The Reverse Cuthill-McKee (RCM) [George 1971], Sloan [Sloan 1986], and Nested Dissection [George 1973] are examples of heuristics based on graph search strategies. Another kind of approach is found in Spectral algorithm [Barnard et al. 1993] which is a heuristic based on the computation of an eigenvector of a special matrix. Metaheuristics like Tabu Search [Mart et al. 2001] and Simulated Annealing [Rodriguez-Tello et al. 2008] have also become useful strategies for solving combinatorial optimization problems like this. Besides, some hybrid algorithms were also introduced: one combining ant colony optimization with hill-climbing [Lim et al. 2006] and another combining particle swarm optimization with hill-climbing [Lim et al. 2007].

Parallel implementations of algorithms for bandwidth minimization problem have been proposed in order to reach greater performance from multi-core processors. As examples, [Lin 2005] introduced a genetic parallel algorithm tailored to this problem, and [Karypis and Kumar 1998] presented a parallel formulation of the multilevel graph partitioning and sparse matrix ordering problem. In this paper, the leveled parallel RCM proposed by [Karantasis et al. 2014] is presented. The original implementation of the algorithm is based on the Galois system [1]. However, in this work, the results obtained by an alternative implementation of it using the OpenMP [2] framework are analised. Furthermore, an optimized version of the leveled RCM is introduced. This second algorithm is based on some suggested enhancements. Firstly, the four main steps of the original algorithm are merged into just two overall phases. This change decreases the number of serial operations performed by the algorithm. Moreover, the FIFO queue data structure used throughout the whole algorithm is replaced by a static Bucket Array. This new evaluated structure leads to significant reordering time improvements.

The outline of the paper is as follow. In the next section some definitions and structures used in this work are presented. The Section 3 is dedicated to the Leveled Parallel RCM algorithm description. In the Section 4, some changes in the original algorithm are proposed and an optimized version of the Leveled RCM is presented. All tests and achieved results are described in the Section 5. Conclusions and future works are addressed in the Section 6.

2. Structures and Definitions

Let A be a structurally symmetric matrix, i. e., if $a_{ij} \neq 0$ then $a_{ji} \neq 0$, but not necessarily $a_{ij} = a_{ji}$. The bandwidth of A denoted by $lb(A)$ is defined as the greatest distance from the first nonzero element to the diagonal, considering all rows of the matrix. The envelope of A, denoted by $env(A)$ is the sum of the distances from the first nonzero element to the diagonal, also considering

[1] Galois is a system that automatically executes serial C++ or Java code in parallel on shared-memory machines [Galois].

[2] OpenMP is a specification for a set of compiler directives, library routines, and environment variables that can be used to specify high-level parallelism in Fortran and C/C++ programs [OpenMP].

all rows of the matrix [Coleman 1984]. More formally,

$$\begin{aligned} b_i &= (i-j) \quad \forall a_{ij} \neq 0; i = 2,3,\ldots,n \\ lb(A) &= \max_{i=2,3,\ldots,n} \{b_i\} \\ env(A) &= \sum_{i=2}^{n} b_i \end{aligned}$$

The Bandwidth Minimization Problem consists of finding a permutation of rows and columns of A so as to bring all nonzero elements of A to reside in a band as close as possible to the main diagonal, that is, $b = \min\{\max\{|i-j| : a_{ij} \neq 0, i = 1..n, j = 1..n\}\}$

In many scientific computations the manipulation of sparse matrices is considered the crux of the design. Generally the nonzero elements in a sparse matrix constitutes a very small percentage of data. This irregular nature of sparse matrix problems has led to development of a variety of compressed storage formats. The Compressed Sparse Row (CSR) is an important storage method which have been widely used in most sources [Saad 2003]. Storing a given matrix A with a CSR scheme requires three one-dimensional arrays AA, JA and IA of length nnz, nnz, and $n+1$ respectively, where n is the number of rows and nnz is the total number of nonzero elements in the matrix A [Farzaneh et al. 2009]. The content of each array is as follow.

