DOI: 10.15439/2017F179
ISSN 2300-5963 ACSIS, Vol. 11

A Non-Speculative Parallelization of Reverse Cuthill-McKee Algorithm for Sparse Matrices Reordering

Thiago Nascimento Rodrigues,
Maria Claudia Silva Boeres, Lucia Catabriga
Federal University of Espírito Santo
Av. Fernando Ferrari, 514 - Goiabeiras, Vitória, 29.075-910, Brazil
Email: {tnrodrigues, boeres, luciac}@inf.ufes.br

***Abstract*—This work presents a new parallel non-speculative implementation of the Unordered Reverse Cuthill-McKee algorithm. Reordering quality (bandwidth reduction) and reordering performance (CPU time) are evaluated in comparison with a serial implementation of the algorithm made available by the state-of-the-art mathematical software library HSL. The bandwidth reductions reached by our parallel RCM were more than 90% for several large matrices out of the ones tested, and the time reordering improvement was up to 57.82%. Speedups higher than 3.0X were achieved with the parallel RCM. The underlying parallelism was supported by the OpenMP framework and three strategies for reducing idle threads were incorporated into the algorithm.**

I. 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 [1]. 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 order to simplify the solution of this type of system, the bandwidth minimization plays an efficient role. This pre-processing method consists of finding a permutation of rows and columns of a 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. In this way, let A be a structurally symmetric matrix, i. e., if $a_{ij} \neq 0$ then $a_{ji} \neq 0$, but not necessarily $a_{ij} = a_{ji}$, whose diagonal elements are all non-zero. The bandwidth of A denoted by $\beta(A)$ is defined as the greatest distance from the first nonzero element to the diagonal, considering all rows of the matrix [1]. 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, $\beta(A) = \max\limits_{i=2,3,\ldots,n} \{b_i(A)\}$.

Since Papadimitrou [2] 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.

The Reverse Cuthill-McKee (RCM) is a traditional heuristic for the bandwidth reduction problem. It was originally presented by [3], and a performance modification for it was proposed by [4] posteriorly. The approach based on looking into a corresponding graph structure is also explored by several other algorithms. Some of the most often referred for the bandwidth minimization problem are Sloan [5] and GPS [6]. They are also able to provide quality solutions in an efficient way.

Classically, algorithms like the aforementioned implement the matrix reordering in a serial way. Nevertheless, the advances toward the massive use of multi-core processors on scientific computation has leveraged significant performance improvements related to the solution of sparse matrices problems. In this context, in 2014 [7] described the first parallelization of the RCM algorithm, which was based on the speculative parallel model. In this parallelism model, a runtime system detects dependence violations between concurrent computations and rolls back conflicting computations as needed [8]. As the RCM is organized around a graph, which is implemented as a pointer-based data structure, it is considered as an irregular algorithm [9]. Algorithms of this type exhibit a complex pattern of parallelism which must be found and exploited at runtime [10]. To explore this kind of parallelism and to reduce the programming burden, [7] use the Galois system [11] which gives support to the speculative parallelism.

Making use of another parallel model, this paper proposes a non-speculative OpenMP-based implementation of the Unordered Parallel RCM algorithm presented by Karantasis et al. [7]. This implementation strategy was considered once the non-speculative parallel model is the traditional manner to speedup every type of algorithm, and the OpenMP [12] framework for parallelism is widely used in industry as well as in academia. To reach an efficient non-speculative parallelization of the RCM, three optimizations for reducing idle threads were incorporated into the implemented algorithm. The performance evaluation of the Unordered RCM algorithm was against the HSL [13], a state-of-the-art mathematical software library that contains a collection of Fortran codes for large-scale scientific computations.

The outline of the paper is as follow. In the next section, an efficient sparse matrix storage format is described. Section III is dedicated to detailing an auxiliary parallel algorithm implemented for pseudo-peripheral nodes finding. The Unordered Parallel RCM algorithm is presented in the subsequent section, as well as the optimizations proposed by this work. In Section V, all tests and achieved results are described. Conclusions and future works are addressed in Section VI.

II. Optimized Storage Format

In many scientific computations, the manipulation of sparse matrices is considered the crux of the design. Generally, the nonzero elements in a sparse matrix constitute a very small percentage of data. This irregular nature of sparse matrix problems has led to the development of a variety of compressed storage formats. The Compressed Sparse Row (CSR) used in this work is an important sparse matrix storage method which has been widely applied in most sources [1]. 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 [14]. The content of each array is as follow. Figure 1 illustrates this technique.

- **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 plus one.

$$A = \begin{bmatrix} 1 & 1 & 5 & 0 & 0 \\ 3 & 4 & 0 & 0 & 0 \\ 6 & 0 & 7 & 8 & 9 \\ 0 & 0 & 3 & 6 & 0 \\ 0 & 0 & 2 & 0 & 5 \end{bmatrix}$$

AA	1	1	5	3	4	6	7	8	9	3	6	2	5
JA	1	2	3	1	2	1	3	4	5	3	4	3	5
IA	1	4	6	10	12	14							

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

III. Parallel Pseudo-peripheral Node Finding

Empirical data show that the quality of reordering algorithms are highly influenced by the nodes chosen as the source for Breadth-First Search (BFS) and RCM algorithms [15]. Often, a heuristic is used for this purpose. Thus, considering $d(x,y)$ the distance between vertices x and y in a graph G, i.e., the length of the shortest path between x and y, the graph diameter is defined as $\delta(G) = max\{d(x,y)|\ x,y \in vertices\ of\ G\}$. Then, ideally, one of two nodes in a pair (x,y) that achieves the diameter, denoted as peripheral nodes, can be used as a starting point. However, these nodes are expensive to determine. Instead, a pseudo-peripheral node, which has approximately the greatest distance from each other in the graph, is picked up as source node for constructing the level set structure[1] of these algorithms.

Moreover, the nodes choice strategy employed in order to select ones to be expanded at each search level also impacts significantly on the reordering quality. In this work, the pseudo-peripheral node finding heuristic described by [16] was implemented for the RCM algorithm. The pseudo-code is presented in Algorithm 1.

