HiLUK: Scalable Incomplete Factorization Utilizing Combinatorial Methods to Reduce Overheads

Joshua Dennis Booth* Sivasankaran Rajamanickam†

Incomplete factorizations are used to approximate the factorization of a sparse coefficient matrix A, such that $A = \bar{L}\bar{U} \approx LU$, and are commonly used as preconditioners for iterative methods, such as GMRES [7]. The approximation is normally achieved by some combination of dropping small value and/or by not allowing fill-in, i.e., zero elements becoming nonzero during factorization, based on levels (**k**) generated in the elimination tree (**ILU-K**). Incomplete factorizations are notorious for scaling poorly due to low computational intensity per communication/synchronization (sync). Due to this, very few implementations exist that scale beyond a handful of threads. However, increasing number of light-weight cores require that incomplete factorizations scale in order to not to be the bottleneck in key operations such as preconditioned GMRES. In this work, we present a new incomplete factorization package `HiLUK` that uses a variety of combinatorial techniques to achieve near linear speedups on x86 and Intel Phi. We only report **ILU-0** here for brevity.

Sparse factorizations have always required the use of advance combinatorial methods such as graph partitioning and ordering. In order to scale on current systems, a combination of these techniques need to be used to exploit both the matrix sparsity pattern and the underlying hierarchies in modern computer architectures [1].

Synchronization. Traditional methods parallelize incomplete factorization include factoring based on level-sets, coloring, or nested-dissection orderings (ND). However, each of these techniques' standard implementation requires all threads to sync between levels, colors, or tree levels. These syncs can dominate the execution time as the computational intensity of incomplete factorization is much lower than full factorization. In Figure 1, we present a scatter plot of both the number of rows vs number of syncs required to factor 7 matrices reordered with ND using 16 threads of OpenMP style barriers with level-sets. We see from the plot that the ratio can be between $5 \sim 50\times$, and thus demonstrating how much of an overhead syncs are for **ILU**.

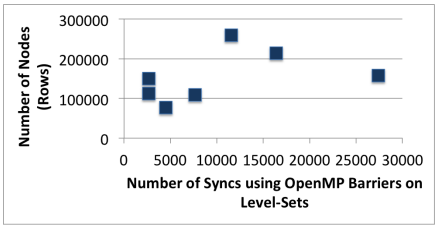

Figure 1: The number of syncs vs number of rows to factor.

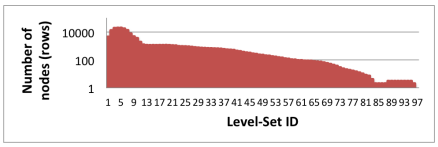

Figure 2: Number of nodes in each level of the level set.

Block Methods. Incomplete factorization tends to be very sparse, and this sparsity may lead to increase in parallelism if effectively utilized. At some point, this parallelism is however limited, and very few parallel incomplete factorization codes take advance of the blocking when parallelism is limited. In Figure 2, we plot a graph of the number of independent rows (y-axis) that can be factored in each level (x-axis) using 16 threads of OpenMP style level-set algorithm for G2_circuit matrix reordered with ND. We see that the tail end can only use a small fraction of a the total number of available threads. Blocking, such as recursive blocking [3], breaks computation into smaller sparse and dense blocks that are cache friendly and can utilize optimized kernel calls. Blocking has been used successfully in a number of dense cases to increase performance and scalability.

HiLUK. `HiLUK` uses both standard implementation techniques, such as level-set and ND ordering, along with other techniques, such as dependency tree pruning, mapping, and blocking, to parallelize up-looking incomplete factorization based on fill-in levels, i.e., **ILU-K**.

Preprocessing. `HiLUK` first reorders A by using an ordering that reduces the number of iterations [2]. Next, a nested-dissection ordering is founding using as few

*jdbooth@sanida.gov
†srajam@sandia.gov, Center for Computing Research, Sandia National Laboratories. Sandia is a multiprogram laboratory operated by Sandia Corporation, a Lockheed Martin Company, for the U.S. Department of Energy under contract DE-AC04-94-AL85000.

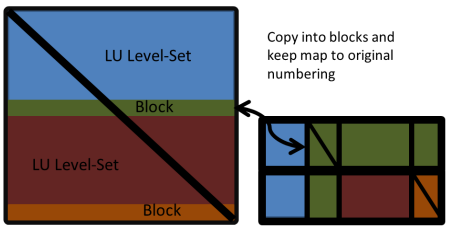

Figure 3: Level-set and block structure of `HiLUK`. Rows in level-set are kept in original order. Rows in blocks are copied to subblock structure with a map to original numbering.

