Tema
(cc) BY
Tendências em Matemática Aplicada e Computacional, **18**, N. 3 (2017), 449-465

www.scielo.br/tema
doi: 10.5540/tema.2017.018.03.0449

Parallel Implementations of RCM Algorithm for Bandwidth Reduction of Sparse Matrices

T.N. RODRIGUES*, M.C.S. BOERES and L. CATABRIGA

Received on November 24, 2016 / Accepted on August 5, 2017

ABSTRACT. The Reverse Cuthill-McKee (RCM) algorithm is a well-known heuristic for reordering sparse matrices. It is typically used to speed up the computation of sparse linear systems of equations. This paper describes two parallel approaches for the RCM algorithm as well as an optimized version of each one based on some proposed enhancements. The first one exploits a strategy for reducing lazy threads, while the second one makes use of a static bucket array as the main data structure and suppress some steps performed by the original algorithm. These related changes led to outstanding reordering time results and significant bandwidth reductions. The performance of two algorithms is compared with the respective implementation made available by Boost library. The OpenMP framework is used for supporting the parallelism and both versions of the algorithm are tested with large sparse and structural symmetric matrices.

Keywords: parallel RCM, bandwidth reduction, sparse matrices.

1 INTRODUCTION

Computation involving sparse matrices have been of widespread use since the 1950s, and its application includes electrical networks and power distribution, structural engineering, reactor diffusion, and, in general, solutions to partial differential equations [29]. The typical way to solve such equations is to discretize them, i.e., to approximate them by equations that involve a finite number of unknowns. The linear systems that arise from these discretizations are of the type $Ax = b$, in which A is a large and sparse matrix, that is, it has very few nonzero entries. In this context, a matrix with a small bandwidth is useful in direct methods for solving sparse linear systems since it allows a simple data structure to be used. It is also useful in iterative methods because the nonzero elements will be clustered close to the diagonal, thereby enhancing data locality. Given a symmetric matrix A, a bandwidth-reduction ordering aims to find a permutation P so that the bandwidth of PAP^T is small. The bandwidth of a matrix A denoted by $\beta(A)$ is defined as the greatest distance from the first nonzero element to the diagonal, considering all

*Corresponding author: Thiago Nascimento Rodrigues – E-mail: nascimenthiago@gmail.com
Centro Tecnológico, Departamento de Informática, UFES – Universidade Federal do Espírito Santo, Av. Fernando Ferrari, 541, 29075-910 ES, Brasil. E-mails: boeres@inf.ufes.br; luciac@inf.ufes.br

rows of the matrix [29]. More formally, for the i-th row of A, $i = 1, 2, \ldots, n$, let $f_i(A) = \min\{j \mid a_{ij} \neq 0\}$, and $b_i(A) = i - f_i(A)$. So, $\beta(A) = \max_{i=2,3,\ldots,n} \{b_i(A)\}$.

Since Papadimitrou [23] proved that the bandwidth minimization problem is NP-complete, several heuristic algorithms have been presented in the literature aiming to find good quality solutions as fast as possible. An important class of these algorithms treats a matrix bandwidth reduction under the perspective of a graph labeling problem. In this way, reordering a sparse matrix is considered a problem of labeling the vertices of the corresponding graph in such way that closest labels are assigned to most linked vertices.

In this paper, the parallel Unordered and Leveled implementations of the RCM algorithm proposed by [16] are presented. In the first one, the underlying matrix graph is completely processed in one step, followed by the permutation vector generation. On the other hand, in the second one, the graph traversing and the vector computation take place iteratively level-by-level. The original implementation of these algorithms is based on the Galois system[1]. However, as the OpenMP[2] framework has widely been used in researches related to shared-memory parallel systems, this work analyses the results obtained by an OpenMP implementation of these algorithms. Furthermore, an optimized version of each one is introduced. Actually, the set of proposed optimizations leads to outstanding improvements on the performance of the algorithms. In the Unordered RCM, a strategy to reduce lazy threads is evaluated. On the other hand, the Leveled RCM is presented with an alternative main data structure – a static Bucket Array [3] – in replacement of FIFO queue [2] proposed originally. Relevant CPU time improvements were observed by means of this change.

The outline of the paper is as follow. In the next section, an overview of the serial RCM is presented. Section 3 is dedicated to description of the Unordered RCM algorithm as well as some proposed optimizations. In Section 4, the Leveled RCM algorithm is presented along with an optimized version of it. All tests and achieved results are described in Section 5. Conclusions and future works are addressed in Section 6.

2 SERIAL REVERSE CUTHILL-MCKEE

The serial Cuthill-McKee algorithm [8] is based on a Breadth-First-Search (BFS) strategy, in which the graph is traversed by level sets[3]. As soon as a level set is traversed, its nodes are marked and numbered. The neighbors of each of these nodes are then inspected. Each time a neighbor of a visited vertex that is not numbered is encountered, it is added to a list and labeled as the next element of the next level set. The order in which each level itself is traversed gives rise to different orderings or permutations of rows and columns. In the Cuthill-McKee ordering, the nodes adjacent to a visited node are always traversed from the lowest to the highest degree.

[1] Galois is a system that automatically executes serial C++ or Java code in parallel on shared-memory machines [10].