Algorithm 1 Parallel Pseudo-Diameter Algorithm

```
Input: Graph g, ShrinkingStrategy strategy, float CHUNK
Output: Node start, Node end
BFS forwardBFS, reverseBFS;
GraphDiameter diameter;
diameter.start = graph.vertexOfMinimunDegree();
diameter.end = -1;  {
forwardBFS = Parallel_BFS(g, diameter.start, CHUNK);
int localDiameter = forwardBFS.height();
List candSet = forwardBFS.verticesAt(localDiameter);
candSet = strategy.shrink(candSet);
int minWidth = MAX_INT;
foreach (Node candidate : candSet)  {
    reverseBFS = Parallel_BFS(g, candidate, CHUNK);
    if (reverseBFS.width < minWidth)  {
        if (reverseBFS.height > localDiameter)  {
            diameter.start = candidate;
            diameter.end = -1;
            break;
        else
            minWidth = reverseBFS.width;
            diameter.end = candidate;
}  }  }

} while (diameter.end == -1);
if (forwardBFS.width > reverseBFS.width)
    return (diameter.end, diameter.start);
return (diameter.start, diameter.end);
```

[1]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.

The pseudo-diameter computation uses two BFS engines (line 1). The $forwardBFS$ always uses the current start vertex as root. The $reverseBFS$ variable uses candidates for the end vertex as root. Initially, the start node is chosen to be any vertex of smallest degree (line 3) and the end node is unknown (line 4). Next, the algorithm enters the main outer loop which does not exit until a suitable end node has been determined and all candidates have been exhausted. For each iteration of the outer loop, a forward breadth-first search (line 6) is performed, the current diameter is set as the height of the level structure, and the list of all vertices that are in the farthest level set ($candSet$) is gotten (line 8).

According to [16], the most important optimization incorporated by this algorithm is the shrinking strategy (line 9). Instead of performing a reverse breadth-first search on all vertices that are farthest away from the start vertex, it is much faster to only try a selected subset. For this work, the heuristic of choosing a single vertex of each degree was adopted [17].

Therefore, after applying a shrinking strategy, the list of candidate nodes is processed. For each candidate for end vertex in candidate list (line 11), a reverse breadth-first search is done. As the aim is to find out the candidate whose reverse breadth-first search has the minimum width, so a local variable $minWidth$ is initialized to an arbitrarily large number (line 10). If it is found a candidate that has a narrower level structure than the forward breadth-first search, then this candidate vertex is promoted to the new start vertex (line 15) and the algorithm is restarted. The $break$ in line 17 affects only the inner loop (lines 11-21) and jumps to the line 36. Since $diameter.end$ is still undetermined, the outer loop (lines 5-21) starts a new iteration. If the reverse breadth-first search is narrower than the most narrow reverse breadth-first search so far (line 18), then a new minimum width has been found (line 19), and the candidate is chosen as the end vertex (line 20).

It is important to observe that the main computation to calculate the pseudo-diameter is performing multiple BFS (lines 5 and 11). In this work, the Unordered Parallel BFS (Algorithm 2) presented in the next section is used as a way to parallelize this essential step of the pseudo-peripheral node finding algorithm. The other steps of this algorithm are executed sequentially.

IV. Unordered Parallel RCM Algorithm

The serial Cuthill-McKee algorithm [3] is based on a BFS strategy, in which the graph is traversed by level sets. 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]. However, in 1971, the Reverse Cuthill-McKee algorithm was presented by [4]. It was empirically observed that reversing the Cuthill-McKee ordering yields a better permutation scheme for matrix reordering problems.

The Unordered Parallel RCM proposed by [7] is based on the construction of a level structure, and an RCM-valid permutation is built after a complete level structure is computed. The four major algorithms steps are presented and detailed in the next sections.

A. Unordered Breadth-First Search (Step 1)

Algorithm 2 presents the non-speculative Unordered BFS including three proposed optimizations. The key aspect of the approach in which the algorithm is based on relates the level of a node with a local minimum in the graph. In fact, excepting the root, the level value of a node corresponds to the highest level among neighbors added of one [18]. Thus, the level computation for a node n may be described as a fixpoint system[2]:

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

In order to explore this 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 process several nodes in parallel. As the iteration over the main worklist (wl) do not have a strict order, it may happen that a node is temporarily assigned a level that is higher than the final value. However, the level will monotonically decrease until it reaches the correct value (a fixed point). This step of repeatedly taking a closest known vertex u and testing if $level[v] \leq level[u] + 1$ for all of its v neighbors (lines 19-25), is called node relaxation, which relaxes constraints on the shortest path between two nodes. Particularly, the absence of order in the node iteration and the fact that nodes can be relaxed many times characterize a chaotic relaxation [20].

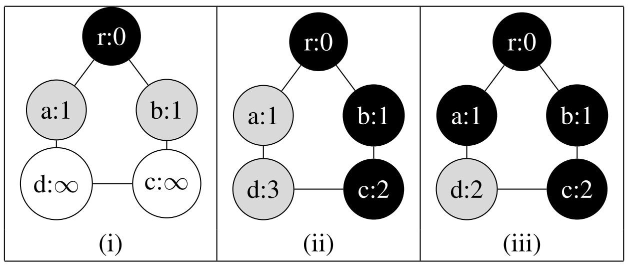


Fig. 2: Fixed Point Iteration Example.

Figure 2 describes an example of the chaotic relaxation process executed by speculative BFS. At the step (i), the root r

[2]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 $f(f(\ldots f(x^*)\ldots)) = f^n(x^*) = x^*$ [19].

