SAND2015-9092C

Basker: A Threaded Sparse LU Factorization Utilizing Hierarchical Parallelism and Data Layouts

Joshua Dennis Booth
Sandia National Laboratories
Albuquerque, New Mexico
jdbooth@sandia.gov

Sivasankaran Rajamanickam
Sandia National Laboratories
Albuquerque, New Mexico
srajama@sandia.gov

Heidi K. Thornquist
Sandia National Laboratories
Albuquerque, New Mexico,
hkthorn@sandia.gov

***Abstract*—Scalable sparse LU factorization is critical for simulations such as those from circuit and powergrid simulations. In this work, we present a new scalable sparse direct LU solver called *Basker* . First, *Basker* introduces a new algorithm to parallelize the Gilbert-Peierls algorithm for sparse LU factorization. Second, as architectures evolve, there exists a need for algorithms that are hierarchical in nature even on a multicore or manycore node to match the hierarchy in thread teams, individual threads, and vector level parallelism. *Basker* is designed to map well to this hierarchy in architectures naturally. Third, there is a need for the data layouts to match multiple levels of hierarchy in memory. *Basker* uses a two-dimensional hierarchical structure of sparse matrices that maps to the hierarchy in the memory architectures and to the hierarchy in parallelism. Finally, we present performance evaluation on Intel SandyBridge and Xeon Phi platforms on circuit and power network matrices taken from the University of Florida sparse matrix collection and from real simulations of the SPICE like simulator Xyce. From this evaluation, *Basker* is able to achieve a geometric mean speedup of $5.91\times$ on CPU (16 cores) and $7.4\times$ on Xeon Phi (32 cores) relative to the solver package KLU. *Basker* outperforms Intel MKL Pardiso (PMKL) by as much as $53\times$ on CPU (16 cores) and $13.3\times$ on Xeon Phi (32 cores) for low fill-in circuit matrices. Furthermore, *Basker* speedups a matrix sequence taken from a real Xyce simulation compared to PMKL and KLU.**

I. Introduction

Scalable sparse direct linear solvers play a pivotal role in the efficiency of simulation codes on many-core systems. Current approaches process multiple columns with similar nonzero structure, i.e., supernodes, with threaded Basic Linear Algebra Subprograms (BLAS) [1], [2]. The one-dimensional approach of using BLAS on these matrices may not be able to extract enough parallelism when the matrix has low fill-in or an irregular nonzero pattern, such as matrices generated by Simulation Program with Integrated Circuit Emphasis (SPICE) [3]. Therefore, a new type of solver is needed that uses a hierarchy of structures to leverage fine-grain parallelism within the irregular nonzero pattern. In this work, we present a new shared-memory sparse direct LU linear solver named *Basker* designed to achieve speedup by using a hierarchy of structure that exploits fine-grain parallelism and naturally fit the hierarchical memory structure of most many-core systems. *Basker* thereby being the first parallel shared-memory solver target at these low fill-in density coefficent matrices.

Sparse factorization of unsymmetric indefinite systems is difficult due the need for numerical pivoting for stability and dynamic nonzero structure generated by such pivoting. Scaling sparse LU therefore depends of efficiently finding concurrent work inside this dynamic nonzero structure while providing enough numerical stability. As a result, speedups achievable for spare factorization is far from ideal [2], [4]. Coefficient matrices with low fill-in are particularly difficult, since there exist limited supernodes. However, a hierarchy of structure can often be found in these matrices that can expose multiple levels of parallelism.

Basker utilizes a hierarchy of two-dimensional sparse blocks designed to fit the multiple levels of nonzero pattern structure that can be found in a coefficient matrix. These patterns can be found with traditional ordering techniques, such as block triangular form [5] and nested-dissection ordering [6]. This hierarchy of two-dimensional sparse blocks design allow *Basker* to accomplish two tasks. The first task is to exploit any fine-grain parallelism found within or between blocks. The second task is to provide a hierarchical data structure that fits the multiple levels of memory found on many-core nodes and divides data among threads. As a result, *Basker* is able to provide new functionally, such as a new algorithm to parallelize Gilbert-Peierls algorithm , and have multiple threads work simultaneous on a single matrix column.

