SAND2016-8033C

A Hybrid Multithreaded Direct Sparse Triangular Solver

Andrew M. Bradley*

Abstract

A triangular solver is an important computational kernel in many applications. On-node parallelism is increasingly important. Multiple approaches underlying multithreaded triangular solvers exist. This article introduces a hybridization approach to combine two classes of parallel triangular solvers, level scheduling and blocking, to efficiently solve triangular systems having a much wider range of sparsity patterns than either algorithm class can efficiently address on its own. The open source software `HTS` implements a hybrid multithreaded triangular solver and is available in Trilinos.

1 Introduction.

This article considers the solution of triangular systems $Tx = b$, where T is an upper or lower triangular matrix, using on-node parallelism up to approximately two hundred hardware threads. I call an algorithm to solve these systems and its software implementation in a package a *triangular solver*. A triangular solver is an important computational kernel in many scientific computations, particularly as all or part of the the preconditioner application phase of an iterative method.

1.1 Setting. This paper focuses on the following computational setting. A sequence of systems $T_i x_i = b_i$ must be solved. The sequence cannot be batched because b_j depends on at least one b_i for $i < j$. Hence substantial parallelism derived from batching is not available. T_i is either the same as T_j, or the two share the same nonzero pattern. Quite likely, exactly the same matrix T is used for at least some equations in sequence—a few to a few tens—and the same nonzero pattern is used for many equations in sequence—hundreds or more.

An example of this setting is the solution of a nonlinear equation over a mesh with fixed topology, by Newton's method, with an iterative method used for the linear equation at each nonlinear iteration. The mesh induces a fixed nonzero pattern for all the T_i, if pivoting is not used in the construction of T. Each nonlinear iteration induces particular numerical entries in T_i. An iteration in time above the nonlinear iteration at each time step may further increase the number of triangular systems having the same nonzero pattern.

This setting permits a solver to use *symbolic analysis* and *numerical* phases to construct and fill data structures to use in the *solution* phase. Phasing is a common work reuse pattern in solvers, such as in UMFPACK [5] and Amesos2 [3] in Trilinos [8]. The time to construct data structures in the first and second phases is amortized over subsequent solutions of equations. In

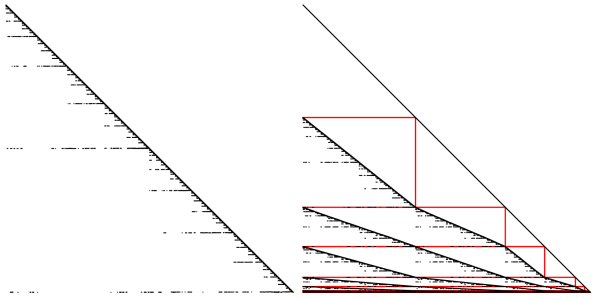

Figure 1: Left, nonzero pattern of a lower triangular matrix. Right, nonzero pattern of this matrix with level schedule ordering. Horizontal red lines separate levels, and vertical red lines separate the MMP blocks from the diagonal blocks.

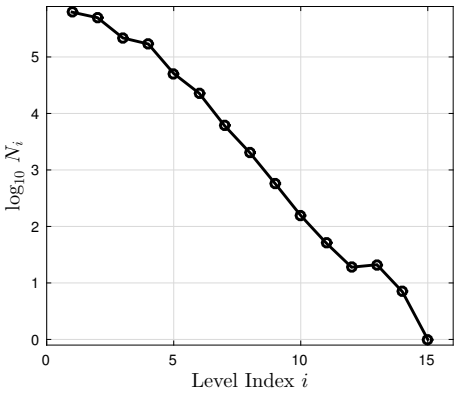

Figure 2: Size of level, N_i, vs. level index, i.

*ambradl@sandia.gov, Center for Computing Research, Sandia National Laboratories, Albuquerque, New Mexico. Sandia National Laboratories is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

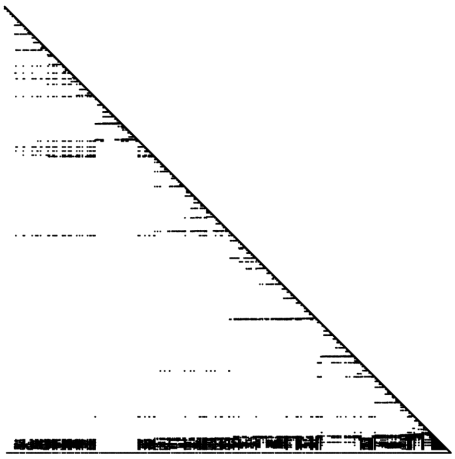

Figure 3: Nonzero pattern of a lower triangular matrix.

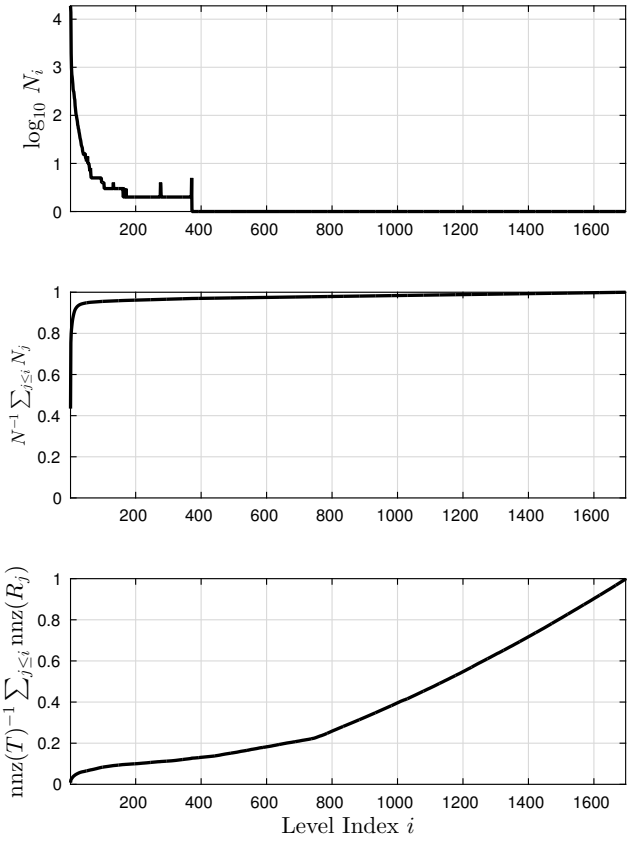

Figure 4: Top, size of level vs. level index i. Middle, cumulative fraction of rows covered through level i. Bottom, cumulative fraction of nonzeros covered through level i.

the setting I consider, the intention is to perform the symbolic analysis phase very occasionally relative to the numerical phase, and the numerical phase somewhat occasionally relative to the solution phase.