Algorithm 2 Parallel Unordered BFS Algorithm

```
Input: Graph G, Node root, float CHUNK
 1: Worklist wl = $\emptyset$;
 2: ENQUEUE(wl, root);
 3: parallel while (wl $\neq \emptyset \vee$ hasUnreachedNodes) {
 4:     // Shifting head
 5:     atomic {
 6:         localHead = wl.head;
 7:         localTail = wl.tail;
 8:         sizeChunk = CHUNK * (localTail - localHead);
 9:         wl.head += sizeChunk;
10:     }
11:     // Work Chunking
12:     Worklist localwl;
13:     while (localwl.size() $<$ sizeChunk) {
14:         Node v = wl.dequeueAtPosition(localHead++);
15:         ENQUEUE(localwl, v);
16:     }
17:     // Fixed Point Iteration
18:     Workset relaxedwl;
19:     foreach (Node n: localwl) {
20:         int level = n.getLevel() + 1;
21:         foreach (Node v: G.neighbors(n)) {
22:             if (level $<$ v.getLevel()) {
23:                 atomic v.setLevel(level);
24:                 ENQUEUE(relaxedwl, v);
25:     } } }
26:     // Relaxing nodes
27:     foreach (Node m: relaxedwl)
28:         atomic ENQUEUE(wl, m);
29: }
```

has been processed (colored black) and nodes a and b in gray are actives in the global list. In the intermediate step (ii), node b has randomly been selected from the global list. After the activation of node c by b, it has been picked up from the global list instead of the another possible active node a. Because of this unordered choice, the node d has become active and its level has temporarily been set as three. In the last step (iii), the last active node a has been selected from the global list and it has updated the level of its neighbor d with the correct value.

In this work, two optimizations suggested by Hassaan, Burtscher, and Pingali [18] (Work chunking and Wasted work reduction) and a new proposed one (Shifted head) were applied in the implemented Unordered BFS algorithm. Each implemented optimization is detailed below.

1) **Work chunking** (lines 12-16). To reduce the overhead of accessing the main worklist, it was adopted the strategy of making each thread able to remove a chunk of active elements from the worklist instead of just one element. In this way, a newly created worklist is cached locally by each thread, and after the entire local chunk is processed (fixed point iteration), a set of new activated (relaxed) nodes is generated. This newest worklist is discharged into the main worklist by the respective thread. With this optimization, each synchronization is executed by a chunk of nodes rather than node by node.
2) **Wasted work reduction** (lines 14 and 28). 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.
3) **Shifted head** (lines 5-10). Aiming the reduction of the lock time spent by each thread, it was implemented an optimization in which the worklist head is shifted to the first position after the chunk size of the current thread. After this shifting, the access lock to the worklist head is released, and the thread starts the dequeue operation itself. Concomitantly, another different thread grabs the access lock and carries out a subsequent head shifting.

Such modifications led to the non-speculative parallel version of BFS algorithm (Algorithm 2). The parallelism begins at line 3 where threads are triggered. The shifted head optimization is carried out by the first atomic section of the algorithm (lines 5 to 10). Every time a thread reaches this region, it stores locally the current memory position of the head ($localHead$) and the tail ($localTail$) of the global main worklist wl. In turn, the global head is shifted $sizeChunk$ positions. The next two stages of the algorithm, work chunking (lines 12-15) and fixed point iteration (lines 18-25), are executed concurrently by each thread. In fact, the cached locally worklist ($localwl$) makes possible the independent nodes processing in each scope of thread. In the second synchronization point of the algorithm (lines 27-28) there is no any dependence with the first one (shifting head). Because of this, the algorithm is able to reduce the idle time of threads.

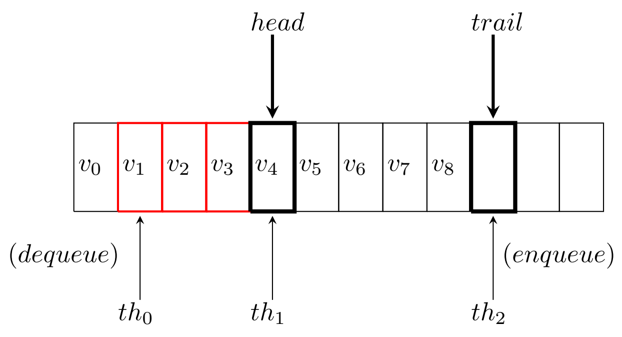


Fig. 3: Multithreading FIFO queue.

Figure 3 describes an iteration of the implemented parallel BFS. At the execution point presented by the figure, a thread th_0 has executed a shifting head and is carrying out the dequeue operation of all its corresponding nodes (red positions

v_1, v_2, v_3). At this moment, the access lock of the main worklist head has already released by th_0 and, furthermore has already grabbed by th_1. Concomitantly, a thread th_2 is executing the respective enqueueing of the nodes processed by it. It is important to notice that all three threads are performing their corresponding operations in a completely parallel way. The synchronization happens just when th_1 must wait the shifting head executed by th_0.

B. Counting Nodes by Level (Step 2)

Computing the number of nodes per level of a graph in a parallel way can be separated in three stages as described by the Algorithm 3 originally presented by [7].

Algorithm 3 Parallel Counting Nodes by Level Algorithm

```
Input: Graph G
Output: Array counts, int max_level
 1: foreach (Node n : G) {
 2:     local_count[th_id][n.level]++;
 3:     local_max[th_id] = max(local_max[th_id], n.level);
 4: }
 5: foreach (int id : threads) {
 6:     max_level = max(max_level, local_max[id]);
 7: }
 8: foreach (int l : [0:max_level]) {
 9:     foreach (int id : threads) {
10:         counts[l] += local_count[id][l];
11: } }
12: return [counts, max_level]
```

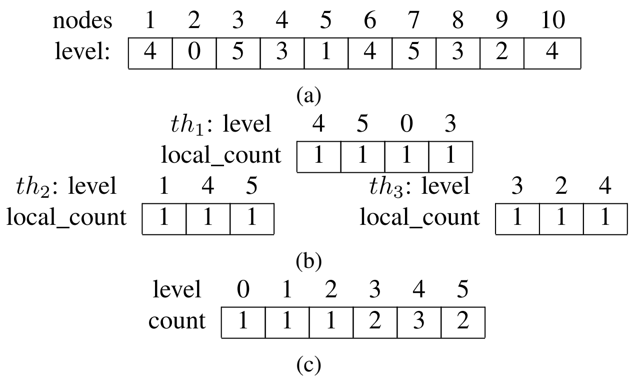


Fig. 4: Example of parallel counting nodes by level.