In this work, we present the algorithm and structures used by *Basker* to achieve hierarchical parallelism. *Basker* is a new shared-memory sparse LU solver implemented in templated C++11 with Kokkos [7], where Kokkos is a package for portability across multiple many-core processors and device backends. The main contributions of this work are:

- A new threaded sparse direct LU solver that out performs Intel MKL's Pardiso and KLU while reducing memory on matrices with low fill-in;
- An overview of how to reveal hierarchical parallelism from sparse coefficient matrices;
- The first parallel Gilbert-Peierls algorithm implementation;
- Empirical evaluation of *Basker* compared to KLU [5] and Intel MKL's Pardiso [4] on Intel SandyBridge and Xeon Phi.
- Performance evaluation with 1000 matrices from a real simulation from Xyce circuit simulation package

The remainder of this paper is as follows. We first present

an overview of needed background and previous work in Section II. We then introduce the hierarchical structured algorithm to extract parallelism from sparse coefficient matrices in Section III. An overview of implementation choices are outlined in Section IV. Section V provides results on speedup, comparison of speedups achievable by supernodal methods on 2/3D mesh problems, and use of *Basker* on a circuit simulation from Xyce. Section VI provides future improvement to *Basker* , and Section VII provides a summery of our findings.

II. Background and Related Work

This section provides an overview of background and related work to the solution of the sparse linear system $Ax = b$, where A is a large sparse coefficient matrix. x is the solution vector, b is the given right-hand side vector.

Orderings. All sparse direct solvers use structural information and assumptions to improve performance and scalability. Coefficient matrices are often reordered to limit fill-in, i.e., zeros becoming nonzero during factorization, or cluster nonzeros into patterns that revealing dependency in computation. Minimal degree orderings, such as approximate minimal degree ordering (AMD) [8], are an ordering subtype that are very efficient in reducing fill-in [9]. Additionally, minimal degree ordering may not directly reveal any submatrices that can be computed in parallel.

Nested-dissection ordering (ND) [6] is a graph representation based ordering, using $G(A)$ when A is symmetric and $G(A+A^T)$ when A is unsymmetric. It is commonly used to provide a tree-structure of submatrices that can be computed in parallel while reducing fill-in. However, the tree-structure can be destroyed during factorization if a pivot is selected from outside the submatrix.

If an unsymmetric matrix does not have the strong Hall property, i.e., if every set of k columns has nonzeros in at least k+1 rows, than the coefficient matrix can be permuted in a block triangular form (BTF) where block submatrices in the lower half are all zeros. A coefficient matrix A permuted by matrices P and Q into BTF has the form:

$$PAQ = \begin{bmatrix} A_{11} & A_{12} & \cdots & A_{1k} \\ & A_{22} & & \vdots \\ & & \ddots & \vdots \\ & & & A_{kk} \end{bmatrix}.$$

This form is very common in very irregular unsymmetric systems, such as those from circuit simulation, and can be found with Dulmage-Mandelsohn decomposition (DM) [10]. In this form, only the submatrix blocks on the diagonal need to be factored resulting in far less work and revealing a great deal of parallelism.

Additionally, permuting a coefficient matrix to limit pivoting by placing nonzeros on the diagonal is common before computation [11]. Finding such a permutation is done through finding a maximum cardinality matching of a bipartite graph representation of the coefficient matrix [12]. However, nonzeros on the diagonal is only one half of the issue; a variant that also tries to maximize the values on the diagonal is often used. We will call this variant maximum weight-cardinality matching ordering (MWCM) [12]. There are multiple types of MWCM based on metrics used to weight the diagonal entries.

Sparse LU. The problem of factoring a symmetric sparse coeffiecent matrix A into LL^T and LDL^T is well studied. However, unsymmetric and indefinite systems, may require numerical pivoting and dynamic discovery of nonzero structure resulting in fewer solver packages. Here, we will look at three solver packages, namely SuperLU-Dist [11], Pardiso [4], and KLU [5], to compare their design choices to *Basker* .

SuperLU-Dist is a distributed memory unsymmetric direct solver by Li and Demmel [11]. SuperLU-Dist uses a two-dimensional data layout and avoids pivoting by using a MWCM that maximizes the sum of the diagonal element (MC64) [12]. In each block matrix, SuperLU-Dist performs a supernodal based LU factorization. Supernodal factorization groups a cluster of columns/rows that will have a similar nonzero structure after factorization together and performs the update using BLAS operations [1]. However, supernodal methods have limitations in sparse LU. These limitations include that a pivot could only be chosen from inside a single supernode, fill-in must be known before hand, and scaling is limited to by the size of supernodes [13]. A share-memory version SuperLU-MT [13] exists, however the shared-memory version only uses a one-dimensional data layout.