- **Array AA:** contains the nonzero elements of A stored row-by-row.
- **Array JA:** contains the column indexes in the matrix A which correspond to the nonzero elements in the array AA.
- **Vector IA:** contains $n+1$ pointers which delimit the rows of nonzero elements in the array AA. The last position of the vector stores the number of nonzero elements of the matrix.

An example of this technique is illustrated in the Figure 1.

$$A = \begin{pmatrix} 1.0 & 0.0 & 0.0 & 2.0 & 0.0 \\ 3.0 & 4.0 & 0.0 & 5.0 & 0.0 \\ 6.0 & 0.0 & 7.0 & 8.0 & 9.0 \\ 0.0 & 0.0 & 10.0 & 11.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & 12.0 \end{pmatrix}$$

AA	1.0	2.0	3.0	4.0	5.0	6.0	7.0	8.0	9.0	10.0	11.0	12.0
JA	1	4	1	2	4	1	3	4	5	3	4	5
IA	1	3	6	10	12	13						

Figure 1: Example of a matrix A represented in CSR format.

Another important structure for this work is the Bucket Array. A bucket array consists in an array B of size n where each cell of B is thought as a "bucket" that is, a collection of key-value pairs. An entry e with a key k is simply inserted into the bucket $B[h(k)]$, where $h(x)$ is a hash function. A hash function maps each key to an integer in the range [0, n - 1]. As initially each bucket is empty, if the hash function does not map any entry to a key k_i, the bucket $B[h(k_i)]$ remains empty. When each $h(k)$ returns an unique integer, then each bucket holds at most one entry. On the other hand, collisions may happen, where two distinct keys k_1 and k_2 have the same hashed value, i. e., $x_1 \neq x_2$ and $h(x_1) = h(x_2)$. Hence, each bucket must be able to accommodate a collection of

elements [Atallah and Fox 1998]. A typical hash function for integer keys is $h(x) = x \bmod n$. In the proposed optimized RCM, this data structure is employed in replacement of FIFO queue used to store the processed children in each iteration. An example of bucket array is shown in Figure 2.

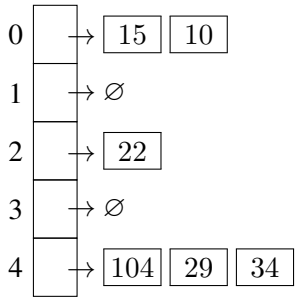

Figure 2: Example of Bucket Array of size 5 storing 6 elements. The hash function is $h(x) = x \bmod 5$.

3. Leveled Parallel Reverse Cuthill-McKee

The Reverse Cuthill-McKee (RCM) operates on the unlabeled graph of the matrix A. The labeled graph of A is a graph having n nodes, labeled from 1 to n, with an edge set E consisting of edges such that $\{x_i, x_j\} \in E$ if only if $a_{ij} \neq 0$ and $a_{ji} \neq 0$. The unlabeled graph of A is simply the graph obtained from A with labels removed. The RCM algorithm generates a label for each node of the unlabeled graph, and hence a reordering of A. The application of the algorithm requires that a starting node be provided. It is usually chosen from the pseudo-peripheral [3] nodes of the graph. Thus, given a starting node r, the algorithm is as follow [Chan and George 1980].

Step 1. Set $x_1 \leftarrow r$.

Step 2. (Breadth-First Search - BFS) For $i = 1, 2, \ldots, n-1$, find all unlabeled neighbors of x_i at level $i + 1$.

Step 3. (Main loop) For $i = 1, 2, \ldots, n$, number all unlabeled neighbors of x_i in increasing order of degree.

Step 4. (Reverse the ordering) The RCM ordering is given by $y_1, y_2, \ldots, y_n$ where $y_i = x_{n+1-i}$, $i = 1, 2, \ldots, n$.