[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 [22]

[3] A level set structure of a graph is defined recursively as the set of all unmarked neighbors of all nodes of a previous level set. Initially, a level set consists of one node.

However, in 1971, the Reverse Cuthill-McKee algorithm was presented by [20]. It was empirically observed that reversing the Cuthill-McKee ordering yields a better permutation scheme for matrix reordering problems. Algorithm 1 presents the pseudo-code of serial RCM.

Algorithm 1 Serial Reverse Cuthill-McKee Algorithm

1: $v_1 = \mathrm{r}$; *// root node*
2: **for** $(i = 1\ to\ n)$ {
3: Find all unlabelled neighbors of the vertex v_i
4: Label the vertices found in increasing order of degree
5: }
6: Reverse the order as $w_1, w_2, \ldots, w_n$ where $w_i = v_{n-i+1}$ for $i = 1, \ldots, n$.

The root r of the level structure is usually chosen from the pseudo-peripheral[4] nodes of the associated graph. In this work, the parallel pseudo-peripheral node finding algorithm presented by [30] was implemented and used for obtaining the root vertex required by RCM.

3 UNORDERED PARALLEL REVERSE CUTHILL-MCKEE

The first RCM parallelization suggested by [16] is composed of four main steps. Based on this, the overall skeleton of the Unordered Parallel RCM is not iterative, that is, there is no a global loop wherewith the algorithm traverses the graph gradually. Instead, the four steps are executed just once and the permutation array is yielded as final output. These related steps are detailed as follow.

Step 1: Generate a level structure through the Unordered BFS algorithm. The Unordered BFS algorithm relies on the perception of local minimums in the graph regarding the levels of the adjacent nodes. More precisely, the level of every node (excepting the root) may be computed as the highest level among the respective neighbors added of one [11]. In this way, the level computation for a node n may be described as a fixpoint system[5]:

$$\begin{cases} \textbf{Initialization:} \\ level(root) = 0; \quad level(k) = \infty, & \forall k \text{ other than root;} \\ \textbf{Fixed Point Iteration:} \\ level(n) = \min(level(m) + 1), & \forall m \in \text{neighbors of n.} \end{cases}$$

Algorithm 2 shows the pseudo-code of the implemented Unordered BFS and the respective optimizations suggested by this work. In order to explore the fixed point feature, an unordered worklist (wl) structure must by maintained by the algorithm. A structure of this type makes possible any node to be picked up. Thereafter, the algorithm is able to processing several nodes in

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

[5] A fixed point iteration $x^{(k+1)} := f(x^{(k)})$ yields a decreasing (increasing) monotonic sequence which converges to a fixed point x^* such that $f(f(\ldots f(x^*)\ldots)) = f^n(x^*) = x^*$ [26].

parallel. Nevertheless, as threads access the worklist in a non-deterministic way, a node may be eventually set with a level higher than the correct final value. Despite this, the successive iterations executed by threads over the worklist generate a monotonic decrement of the level of each processed nodes. This decrement stabilizes at the appropriated node level (fixed point iteration).

This process is exemplified by Figure 1. Initially at step (i), three nodes have already been inspected by the algorithm. The root r is the first node processed (black background) by a thread, and its respective neighbors are active (gray background) in the worklist. The level of them has been defined as the level of the parent (zero) added of one. At this moment, both nodes (a and b) may be taken by a thread. Eventually, in the step (ii), the node b was selected and processed by a thread. Consequently, its unprocessed neighbors (just the node c) were selected and became actives in the worklist. Next, two nodes are available to be processed: a (from the previous step) and c (from the current step). As there is no a specific order to process the nodes, anyone may be taken. In this case, the node c was selected and its not processed neighbor (node d) become active with the level defined as three. Finally, at step (iii), just the node a remains in the worlist. So, it is taken by a thread, and the level of its not processed neighbor (node d) is set to the appropriated value, i.e., two.

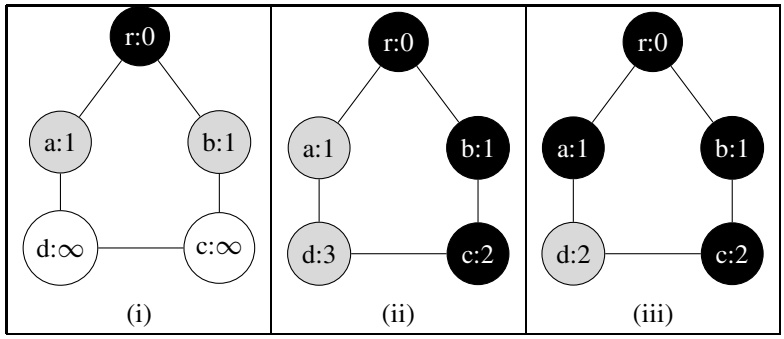

Figure 1: Fixed Point Iteration Example.

Two optimizations previously presented by [11] were incorporated into the Unordered BFS (Algorithm 2).