Pardiso [4] is shared-memory sparse LU solver that uses a number of advanced techniques to achieve high performance. These techniques include using a left-right looking strategy to reduce synchronization and provide three levels of parallelism, namely from the etree, hybrid (left-right) at top levels, and pipelining parallelism. This strategy is applied using a supernodal method similar to SuperLU-Dist . One version of this solver has been added to Intel MKL library, and we compare against it in Section V.

KLU [5] is a serial Gilbert-Peierls algorithm solver designed for circuit simulation problems. It achieves good performance by permuting the circuit coefficient matrix first into BTF . It then uses the Gilbert-Peierls algorithm to discover the nonzero pattern due to fill-in during numeric factorization in time propoortional to arithmetic operations. However, KLU has no method to factor any part in parallel. *Basker* was designed to replace KLU for circuit simulation problems by adding parallel execution both between blocks and within blocks of BTF . It will be added to Trillinos through both Amesos2 and ShyLU [14] packages.

Basker differs from the above solvers in the following way:

- *Basker* is a nonsupernodal factorization unlike SuperLU-Dist and Pardiso ;
- *Basker* uses a heirarchy of structures and revealing orders;
- *Basker* uses both a MWCM and pivoting unlike SuperLU-Dist and Pardiso ;
- *Basker* is a templeted C++ solver using a a many-core portable package supporting muliple backends such as OpenMP, PThreads and QThreads.

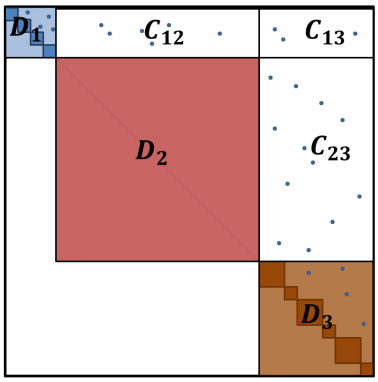
III. Basker Structure

The nonzero pattern in the coefficient matrix determines not only the work but the maximum performance and speedup achievable by sparse factorization. Most shared-memory LU factorization codes utilize a flat one-dimensional layout of blocks where blocks are derived from some ordering or the $etree$ [1]. However, using only one-dimensional methods limit the exploitation of sparsity patterns within and between block structures. For instance, a supernodal factorization's speedup will be either multithreaded BLAS calls or efficiently overlapping serial BLAS calls. Furthermore, algorithms that do not use multthreaded BLAS will have speedups limited by the number of columns that need to be factorized in serial, which can be greater than 10% [6].

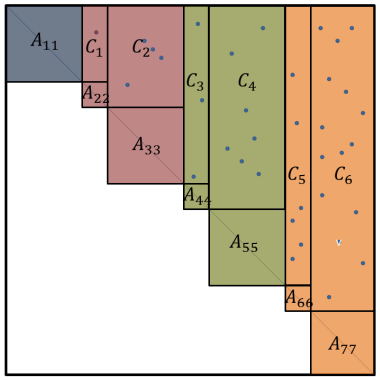
Due to this observation, *Basker* uses a variety of structuring or reordering methods, such as BTF and ND , to derive a hierarchy of two-dimensional sparse blocks. This structuring allows *Basker* to fit the irregular nonzero pattern into a hierarchy of blocks that fit the memory structure of many-core nodes. Additionally, this structure breaks columns into multiple submatrices allowing for multiple threads to work on a column that would have been serial in a nonsupernodal method, or efficiently use multiple calls of serial BLAS. Here, we will only focus on two levels of structures, i.e., structure from BTF and ND . BTF is the first coarse structure and ND is the second fine structure. However, additional levels of structures could also be used, such as an additional level of supernodes inside of the ND structure when two-dimensional blocks become dense. On top of this structure, `BASKER_FACTOR` (Alg. 1) is called. Additionally, the fine structure of ND is used to provide a parallel Gilbert-Peierls algorithm in `NFACT_LARGEBLK`.