The output of the algorithm is a permutation of the vertices of G stored in the vector $y = (y_1, y_2, \ldots, y_n)$, which labels the node v with label i if $y_i = v$. The Leveled Reverse Cuthill-McKee (L-RCM) proposed by [Karantasis et al. 2014] follows the general structure of the serial RCM algorithm. It process nodes level by level and the parallelism is restrict to the current processing level. Before to finalize an iteration, i. e., before advancing to a next level, the algorithm stores each processed node in the RCM permutation array. The L-RCM pseudocode is shown in Algorithm 1. The main general steps are [Karantasis et al. 2014]:

1. The **expansion** step works as a Breadth-First-Search. For each node (parent) in a processing level, the children of it are analyzed. The correct level of each child is calculated based on the level of the respective parent. If the level of a child is updated, the node is added to the list (generation) which will be processed in the next iteration. As a child node might have multiple parent nodes, the parent saved is the one closer to the source node in the permutation array.

2. In the **reduction** step, each parent node computes its respective number of children.

[3] A pseudo-peripheral node is one of the pairs of vertices that have approximately the greatest distance (graph diameter) from each other in the graph (the graph distance between two nodes is the number of edges on the shortest path between nodes).

3. A **prefix sum**[4] step uses the output of the previous step to calculate each child position in the permutation array according to the respective parent. Actually, it is created a correspondence between the number of children of each parent and an appropriated index in the permutation array.

4. In **placement** step, all nodes from the current iteration are stored in the permutation array. Using the indexes generated in the prefix sum step as initial position, each child is allocated in the correct range of positions. The sequence of these ranges respects the RCM ordering of parents. After the last child of a parent is stored in the permutation array, the respective set of children nodes is sorted in ascending degree.

Algorithm 1 Leveled RCM

```
1:  Graph G = input();    // Read in graph
2:  P[0] = source;
3:  while (P.size < G.size)
4:      // Expansion
5:      List generation;
6:      foreach (Node parent: P[parent1:parentN]) {
7:          for (Node nb: parent.neighbors) {
8:              if (nb.level > parent.level) {
9:                  if (nb.level > parent.level + 1) {
10:                     // Atomic check
11:                     atomic nb.level = parent.level + 1;
12:                     generation.push(child);
13:                 }
14:                 if (parent.order < child.parent.order)
15:                     atomic child.parent = parent;
16:     } } }
17:     // Reduction
18:     foreach (Node child: generation) {
19:         atomic child.parent.chnum++;
20:     }
21:     // Prefix Sum
22:     foreach (int threads: thread) {
23:         Prefix sum of parent.chnum into parent.index
24:     }
25:     // Placement
26:     foreach (Node child: generation) {
27:         atomic index = child.parent.index++;
28:         P[index] = child;
29:         if (child == child.parent.lastChild)
30:             sort(P[children1:childrenM]);
31:     } }
```

4. Optimized Leveled Parallel RCM

An optimized version of the Leveled Parallel RCM (OL-RCM) was obtained applying some changes in the original algorithm. Besides, an alternative data structure was tested in replace-

[4]**Prefix sum:** The prefix sum operation takes a binary associative operator $\oplus$, and an ordered set of n elements $[a_0, a_1, \ldots, a_{n-1}]$ and returns the ordered set $[a_0, (a_0 \oplus a_1), \ldots, (a_0 \oplus a_1 \oplus \ldots \oplus a_{n-1})]$.

ment of Linked List (FIFO queue) used for store nodes in each iteration. A main difference between the algorithms is the number of steps. The original Leveled RCM is divided in four steps. In the optimized version of the algorithm, just expansion and placement steps are present. The pseudocode is shown in Algorithm 2.