1. **Work chunking:** In order to minimize the access to the global worklist ws by threads, it was adopted the strategy of making each thread able to dequeue a bunch of nodes (*WORK_CHUNK*) from the worklist rather than only one at a time (lines 6-10). As each thread caches a local worklist $threadws$ (a subset of ws), they can process it independently of each other. Thus, a local set of new actived nodes ($relaxedws$) is built by the running threads, and finally, each $relaxedws$ is merged into the global worklist (lines 20-22). The size of the bunch of nodes picked up by each thread is dynamically defined in the algorithm. Actually, empirical test points out to a better algorithm performance when considering the *WORK_CHUNK* size as fifty percent of the total of nodes available in the global worklist [24].

Algorithm 2 Optimized Unordered BFS Algorithm

```
1:  Graph G = input();     // Read in graph
2:  Worklist ws;
3:  ws.add(root);
4:  while (not ws.isEmpty()) {
5:      // Work Chunking
6:      Worklist threadws;
7:      foreach (Node n: ws) {     // Atomic access to ws
8:          if (threadws.size() > WORK_CHUNK) break;
9:          threadws.add(n);
10:     }
11:     // Fixed Point Iteration
12:     Worklist relaxedws;
13:     foreach (Node n: threadws) {
14:         foreach (Node v: G.neighbors(n)) {
15:             int level = n.getLevel() + 1;
16:             if (level > v.getLevel()) {
17:                 v.setLevel(level);
18:                 relaxedws.add(v);
19:     } } }
20:     // Relaxing nodes
21:     foreach (Node m: relaxedws) {
22:         atomic ws.add(m);
23: } } } }
```

2. **Wasted work reduction:** It was implemented a strategy to reduce the time wasted by each waiting thread (idle threads) in which all threads remove active elements from one end of the worklist and add to the other. The concurrent access of each worklist end is managed by two distinct access lock. Naturally, this approach relaxes the strict order in which the worklist is processed. However, to ensure this strict order increases the access time of the worklist beyond the benefit of reducing the amount of wasted work.

These two related optimizations were earlier explored in the work [24]. But the comparison of the parallel Unordered RCM was made against a correspondent serial algorithm implemented by [13].

Step 2: Count nodes by level. Initially, all nodes of the graph are divided among the set of threads. Each thread counts locally how many nodes, from its respective subset of nodes, belong to each level. Moreover, a local maximum level is determined by each thread. Next, the global maximum level is computed through the comparison of each maximum local level of each thread. Finally, as the number of levels of the graph is already computed, thus a range of levels is assigned

to each thread that, in turn, counts how many nodes were computed by all threads in its respective range. The result is stored in the global array.

Step 3: Compute the prefix sum of levels. The prefix sum [6] calculus implemented in this work is based on the algorithm proposed by [1]. Initially, each thread computes the prefix sums of the $\frac{n}{p}$ elements it has locally. The total number of elements (n) corresponds to the maximum level accounted for by the previous step of the Unordered RCM algorithm. The value p is related to the number of threads. Hereafter, the last prefix sum of each thread is assigned to arrays responsible for guiding the data exchanging process among the threads. In fact, the local prefix sum values are exchanged and each thread accumulates the respective received value. The rule to determine a pair of threads that is going to communicate is through a XOR (exclusive OR) bitwise operation between the unique identifier of the sender thread and a constant related to the group of the receiver thread. Finally, each thread merges the result from the accumulated prefix sums with each local prefix sum initially computed.

Step 4: Build the permutation array. The underlying concept behind the operation of this phase is the pipelining of threads actions among the levels of the graph. For this, one thread is assigned for each level, and the communication among them takes place in pairs: a thread responsible for a level l plays a producer (writer) role, while a thread assigned to the level $l + 1$ acts as a data consumer (reader). Every read/write operation happens over the permutation array. The controller of this implemented producer/consumer paradigm is done through the prefix sum array ($sums$) generated in the previous step. Actually, it is never changed once its values are used as bounds for threads operations. In this way, this pipeline makes possible the construction of the permutation array in a parallel way: while a thread writes the children nodes from a level l in the permutation array, another thread reads these ones in order to write the corresponding neighbors of them at level $l + 1$ in the permutation array.

4 LEVELED PARALLEL REVERSE CUTHILL-MCKEE

The second parallelization of RCM presented by [16] makes use of a different approach. There is no an unordered execution of threads through the graph as a whole. Instead, the graph is traversed level by level, and the parallelism is restricted to one level per time. In each level-iteration, the permutation array is built gradually. The step-by-step executed by the Leveled RCM is detailed in Algorithm 3.

Iterations are composed of four steps carried out inside the outermost parallel loop (line 3). Initially, threads explore the neighbors of the parent nodes previously stored in the permutation array (lines 6-16). This expansion step sets the appropriated level of each explored neighbor. In the two next steps, the number of children (neighbors) per level is accounted (reduction step – lines 18-20), and the prefix sum is generated subsequently (lines 22-24). Finally, the neighbors processed in the current iteration are stored in the permutation array (placement step).

[6] 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 $[a_0, (a_0 \oplus a_1), \ldots, (a_0 \oplus a_1 \oplus \ldots \oplus a_{n-1})]$.

