The Reverse Cuthill-McKee Algorithm in Distributed-Memory

Ariful Azad, Aydın Buluç, Mathias Jacquelin, Esmond G. Ng
E-mail: {azad, abuluc, mjacquelin, egng}@lbl.gov
Computational Research Division
Lawrence Berkeley National Laboratory

We design and develop a distributed-memory parallel implementation of the reverse Cuthill-McKee (RCM) algorithm [1]. The RCM algorithm reorders a symmetric sparse matrix so that the permuted matrix has a small profile. A matrix with a small profile is useful in sparse direct methods 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.

Algorithms. The RCM algorithm involves repeatedly labeling vertices adjacent to the current vertex v_i until all have been labeled and reversing the labeling to obtain the final ordering. The level-by-level exploration of vertices is performed by a variant of the standard breadth-first search (BFS). The original Cuthill-McKee algorithm labels vertices in each BFS level using the following two rules. Within a level, vertices explore their unvisited neighbors in increasing values of their labels. Vertices with the same parent are labeled in increasing degree (that is, a vertex is labeled before its siblings with higher degrees). The RCM ordering is obtained by reversing the order obtained by the Cuthill-McKee algorithm. The ordering of vertex exploration within the BFS frontier (the set of current active vertices) and prioritizing siblings based on their degrees make the RCM algorithm more challenging than the standard BFS. In practice, the RCM algorithm starts with a *pseudo-peripheral vertex* - a vertex displaying a high eccentricity, as close to the graph diameter as possible [2], [1]. A pseudo-peripheral vertex in a graph can be found by computing a rooted level structure using BFS [2].

Similar to many sparse matrix computations, RCM ordering has been shown to be a difficult problem to parallelize [3]. The computational load of BFS used in the RCM algorithm and the pseudo-peripheral vertex computation is highly dynamic, especially if the graph has high diameter. The problem exacerbates on higher concurrency where load imbalance and communication overhead degrade the performance of the parallel algorithm. Here, we aim to overcome these challenges by using the graph-matrix duality and replacing unstructured graph operations by structured matrix/vector operations. The most expensive matrix/vector operations used in our algorithm are (a) sparse matrix-sparse vector multiplication (SPMSPV) and (b) sorted permutation (SORTPERM). SPMSPV is used to traverse vertices from the BFS frontier and SORTPERM labels vertices within a level in lexicographically sorted order based on (parent's order, degree) pair. Other matrix and vector operations used in designing our RCM algorithm do not contribute significantly to the total runtime. We use the Combinatorial BLAS library [4] to implement the linear-algebraic kernels. Our implementation is based on hybrid OpenMP-MPI where in-node multithreading is used to take advance of the shared-memory parallelism available in a node of modern supercomputers.

Results. We evaluate the performance of RCM algorithms using a set of real applications such as Delaunay triangulation, nonlinear optimization, and finite element computations. They are chosen to represent a variety of different structures and nonzero densities. We evaluate the performance of parallel RCM algorithm on Edison, a Cray XC30 supercomputer at NERSC.

We have evaluated the quality and runtime of our algorithm with a shared-memory implementation in SpMP (Sparse Matrix Pre-processing) by Park et al. [5], which is based on the algorithm presented in [6]. For five out of seven matrices that we have tested, the RCM ordering from our distributed-memory algorithm yields smaller bandwidth than SpMP. SpMP is faster than our implementation for up to 12 threads due to our distributed-memory parallelization overheads. However, SpMP does not scale well across multiple NUMA domains (i.e., beyond 12 cores on Edison). For example, SpMP slows down by a factor of $20\times$ for the `delaunay_n24` matrix on 24 cores compared to 6 cores.

We ran the distributed-memory RCM algorithm on up to 4096 cores of Edison. Figure 1 shows the strong scaling of the distributed-memory RCM algorithm for six selected matrices. To better understand the performance, we break down the runtime into five parts at each concurrency where the height of the bars denote the total runtime of identifying a pseudo-peripheral vertex and computing the RCM ordering. Our distributed algorithms scale up to 1024 cores on four out of the six graphs in Figure 1.