Algorithm 2 Optimized Leveled RCM

```
1:  Graph G = input();    // Read in graph
2:  P[0] = source;
3:  while (P.size < G.size)
4:      // Expansion
5:      BucketArray generation;
6:      foreach (Node parent: P[parent1:parentN]) {
7:          for (Node nb: parent.neighbors) {
8:              if (nb.level > parent.level) {
9:                  if (nb.level > parent.level + 1) {
10:                     // Atomic check
11:                     atomic nb.level = parent.level + 1;
12:                     if (nb.status == UNLABELED) {
13:                         nb.status = LABELED;
14:                         generation[hash(parent)].push(nb);
15:     } } } } }
16:     // Placement
17:     foreach (Vector children: generation[bucket1:bucketN].children) {
18:         atomic index = P.size + permOffSet;
19:         atomic permOffSet += children.size;
20:         sort(children);
21:         foreach (Node child: children)
22:             P[index++] = child;
23:     } }
```

Initially, the graph is read and the root node is set in the first position of the permutation array (P). The main loop is executed until each node of the graph is allocated in the permutation array. For each parent node already processed at level i, the respective neighborhood is obtained and the level of children is set as $i + 1$. As a child node may have more than one parent and as each parent is picked up by different threads, this operation of setting the child level may generate conflicts. So, it is performed as an atomic operation. Afterwards, processed children are labeled (if they do not have a label yet), pushed to the parent array (they are parents of the next level of nodes), and allocated at the bucket of its respective parent. From this point, all processing is done over the bucket array (generation). As all processed children are allocated at the appropriated buckets, they may be processed by different threads. Each thread uses the size of the children array of a bucket to determine the positions of them in the permutation array. Thus, the threads sort each respective set of children by degree, and place each node in the permutation array using the previous calculated range.

The expansion phase remained essentially the same. As in BFS, when neighbors are accessed for the first time, theirs distance from the source are recorded, the correct parent is set, and they are labeled. Additionally, however, the reduction step of the original algorithm is incorporated by this phase. Instead of checking all children to count the number of neighbors by parent in a separated phase, this process is performed during the analysis of each parent in expansion phase. In fact, when a neighbor is labeled, it is immediately allocated in the bucket corresponding to its parent.

The appropriated bucket is determined by a hash function which is a simply mapping between the parent position in permutation array and the corresponding parent position in the bucket array.

Through the use of a static data structure like bucket array, each thread has an optimized way to determine the appropriated bucket to access. Actually, the cost involved in accessing a bucket is related to an arithmetic operation performed by the hash function, added to the cost of accessing an specific position of array of buckets. Another aspect that aggregates high performance in the use of the bucket array is the static way to store children in buckets. It was possible by oversizing each bucket. Each one was defined with size equals to the degree of the respective parent. Thus, adding a child in an appropriated bucket corresponds to the cost of accessing one array position.

The placement stage also differs between the two algorithms. In the L-RCM, threads process child by child. As the designated position in permutation array depends of an atomic operation related to an index stored by the respective parent, each child placement implies a new synchronized operation. Furthermore, there is the spent time in the previous step responsible to designate a range of indexes for each parent. In the OL-RCM, the prefix sum step is removed. As the placement operation is done by threads processing bucket by bucket, the range of positions in the permutation array is calculated in its bucket iteration. Thus, the number of synchronized operations decreases and an unnecessarily step (prefix sum) is eliminated.

5. Experimental Results

The performance evaluation of the OL-RCM algorithm was against a traditional serial implementation of RCM and the Leveled Parallel RCM proposed by [Karantasis et al. 2014]. A set of nine symmetric and square matrices was selected from the University of Florida Sparse Matrix Collection [Davis and Hu 2011]. These matrices cover multiple kinds of problems in order to increase the dataset variety. The set of tested matrices is shown in Table 1. The columns present matrices and some characteristics of them: dimension, number of non-zeros, percentage of sparsity, and bandwidth. The program was coded in the C language and the parallelism was supported by OpenMP framework. The experiments were performed on a PC with Intell i7-3610QM 8 core processor with 2.3 GHz of CPU and 8 GB of main memory. The operational system was Ubuntu 14.04.3 LTS 64-bit with Linux Kernel 3.19.0-31. The code was compiled with GNU gcc version 4.8.4.