In the initial stage, 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 (lines 1-4). In the subsequent stage (lines 5-7), the global maximum level is computed through the comparison of each maximum local level of each thread. In the final stage (lines 8-10), 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 *counts* array.

Figure 4 describes an example of the parallel process of counting nodes per level. The respective level of each node is stored in the array of Figure 4(a). In Figure 4(b), a range of the levels is assigned to each one of three threads (th_1, th_2, th_3) and the number of nodes by level is locally computed. The Figure 4(c) presents the final array as a result of the merge of each locally counting carried out by the threads.

C. Prefix Sum (Step 3)

In this work, the Algorithm 4 was implemented for the prefix sum[3] calculus. It is based on the algorithm proposed by [21]. Initially, each thread computes the prefix sums of the $\frac{n}{p}$ elements it has locally (lines 3-5). The total number of elements (n) corresponds to the maximum level (max_level) accounted for by the previous step of the Unordered RCM algorithm. The value p is related to the number of threads.

Algorithm 4 Parallel Prefix Sum Algorithm

```
Input: Array counts, int max_level
Output: Array prefix_sum
 1: int num_changes = $log_2$(threads.size());
 2: int chunk = threads.size() / max_level;
 3: for (int i = thId; i < thId + chunk; i++) {
 4:     prefix_sum[i] = prefix_sum[i-1] + counts[i];
 5: }
 6: cPrefix[thId] = cTotal[thId] = prefix_sum[thId + chunk];
 7: lPrefix[thId] = lTotal[thId] = prefix_sum[thId + chunk];
 8: for (i = 0; i < num_changes - 1; i++) {
 9:     thId' = thId $\otimes$ $2^i$;
10:     if (thId' < threads.size() $\wedge$ thId' $\neq$ thId) {
11:         if (thId' < thId) {
12:             lPrefix[thId'] += cTotal[thId];
13:             lTotal[thId'] += cTotal[thId];
14:         else
15:             lTotal[thId'] += cTotal[thId];
16:         }
17:         cPrefix[thId] = lPrefix[thId];
18:         cTotal[thId] = lTotal[thId];
19: } }
20: for (int i = thId; i < thId + chunk; i++)
21:     prefix_sum[i] += lPrefix[i];
22: return prefix_sum;
```

In the second phase of the algorithm, the last prefix sum of each thread is assigned to four arrays (lines 6-7) which are 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 (lines 8-19). The rule to determine a pair of threads that are going to communicate is through a XOR (exclusive OR, denoted by $\otimes$)

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

bitwise operation between the unique identifier of the sender thread and a constant related to the group of the receiver thread (line 9). Finally, each thread combines the result from the accumulated prefix sums with each local prefix sum initially computed (lines 20-21).

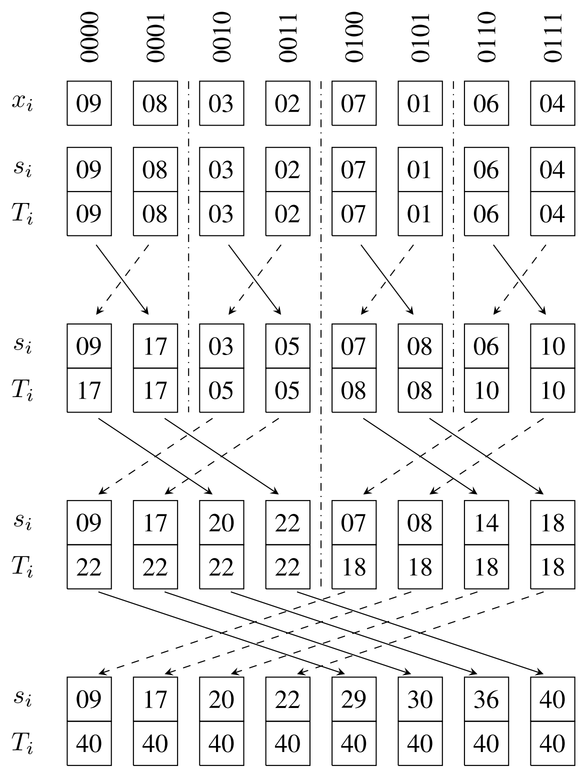


Fig. 5: Parallel prefix sum example.

Figure 5 presents an example of the implemented parallel prefix sum algorithm. The prefix sum is carried out by a set of eight threads (the unique identifier of each thread is shown in binary notation). The values indicated by x_i line correspond to the values initially assigned to each thread $i = 1, \ldots, 8$. In the other words, the prefix sum is going to be executed on the ordered x_i set. Each line labeled with s_i is related to the prefix sum value stored by the thread i. The label T_i indicates the total sum value calculated by a thread i. The parallel prefix sum is executed in three phases of data exchanging. In the first one, threads are divided among groups of size two, and the data exchanging carries out inside each group. In the second and third phases, the group size is increased to four and eight respectively. When the unique identifier of a sender thread is lower than a receiver thread, both the values of local prefix sum and local total sum are updated. Otherwise, only the local prefix sum of the receiver thread is incremented.

D. Nodes Placement (Step 4)

The fourth step is described by the Algorithm 5. It was originally proposed by [7]. 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. Two copies ($read\_offset$ and $write\_offset$) of this array are created (line 1) in order to control the number of nodes to read from a level, and the number of nodes to write from the next level. The original $sums$ array is never changed once its values are used as bounds for threads operations.

The process starts assigning the source node to the first position of the permutation array. As there is a write operation related to the level 0, the corresponding position in $write\_offset$ array is incremented (line 3). Thereafter, every time the $read\_offset[l]$ is different of $sums[l+1]$ (line 6), the thread assigned to the level l becomes able to read the node from the permutation array at position $read\_offset[l]$ (line 8). Actually, this condition indicates that there are $sums[l+1] - read\_offset[l]$ nodes whose children must be placed in the permutation array. In this way, the reading of each node at level l generates an increment of the $read\_offset$ array at position l (line 9). Next, the respective thread gets the neighbors of the read node (line 10), sort them by degree (line 11), and place them in the permutation array (line 13). Each write operation produces an increment of the $write\_offset$ array at position $l+1$ (line 14). Therefore, 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.

