Proceeding Series of the Brazilian Society of Computational and Applied Mathematics

An Implementation of the Unordered Parallel RCM for Bandwidth Reduction of Large Sparse Matrices

Thiago Nascimento Rodrigues[1]
Maria Claudia Silva Boeres[2]
Lucia Catabriga[2]
Department of Informatics, Federal University of Espírito Santo - UFES, Vitória, ES

Abstract. This paper describes an implementation of the Unordered Parallel Reverse Cuthill-McKee algorithm which is compared with its well-known serial version. The OpenMP framework is used for supporting the parallelism and a strategy for reducing lazy threads is evaluated. Large sparse matrices are used to test sequential and parallel approaches. The computational cost reduction and the quality of matrices bandwidth minimization are validated by CPU time and speedup.

Keywords. Parallel RCM, Sparse Matrix, Unordered Breadth-first Search, OpenMP

1 Introduction

Heuristics for bandwidth reduction of matrices are used to reduce computational and storage costs of the large sparse linear systems resolution. The bandwidth of a matrix is related to the concentration of nonzero elements around the main diagonal. For a symmetric matrix $A = \{a_{ij}\}$ of size n, the bandwidth reduction problem consists of finding a permutation of rows and columns of A so as to bring all the non-zero elements of A to reside in a band that is as close as possible to the main diagonal, that is to $b = \min\{\max\{|i-j| : a_{ij} \neq 0, i = [1..n], j = [1..n]\}\}$. Unfortunately, determining the optimal ordering is a NP-complete problem [12], so heuristic algorithms are typically used instead.

Most methods designed to reduce the matrix bandwidth are based on the corresponding graph formulation: find a labeling of the vertices of a graph such that most connections are between vertices having close labels. An important and well-known example of a bandwidth reduction algorithm is that proposed by Cuthill and McKee [2] and its many variants. The Cuthill-McKee algorithm uses Breadth-First-Search to constructs a level set structure of the graph and labels the nodes according to these levels. Other algorithms very frequently referenced in the literature for bandwidth minimization are Sloan [13] and Nested Dissection [4]. All of them also use graph search strategies and provide high quality solutions. Another main class of reordering algorithms is based on the minimum degree

[1]nascimenthiago@gmail.com
[2]{boeres, luciac}@inf.ufes.br

algorithm, which uses computationally cheap bounds on the minimum degree in place of the exact minimum degree.

In classical implementations of these algorithms, matrix reordering is performed sequentially. However, the widespread use of multi-core processors has conducted great performance improvements related to matrix computation. In this context, several parallel strategies has been proposed for solving the bandwidth reduction problem. The use of Genetic Algorithms [10] and multilevel graph partitioning methods [8] are some examples.

In this paper, an implementation of the parallel RCM algorithm proposed by [7] is presented. The OpenMP framework is used for support the parallelism and a strategy to reducing lazy threads was included in the implemented algorithm. So, this paper is organized as follow. In the next section, the unordered RCM algorithm is described in details. The Section 3 is dedicated to present experimental results on a set of test matrices. Conclusions and future works are presented in the last section.

2 Unordered Parallel Reverse Cuthill-McKee

The Cuthill-McKee algorithm has as input a matrix A and uses a refinement of Breadth-First-Search (BFS) to produce a permutation of the rows and columns of A. It divides the matrix associated graph into level sets and a level structure rooted at a node r is defined as the partitioning of V into levels $l_1(r), l_2(r), \dots, l_h(r)$ such that $l_1(r) = \{r\}$ and for $i > 1, l_i(r)$ is the set of all nodes that are adjacent to nodes in $l_{i-1}(r)$ but are not in $l_1(r), l_2(r), \dots, l_{i-1}(r)$. Cuthill-McKee orders within each level set $l_i(r)$ by sorting first neighbors of the first node in $l_{i-1}(r)$, then those that are neighbors of the second node in $l_{i-1}(r)$, and so on. The Reverse Cuthill-McKee (RCM) algorithm uses as heuristic the opposite ordering relation so nodes are visited in descending degree order. Such reversion has been proved with experimental results that RCM produces permutations of better or equal quality to the original Cuthill-McKee. The root r of the level structure is usually chosen from among the pseudo-peripheral [3] nodes of the associated graph.

The Unordered Parallel RCM proposed by [7] uses a different BFS algorithm as starting point. Other three general steps complete the algorithm and they are as follow.