Table 1: Sparse Tested Matrices

Matrix	Dimension	Non-zeros	Sparsity (%)	Bandwidth
dw8192	8,192	41,746	99.938	4,160
FEM_3D_thermal1	17,880	430,740	99.865	13,787
rail_79841	79,841	553,921	99.991	79,811
Dubcova3	146,689	3,636,643	99.983	146,356
inline_1	503,712	36,816,170	99.985	502,403
audikw_1	943,695	77,651,847	99.991	925,946
dielFilterV3real	1,102,824	89,306,020	99,993	1,036,475
atmosmodj	1,270,432	8,814,880	99.999	21,904
G3_circuit	1,585,478	7,660,826	99.999	947,128

The Table 2 shows a performance comparison between a serial and the two versions of parallel Leveled RCM (L-RCM and OL-RCM). The programs were performed five times for each pair (m_i, t_j), where m_i is a matrix of the Table 1, and t_j is the number of threads between 4 and 128 (in steps of 2) used by each program. For each (m_i, t_j) tested pair, the minimum and maximum reported values were discarded, and the average was calculated from the considered values. In

order to compare both algorithms, for each matrix m_i it was selected the number of threads t_j that reached the best time reordering for the L-RCM algorithm. This same number of threads t_j was used to select the corresponding (m_i, t_j) tested pair from the OL-RCM algorithm. Each number of thread t_j chosen for the comparison of the two algorithms is indicated in the column #Threads of the Table 2. Moreover, the Compressed Sparse Row format was the mechanism used to store each tested matrices. The operations applied on them were also performed using this format.

The three columns grouped by Reorder Time column in the Table 2 show the elapsed time by the algorithms to reorder each matrix. It is relevant to point out that the time spent in each pseudoperipheral computation was excluded from the programs. With the L-RCM it was possible to reach a time reduction ranging between 16.84% ($rail\_79841$) and 81.23% ($G3\_circuit$) when comparing with times obtained for the serial implementation. On the other hand, the reorder performance reached by the OL-RCM was highest for all tested matrices. In fact, for the six largest matrices the attained time reduction was higher than eighty-four percent varying between 84.65% for the $Dubcova3$ and 98.45% for the $G3\_circuit$. Including the smallest matrices, the reordering time performance of the OL-RCM was superior. Actually, the experimental results shown a time reduction of 40%, 68.57%, and 58.95% for the $dw8192$, $FEM\_3D\_thermal1$, and $rail\_79841$ respectively. Nevertheless, no significant speedup was observed for the two parallel algorithms. One reason for this result is that both algorithms use the same general strategy based on a parallelism by level [Karantasis et al. 2014]. As a barrier must be placed between each level, increasing the number of threads do not necessarily lead to an improvement of performance.

Table 2: Results Comparison

Input		Reorder Time (sec.)			Bandwidth Reduction (%)		
Matrix	#Threads	RCM	L-RCM	OL-RCM	RCM	L-RCM	OL-RCM
dw8192	4	0.005	0.003	0.003	96.490	93.462	92.764
FEM_3D_thermal1	4	0.038	0.020	0.012	95.046	95.322	95.046
rail_79841	4	0.095	0.079	0.039	99.311	99.308	99.311
Dubcova3	32	1.199	0.469	0.184	98.442	98.425	98.442
inline_1	8	9.04	4.007	1.096	98.807	98.720	98.807
audikw_1	16	84.833	34.449	2.441	96.283	95.038	96.283
dielFilterV3real	16	88.298	35.521	2.661	97.548	97.542	97.548
atmosmodj	16	32.732	20.193	1.126	64.513	64.235	64.513
G3_circuit	8	84.832	15.922	1.318	99.464	99.463	99.464