Algorithm 5 Parallel Nodes Placement Algorithm

Input: Graph G, Node source, int dist, Array sums
Output: Array perm

```
1:  int read_offset = write_offset = sums;
2:  int perm[0] = source;
3:  write_offset[0] = 1;
4:  foreach (int thread : threads) {
5:      for (int l = thread; l < dist; l += threads.size()) {
6:          while (read_offset[l] ≠ sums[l + 1]) {
7:              while (read_offset[l] == write_offset[l]) { }
8:              Node n = perm[read_offset[l]];
9:              ++read_offset[l];
10:             children = G.neighborsAtLevel(n, level+1);
11:             sort(children); // Sort children by degree
12:             foreach (Node c : children) {
13:                 perm[write_offset[l+1]] = c;
14:                 ++write_offset[l+1];
15: } } }
```

V. Experimental Results

The program was coded in the C language and the parallelism was supported by OpenMP framework - version 4.0. The experiments were performed on a PC running Ubuntu Linux, version 14.04.5 LTS, with Kernel version 3.19.0-31. It consists of one Intel i7-3610QM processor of 4 cores (two threads per core), operating at 2.3 GHz. Each core has a unified 256KB L2 cache and each processor has a shared 6MB L3 cache. The PC contains 8GB of main memory and the code was compiled with GCC version 5.4.0, and with the $-O3$ optimization flag turned on. The complete source code is available on GitHub repository [22].

A. Methodology

A set of twenty structural symmetric and square matrices was selected from the University of Florida Sparse Matrix Collection [23]. These matrices cover multiple types of problems in order to increase the dataset variety and the percentage of sparsity of each one is higher than 99.95%. The set of tested matrices is shown in Table I. The columns tabulate the matrice's name, as well as the dimension, the number of non-zeros, and the average of non-zeros per row (NNZ/row) of them.

TABLE I: Tested sparse matrices.

#	Matrix	Dimension	Non-zeros	NNZ/row
01	m_t1	97,578	9,753,570	100
02	filter3D	106,437	2,707,179	25
03	SiO2	155,331	11,283,503	73
04	d_pretok	182,730	1,641,672	9
05	CO	221,119	7,666,057	35
06	offshore	259,789	4,242,673	16
07	Ga41As41H72	268,096	18,488,476	69
08	F1	343,791	26,837,113	78
09	mario002	389,874	2,097,566	5
10	msdoor	415,863	19,173,163	46
11	inline_1	503,712	36,816,170	73
12	gsm_106857	589,446	21,758,924	37
13	Fault_639	638,802	27,245,944	43
14	tmt_sym	726,713	5,080,961	7
15	boneS10	914,898	40,878,708	45
16	audikw_1	943,695	77,651,847	82
17	nlpkkt80	1,062,400	28,192,672	27
18	dielFilterV2real	1,157,456	48,538,952	42
19	Serena	1,391,349	64,131,971	46
20	G3_circuit	1,585,478	7,660,826	5

The algorithms were performed five times for each pair (m_i, t_j), where m_i is a sparse matrix, and t_j is the number of threads between 1 and 12 (in steps of 2). For each (m_i, t_j) tested pair, the average was calculated from the reported values. In order to confront the algorithms, for each matrix m_i, it was selected the number of threads t_j that reached the best value considering the CPU time. The Compressed Sparse Row format (Section II) was the mechanism used to store each tested matrix. For the starting point of the algorithms (source node), it was used a pseudo-peripheral node obtained by the heuristic described in Section III. Moreover, the speedup S computed for the parallel RCM algorithm was calculated according to expression $S(n) = \frac{T_1}{T_n}$, where T_1 is the run-time of the parallel RCM executed with one thread, and T_n is the run-time of the same algorithm executed with n threads.

B. Environment Variables Setup

Some OpenMP variables that affect the execution of OpenMP programs were configured to guide the threads behavior. According to OpenMP Language Working Group [24], all settings must be done before the program has started. Otherwise, modifications to the environment variables are ignored. In this work, the OpenMP configured variables are described below.

- **OMP_DYNAMIC**: This environment variable controls dynamic adjustment of the number of threads inside parallel regions. As the executed experiments involve a specific number of threads, this variable was set to **FALSE**.
- **OMP_WAIT_POLICY**: It provides a hint to the OpenMP implementation about the desired behavior of waiting threads. For all experiments of this work, the behavior of waiting threads was set to **PASSIVE**. This value specifies that waiting threads should mostly be passive, not consuming cycles, while waiting.
- **OMP_PROC_BIND**: It enables or disables threads binding to processors. In this work, the value **TRUE** was defined for this variable. With this configuration, the execution environment does not move OpenMP threads between OpenMP places.

C. Reordering Quality

Table II shows reordering quality (final bandwidth columns) comparison between the serial HSL library and the implemented Unordered Parallel RCM algorithm (URCM). Columns reduction display the bandwidth percentage reduction attained by each algorithm in relation to the original bandwidth value.