1.2 Background and Motivation. Factorization packages provide triangular solvers for factorization-specific data structures, such as the elimination tree and supernodes, and can parallelize the solution phase using these data structures. This article considers *black box* triangular solvers, which have access only to the triangular matrices in a generic input format, such as compressed row storage (CSR).

Level scheduling (see, e.g., [14], [16], [12]) is a common method to expose parallelism in the solution of a very sparse triangular system. In the symbolic analysis phase, it finds a sequence of variable sets R_i that can be solved for simultaneously. Graph coloring induces sets having this same property. A small sequence of large sets is desired. If a matrix is not very sparse, level scheduling quickly arrives at a large serial bottleneck. Level scheduling can be effective on triangles arising from low-fill incomplete factorizations or in smoothers, and tends to be ineffective on matrices arising from complete factorizations.

For not very sparse matrices, another class of approaches is used: blocking. The triangular matrix is blocked by one of multiple means (see, e.g., [11]). Blocks are of one of two mathematical types: a block to which is associated a product between a sparse matrix and a dense vector or matrix (subsequently, an *MMP*) that scatters the solution obtained so far into the remaining right hand side (RHS), and a block to which is associated a smaller triangular system. Blocks must be handled in sequence, but within the first kind of block, and sometimes the second, parallelism is possible. Blocking methods tend to be inefficient on very sparse matrices because of poor data access locality in the right hand side and solution, and little work in each block over which to parallelize. In contrast, blocking methods can be very efficient on not very sparse matrices, and certainly on dense matrices.

Notation. A sequence is indexed by i. A set of rows (equivalently, variables) is R_i. A set's cardinality is N_i. A matrix has N rows, and if R_i partitions the matrix, $N = \sum_i N_i$. R_i *covers* the rows contained in R_i and the nonzeros contained in these rows. $\mathrm{nnz}(A)$ is the number of nonzeros in the nonzero pattern of A.

Figure 1 shows a sparse lower-triangular matrix (left) and a level schedule for this matrix (right). This matrix, as are all matrices in this article except the one used in Figure 6, is derived from one in the UF Sparse Matrix collection [6]. Figures 1 and 2 use `AMD/G3_circuit`. Nodal nested dissection using Parmetis [10] was applied to the level-0 pattern of

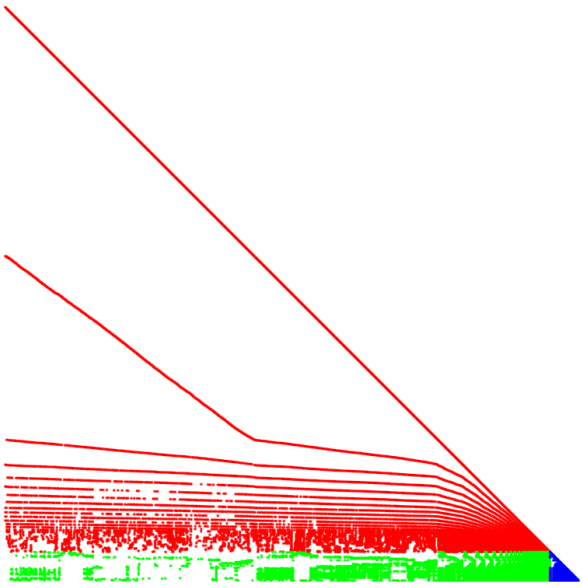

Figure 5: Pattern of a lower triangular matrix with hybrid ordering and partitioning. Red, level scheduled part; blue, blocked part; green, MMP between these two.

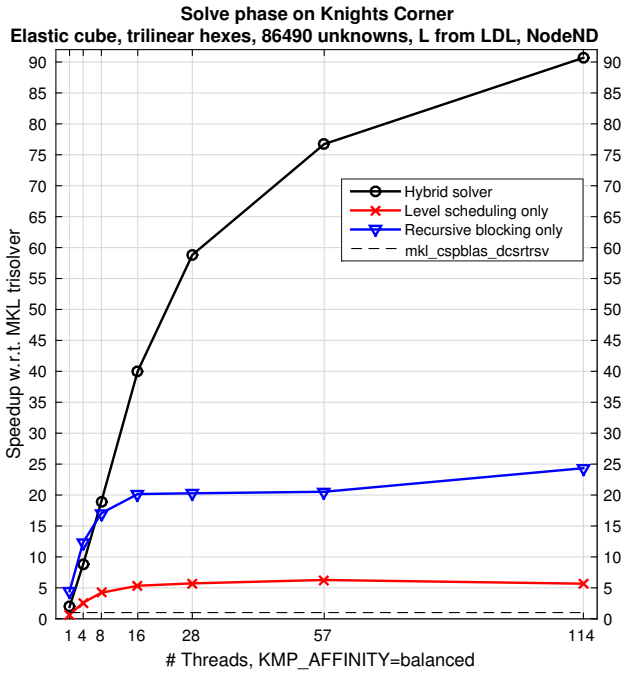

Figure 6: Scaling results on a 57-core Intel Xeon Phi Knights Corner for solution phase of an LDL factorization. Matrix is from discretization of a 3D elastic cube, with nodal nested dissection from Parmetis. Trilinear hexahedral elements were used. Speedup is relative to MKL's serial sparse triangular solver. Hence speedup for that solver is always 1.

the matrix. A level schedule partitions the triangle into rectangles corresponding to MMP and on-diagonal triangular matrices. The triangular matrices are in fact diagonal. Hence each of these blocks permits perfect parallelization within a block. Figure 2 plots the size of each set, N_i, against the set's index, i, for the lower triangle of the matrix. This triangle has 15 levels. In the first five, each has $N_i > 10^4$. In the next five, each has $N_i > 10^2$. Each of the final five is small.