Algorithm 3 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:     } }
```

4.1 Optimized Leveled Parallel RCM

The proposed new version of the Leveled Parallel RCM follows the same strategy of the original one. It is based on a graph traversing level-by-level and the parallelism is restricted to each level. However, an alternative main data structure is used in order to conduct the nodes processing iteratively [25]. Actually, during each iteration, the processing relies on a Bucket Array in which inspected nodes are stored in specific buckets or cells. In order to determine the correct bucket for

storing a node, it is used a hashing function. As this type of function may generate a same output for distinct inputs, each bucket array position is able to enqueue nodes. Figure 2 presents an operating scheme of the bucket array implemented for the optimized Leveled RCM. Furthermore, the use of the Bucket Array as main data structure produces an important impact on the design of the new proposed algorithm. In effect, the number of steps of the original algorithm were merged into just two. The algorithm idea is presented in Algorithm 4.

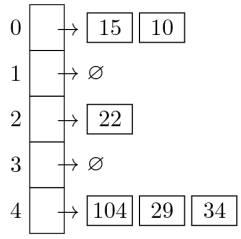

Figure 2: Bucket Array with hash function $h(x) = x \bmod 5$.

Algorithm 4 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:     } }
```

The algorithm starts setting a pseudo-peripheral node (*source* node) at the first position of the permutation array. Hereafter, the parallel processing comes into play inside the outermost loop (line 3). Every iteration is composed of two steps. In the first one, all nodes to be processed are read from the permutation array (P), and the respective children of each node are inspected. A global range (*parent1:parentN* – line 6) is kept in order to control the parents to be processed in each iteration. Thus, the appropriated level is set to every $UNLABELED$ child (line 11), the status of each one is updated to $LABELED$ (line 13), and they are stored in the correct positions in the bucket array $generation$ (line 14). Two features related to the use of a bucket array in this step are ever relevant for the algorithm performance:

- **Implicit reduction step.** As each processed child node is stored in the bucket of its respective parent, the number of children per parent becomes automatically available through a simple node counter. It is incremented at each bucket insertion. Thus, there is no necessity of a separated step (reduction) for to execute this computation, as is done by the original algorithm.
- **Cheap hashing function.** Each cell or bucket of the main data structure corresponds to a parent node of some inspected node in the current iteration. So, the hashing function employed in mapping nodes to buckets involves just a memory address computation. In order words, to find out the correct position of a node in the bucket array is related to the cost of one memory access.

After all threads conclude the building bucket array process by means of the parent nodes inspection, they advance to the second iteration step. In this phase, each thread takes a bucket and stored the children mapped to it in the final permutation array (lines 21-22). This way of traversing the processed nodes, i.e. bucket-by-bucket instead of node-by-node, represents an important improvement for the performance of the algorithm. In effect, this strategy makes possible threads execute all placement work independently of each other. This because only two synchronized operations (lines 18 and 19) are necessary. Both of them are related to updating the range of the permutation array to be made available for the next iteration. In contrast, the original algorithm guides the threads work through the prefix sum computed in a previous step. This specific step is not necessary in the bucket-by-bucket computation proposed by the optimized algorithm.

All aforementioned modifications implemented in the original algorithm were early evaluated in a previous work [25]. However, the comparison was made against a free OpenMP-based implementation of the original algorithm. The evaluation shown that the Leveled RCM speedup is critically compromised by a straightforward replication of it to the OpenMP platform.

5 EXPERIMENTAL RESULTS

The performance evaluation of the Optimized **Leveled** and the Optimized **Unordered** RCM algorithms was against a serial implementation of RCM made available by the Boost Library [28]. A set of twenty structural symmetric and square matrices was selected from the University of

Florida Sparse Matrix Collection [9]. These matrices cover multiple type of problems in order to increase the dataset variety and the percentage of sparsity of each one is higher than 99.97%. The set of tested matrices is shown in Table 1. The columns present matrices and some characteristics of them: dimension, number of non-zeros, and average of non-zeros per row (NNZ/row). The program was coded in the *C* language and the parallelism was supported by OpenMP framework. The experiments were performed on a PC running Ubuntu Linux, version 14.04.5 LTS, with Kernel version 3.19.0-25. It consists of 2 Xeon processors of 8 cores each, operating at 2.4 GHz. Each core has a unified 256KB L2 cache and each processor has a shared 20MB L3 cache. The PC contains 32GB of main memory and code was compiled with GNU gcc version 4.8.4. The complete source code is available on GitHub repository [31].

Table 1: Sparse Tested Matrices.

Matrix	Dimension	Non-zeros	NNZ/row
benzene	8,219	242,669	30
FEM_3D_thermal1	17,880	430,740	24
rail_79841	79,841	553,921	7
thermomech_TC	102,158	711,558	7
Dubcova3	146,689	3,636,643	25
Ga41As41H72	268,096	18,488,476	69
F1	343,791	26,837,113	78
helm2d03	392,257	2,741,935	7
msdoor	415,863	19,173,163	46
inline_1	503,712	36,816,170	73
gsm_106857	589,446	21,758,924	37
Fault_639	638,802	27,245,944	43
tmt_sym	726,713	5,080,961	7
tmt_unsym	917,825	4,584,801	5
audikw_1	943,695	77,651,847	82
nlpkkt80	1,062,400	28,192,672	27
dielFilterV3real	1,102,824	89,306,020	81
dielFilterV2real	1,157,456	48,538,952	42
Serena	1,391,349	64,131,971	46
G3_circuit	1,585,478	7,660,826	5

Tables 2 and 3 show a quality and performance comparison, respectively, among the serial Boost Library implementation of RCM and the two modified parallel versions of the algorithm, Unordered and Leveled, which have been proposed in this work. The algorithms were performed five times for each pair (m_i, t_j), where m_i is a matrix of Table 1, and t_j is the number of threads between 1 and 12 (in steps of 2) used by each algorithm. For each (m_i, t_j) tested pair, the average was calculated from the reported 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 Unordered algorithm. This same number of threads t_j was used to select the corresponding (m_i, t_j) tested pair from the Leveled algorithm. Moreover, the Compressed Sparse Row format was the mechanism used to store each tested matrix. The operations applied on them were also performed using this format.