TABLE II: Bandwidth Comparison after Reordering

Matrix		Final Bandwidth		Reduction (%)	
Name	Bandwidth	HSL	URCM	HSL	URCM
m_t1	6,482	6,807	**6,482**	-5.01	0.00
filter3D	8,276	**3,492**	3,613	57.81	56.34
SiO2	55,068	21,647	**19,572**	60.69	64.46
d_pretok	129,917	**2,564**	2,577	98.03	98.02
CO	26,470	20,734	**19,116**	21.67	27.78
offshore	237,738	23,923	**21,617**	89.94	90.91
Ga41As41H72	40,195	35,164	**34,139**	12.52	15.07
F1	343,754	14,970	**10,052**	95.65	97.08
mario002	387,647	1,191	**1,178**	99.69	99.70
msdoor	291,114	6,088	**5,823**	97.91	98.00
inline_1	502,403	6,468	**6,002**	98.71	98.81
gsm_106857	588,744	18,132	**17,742**	96.92	96.99
Fault_639	19,988	17,016	**15,749**	14.87	21.21
tmt_sym	1,921	1,141	**1,139**	40.60	40.71
boneS10	8,969	15,789	**13,751**	-76.04	-53.32
audikw_1	925,946	39,441	**35,102**	95.74	96.21
nlpkkt80	550,481	37,522	**37,445**	93.18	93.20
dielFilterV2real	948,032	**18,014**	18,045	98.10	98.10
Serena	81,578	**81,360**	81,647	0.27	-0.08
G3_circuit	947,128	**5,069**	**5,069**	99.46	99.46

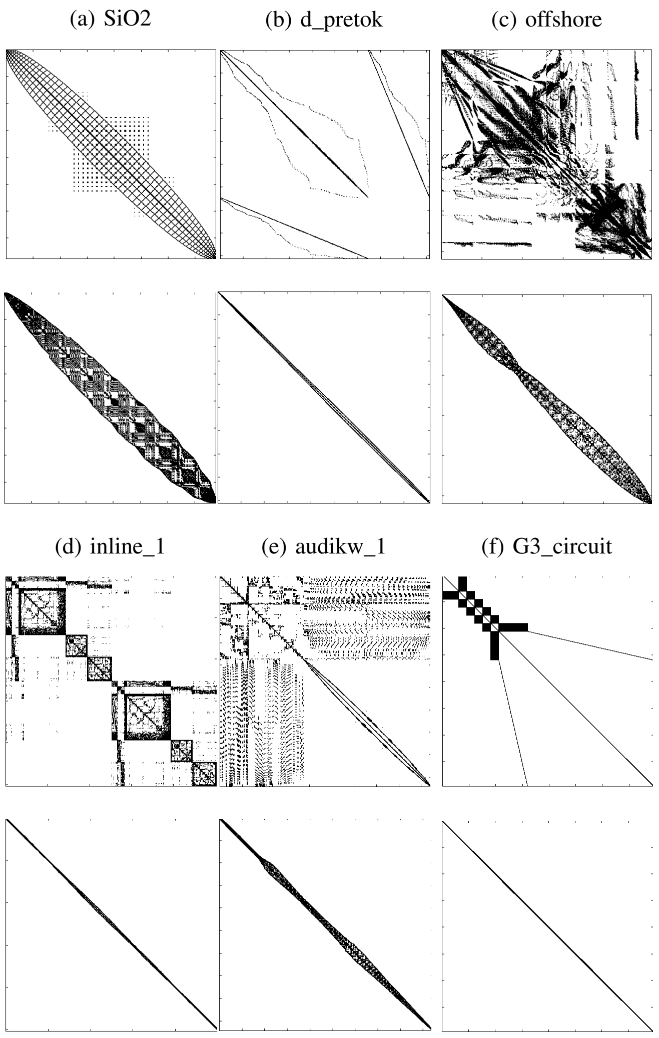


Fig. 6: Sparse matrix pattern yielded by the Unordered RCM.

The results displayed in Table II highlight the efficiency of the implemented algorithms for solving the bandwidth minimization problem. The HSL's sequential RCM and the Unordered RCM produce very similar bandwidth numbers. The HSL library reached a better solution only for four matrices. For eleven matrices, the percentage of bandwidth reduction attained by the URCM algorithm was higher than 90%. For the other matrices, there was a lower bound of 15.07% for the bandwidth reduction. Just three exceptions were observed: (i) Despite the URCM has not reached any bandwidth reduction with the m_t1 matrix, the result achieved by HSL was worse. The library increased the matrix bandwidth; (ii) For the $boneS10$ matrix, both algorithms produced a bandwidth higher than the original; and (iii) For the $Serena$ matrix, the URCM algorithm also generated a final bandwidth worse than the original.

The reordering quality produced by the implemented algorithm may also be graphically attested through Figure 6. It presents some examples of sparse matrix pattern yielded by the URCM algorithm. The first row of each subfigure presents the matrix sparsity before reordering. In the below rows, each respective matrix is exhibited as result of a permutation of rows and columns. The first set of matrices (Figures 6(a), 6(b), and 6(c)) are samples out of smallest matrices (order up to around 500.000). The second group of matrices (Figures 6(d), 6(e), and 6(f)) corresponds to some of the highest ones (with order of 1.500.000 approximately). The bandwidth reduction reached with these six matrices varied from 64.46% ($SiO2$) to 99.46% ($G3\_circuit$).

D. Reordering Performance

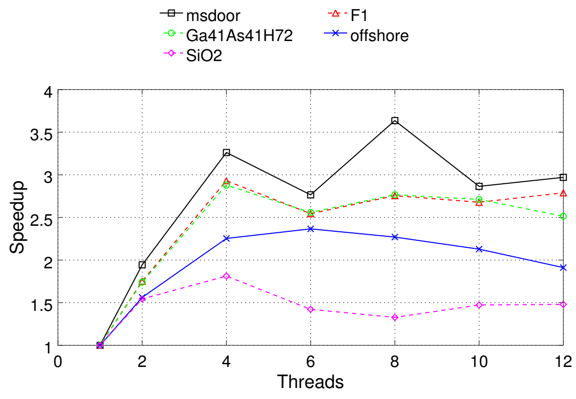
Table III shows a performance comparison of the two algorithms. The reordering times are presented in scale of 10^{-3} seconds, and the best values in terms of CPU time are highlighted in bold. The numbers in parentheses indicate the number of threads used to reach the respective value. The column Reduction presents the time reduction percentage achieved by the Unordered RCM in comparison with HSL.

TABLE III: CPU time comparison (x10^{-3} sec.)