Figure 3 shows the nonzero pattern of the matrix L derived from factoring another matrix, `Schenk_IBMNA/c-60`, with UMFPACK [5]. (This smaller matrix was used for greater clarity in the figures showing nonzero patterns.) The pattern shows the typical fill toward the bottom of L. Figure 4 plots three quantities against the level index i in a level schedule for this triangle. There are over 10^3 levels. Referring to the top plot, the largest level has over 10^4 variables. But each set after the first approximately 20 levels has fewer than 10^2 variables. The middle plot shows cumulative fraction of rows covered by levels. Over 90% of the rows are covered within the first approximately 10 levels. The bottom plot shows cumulative fraction of nonzeros in those rows. Fewer than 10% of nonzeros are covered by level sets having at least 10 variables. After about level index 400, each set has just one row, meaning the algorithm is serialized. At this index, more than 80% of the nonzeros remain uncovered. UMFPACK factorization of matrix `AMD/G3_circuit`, used in Figures 1 and 2, yields a lower triangular matrix having over 10^4 levels, with more than 30% of nonzeros in the fully serialized section.

1.3 Hybrid Triangular Solver. Between level scheduling and blocking, we have two methods that tend to perform well at two different ends of the sparsity spectrum. Level scheduling has another use in addition to solving a triangular system: it induces a symmetric permutation of the triangular system. The right triangle in Figure 1 illustrates this reordering. This permutation tends to further sparsify part of the triangle, and densify another part.

In this article, I exploit these observations to design a hybrid triangular solver that uses both level scheduling and blocking. In addition, I describe a particular software implementation, `HTS`, and results produced by this software. `HTS` is open source software available in Trilinos [8].

Figure 5 shows the triangle in Figure 3 reordered using a *hybrid permutation*. In the case of a lower trian-

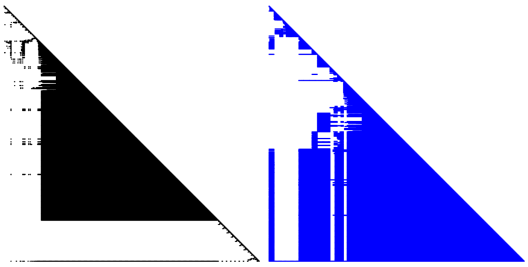

Figure 7: Right, blocked part after hybrid level-scheduling ordering. Left, lower-right corner of same size from original triangle, for comparison.

gle, the hybrid permutation is a set of variable indices in which the first part is a level schedule permutation for part of the triangle, and the second part is the set of remaining variables. An upper triangle can be understood as the transpose of a lower triangle, and subsequent discussion will be relative to lower triangles only. In Figure 5, the red part is level scheduled (LS); the blue part is blocked; and the green part scatters the solution from the LS part into the blocked part's RHS.

Figure 6 shows `HTS`'s speedup relative to Intel MKL's serial triangular solver `mkl_cspblas_dcsrtrsv` on a 57-core Intel Xeon Phi Knights Corner for a particular problem. (Hardware, software environment, and numerical experiments are described comprehensively in §3; however, Figure 6 was created using data from a run on an older Knights Corner than used in §3.) Each datum corresponds to the solution of the lower triangular system arising from the LDL factorization of a matrix. The matrix results from the trilinear hexahedral discretization of an elastic cube. It is ordered using Parmetis's [10] nested nodal dissection. Speedup is shown as a function of number of hardware threads. The thread affinity fills each core before adding another core. `mkl_cspblas_dcsrtrsv` has a speedup factor of 1 because it is serial and its solution time is the unit of time in these data. Level scheduling alone achieves a parallel speedup factor of just over 5. Blocking alone achieves a speedup factor of approximately 25. Each of these is obtained by forcing `HTS` to use just one approach or the other. Finally, `HTS`'s black box hybrid solver achieves a speedup factor of over 90 when the full computer is used with two hardware threads per core.

A key observation is that even if only a small fraction of nonzeros belong to level scheduled rows, the hybrid reordering resulting from the level schedule densifies the part that is recursively blocked. It is denser because the symmetric permutation induced by the level schedules pushes dense rows downward and dense columns to the right (for a lower triangle), and so into the blocked part. This explains the jump in speedup from 25 to 90; it is not just that level scheduling is used to good effect on part of the triangle, but also that the remaining part is denser than in the original ordering of the matrix. Hence level scheduling has two roles: as a means of solving part of the system using graph-based parallelism, and as a reordering to expose greater parallelism in the blocked part. Figure 7 right shows the blocked part; at left, for comparison, is the identically sized bottom-right part of the triangle in its original order. The dense subtriangle is preserved, and additional dense blocks are brought into the part. Storing large portions of this blocked triangle in dense format is efficient.

1.4 Related Work. There are a number of approaches to developing on-node parallelized triangular solvers. Alvarado and Schreiber [1] and Van Duin [15] factor the inverse of the triangle into a product of matrices. Each matrix corresponds to a sparse MMP. Factors have no fill. The method is applicable across the spectrum of matrix sparsity. Stability could be a concern since inverses are used, but Higham and Pothen [9] show that the method is stable if the triangle is well conditioned. I believe implementations of this method on contemporary architectures should be studied.

Wolf, Heroux, and Boman [16] study level scheduling with barrier synchronization between levels. They examine performance in various multithreaded implementation approaches. Naumov [12] also studies level scheduling with barrier, but on a GPU; each level corresponds to a kernel execution. Park et al. [13] use point-to-point synchronization with edge pruning to implement level scheduling without barriers. Instead, each level is broken into tasks, and edges in the resulting task directed acyclic graph (DAG) indicate data dependencies. Pruning removes redundant edges.

All of these methods implement symbolic analysis, numerical, and solution phases. In contrast, Chow and Patel [4] and Antz et al. [2] solve triangular systems in place, using a Jacobi iteration or asynchronous, nondeterministic Gauss-Seidel-like iteration. An analysis phase is not necessary because each iteration is essentially an MMP. The Jacobi iteration has assured convergence; at worst, N iterations must yield the solution.

There has been a large amount of work on blocked solvers for both dense and sparse formats. `HTS` follows an approach similar to that in [11]: the matrix is reformatted so that each block's data are local to the block.

2 Algorithms.

The hybrid trisolver must act as a black box; it must determine how to reorder and partition the triangle algorithmically, without input from the user. In principle, a solver could alternate between level-scheduled and blocked parts; in this article, at most one of each part is permitted.