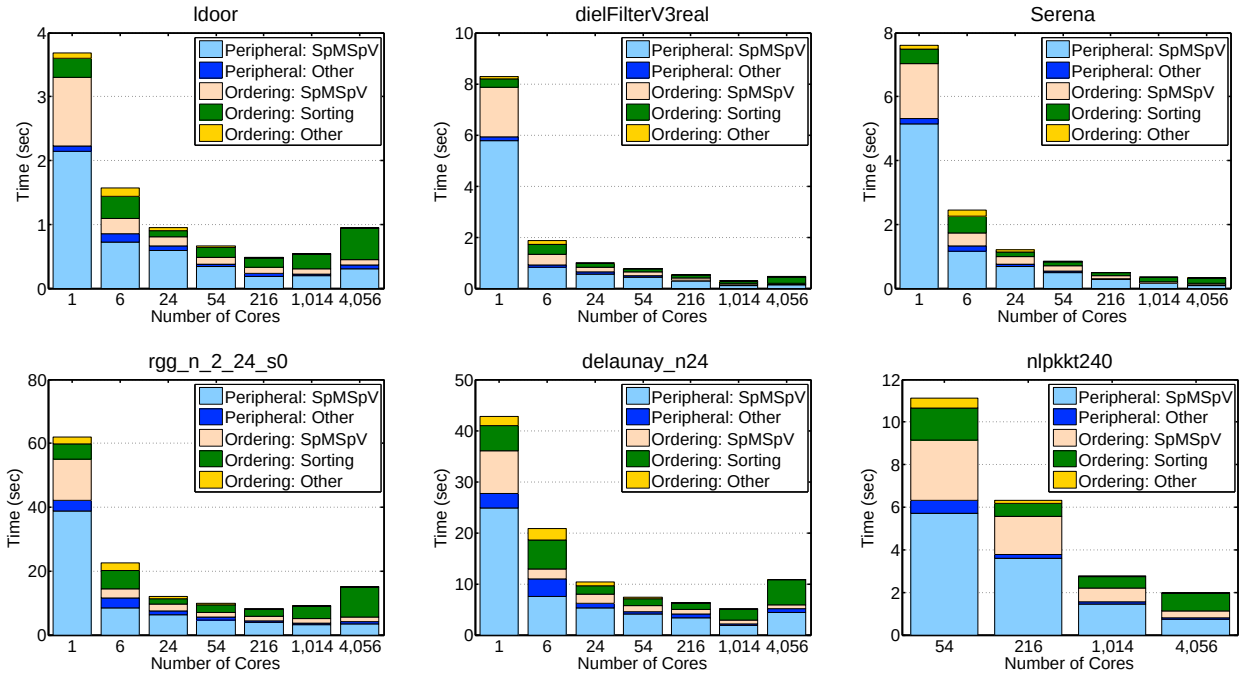

Fig. 1: Runtime breakdown of distributed-memory RCM on Edison. Six threads per MPI process are used when the number of cores is greater than or equal to six. `nlpkkt240` ran out of memory on a single node of Edison.

The RCM algorithm attains the best speedup of 30× for `dielFilterV3real` on 1024 cores. By contrast, it achieves 8× and 10× speedups on `rgg_n_2_24_s0` and `delaunay_24`, respectively. The sharp drop in parallel efficiency on these graphs are due to their high diameters. The level-synchronous nature of our BFS incurs high latency costs and decreases the amount of work per processor on high-diameter graphs. Figure 1 shows SPMSPV is usually the most expensive operation on lower concurrency. However, SORTPERM starts to dominate on high concurrency because it performs an AllToAll among all processes, which has higher latency.

Conclusions. We present a scalable distributed-memory algorithm for RCM ordering. Our algorithm relies on a small set of parallel primitives that are optimized for both shared-memory and massively parallel distributed memory systems. The quality (bandwidth and envelope) of ordering from our distributed-memory implementation is comparable to the state of the art and remains insensitive to the degree of concurrency. We provide a hybrid OpenMP-MPI implementation of the RCM ordering that attains up to 30× speedup on real matrices on 1024 cores of a Cray XC30 supercomputer. Our performance evaluation sheds light on the performance bottlenecks and opportunities for future research.

Acknowledgments

We thank Hari Sundar for sharing the source code of HykSort. This work is supported by the Applied Mathematics and the SciDAC Programs of the DOE Office of Advanced Scientific Computing Research under contract number DE-AC02-05CH11231. We used resources of the NERSC supported by the Office of Science of the DOE under Contract No. DE-AC02-05CH11231.

References

[1] A. George and J. W.-H. Liu, *Computer Solution of Large Sparse Positive Definite Systems*. Englewood Cliffs, New Jersey: Prentice-Hall Inc., 1981.

[2] A. George and J. W. H. Liu, "An implementation of a pseudoperipheral node finder," *TOMS*, vol. 5, no. 3, pp. 284–295, Sep. 1979.

[3] K. I. Karantasis, A. Lenharth, D. Nguyen, M. J. Garzaran, and K. Pingali, "Parallelization of reordering algorithms for bandwidth and wavefront reduction," in *SC14*. IEEE, 2014, pp. 921–932.

[4] A. Buluç and J. R. Gilbert, "The Combinatorial BLAS: Design, implementation, and applications," *IJHPCA*, vol. 25, no. 4, 2011.

[5] J. Park et al., "SpMP: Sparse Matrix Pre-processing." [Online]. Available: https://github.com/jspark1105/SpMP

[6] K. I. Karantasis, A. Lenharth, D. Nguyen, M. J. Garzaran, and K. Pingali, "Parallelization of reordering algorithms for bandwidth and wavefront reduction," in *SC'14*. IEEE, 2014, pp. 921–932.