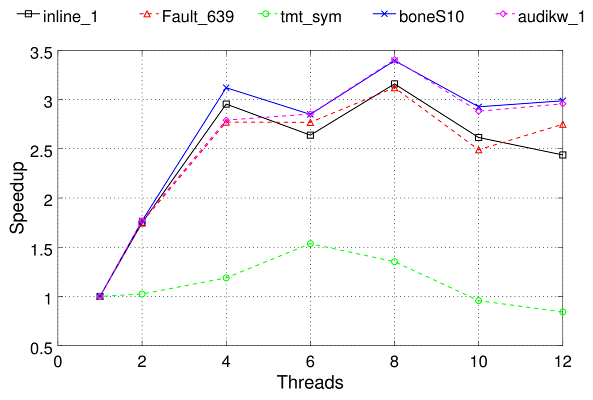
Matrix	Reordering Time		
Name	HSL	URCM	Reduction (%)
m_t1	0.871	**0.628 (04)**	28.64
filter3D	0.880	**0.394 (04)**	54.76
SiO2	2.339	**1.668 (04)**	28.69
d_pretok	0.746	**0.585 (04)**	21.58
CO	2.095	**1.020 (08)**	51.31
offshore	2.432	**1.082 (06)**	55.51
Ga41As41H72	3.988	**1.794 (04)**	55.02
F1	3.394	**2.414 (04)**	28.87
mario002	1.507	**1.349 (06)**	10.48
msdoor	2.381	**1.838 (08)**	22.81
inline_1	4.140	**3.100 (08)**	25.12
gsm_106857	5.780	**2.840 (08)**	50.87
Fault_639	3.250	**2.900 (08)**	10.77
tmt_sym	**2.040**	2.290 (06)	-12.25
boneS10	10.480	**4.420 (08)**	57.82
audikw_1	12.590	**5.910 (08)**	53.06
nlpkkt80	8.320	**3.670 (08)**	55.89
dielFilterV2real	10.390	**5.170 (08)**	50.24
Serena	11.170	**5.920 (08)**	47.00
G3_circuit	5.120	**4.750 (06)**	07.22

As displayed in Table III, the Unordered RCM achieved outstanding performance results. In fact, the rate of time reordering reduction of the algorithm varies from 10.48% ($mario002$) to 57.82% ($boneS10$). The time reordering improvement presented by five matrices was very significant. With these matrices, the algorithm reached speedups superior to 3.0X, i.e., 3.84X ($boneS10$), 3.64X ($msdoor$), 3.40X ($audikw\_1$), 3.15X ($inline\_1$), and 3.12X ($Fault\_639$).

Figure 7 shows two sets of speedup curves generated by experiments with the Unordered RCM processing ten matrices. Figure 7(a) presents the five matrices that have shown the best speedup ratio out of the smallest ones tested (matrix order up to five hundred thousand). In this set of matrices, the performance improvement was more impacted by the matrices order than by the number of non-zeros per row. Actually, the best speedup ratios observed in this set of matrices were 3.64X ($msdoor$), 2.93X ($F1$), and 2.88X ($Ga41As41H72$). Although the matrix $SiO2$ has the most significant average of NNZ/row among these five ones (about 76 - see Table I), the Unordered RCM achieved the lowest speedup ratios with this matrix (1.81X running with just 4 threads).



(a) Smallest matrices



(b) Largest matrices

Fig. 7: Speedup of Unordered RCM.

A different behavior was observed with largest matrices. Figure 7(b) shows the speedup of a second set of five matrices whose order varies from 500,000 to 1,500,000 approximately. The best performance improvement was reached with the matrices with a high average of non-zeros per row. It was the case of $inline\_1$, $Fault\_639$, $boneS10$, and $audikw\_1$. The same was not observed with tmt_sym matrix - it has an average of just 7 nonzeros per row. These different ratios of performance observed with matrices of distinct orders and distinct average of non-zeros per row suggest that 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. Nevertheless, for lower order graphs, the parallelism overhead impacts heavily on the CPU time improvement.

VI. Conclusion

This paper analyzed a parallel strategy for a traditional reordering algorithm. The obtained results show the benefits related to improving reordering time. In fact, for the set of tested matrices, the attained time reduction varies between 10.48% and 57.82%. Other significant results show the unordered RCM algorithm achieving speedups up to 3.84X with 6 threads. About the quality of solutions, the bandwidth reduction reached by the implemented algorithm was not superior to HSL just for one tested matrix. Therefore, the new parallel implementation proposed by the RCM algorithm may be considered as an efficient approach for the bandwidth minimization problem applied on large sparse matrices.

Some works in the literature have addressed the reordering problem through the use of other data structures and alternative breadth-first search (BFS) strategies have been proposed for the parallelism of RCM. As example, relevant results have been reached with a wavefront BFS implementation [18], and a novel implementation of a worklist data structure, called bag, has been used in place of FIFO queue usually employed in BFS algorithms [25]. The use of these new structures and strategies may promote more improvements to the algorithm studied in this work.

References

[1] Y. Saad, *Iterative methods for sparse linear systems*, 2nd ed. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 2003. ISBN 0898715342

[2] C. H. Papadimitriou, “The np-completeness of the bandwidth minimization problem,” *Computing*, vol. 16, no. 3, pp. 263–270, 1976. doi: 10.1007/BF02280884. [Online]. Available: http://dx.doi.org/10.1007/BF02280884

[3] E. Cuthill and J. McKee, “Reducing the bandwidth of sparse symmetric matrices,” in *Proceedings of the 1969 24th National Conference*, ser. ACM ’69. New York, NY, USA: ACM, 1969. doi: 10.1145/800195.805928 pp. 157–172. [Online]. Available: http://doi.acm.org/10.1145/800195.805928

[4] W. Liu and A. H. Sherman, “Comparative analysis of the Cuthill-McKee and the Reverse Cuthill-McKee ordering algorithms for sparse matrices,” *SIAM Journal on Numerical Analysis*, vol. 13, no. 2, pp. 198–213, May 1974. doi: 10.1137/0713020. [Online]. Available: http://dx.doi.org/10.1137/0713020