The triangular system associated with each part can be solved using any of a number of algorithms and variants. I have attempted to give `HTS` high-quality implementations of state-of-the-art level scheduling and blocking algorithms. While this article's focus is the utility of the hybrid approach, this section also describes some of the details of these implementations.

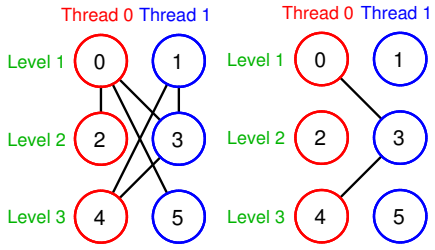

Figure 8: Left, original task graph; right, pruned.

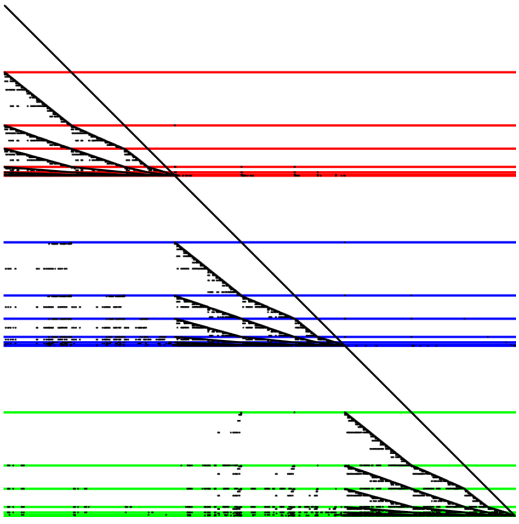

Figure 9: Nonzero pattern of a level-scheduled matrix after permutation for data locality in each of three threads. Color corresponds to thread. Levels within a thread are separated by horizontal lines.

2.1 Hybrid Transition.

`HTS` uses the following procedure to determine the partition. First, the triangle is level scheduled following equation 18 in [14]. Associated with the schedule is a sequence N_i of level sizes with i the level index. The number of rows in the triangle is $N = \sum_i N_i$. Second, two parameters, n_{good} and f_{bad}, are chosen. n_{good} is the minimum set size considered *good*. f_{bad} is the fraction of *bad rows* in the level schedule; a bad row is one that belongs to a set whose size $N_i < n_{\text{good}}$. In §3 and in HTS's default settings, $n_{\text{good}} = 10$ and $f_{\text{bad}} = 0.01$, but results are not very sensitive to these parameter values. Third, $C_i \equiv \sum_{j \le i} N_j$ and $C_i^{\text{bad}} \equiv \sum_{j \in J} N_j$, $J \equiv \{j : j \le i, N_j < n_{\text{good}}\}$. Fourth, the largest i is found such that $N_i \ge n_{\text{good}}$ and $C_i^{\text{bad}} \le f_{\text{bad}} C_i$. Finally, all levels through i are used, and rows corresponding to subsequent levels are blocked in the reordered matrix.

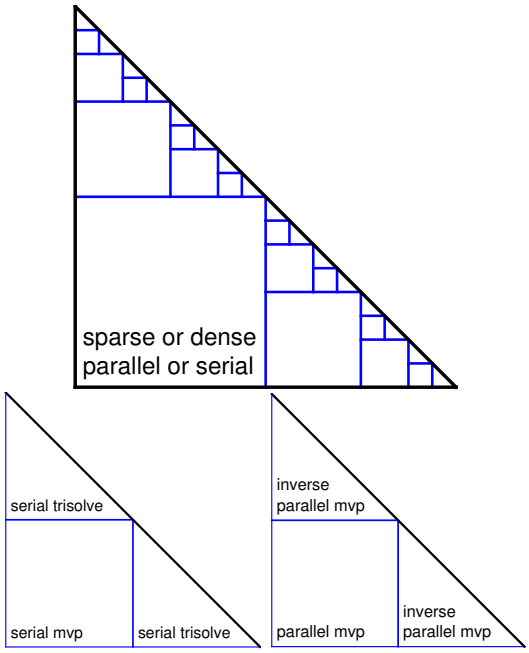

Figure 10: Recursive blocking structure. Top, full structure. Bottom, two possible implementations of on-diagonal blocks.

For a hybrid algorithm that permits at most one transition, an effective algorithm is robust to downward and upward spikes in the plot of N_i. Examples of ineffective algorithms are to transition at the earliest level i such that $N_i < n_{\text{good}}$ or the latest such that $N_i \ge n_{\text{good}}$. `HTS`'s algorithm is insensitive to spikes.

Level size as a function of level depends strongly on matrix ordering. To capture nonmonotonicity of this function, a hybrid algorithm could switch between two methods multiple times. At present, `HTS` transitions just once. Its transition method leads to useful partitions for both noisy monotonically decreasing and concave functions.

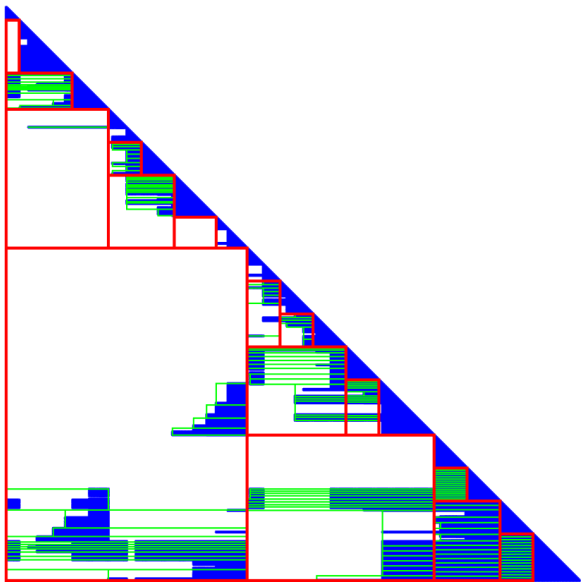

Figure 11: Data structures for recursively blocked part of L factor. Blue, nonzero pattern; red, block structure; green, data partition among threads in an MMP block.

2.2 Level Scheduling. `HTS` follows Park et al.'s [13] pruned point-to-point level-scheduling algorithm for its LS block. Their approach has two key ideas. First, instead of using a global thread barrier between each level, partition each level into tasks and implement data dependencies with point-to-point synchronization. Second, create a static task DAG in the symbolic analysis phase, and prune it to remove redundant edges. Each edge corresponds to a shared-memory variable on which to spin-wait; hence pruning edges reduces the number of variables and spin waits.