The three columns related to Bandwidth Reduction in Table 2 shown the percentage of band reduction obtained by each algorithm. This percentage was calculated as the ratio between the band obtained after to apply the permutation generated by the algorithm and the original band. The OL-RCM did not reach the best percentage of reduction just for the two smallest matrices. In fact, the performance of L-RCM algorithm was superior only for the $FEM\_3D\_thermal1$ matrix, and for the smallest matrix ($dw8192$) the best band reduction was achieved by the serial RCM. For the other seven matrices, as the serial implementation as the OL-RCM algorithm attained the same quality of results. This quality may by graphically attested through Figs 3, 4, and 5. Matrices were grouped according to the size of them: three smallest, three largest, and three of intermediate size. The first row of each group presents the matrix sparsity before reordering. In the below rows, each respective matrix is exhibited as result of a permutation of rows and columns derived from the new proposed Leveled RCM algorithm.

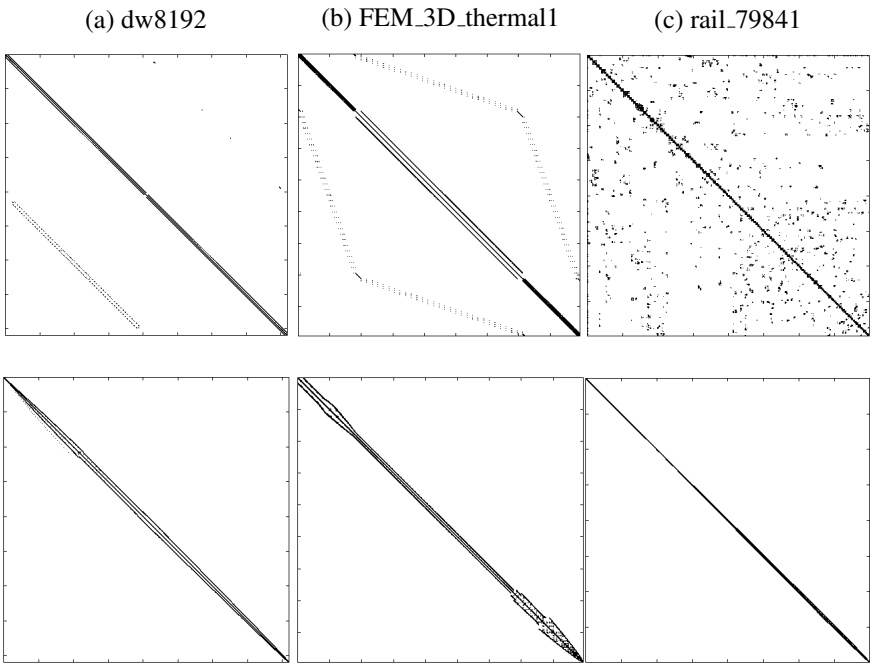

Figure 3: Matrices of size smaller than a hundred thousand

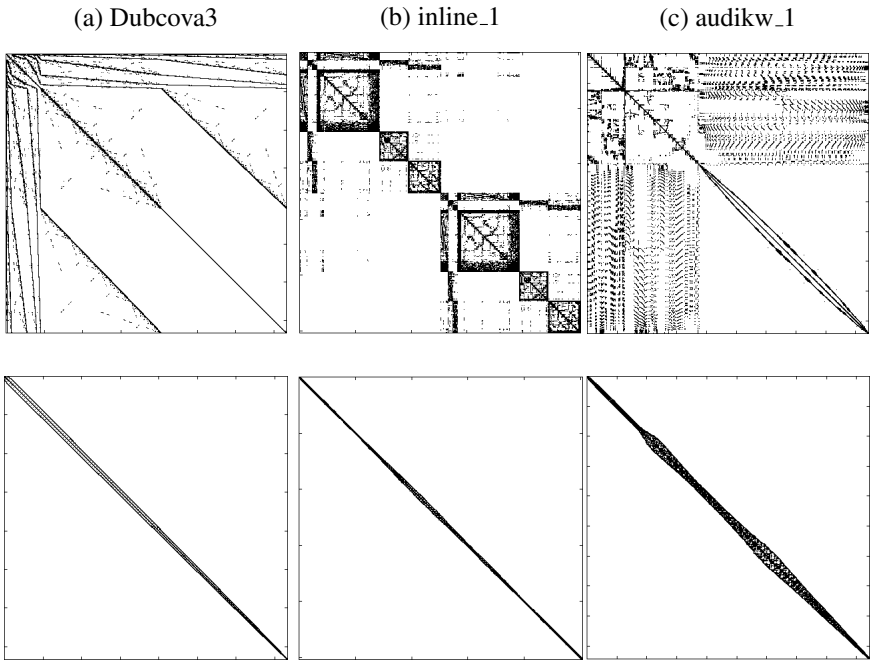

Figure 4: Matrices of size between a hundred thousand and one million

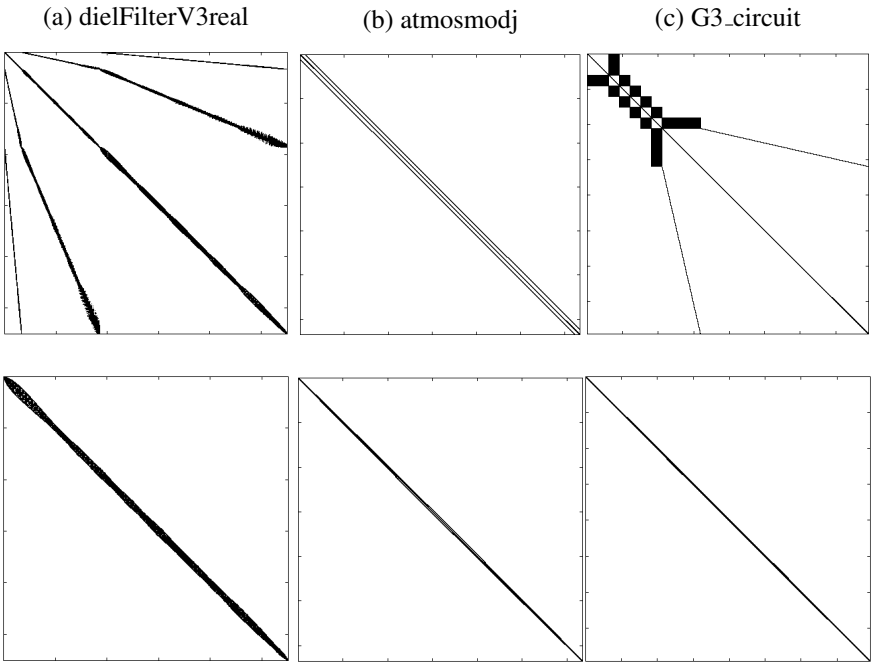

Figure 5: Matrices of size larger than one million

6. Conclusion and Future Work