[5] S. W. Sloan, “An algorithm for profile and wavefront reduction of sparse matrices,” *International Journal for Numerical Methods in Engineering*, vol. 23, no. 2, pp. 239–251, 1986. doi: 10.1002/nme.1620230208

[6] N. E. Gibbs, W. G. Poole, and P. K. Stockmeyer, “An algorithm for reducing the bandwidth and profile of a sparse matrix,” *SIAM Journal on Numerical Analysis*, vol. 13, no. 2, pp. 236–250, 1976. [Online]. Available: http://www.jstor.org/stable/2156090

[7] K. I. Karantasis, A. Lenharth, D. Nguyen, M. Garzarán, and K. Pingali, “Parallelization of reordering algorithms for bandwidth and wavefront reduction,” in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC ’14. Piscataway, NJ, USA: IEEE Press, 2014. doi: 10.1109/SC.2014.80. ISBN 978-1-4799-5500-8 pp. 921–932. [Online]. Available: http://dx.doi.org/10.1109/SC.2014.80

[8] D. Padua, *Encyclopedia of parallel computing*. Springer Publishing Company, Incorporated, 2011. ISBN 0387097651

[9] K. Pingali et al., “The tao of parallelism in algorithms,” in *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI ’11. New York, NY, USA: ACM, 2011. doi: 10.1145/1993498.1993501. ISBN 978-1-4503-0663-8 pp. 12–25. [Online]. Available: http://doi.acm.org/10.1145/1993498.1993501

[10] M. Kulkarni, M. Burtscher, R. Inkulu, K. Pingali, and C. Casçaval, “How much parallelism is there in irregular applications?” *SIGPLAN Not.*, vol. 44, no. 4, pp. 3–14, Feb. 2009. doi: 10.1145/1594835.1504181. [Online]. Available: http://doi.acm.org/10.1145/1594835.1504181

[11] M. Kulkarni et al., "Optimistic parallelism requires abstractions," in *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '07. New York, NY, USA: ACM, 2007. doi: 10.1145/1250734.1250759. ISBN 978-1-59593-633-2 pp. 211–222. [Online]. Available: http://doi.acm.org/10.1145/1250734.1250759

[12] L. Dagum and R. Menon, "Openmp: An industry-standard api for shared-memory programming," *IEEE Comput. Sci. Eng.*, vol. 5, no. 1, pp. 46–55, Jan. 1998. doi: 10.1109/99.660313. [Online]. Available: http://dx.doi.org/10.1109/99.660313

[13] HSL, "A collection of fortran codes for large scale scientific computation," 2011. [Online]. Available: http://www.hsl.rl.ac.uk/

[14] A. Farzaneh, H. Kheiri, and M. A. Shahmersi, "An efficient storage format for large sparse matrices," *Communications Series A1 Mathematics & Statistics*, vol. 58, no. 2, pp. 1–10, Jul. 2009.

[15] J. K. Reid and J. A. Scott, "Ordering symmetric sparse matrices for small profile and wavefront," *International Journal for Numerical Methods in Engineering*, vol. 45, pp. 1737–1755, Feb. 1999.

[16] G. K. Kumfert, "An object-oriented algorithmic laboratory for ordering sparse matrices," Ph.D. dissertation, Lawrence Livermore National Laboratory and United States. Department of Energy and United States. Department of Energy. Office of Scientific and Technical Information, 2000.

[17] I. S. Duff, J. K. Reid, and J. A. Scott, "The use of profile reduction algorithms with a frontal code," *International Journal for Numerical Methods in Engineering*, vol. 28, no. 11, pp. 2555–2568, 1989. doi: 10.1002/nme.1620281106. [Online]. Available: http://dx.doi.org/10.1002/nme.1620281106

[18] M. A. Hassaan, M. Burtscher and K. Pingali, "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*, ser. PPoPP '11. New York, NY, USA: ACM, 2011. doi: 10.1145/1941553.1941557. ISBN 978-1-4503-0119-0 pp. 3–12. [Online]. Available: http://doi.acm.org/10.1145/1941553.1941557

[19] B. S. W. Schröder, "Algorithms for the fixed point property," *Theoretical Computer Science*, vol. 217, no. 2, pp. 301 – 358, 1999. doi: http://dx.doi.org/10.1016/S0304-3975(98)00273-4. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0304397598002734

[20] D. Chazan and W. Miranker, "Chaotic relaxation," *Linear Algebra and its Applications*, vol. 2, no. 2, pp. 199 – 222, 1969. doi: http://dx.doi.org/10.1016/0024-3795(69)90028-7. [Online]. Available: http://www.sciencedirect.com/science/article/pii/0024379569900287

[21] S. Aluru, "Teaching parallel computing through parallel prefix," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, ser. SC12, 2012. [Online]. Available: http://sc12.supercomputing.org/hpceducator/ParallelPrefix/ParallelPrefix.pdf

[22] T. N. Rodrigues, "tnas/reordering-library: Federated Conference on Computer Science and Information Systems 2017," May 2017. [Online]. Available: https://doi.org/10.5281/zenodo.570225

[23] T. A. Davis and Y. Hu, "The university of florida sparse matrix collection," *ACM Trans. Math. Softw.*, vol. 38, no. 1, pp. 1:1–1:25, Dec. 2011. doi: 10.1145/2049662.2049663. [Online]. Available: http://doi.acm.org/10.1145/2049662.2049663

[24] OpenMP Language Working Group, "Openmp technical report 4," OpenMP Architecture Review Board, Tech. Rep. TR-4 Version 5 Preview 1, 2016.

[25] C. E. Leiserson and T. B. Schardl, "A work-efficient parallel breadth-first search algorithm (or how to cope with the nondeterminism of reducers)," in *Proceedings of the Twenty-second Annual ACM Symposium on Parallelism in Algorithms and Architectures*, ser. SPAA '10. New York, NY, USA: ACM, 2010. doi: 10.1145/1810479.1810534. ISBN 978-1-4503-0079-7 pp. 303–314. [Online]. Available: http://doi.acm.org/10.1145/1810479.1810534