Figure 8 illustrates edge pruning. At left, each of three levels has been split into two tasks, one per thread, for a total of six tasks. Edges between task nodes encode data dependencies; for example, task 4 depends on variables solved for by tasks 1 and 3. First, every edge between tasks in the same thread can be pruned because program execution order of the thread assures that all variables are solved for in the correct order. Second, one edge in a triangle of edges can be pruned. For example, tasks 1, 3, and 4 form a triangle. The edge between 1 and 4 can be pruned because task 1 is assuredly done when task 3 is done. Third, a triangle can be created simply to induce a pruned edge. For example, both tasks 3 and 5 depend on 0. 5 occurs in program order after 3, so we can add an edge between tasks 3 and 5. This forms a triangle, allowing us to remove the edge between 0 and 5. Pruning can be done on other than triangles in the graph, but Park et al. [13] show that, in practice, pruning based on only triangles and simpler rules handles almost all edges while saving the cost of more analysis.

The number of edges can grow quadratically with the number of threads because it is possible for a task to depend on all tasks in the previous level. Symbolic analysis can be done in parallel, but quadratic growth in edges implies the approach is not naively scalable. Practical details mitigate this problem. First, LS is either applied only to problems for which it is effective or, in a hybrid triangular solver, to only the part of the problem for which it is effective. Hence the quadratic-growth case tends not to occur in practice. Second, edge pruning work can be reduced substantially by the following procedure. Consider a task t with dependencies on tasks $d \equiv \{d_i\}$; each task d_i is dependent on tasks $c_i \equiv \{c_{i,j}\}$. A key step in edge pruning is to intersect the set d against the set c_i. If a dependency is common to the two, then there is a triangle of edges, and one edge can be removed. An important implementation detail is to preprocess d to produce a smaller set d' with which to perform intersections. First, suppose t depends on multiple tasks owned by the same thread; then d' contains only the task at the latest level in the schedule. Second, if a dependency d_i is only one level shallower than t, then it cannot be pruned, and also it will not appear in any c_i, so d' does not contain it. A final implementation detail supports these dependency pruning rules: order dependency lists by thread, then by level.

Another implementation detail that is important in the solution phase is to symmetrically permute the LS part to group all rows that belong to a thread together [13], producing a nonzero pattern like that illustrated in Figure 9. Each color corresponds to a thread, and horizontal lines separate tasks, and so levels, within a thread. The nonzeros are predominantly in diagonal blocks, increasing data locality in access to the RHS and solution relative to the original ordering.

2.3 Recursive Blocking. `HTS` uses recursive blocking (RB) for the blocked part. Each block stores its data in any of a number of formats, and computes in any of a number of ways, depending on just its data.

Figure 10 summarizes choices schematically. At top, a triangle is blocked recursively. At each level of the recursion, there is one MMP block and two on-diagonal triangles. Each triangle may then be blocked. Each MMP block contains data in sparse (CSR in `HTS`) or dense (row major in `HTS`) format. In addition, each block's MMP can be computed in serial or parallel. At Figure 10 bottom, on-diagonal triangles and MMP blocks at the leaf level can be treated in two ways.

At left, each block's computation is serial: a serial sparse or dense triangular solution, and a serial sparse or dense MMP. The MMP block is serial because otherwise it would not be at the leaf level of the recursion tree. At Figure 10 right, the on-diagonal triangles are inverted in the numerical phase; in the solution phase, a parallel MMP is computed. The inverse of an on-diagonal triangular matrix is computed by substitution independently by column, corresponding to Method 1 in [7].

Recursive partitioning depends on three values. The first is the minimum block size. The second is the size of the block to split; it is bisected if the third value is not relevant and if the block is larger than the minimum block size. The third value is a list of favorable split points computed before recursive partitioning. A favorable split point is one that exploits on-diagonal separability. Figure 11 illustrates these procedures. A triangle with small dense blocks (nonzero pattern is blue) is recursively partitioned. Blocks are outlined in red. Notice the first split is not an even bisection of the triangle; at the top-right corner of the largest MMP block, we see that the split exploits structure in the on-diagonal values. There are at least five instances of this behavior in Figure 11. A green rectangle outlines the row-oriented data that belong to a thread within an MMP block. Each thread can represent its data in dense row-major or CSR formats. A thread owns a block of contiguous rows. In `HTS`, currently density is determined based on fraction of nonzeros in a minimal rectangle encompassing the nonzeros in these rows; an improved implementation would further partition nonzeros in the column direction to find dense subblocks within a thread's data. In this example, the on-diagonal triangles store dense inverses. These data are also partitioned among threads, with load balance based on nonzeros, but the corresponding green rectangles are omitted in Figure 11. Both MMP and on-diagonal triangular blocks can choose to use fewer threads than are available, if there is not sufficient data in the block to use all threads.

2.4 Phases. The symbolic analysis phase determines the level schedule, the hybrid transition, the recursive block structure, and the task DAG for the level schedule. The symbolic analysis phase is parallelized. However, because it does substantial work, it requires substantially more time than the other two phases. For this reason, `HTS` is best suited to problems having a sequence of triangular systems with fixed nonzero pattern.

The numerical phase moves values from the caller's triangle into the internal data structures created in the symbolic analysis phase. This is an embarrassingly parallel operation. The numerical phase is separate from the symbolic analysis phase so that a sequence of triangular matrices sharing the same nonzero pattern can use just one symbolic analysis.

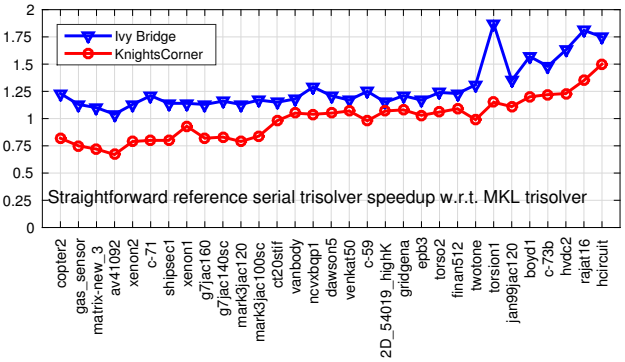

Figure 12: Comparison of benchmark serial trisolvers.