This paper analysed a parallel strategy for the traditional Reverse Cuthill-McKee reordering algorithm. Two implementations were presented and the results achieved by both represent a significant improvement on reordering time. With the original studied algorithm (L-RCM), it reached a time reduction of until 81.23% of the serial time. The changes proposed on this algorithm led to performance enhancements. Actually, the optimized algorithm (OL-RCM) achieved a time reordering higher than 84% for six of tested matrices. For the remaining matrices, the results were also superior when compared with the original algorithm. About the reordering quality, both implementations attained relevant bandwidth reduction. In fact, an exception of one matrix - $atmosmodj$, the permutation generated by both algorithms meant a band reduction superior to ninety percent. Therefore, the original Leveled RCM as well as the proposed optimized version of it might be considered as a efficient approach for the bandwidth reduction problem.

The implementation of the Leveled RCM, as well as the proposed enhancements presented in this work were supported by the OpenMP parallel framework. However, some works have addressed the reordering problem through the use of another kind of parallelism tool. As example, [Chevalier and Pellegrini 2008] has developed a parallel tool for graph partitioning, and several works have discussed the parallelization of algorithms by the Galois System [Hassaan et al. 2011]. Therefore, an evaluation of alternative implementations developed over parallel platforms like these, might aggregate more improvements to the studied algorithms.