Step 1: Compute the levels for each node using the unordered BFS algorithm. The BFS modified algorithm is obtained by noticing that the level of a node is a local minimum in the graph, i.e., the level of a node (except the root) is one higher than the minimum level of this neighbors [5]. In the other words, the computation of level for node n can be described as: $level(root) = 0; level(k) = \infty, \forall k$ other than root; and $level(n) = \min(level(m) + 1), \forall m \in$ neighbors of n. Because of this property, the algorithm maintains a workset (ws) of nodes from which any arbitrary element can be selected. In this case, multiple nodes can be processed in parallel and data structure used as workset must be an unordered set. Initially, the level of the *root* is set to zero, the level of all other nodes is set to ∞, and only the *root* node is in the workset. For each

[3] 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).

iteration of the algorithm over the workset, a thread removes an arbitrary node n from ws and calculates the appropriated level to neighbors of n. Thus the calculated level is set for each neighbor with higher level. The updated neighbors are added in ws and the algorithm iterates until ws is empty.

Step 2: Count the number of nodes at each level. Each position of the generated array is related to a level. Hence, each value stored in the respective position of the array corresponds to the number of nodes at a specific level. In the implemented algorithm, the array obtained from Step 1 is divided among threads. Each one counts how many nodes there are at each level locally. Thus, the local array from each thread is aggregated in a global count array.

Step 3: Compute the prefix sum[4] of levels. The prefix sum gives the initial offset in the final permutation array for nodes in any given level. The algorithm used for calculating prefix sum is based on the algorithm proposed by [1]. Initially, each processor computes the prefix sums of the $\frac{n}{p}$ elements it has locally (n is the number of elements, and p is the number of processors). Next the local prefix sum values are exchanged among the processors and each one accumulates the respective received value. For this exchange the processors are divided in dinamically sized groups. Finally, each processor combines the result from the accumulated prefix sums with each local prefix sum initially computed.

Step 4: Place nodes in the permutation array. This placement phase is parallelized by pipelining. A single thread is assigned to each level, and threads communicate to the next level through a single-producer, single-consumer queue. In this way, a thread at level l writes out children in a RCM order for a thread at level $l + 1$ to consume. A thread receives nodes in the correct order for its level and produces nodes in the correct order for the next level. Parallelization is achieved because each level can be populated as soon as nodes are produced by the previous level.

In the related work [5], some optimizations were proposed for other unordered algorithms. These same optimizations were evaluated and applied in the implemented parallel RCM. Firstly, to reduce the overhead of accessing the workset, each thread removes a chunk of active elements from workset instead of just one element. Newly created work is cached locally, and after the entire chunk is processed, this work is added to the global workset. The chunk size is determined dynamically by the program. Each thread gets a number of elements corresponding to fifty percent of the current global workset size. Although other percentages were tested, the best performance was reached using this empirically defined value. Another implemented optimization is the reduction of amount of wasted work by each thread. As the workset is accessed by multiple threads without a strict order, it can be eventually empty. In this case, can there be lazy threads. Thus, the used strategy was to maintain an order in which all threads remove active elements from one end and add to the other.

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

3 Experimental Results

The evaluation of the Unordered RCM performance was against a traditional serial implementation of RCM [11]. A set of six symmetric matrices was selected from the University of Florida Sparse Matrix Collection [3]. The program was coded in the *C* language and the parallelism was supported by OpenMP framework. The experiments were performed on a PC with Intel 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: Results Comparison

Input			Band Reduction	Time (sec./#threads)	
Matrix	Dimension	Sparsity (%)	(%)	Serial	Parallel
Dubcova3	146,689	99.983	98.442	1.199	0.230 (32)
inline_1	503,712	99.985	98.807	9.040	1.965 (64)
audikw_1	943,695	99.991	96.283	84.833	8.244 (128)
dielFilterV3real	1,102,824	99.993	97.548	88.298	13.308 (128)
atmosmodj	1,270,432	99.999	64.513	32.732	1.895 (128)
G3_circuit	1,585,478	99.999	99.464	84.832	18.007 (128)

Table 1 shows a performance comparison between serial and parallel RCM. The three first columns present selected matrices and some characteristics of them (dimension and percentage of sparsity). The Band Reduction column shows the percentage of the bandwidth reduction for each matrix for both algorithms, once serial and parallel implementations got the same reordering for all tested matrices. Naturally, each achieved best execution time is related to appropriated choice of the number of threads (number into parentheses at Parallel Time column). Moreover, two features are relevant to point out: (1) The time spent in each pseudo-peripheral computation was excluded from the parallel as well as the sequential program; (2) The BFS (Step 1) and the Placement (Step 4) are the most expensive steps of the algorithm, corresponding in general to around 50% and 40% of total algorithm cost, respectively.