The solution phase solves a triangular system for one or more right hand sides in batch, and is intended to be called sequentially many times.

`HTS`'s current implementation requires that the number of threads be the same in each phase. A useful future improvement would be to allow more threads—typically, twice as many—in the numerical and solution phases than in the symbolic analysis phase. Using both hardware threads within a conventional CPU core tends to give a little speedup in the numerical and solution phases over using just 1 thread per CPU core, but it tends to slow the symbolic analysis phase substantially, as demonstrated in §3.

2.5 Software. `HTS` is open source software available in Trilinos [8]. It uses C++98 and OpenMP 3.

The caller indicates the number of threads `HTS` may use, but `HTS` internally may throttle the number of threads at the granularity of level set and block.

3 Results.

The Intel compiler suite version 15.0.2 was used. Two platforms were used: an Intel Xeon Ivy Bridge, CPU E5-2670 v2, 2.50GHz, 20 cores, 10 cores per socket, 2 hardware threads per core, 256 GB memory; and Intel Xeon Phi Knights Corner, 1.238 GHz, 61 cores, 4 hardware threads per core, 16 GB memory.

Results were obtained for triangular systems induced by the ordering and LU factorization of UMFPACK [5] integrated into the `lu` command in Matlab. However, `HTS` does not know about the factorization; it analyzes and solves a triangular system based only on the nonzero pattern of the triangular matrix. Each datum corresponds to a computation including

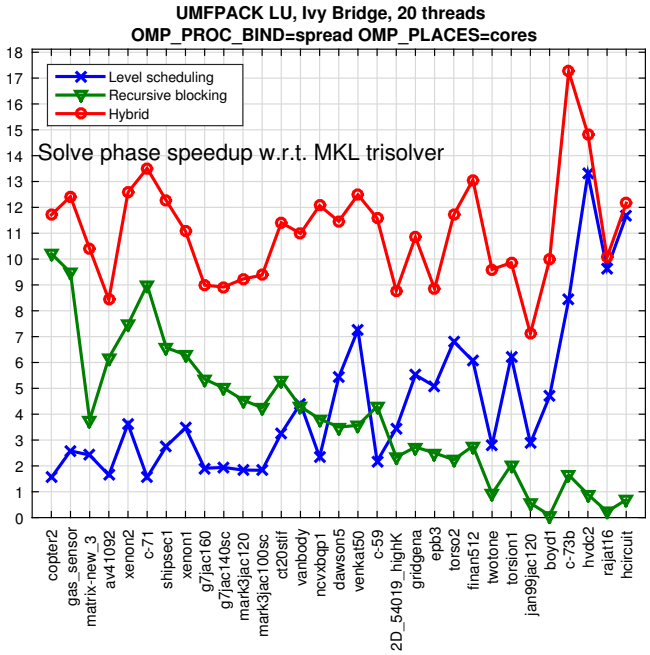

Figure 13: Results on 20-core Intel Ivy Bridge. Matrices are ordered by decreasing $(\text{nnz}(L) + \text{nnz}(U))/N$. Recursive blocking is favored to left and level scheduling to right; monolithic solver performance crosses at about middle.

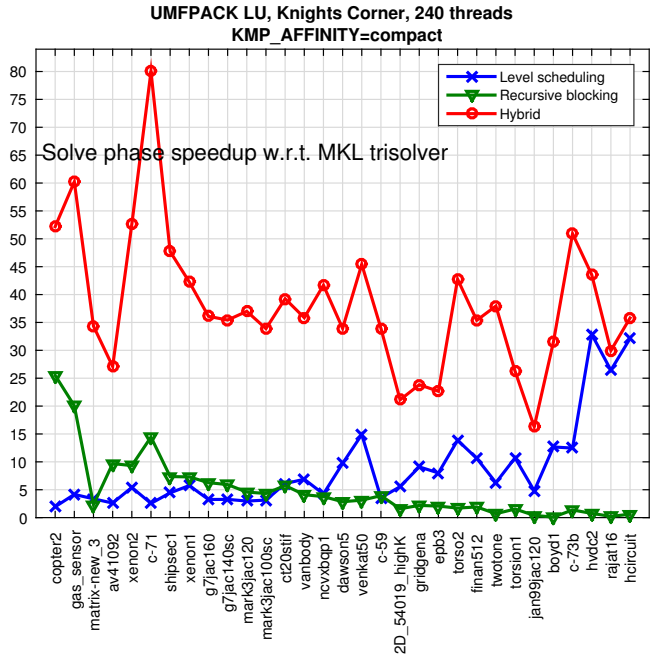

Figure 14: Results on 61-core Intel Xeon Phi Knights Corner.

permutation and scaling of the RHS, solution of the L and U systems, and permutation of the solution. A datum is speedup of this sequence of computations using `HTS` relative to that using Intel MKL's serial `mkl_cspblas_dcsrtrsv`. However, this speedup can be transformed to one with respect to a straightforward serial trisolver, provided by `HTS` for reference, by using the comparison in Figure 12. Finally, a datum is a median of measurements from at least five runs, with all matrices run once before the next run to capture system variability. Within each run, the solution phase is run a few times to bring the system into steady state; then the mean time over several tens of solutions is recorded.

Matrices are from the University of Florida Sparse Matrix Collection [6]. For detailed analysis, 31 matrices meeting certain criteria were randomly selected from the set of over 800 matrices used overall. These matrices have $N \geq 40$ thousand, $N < 190$ thousand, with a median of approximately 55 thousand. In figures, they are ordered by decreasing $(\text{nnz}(L) + \text{nnz}(U))/N$. Thus, in figures, recursive blocking is generally more effective to the left, and level scheduling is generally more effective to the right.

Figures 13 and 14 show speedup of the monolithic LS, monolithic RB, and hybrid methods on, respectively, Ivy Bridge and Knights Corner. (All figures list thread and thread affinity data.) `HTS` was used as the implementation for all three solvers. The order of matrices from left to right induces a crossover point in about the middle of each plot at which the two monolithic approaches switch. The hybrid method is uniformly the best.