A. Block Triangular Structure, Level 1

The first level of structure used by *Basker* is the block triangular form (BTF) of the coefficient matrix. This form is commonly used when dealing with unsymmetric coefficient matrices, particularly those from circuit and powergrid simulations. *Basker* first permutes the coefficient matrix based on an ordering found from MWCM (P_{m1}) in order to ensure that the system does not become structurally singular. Next, the strongly connected components are found, and coefficient matrix is reordered (P_c) so that each strongly connected component becomes a block on the diagonal. The reordered coefficient matrix, i.e., $P_c P_{m1} A P_c$, produces a structure similar to that in Figure 1(a). The form in Figure 1(a) is analogous to the fine decomposition of square center block found in Dulmage-Mandelsohn decomposition [10]. We note that if the matrix is structurally singular, than other blocks from the coarse decomposition of DM will exist, but will not for *Basker* if the matrix is structurally nonsingular. Additionally, any of the three diagonal blocks may not exist if it is structurally nonsingular. For example, coefficient matrices coming from many solid mechanic codes using finite elements will only have the center block.



(a) Level 1 of hierarchy structure, BTF ($P_c P_m A P_c^T$). The first level allows *Basker* to both reduce factorization work by only factoring the diagonal blocks, and exposes that blocks within $D_1, D_2, and D_3$ can be factorized independently.



(b) Representation of SMALLBLK layout, i.e., D_1 and D_3. The SMALLBLK have multiple small subblocks on the diagonal that can be factored independently. The off-diagonal subblocks do not need factored. Each thread takes a group of blocks based on estimated float-point operations. The coloring of the blocks suggest one possible mapping of thread and blocks.

Fig. 1. Level 1 coarse (BTF) hierarchy structure and fine Level 2 structure within D_1 and D_3. Top level structure used in Algorithm 1 `BASKER_NFACTOR`.

In Figure 1(a), a two-dimensional structure exists with three diagonal blocks exist, namely D_1, D_2, and D_3, along with upper off-diagonal blocks C_{12}, C_{13}, and C_{23}. Each block has a sparse nonzero pattern. The blocks D_1 and D_3 consists of multiple tiny subblocks on the diagonal, and the block D_2 consists of one or more large subblocks. These multiple tiny subblocks in D_1 and D_3 provide enough parallelism by using their derived ordering from BTF as their second level structure as well. *Basker* uses `SMALLBLK` methods for these structures. In contrast, D_2 must be further structured, and will use `LARGEBLK` methods depending on this second level. An overview of the complete *Basker* algorithm using these methods is listed in Algorithm 1

SMALLBLK. The substructure of D_1 and D_3 are easily dealt with as subblocks are independent of each other. Therefore, the sparsity pattern of each subblock and factorization can be computed concurrently. A typical representation of this block is given in Figure 1(b), where diagonal submatrices

Algorithm 1 BASKER_FACTOR

1: Based on level 1 structure, BTF
2: **CALL** SYMBFACT_SMALL on D_1 and D_3
3: **CALL** SYMBFACT_LARGE on D_2
4: **CALL** NFACT_SMALL on D_1 and D_3
5: **CALL** NFACT_LARGE on D_2

A_{ii} needs to be factored. A two-dimensional sparse block structure is used here as well. This structure allows for each diagonal submatrix to be handled independently. The off-diagonal blocks are stored so they can be easily used with sparse matrix-vector multiplication when solving for a given right-hand side vector. We note that these off-diagonal submatrices could further be split, however they tend to be very sparse as they are in the original nonzero pattern and further blocking and using multiple threads will be typically inefficient.

Basker first performs a symbolic factorization of this block as in Algorithm 2. This symbolic factorization first reorders each diagonal submatrix using AMD . Next, symbolic factorization finds the number of nonzeros of each column and an estimates the number of float-point operations required to factor. From the number of float-point operations, *Basker* partitions the submatrices among the threads and memory for LU is allocated. The coloring in Figure 1(b) provides one division of how four threads may be divided among the blocks.

Algorithm 2 SYMBFACT_SMALLBLK

1: **for all** subblocks on diagonal (A_{ii}) **do**
2: **CALL** AMD on $A_{ii} \rightarrow P_{amd}$
3: **CALL** COL_COUNT on $P_{amd}A_{ii} \rightarrow CC[i][], OP[i][]$
4: **end for**
5: //Based on estimated operation in $OP[][]$ divide subblocks equally among p threads.
6: **for all** p threads **do**
7: Initialize LU structure based on $CC[][]$
8: **end for**

NFACT_SMALLBLK. After the symbolic factorization, the numeric factorization simply uses the same thread mapping to submatrices to call sparse LU factorization using Gilbert-Peierls algorithm . No algorithm block is given, since it is a simple parallel for loop over the diagonal submatrices.