5.1 Reordering Quality

Table 2 shows the new bandwidth obtained after applying the permutation generated by each algorithm. In the column Bandwidth the original bandwith of the matrices is presented. The same band reordering values were reached by the three implementations for five of tested matrices. The Boost library overcomes both parallel algorithms only for three matrices: $Ga41As41H72$, $audikw\_1$, and $dielFilterV2real$. For the remaining matrices, both proposed algorithms attained the smallest values for the bandwidth. In particular, only for the *Serena* matrix, the algorithms reached a less expressive reordering result around 0.17%. Five matrices achieved an intermediate reduction rate (up to 56.04%): $Fault\_639$, $Ga41As41H72$, tmt_sym *benzene*, and tmt_unsym. With the other matrices, the percentage of reduction various between 95.60% ($nlpkkt80$) until 99.75% ($helm2d03$).

Table 2: Reordering Quality Comparison.

Matrix		Bandwidth After Permutation		
Name	Bandwidth	Boost	Unordered	Leveled
benzene	2,898	**1,699**	**1,699**	**1,699**
FEM_3D_thermal1	13,787	636	**635**	**635**
rail_79841	79,811	**418**	**418**	**418**
thermomech_TC	102,138	**262**	**262**	**262**
Dubcova3	146,356	2,130	**1,942**	2,268
Ga41As41H72	40,195	**32,974**	33,112	33,016
F1	343,754	**10,052**	**10,052**	**10,052**
helm2d03	391,403	**994**	**994**	**994**
msdoor	291,114	6,032	**5,823**	5,970
inline_1	502,403	**6,002**	**6,002**	**6,002**
gsm_106857	588,744	17,865	**17,742**	**17,742**
Fault_639	19,988	17,892	17,028	**16,324**
tmt_sym	1,921	1,140	**1,139**	**1,139**
tmt_unsym	2,161	**942**	**942**	**942**
audikw_1	925,946	**35,084**	35,102	35,102
nlpkkt80	550,481	32,726	35,258	**28,243**
dielFilterV3real	1,036,475	23,691	23,652	**23,633**
dielFilterV2real	948,032	**17,999**	18,045	18,045
Serena	81,578	81,673	**81,438**	**81,438**
G3_circuit	947,128	**5,068**	**5,068**	**5,068**

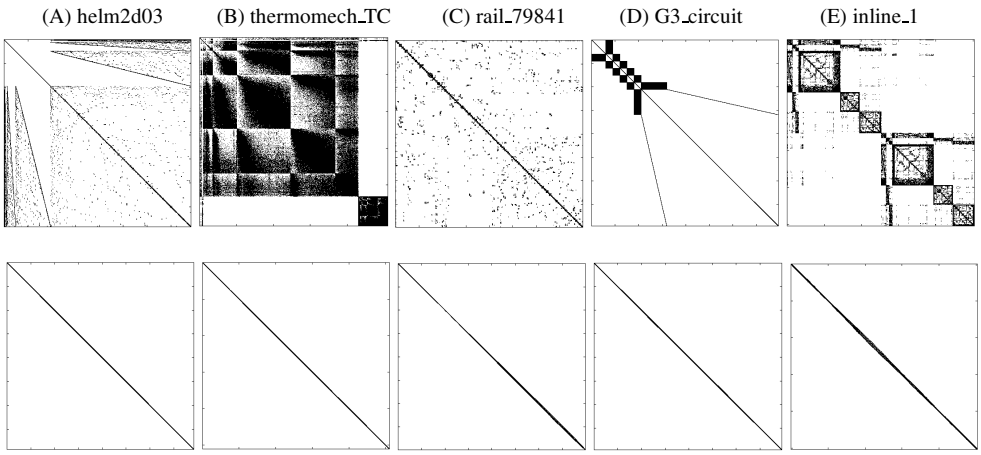

Figure 3: Sparse matrix pattern yielded by the Unordered RCM.

A strict comparison between the two proposed algorithms emphasizes the efficiency of both approaches in sense of quality reordering. Both of them reached the best bandwidth value with twelve matrices. But, for two matrices ($Dubcova3$ and $msdoor$), only the Unordered RCM attained the best reduction values. On the other hand, the Leveled RCM generates the best permutation for three other ones: $Fault\_639$, $nlpkkt80$, and $dielFilterV3real$. The presented values of bandwith reduction highlight the efficiency of all approaches in sense of quality reordering.

The quality reordering may also be graphically attested through Figure 3. It shows the five matrices with best bandwidth reduction rate reached by the algorithms. 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 Unordered RCM algorithm. Actually, the reordering pattern reached by any one of the three tested algorithms might be used to depicts the quality of applied permutation. This because the differences among the bandwith reduction reached by each one is less than 0.002% and may be negligible.