Figures 15 and 16 show results again for each architecture. For Ivy Bridge, results are shown for 10 threads (1 thread/core, 1 socket), 20 threads (1 thread/core, 2 sockets) and 40 threads (2 threads/core, 2 sockets). For Knights Corner, compact thread affinity is used so that 60 threads use 15 cores with 4 threads/core, 120 threads use 30 cores, and 240 threads use 60 of the 61 cores. In each figure, the top plot shows solution phase speedup relative to MKL's serial triangular solver. The middle plot shows the numerical phase time, i.e., the amount of time it takes to load new numbers into `HTS`'s data structures. Time is expressed in terms of parallel solution time. On Ivy Bridge, it takes about 8 times longer to load new numbers into `HTS` than too solve an equation; on Knights Corner, the relative time is slightly less. The bottom plot shows the time of the symbolic analysis phase in terms of MKL solution time. On Ivy Bridge, using hardware threads in this phase gives poor performance. At 20 threads, symbolic analysis takes about 5 times longer than a serial solve. On Knights corner, the relative time is again a little less.

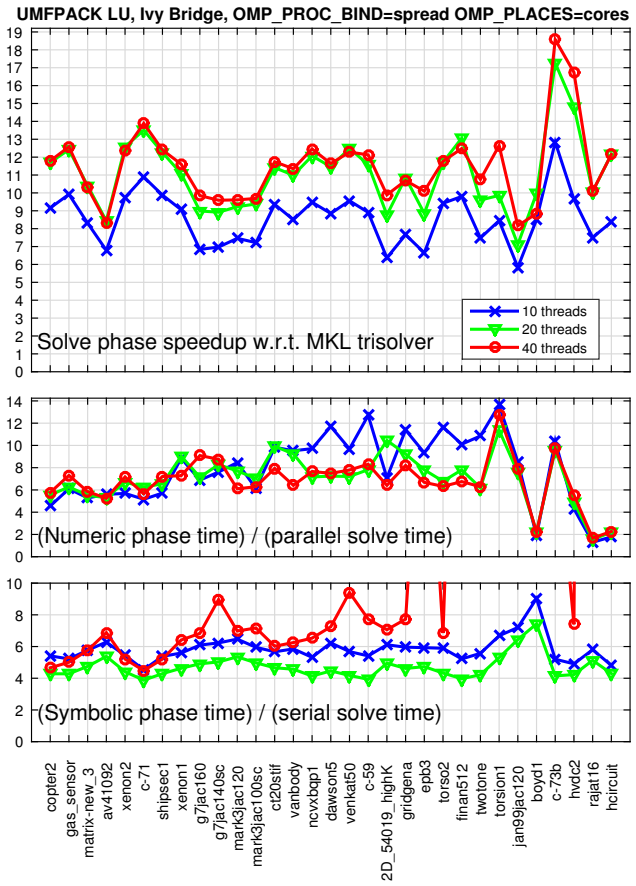

Figure 15: Results on 20-core Intel Ivy Bridge. Hardware threads are used for 40-thread results.

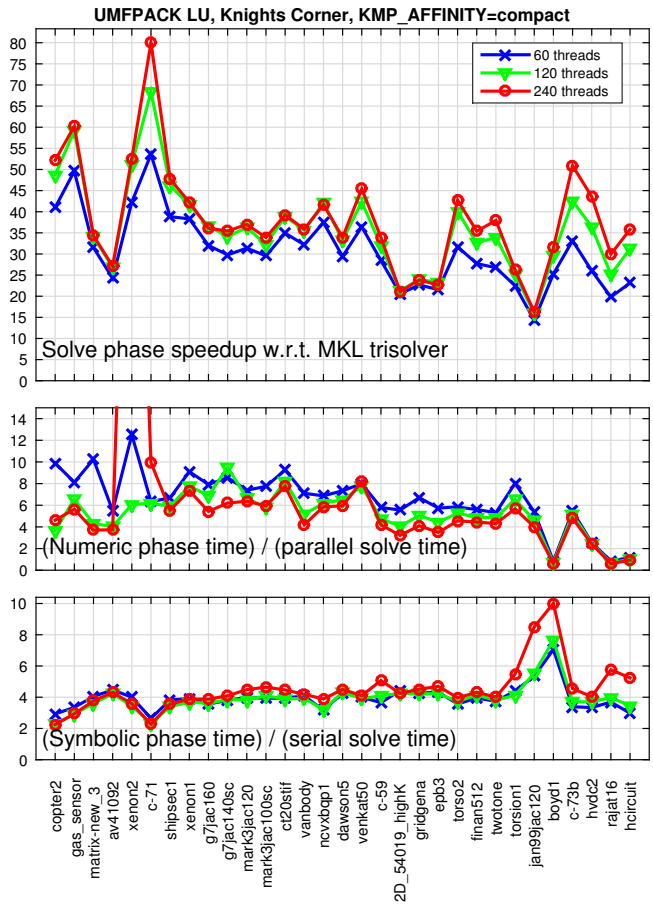

Figure 16: Results on 61-core Intel Knights Corner. Affinity setting fills all four threads of one core before using another core. Hence 60 threads use 15 cores.

Figures 17 and 18 show results for over 800 matrices from the collection. Each point corresponds to a matrix. Speedup is plotted against size of a matrix N. Although 20 and 240 threads, respectively, were provided to `HTS`, it is unlikely that they were all used on the smallest matrices because `HTS` throttles thread count on small problems. Larger matrices provide greater opportunity for efficient parallelization. The red line at N shows the median speedup for all matrices having at least N rows. Hence, for Ivy Bridge, the median speedup factor for all matrices with $N \geq 10^4$ is approximately 9.

4 Conclusions.

Figures 13 and 14 provide substantial evidence that a hybrid triangular solver is robust across a range of problem types. Because the monolithic solvers are `HTS` with parameters set to force monolithic solution algorithms, we conclude that a hybrid approach yields a triangular solver that is often substantially faster than the fastest of the subsolvers for a given problem. Hybridization gives each subsolver a subproblem better suited to it than the full problem.

There are at least four broad areas for further investigation. First, a hybrid triangular solver could be designed to transition between two or more subsolvers multiple times. Second, other algorithms for the determination of transition points can be explored. Third, various subsolvers could be plugged into a hybrid triangular solver rather than the fixed two in `HTS`. Fourth, hybridization is independent of architecture or computation model; for example, two triangular solvers with GPU implementations could be combined to create a GPU triangular solver that is effective across a broader range of matrices than each subsolver separately.