B. Nested-Dissection Structure, Level 2

The block D_2 in Figure 1(a) contains one or more of the largest strongly-connected components in the coefficient matrix. This whole block must be factored, i.e., $D_2 = L_{D_2}U_{D_2}$, in order to solve for a given right-hand side vector. Additionally, this block dominates factorization time, but provides no direct way with natural ordering to factor with multiple threads. In KLU , this block would simply be ordered with AMD and factored by one thread serially after the blocks before it. We note that this block constitutes an average of 68.4% of the total matrix size in our circuit problem test suite (see Section V). Therefore, factoring this block in a single thread does not provide enough parallelism to achieve acceptable speedup; *Basker* devises a nested method using Gilbert-Peierls algorithm to solve this problem.

We use nested-dissection ordering on D_2 in order to discover smaller independent subblocks to factor in parallel. *Basker* first permutes D_2 using a MWCM (P_{m2}) to find the locally best matching and reduce the need to pivot. Next, *Basker* compute ND ordering on the graph of D_2+D_2^T with a separator tree that contains p leaves, where p is the number of threads available. The resulting ND ordering (P_{nd}) if applied resulting in $P_{nd}P_{m2}D_2P_{nd}^T$. The structure of the reordered coefficient matrix is shown in Figure 2(a) for four threads. This two-dimensional structure of sparse matrices is used to store both the reordered matrix and factorization. The colors suggest one possible layout where blocks of a particular color are shared by a thread.

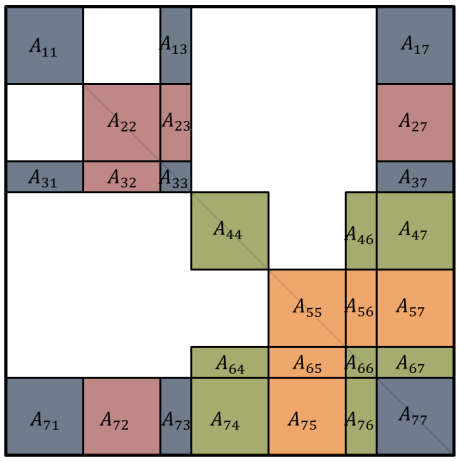
Figure 2(b) provides a modified separator tree of the nested-dissection ordering. This tree is modified by providing one extra level that allows us to write the dependencies of the two-dimensional structure more efficiently. Dependencies of submatrices within a treenode are based on line position from top to bottom, i.e., matrices on line two require matrices found in line one. Dependencies between treenodes are presented as arrows. The modified tree may also be used to statically assign threads, and one such assignment is indicated by the node colors. Additionally, a constrained fill reduced ordering, such as constrained AMD (P_{amd}), is used where the constraints are based on levels in the tree.

LARGEBLK. After D_2 has been moved into the two-dimensional structure, a symbolic analysis is used to estimate the number of nonzeros in the L and U corresponding to each block. This is done by following the ND tree-structure from the leaf nodes to the root. The general idea of the algorithm is to first find the structure of the diagonal submatrix in the leaf node in Figure 2(b) using an $etree$ (Lines 3-5 in Alg 3). Next, the algorithm finds nonzero counts of off-diagonal submatrices by walking paths in the $etree$ (Lines 8-29 in Alg 3).

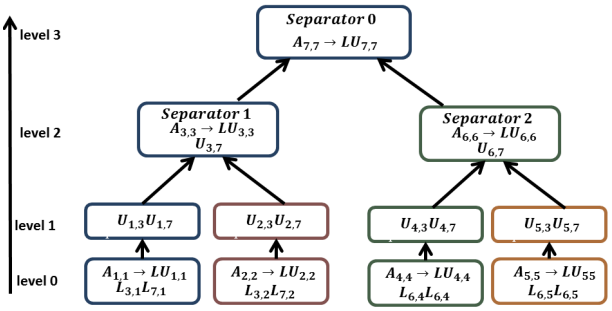
In more detail, *Basker* performs symbolic factorization on the leaf nodes, i.e., level 0 in Figure 2(b), by forming $etree(A_{ii})$, $etree(A_{ii}$+$A_{ii}^T)$, or $etree(A_{ii}A_{ii}^T)$ depending on the level of symmetry and pivoting needed. COL_COUNT in lines 3-5 of Algorithm 3 provides the nonzero count for each submatrix A_{ii} by simple transversal of the $etree$ [15].