As aforementioned and depicted by Table 1, the reordering results obtained by running the two algorithms for each matrix were the same. This quality might be graphically attested through Figs. 1 and 2. Matrices were divided in groups according to the size of them (smaller and larger than one million). 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 Unordered RCM algorithm.

For each matrix running with a number of threads between 4 and 128 (in steps of 2), the program was ran five times. The minimum and maximum reported values was excluded and the average time was calculated from remaining values. Based on this methodology, the Figure 3 presents how speedups scale as the number of processors increases. Using the size of 1 million as baseline, the matrices were grouped according to the dimension one

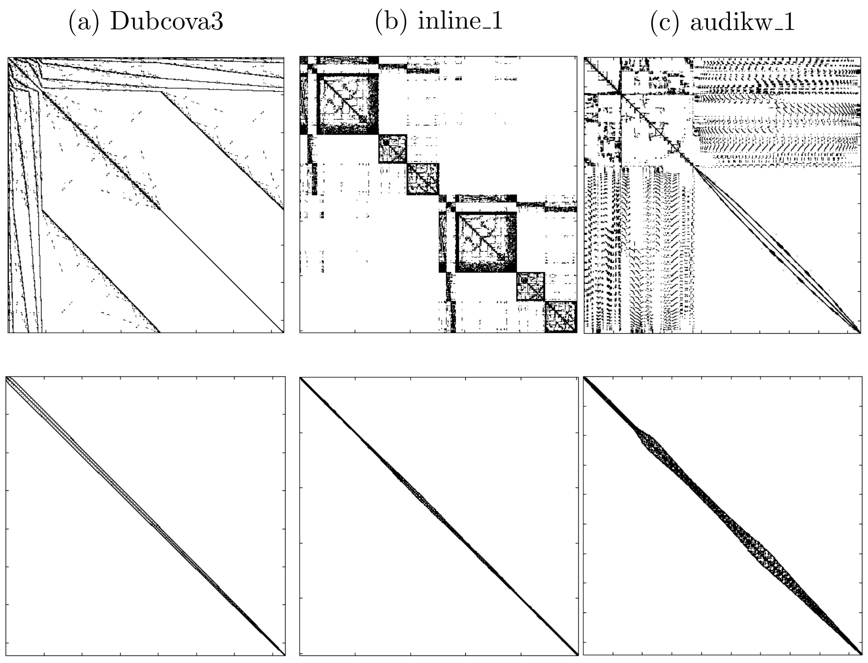

Figure 1: Matrices of size smaller than one million

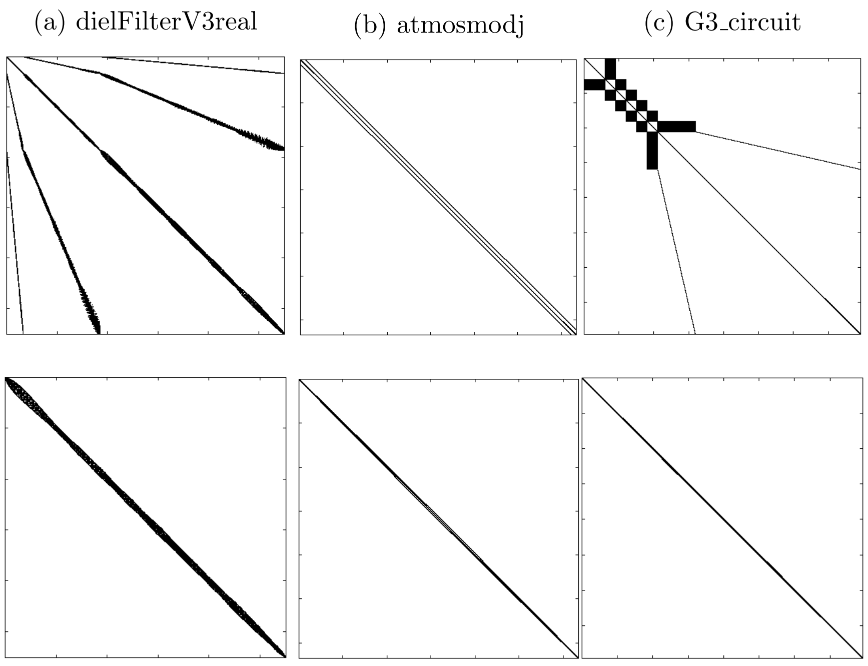

Figure 2: Matrices of size larger than one million

more time: three smaller than 1 million and three larger than 1 million. As shown by the Figure 3, *audikw_1* and *atmosmodj* are matrices with best speedups for Unordered RCM. Another feature is related to speedup upper bound reached by the matrices of dimension around five hundred thousand. This is the case of *Dubcova3* and *inline_1* matrices, which speedup increases until 32 and 64 threads respectively. For the remaining largest matrices speedup scales until 128 threads. On the other hand, the algorithm did not get relevant performance improvements when testing small matrices. Actually, for matrices of size smaller than *Dubcova3* matrix dimension (around 146 thousand rows), there is a speedup reduction and an increasing of the reordering time.

(a) Smallest Matrices (b) Largest Matrices

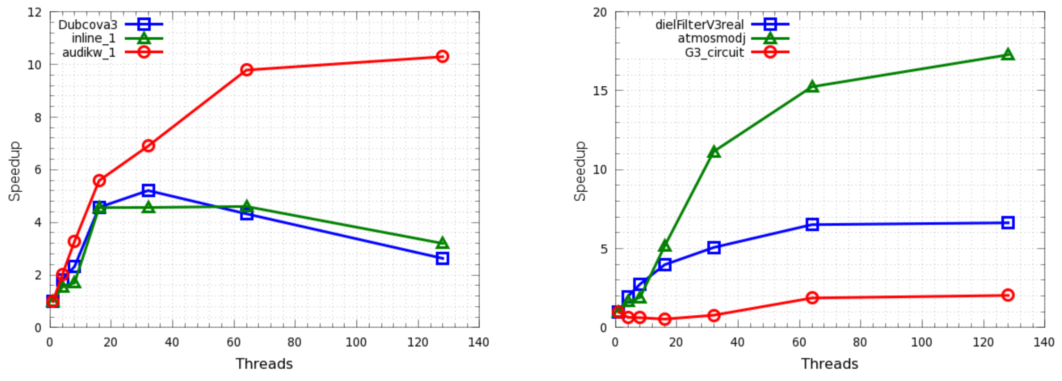

Figure 3: Matrices Speedup

4 Conclusions