References

Atallah, M. J. e Fox, S., editores (1998). *Algorithms and Theory of Computation Handbook*. CRC Press, Inc., Boca Raton, FL, USA, 1st edition.

Barnard, S. T., Pothen, A. e Simon, H. D. (1993). A spectral algorithm for envelope reduction of sparse matrices. In *Proceedings of the 1993 ACM/IEEE Conference on Supercomputing*, Supercomputing '93, p. 493–502, New York, NY, USA. ACM.

Chan, W. M. e George, A. (1980). A linear time implementation of the reverse cuthill-mckee algorithm. *BIT Numerical Mathematics*, 20(1):8–14.

Chevalier, C. e Pellegrini, F. (2008). Pt-scotch: A tool for efficient parallel graph ordering. *Parallel Comput.*, 34(6-8):318–331.

Coleman, T. F. (1984). *Large Sparse Numerical Optimization*, volume 165 of *Lecture Notes in Computer Science*. Springer.

Davis, T. A. e Hu, Y. (2011). The university of florida sparse matrix collection. *ACM Trans. Math. Softw.*, 38(1):1:1–1:25.

Farzaneh, A., Kheiri, H. e Shahmersi, M. A. (2009). An efficient storage format for large sparse matrices. *Communications Series A1 Mathematics & Statistics*, 58(2):1–10.

Galois. `http://iss.ices.utexas.edu/?p=projects/galois`.

George, A. (1973). Nested dissection of a regular finite element mesh. *SIAM Journal on Numerical Analysis*, 10(2):345–363.

George, J. A. (1971). *Computer Implementation of the Finite Element Method*. PhD thesis, Stanford, CA, USA. AAI7205916.

Hassaan, M. A., Burtscher, M. e Pingali, K. (2011). Ordered vs. unordered: A comparison of parallelism and work-efficiency in irregular algorithms. In *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming*, PPoPP '11, p. 3–12, New York, NY, USA. ACM.

Karantasis, K. I., Lenharth, A., Nguyen, D., Garzarán, M. e Pingali, K. (2014). Parallelization of reordering algorithms for bandwidth and wavefront reduction. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '14, p. 921–932, Piscataway, NJ, USA. IEEE Press.

Karypis, G. e Kumar, V. (1998). A parallel algorithm for multilevel graph partitioning and sparse matrix ordering. *J. Parallel Distrib. Comput.*, 48(1):71–95.

Lim, A., Lin, J., Rodrigues, B. e Xiao, F. (2006). Ant colony optimization with hill climbing for the bandwidth minimization problem. *Appl. Soft Comput.*, 6(2):180–188.

Lim, A., Lin, J. e Xiao, F. (2007). Particle swarm optimization and hill climbing for the bandwidth minimization problem. *Applied Intelligence*, 26(3):175–182.

Lin, W. (2005). Improving parallel ordering of sparse matrices using genetic algorithms. *Appl. Intell.*, 23(3):257–265.

Mart, R., Laguna, M., Glover, F. e Campos, V. (2001). Reducing the bandwidth of a sparse matrix with tabu search. *European Journal of Operational Research*, 135(2):450–459.

OpenMP. `http://openmp.org`.

Papadimitriou, C. H. (1976). The np-completeness of the bandwidth minimization problem. *Computing*, 16(3):263–270.

Rodriguez-Tello, E., Hao, J. e Torres-Jimenez, J. (2008). An improved simulated annealing algorithm for bandwidth minimization. *European Journal of Operational Research*, 185(3):1319–1335.

Saad, Y. (2003). *Iterative Methods for Sparse Linear Systems*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2nd edition.

Sloan, S. W. (1986). An algorithm for profile and wavefront reduction of sparse matrices. *International Journal for Numerical Methods in Engineering*, 23(2):239–251.