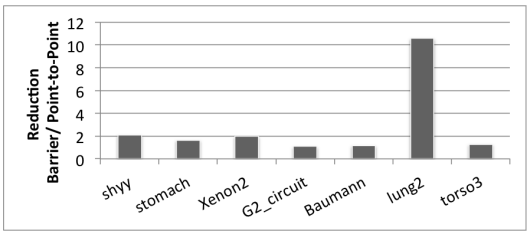

Figure 4: Reduction. OpenMP Barrier / Point-to-Point level-sets.

levels of node partitioning as possible, i.e., the number of leaf nodes is $O(|threads|)$. We have found that by limiting the number of ND levels that the ND ordering has limiting negative effects on iteration count.

Next, the fill pattern is found for a desired fill level. This is currently done in serial, but we are working on a parallel version based on Hysom and Pothen [4]. The thread incomplete Cholesky package, Tacho [5], has found that this implementation is scalable in shared memory. The level-sets of the graph representation of L's fill-in pattern are found. Nodes in level-sets towards the end with limited parallelism are mapped to a block structure, if they exist. These nodes (rows) are not reordered toward the end of the matrix, but are only copied in the block structure with a map that keeps the original ordering. We present this structure in Figure 3.

Level-Set Parallel Execution. Parallel execution based on level-sets is a commonly used approach. However, a data-parallel method, such as using OpenMP with barriers, may have many sync points, as any node in a level might depend on only a few nodes in previous level. Moreover, if the node in the next level is assigned to the same thread as the node in the previous level it depends on, the execution order would take care of the dependency, and no sync would be needed. Using these two observations, a strategy that first assigns nodes to threads based on level-set and next builds the full dependency graph of nodes in the level-set. The found dependency graph is pruned to remove transitive edges from non-neighboring levels and to remove edges from nodes in the same thread based on execution order. Now, the edges remaining are the only needed syncs that can be done in a point-to-point manner similar sparse trisolve as in Park et al. [6]. Figure 4 shows the reduction in syncs using OpenMP barrier to using point-to-point for 7 matrices on average of about 2×. Even G2_circuit which does not have a high reduction in syncs scales better because threads do no have to wait for others when there is load imbalance in a level.

Recursive Block Parallel Execution. In Figure 3, we present a figure that represents the execution on blocks. We note that multiple threads can work on the factorization at the same time using this blocking method, and has been used by many different kernels. This is critical for many matrices with increasing fill-in.

Acknowledgment. We would like to thank Andrew Bradley who first experimented with this framework for sparse trisolve (HTS). We thank Erik Boman and Kyungjoo Kim for many helpful discussions aiding in the initial framework and insight into scaling incomplete factorizations.

References

[1] J. D. Booth, S. Rajamanickam, and H. K. Thornquist, *Basker: A threaded sparse LU utilizing hierarchical parallelism and data layouts*, CoRR, abs/1601.0525 (2016).

[2] I. S. Duff and G. A. Meurant, *The effect of ordering on preconditioned conjugate gradients*, BIT Numerical Mathematics, 29 (1989), pp. 635–657.

[3] E. Elmroth, F. Gustavson, I. Jonsson, and B. Kgstrm, *Recursive blocked algorithms and hybrid data structures for dense matrix library software*, SIAM Review, 46 (2004), pp. 3–45.

[4] D. Hysom and A. Pothen, *A scalable parallel algorithm for incomplete factor preconditioning*, SIAM Journal on Scientific Computing, 22 (2001), pp. 2194–2215.

[5] K. Kim, S. Rajamanickam, H. C. Edwards, S. Olivier, and G. Stelle, *Task Parallel Incomplete Cholesky Factorization using 2D Partitioned-Block Layout*, CoRR, abs/1460309 (2016).

[6] J. Park, M. Smelyanskiy, N. Sundaram, and P. Dubey, *Sparsifying synchronization for high-performance shared-memory sparse triangular solver*, in Proceedings of the 29th International Conference on Supercomputing - Volume 8488, ISC 2014, New York, NY, USA, 2014, Springer-Verlag New York, Inc., pp. 124–140.

[7] Y. Saad and M. Schultz, *Gmres: A generalized minimal residual algorithm for solving nonsymmetric linear systems*, SIAM Journal on Scientific and Statistical Computing, 7 (1986), pp. 856–869.