5.2 Reordering Performance

Table 3 shows the elapsed time (Time columns) by the algorithms, in scale of 10^{-3} seconds, to reorder each matrix considering the best set of threads (column #Th.). Other additional columns (Reduction) present the time reduction percentage reached by the respective parallel algorithm. The pseudo-peripheral vertex used as source node by the three evaluated algorithms is obtained through the algorithm mentioned in Section 2. Because this, the time spent in each pseudo-peripheral computation was not included in this performance analysis. As showed by the Table 3, for all tested instances both proposed algorithms attained better performance than Boost library. In fact, the Unordered RCM decreased the time reordering in two order of magnitude for twelve tested matrices. For other seven matrices, there was one order of magnitude in the time reduction. The Leveled RCM also presented an outstanding efficiency for the set of matrices.

Table 3: Performance Comparison. URCM: Unordered RCM, LRCM: Leveled RCM.

Matrix	#Th.	Time (x10^{-3} sec.)			Reduction (%)	
		Boost	URCM	LRCM	URCM	LRCM
benzene	2	1.782	0.096	0.132	94.16	92.59
FEM_3D_thermal1	6	1.961	0.095	0.260	95.16	86.74
rail_79841	2	2.720	0.788	0.542	71.03	80.07
thermomech_TC	4	3.818	0.483	1.093	87.35	71.37
Dubcova3	10	16.593	0.951	1.926	94.27	88.39
Ga41As41H72	6	85.534	1.641	11.054	98.08	87.08
F1	6	120.952	2.319	8.591	98.08	92.90
helm2d03	2	13.858	1.585	1.498	88.56	89.19
msdoor	4	89.861	2.402	6.323	97.33	92.96
inline_1	8	164.875	4.374	13.725	97.35	91.68
gsm_106857	8	103.744	3.689	10.543	96.44	89.84
Fault_639	8	128.757	4.173	11.758	96.76	90.87
tmt_sym	2	25.636	3.412	3.775	86.70	85.27
tmt_unsym	2	24.53	4.41	3.832	82.02	84.38
audikw_1	6	348.895	8.489	43.130	97.57	87.64
nlpkkt80	4	139.195	5.903	15.35	95.76	88.97
dielFilterV3real	8	404.739	9.611	53.626	97.63	86.75
dielFilterV2real	6	231.527	7.405	19.924	96.80	91.39
Serena	6	294.792	7.038	40.062	97.61	86.41
G3_circuit	2	40.042	12.080	8.923	69.83	77.72

A direct performance comparison between the two parallel approaches points out the superior efficiency of the Unordered algorithm. Indeed, the best times were reached by the Unordered RCM for sixteen tested matrices. On the other hand, the Leveled RCM obtained the best values for another subset of four instances. The time reduction percentage reached by the Unordered algorithm various between 69.83% ($G3\_circuit$) and 98.08% ($F1$). Analogously, the reduction ratio obtained by the Leveled RCM was from 71.37% ($thermomech\_TC$) to 92.96% ($msdoor$).

Figure 4 presents how speedups of the Unordered RCM scale as the number of threads is increased. Five matrices with the highest rate of speedup were selected. As shown by this figure, $audikw\_1$ and $dielFilterV3real$ are the matrices with best speedups of this group. It is relevant to notice the related matrices are able to scale until 12 threads, and the maximum speedup ratio reached by each one was 4.48X and 4.07X respectively. The speedup ratio reached by the other three matrices was of 3.06X ($Ga41As41H72$), 3.02X ($F1$), and 2.46X ($msdoor$). An important feature to point out about these five matrices is the large number of nonzeros per row of each one (approximately an average of 71 NNZ/row – for details see Table 1). Therefore, speedups of parallel algorithms like Unordered RCM which are based on a BFS approach are higher for graphs with a larger number of edges per node.