This paper analysed 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 various between 78.26% and 94.21%. Other significant results show the unordered RCM algorithm achieving speedups between 2.04X to 17.21X on 8 cores and 128 threads respectively. About the quality of solutions, the reached bandwidth reduction was the same for sequential and parallel implementations. Therefore, the studied algorithm might be considered as a good option for computation of bandwidth reduction problem applied on large sparse matrices.

The presented algorithm was not compared against the state-of-the-art sequential implementation of RCM, i.e., the HSL software library [6]. Thus, a more accurate performance comparison might be achieved by confrontating both algorithms. Moreover, other data structures and BFS strategies have been proposed for the parallelism of RCM. In fact, Hassam et al [5] presents relevant results from a wavefront BFS implementation. And [9] proposes a novel implementation of a workset data structure, called a "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 studied algorithm.

References

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

[2] E. Cuthill and J. McKee. Reducing the bandwidth of sparse symmetric matrices. In *Proceedings of the 1969 24th National Conference*, ACM '69, pages 157–172, New York, NY, USA, 1969. ACM.

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

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

[5] 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*, PPoPP '11, pages 3–12, New York, NY, USA, 2011. ACM.

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

[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*, SC '14, pages 921–932, Piscataway, NJ, USA, 2014. IEEE Press.

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

[9] 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*, SPAA '10, pages 303–314, New York, NY, USA, 2010. ACM.

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

[11] B. Lugon and L. Catabriga. Algoritmos de reordenamento de matrizes esparsas aplicados a precondicionadores ILU(p). In *XLV Simpósio Brasileiro de Pesquisa Operacional*, pages 2343–2354, 2013.

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

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