Acknowledgments. I thank Erik Boman, Joshua Booth, Eric Cyr, Clark Dohrmann, William Held, Mark Hoemmen, Kyungjoo Kim, Stephen Olivier, Aftab Patel, Andrey Prokopenko, and Siva Rajamanickam for many helpful discussions, and three anonymous review-

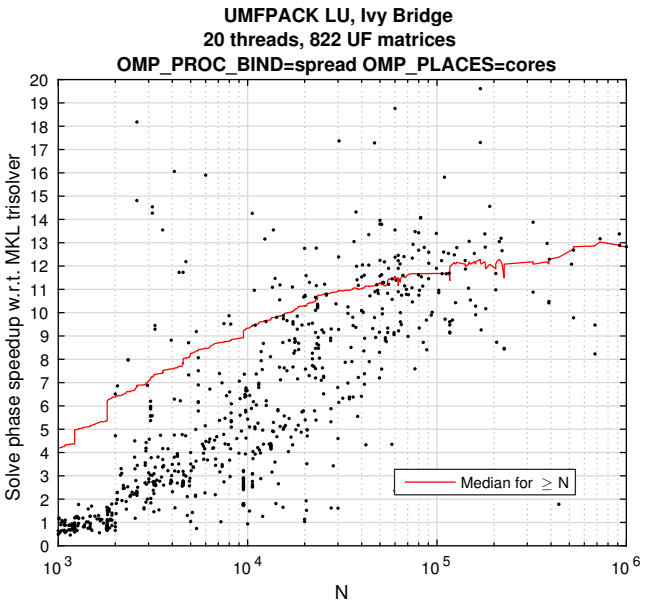

Figure 17: Results on 20-core Ivy Bridge for solve phase of 822 UF matrices. Ordinate is speedup relative to MKL serial trisolver, obtained as maximum speedup over all thread trials. Coordinate is number of rows N. A dot corresponds to one matrix. Red line at N is median speedup for all matrices having at least N rows.

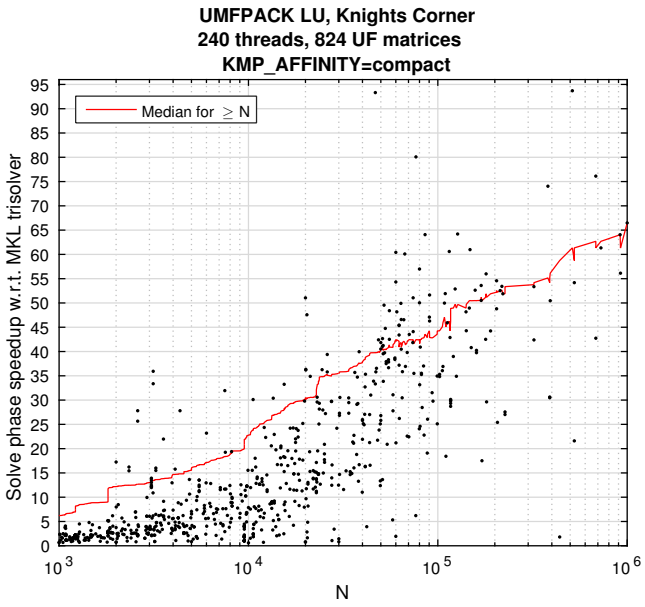

Figure 18: Results on 61-core Intel Knights Corner. See Fig. 17 for explanation.

ers for their helpful comments.

References

[1] F. L. Alvarado and R. Schreiber, *Optimal parallel solution of sparse triangular systems*, SIAM J. Sci. Comput., 14 (1993), pp. 446–460.

[2] H. Anzt, E. Chow, and J. Dongarra, *Iterative sparse triangular solves for preconditioning*, Euro-Par 2015: Parallel Processing, Springer Berlin Heidelberg, 2015, pp. 650–661.

[3] E. Bavier, M. Hoemmen, S. Rajamanickam, and H. Thornquist, *Amesos2 and Belos: Direct and iterative solvers for large sparse linear systems*, Scientific Programming, 20 (2012), pp. 241–255.

[4] E. Chow and A. Patel, *Fine-grained parallel incomplete LU factorization*, SIAM J. Sci. Comput., 37 (2015), pp. 169–193.

[5] T. A. Davis, *Algorithm 832: UMFPACK V4.3—an unsymmetric-pattern multifrontal method*, ACM TOMS, 30 (2004), pp. 196–199.

[6] T. A. Davis and Y. Hu, *The University of Florida Sparse Matrix Collection*, ACM TOMS, 38 (2011), pp. 1–25.

[7] J. J. Du Croz and N. J. Higham, *Stability of methods for matrix inversion*, IMA J. Num. Ana., 12 (1992), pp. 1–19.

[8] M. A. Heroux, et al, *An overview of the Trilinos project*, ACM TOMS, 31 (2005), pp. 397–423.

[9] N. J. Higham and A. Pothen, *Stability of the partitioned inverse method for parallel solution of sparse triangular systems*, SIAM J. Sci. Comput., 15 (1994), pp. 139–148.

[10] G. Karypis and V. Kumar, *A fast and high quality multilevel scheme for partitioning irregular graphs*, SIAM J. Sci. Comput., 20 (1999), pp. 359–392.

[11] M. Martone, S. Filippone, S. Tucci, M. Paprzycki, and M. Ganzha, *Utilizing recursive storage in sparse matrix-vector multiplication—preliminary considerations*, in CATA, (2010), pp. 300–305.

[12] M. Naumov, *Parallel solution of sparse triangular linear systems in the preconditioned iterative methods on the GPU*, Tech. Report NVR-2011-001, NVIDIA, 2011.

[13] J. Park, M. Smelyanskiy, N. Sundaram, and P. Dubey, *Sparsifying synchronizations for high-performance shared-memory sparse triangular solver*, ISC, 2014.

[14] Y. Saad, *Krylov subspace methods on supercomputers*, SIAM J. Sci. and Stat. Comput., 10 (1989), pp. 1200–1232.

[15] A. C. N. Van Duin, *Scalable parallel preconditioning with the sparse approximate inverse of triangular matrices*, SIAM J. Matrix Ana. and App., 20 (1999), pp. 987–1006.

[16] M. Wolf, M. Heroux, and E. Boman, *Factors impacting performance of multithreaded sparse triangular solve*, in High Performance Computing for Computational Science, VECPAR 2010, Lec. Notes in Comp. Sci., Springer Berlin Heidelberg, 6449, 2011, pp. 32–44.