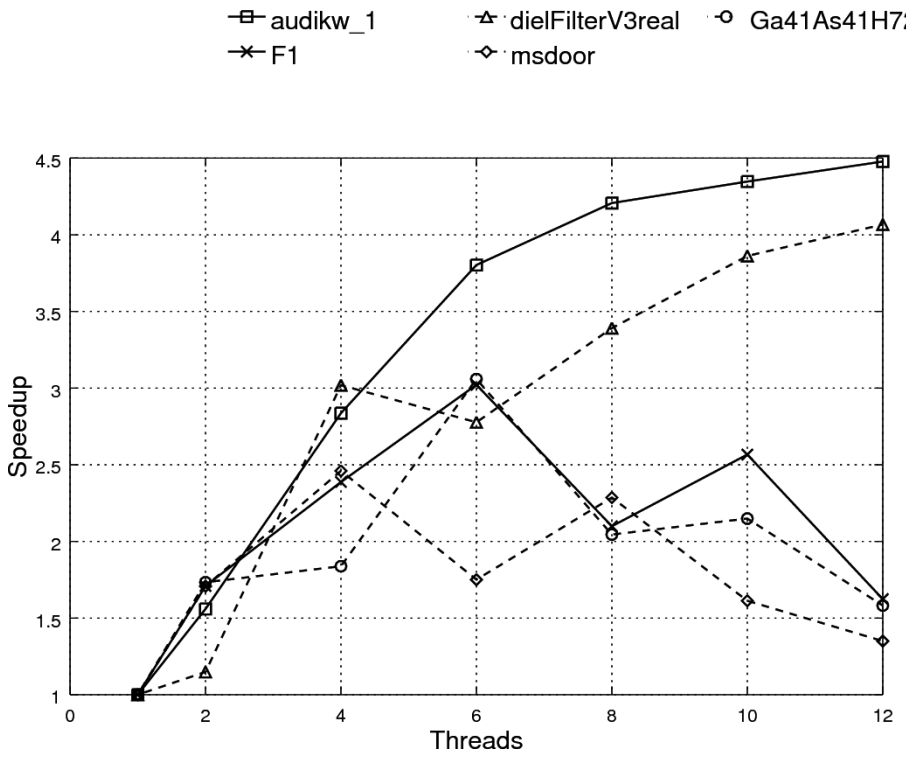

Figure 4: Speedup of Smallest Matrices.

Another feature is related to the no significant speedup observed for the Leveled RCM. As it uses a different general strategy based on a parallelism by level [16], a barrier must be placed between each level. Because of this, increasing the number of threads do not necessarily lead to an improvement of performance.

6 CONCLUSIONS AND FUTURE WORKS

This paper analysed two parallel strategies for the traditional Reverse Cuthill-McKee reordering algorithm. Two modified implementations were presented and the results achieved by both algorithms represent a significant improvement on the reordering time. With the Unordered RCM algorithm, it reached a time reduction of until 98.08% of the serial time obtained by the Boost library. Other significant results show the Unordered RCM achieving speedups until 4.478 with 12 threads. A relevant performance was also achieved by the Leveled RCM algorithm. The permutations generated by it reduced the bandwidth until 99.75%. It is noticeable the changes proposed for the two parallel algorithms led to outstanding performance enhancements. About the reordering quality, both implementations attained relevant bandwidth reduction. Therefore, the two modified parallel RCM algorithms may be considered as efficient approaches for the bandwidth minimization problem.

Both parallel implementations of the RCM presented in this work were supported by the OpenMP framework. However, some works have addressed the reordering problem through the use of another kind of parallelism tool. As example, [7] has developed a parallel tool for graph partitioning, and several works have discussed the parallelization of algorithms by the Galois System [11]. Moreover, other data structures and BFS strategies have been proposed for the parallelism of RCM. In fact, Hassam et al. [11] present relevant results from a wavefront BFS implementation, and Leiserson and Schardl [18] propose a novel implementation of a workset data structure, called "bag", in place of FIFO queue usually employed in BFS algorithms. The use of these new structures and strategies might promotes more improvements to the algorithms studied in this work. Moreover, additional comparisons against other mathematical libraries may ever endorse the relevant performance reached by the algorithms. In this way, a confrontation with the Harwell Subroutine Library (HSL) [12] – a collection of state-of-the-art packages for large-scale scientific computation – constitutes an important complementary test to be made.

RESUMO. O algoritmo Reverse Cuthil-McKee (RCM) constitui uma heurística bem conhecida para o reordenamento de matrizes esparsas. Ele é tipicamente aplicado para a melhoria do desempenho da computação de sistemas lineares de equações. Este artigo descreve duas abordagens paralelas propostas para o algoritmo Reverse Cuthill-McKee, assim como versões otimizadas baseadas em alguns aprimoramentos propostos. Na primeira abordagem há a exploração de uma estratégia para a redução de *threads* ociosas, enquanto a segunda abordagem faz uso de um *bucket array* estático como estrutura de dados principal, além de suprimir algumas etapas do algoritmo original. As modificações apresentadas conduzem a resultados relevantes tanto em termos do tempo de reordenamento quanto na redução da largura de banda. O desempenho de cada algoritmo é comparado com a sua respectiva implementação disponibilizada pela biblioteca *Boost*. O paralelismo é suportado pelo *framework* OpenMP e ambas as versões do algoritmo são testadas com matrizes esparsas de grande porte e estruturalmente simétricas.

Palavras-chave: RCM paralelo, redução de largura de banda, matrizes esparsas.

REFERENCES

[1] S. Aluru. Teaching parallel computing through parallel prefix, The International Conference for High Performance Computing, Networking, Storage and Analysis, (2012), Salt Lake City, USA.

[2] G. Anderson and D. Ferro & R. Hilton. Connecting with Computer Science – Second Edition. Course Technology, Cengage Learning, Boston, (2011).

[3] M.J. Atallah & S. Fox. Algorithms and Theory of Computation Handbook. CRC Press, Inc., Boca Raton, FL, USA, (1978).

[4] P.R. Amestoy, T.A. Davis & I.S. Duff. An Approximate Minimum Degree Ordering Algorithm. *SIAM J. Matrix Anal. Appl.*, **17**(4) (1996), 886–905.

[5] L. Cardelli & X. Leroy. Abstract Types and The Dot Notation, Research report 56, Digital Equipment Corporation, Systems Research Center, March 10, (1990).

[6] D. Chazan & W. Miranker. Chaotic Relaxation. *Linear Algebra and its Applications*, **2**(2) (1969), 199–222.

[7] C. Chevalier & F. Pellegrini. PT-Scotch: A Tool for Efficient Parallel Graph Ordering. *Parallel Comput.*, **34**(6-8) (2008), 318–331.

[8] E. Cuthill & J. McKee. Reducing the Bandwidth of Sparse Symmetric Matrices, at "Proceedings of the 24th ACM National Conference", pp. 157–172, ACM '69, (1969).

[9] T.A. Davis & Y. Hu. The University of Florida Sparse Matrix Collection. *ACM Trans. Math. Softw.*, **38**(1) (2011), 1:1–1:25.

[10] M. Kulkarni, K. Pingali & B. Walter et al. Optimistic parallelism requires abstractions, at "Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation", pp. 211–222, PLDI '07, (2007).

[11] M.A. Hassaan, M. Burtscher & K. Pingali. Ordered vs. Unordered: A Comparison of Parallelism and Work-efficiency in Irregular Algorithms, at "Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming", pp. 3–12, PPoPP '11, (2011).

[12] Harwell Subroutine Library. A collection of Fortran codes for large scale scientific computation, (2011), `http://www.hsl.rl.ac.uk/`.

[13] B. Lugon & L. Catabriga. Algoritmos de reordenamento de matrizes esparsas aplicados a precondicionadores ILU(p), at "Simpósio Brasileiro de Pesquisa Operacional", pp. 2343–2354, XLV SBPO, (2013).

[14] A. George. Nested Dissection of a Regular Finite Element Mesh. *SIAM Journal on Numerical Analysis*, **10**(2) (1973), 345–363.

[15] A. George & J.W.H. Liu. A Fast Implementation of the Minimum Degree Algorithm Using Quotient Graphs. *ACM Trans. Math. Softw.*, **6**(3) (1980), 337–358.

[16] K.I. Karantasis, A. Lenharth, D. Nguyen, M. Garzarán & K. Pingali. Parallelization of Reordering Algorithms for Bandwidth and Wavefront Reduction, at "Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis", pp. 921–932, SC '14, (2014).

[17] G. Karypis & V. Kumar. A Parallel Algorithm for Multilevel Graph Partitioning and Sparse Matrix Ordering. *J. Parallel Distrib. Comput.*, **48**(1) (1998), 71–95.

[18] C.E. Leiserson & T. B. Schardl. A Work-efficient Parallel Breadth-first Search Algorithm (or How to Cope with the Nondeterminism of Reducers), at "Proceedings of the Twenty-second Annual ACM Symposium on Parallelism in Algorithms and Architectures", pp. 303–314, ACM, (2010).

[19] W. Lin. Improving Parallel Ordering of Sparse Matrices Using Genetic Algorithms. *Appl. Intell.*, **23**(3) (2005), 257–265.

[20] W. Liu & A.H. Sherman. Comparative Analysis of the Cuthill-McKee and the Reverse Cuthill-McKee Ordering Algorithms for Sparse Matrices. *SIAM Journal on Numerical Analysis*, **13**(2) (1976), 198–213.

[21] U. Meyer, A. Negoescu & V. Weichert. New Bounds for Old Algorithms: On the Average-Case Behavior of Classic Single-Source Shortest-Paths Approaches, at "Theory and Practice of Algorithms

in (Computer) Systems: First International ICST Conference – Proceedings", pp. 217–228, ICST, (2011).

[22] L. Dagum & R. Menon. OpenMP: An industry-standard API for shared-memory programming. *IEEE Comput. Sci. Eng.*, **5**(1) (1998), 46–55.

[23] C.H. Papadimitriou. The NP-Completeness of the bandwidth minimization problem. *Computing*, **16**(3) (1976), 263–270.

[24] T.N. Rodrigues, M.C.S. Boeres & L. Catabriga. An Implementation of the Unordered Parallel RCM for Bandwidth Reduction of Large Sparse Matrices, at "Congresso Nascional de Matemática Aplicada e Computacional", XXXVI CNMAC, (2016), (in press).

[25] T.N. Rodrigues, M.C.S. Boeres & L. Catabriga. An Optimized Leveled Parallel RCM for Bandwidth Reduction of Sparse Symmetric Matrices, at "Simpósio Brasileiro de Pesquisa Operacional", XLVIII SBPO, (2016), (in press).

[26] B.S.W. Schröder. Algorithms for the Fixed Point Property. *Theoretical Computer Sciense*, **217**(2) (1999), 301–358.

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

[28] J.G Siek, L.-Q. Lee & A. Lumsdaine. The Boost Graph Library: User Guide and Reference Manual, Addison-Wesley Longman Publishing, Boston, MA, USA (2002).

[29] Y. Saad. Iterative methods for sparse linear systems, Addison-Wesley Society for Industrial and Applied Mathematics, Philadelphia, PA, USA (2003).

[30] G.K. Kumfert. *An object-oriented algorithmic laboratory for ordering sparse matrices*, Lawrence Livermore National Laboratory and United States, (2000).

[31] T.N. Rodrigues. tnas/reordering-library: Trends in Applied and Computational Mathematics, `doi:10.5281/zenodo.583729`, `https://doi.org/10.5281/zenodo.583729`.