Basker now moves up the tree with the loop found on line 8 in Alg. 3. The nonzero counts of U_{ij} in the levels are found with a call to COL_COUNT_UPPER on line 13. For $U_{ij}(k)$ where k is the column, the nonzero count is found by walking the $etree$ of A_{ii} starting at points corresponding to nonzero entries in column k. We note that all of the $etree$ need not be walked as already explored paths are marked to limit computation. Additionally, we keep track of the beginning of paths in the $etree$ fo the submatrix. This bookkeeping allows us to estimate fill-in in parent blocks.

After computing the nonzero estimates of U_{ij}, the nonzero count estimate is found for L_{ji} on line 14 with a call to COL_COUNT_LOWER. Again, the $etree$ of A_{ii} is used, but in a slightly different manner. For each column k in the submatrix,



(a) Matrix in nested-dissection ordering, $P_{amd}P_{nd}P_{m2}D_2P_{nd}^TP_{amd}^T$, of D_2. Each submatrix is stored by Basker as a sparse matrix. One possible thread layout indicated by color.



(b) Modified nested-dissection tree. Submatrix dependencies between tree nodes mark with arrows. Dependencies within nodes based on line.

Fig. 2. Level 2 fine (ND) structure used to extract fine-grain parallelism from a single large connected-component. This structure is used within BASKER_NFACTOR by NFACT_LARGEBLK to perform a parallel Gilbert-Peierls algorithm .

the $etree$ is traversed backwards to determine which columns may contribute to the nonzero structure (Str) of $L_{ji}(k)$. More precisely, this is written as: $Str(L_{ji}(k)) = Str(A_{ji}(k)) \cup_v Str(L_{ji}(v))$, where v are columns corresponding to nodes preceding the $etree$ node of k. Furthermore, bookkeeping of the starting and ending of paths in the submatrix is stored.

Next, *Basker* has to find nonzero counts for more off-diagonal submatrices if level $< log_2(p)$ where p is the number of threads. If the nonzeros of more off-diagonal submatrices are needed, then the loop on line 16 is executed. If not, the loop is skipped over. Either way, a reduction of the paths found while finding nonzero count of in lines 13 and 14 need to be found. REDUCE_PATH on lines 19 and 25 matches the stored starting path of U_{ij} with a path stored in the exploration of L_{ji}. If in the inner loop, then COL_COUNT_UPPER and COL_COUNT_LOWER is called. Else, COL_COUNT_SEP is used to estimate the nonzero in the diagonal separator submatrix by using the reduction path and adding the nonzero structure of the submatrix. . We additionally note that for many submatrices on the diagonal corresponding to separators can be assumed to be dense with very little memory impact.

Algorithm 3 SYMBFACT_LARGEBLK

```
1: //Find leaf node etree (Domains)
2: for all p threads do
3:     Map p → i where i is a leaf node
4:     CALL COL_COUNT → etree_i
5: end for
6: **ALLSYNC**
7: //Get counts of all parent nodes (Separators)
8: for all level = 1 : log_2(p) do
9:     Map level → j
10:    //Factor Domain Us
11:    for all p threads do
12:        Map p → i where i is a leaf node
13:        CALL COL_COUNT_UPPER → upaths_i
14:        CALL COL_COUNT_LOWER → lpaths_i
15:        //Update Parent Separators In Same Column
16:        for all sublevel = 1 : level − 1 do
17:            Map sublevel → l
18:            **ALLSYNC**
19:            CALL REDUCE_PATH → paths_i
20:            CALL COL_COUNT_UPPER → upaths_i
21:            CALL COL_COUNT_LOWER → lpaths_i
22:        end for
23:        //Find nnz for submatrix A_jj
24:        **ALLSYNC**
25:        CALL REDUCE_PATH → paths_i
26:        CALL COL_COUNT_SEP
27:    end for
28: end for
```

NFACT_LARGEBLK. We now provide an overview of the numerical factorization algorithm. The numerical factorization algorithm has two main phases, i.e., factorization of blocks corresponding to leaf nodes and factorization of blocks corresponding to separator nodes. Leaf node matrices, i.e., block matrices at level 0 in Figure 2(b), are processed by factoring the submatrices in the node column-by-column reading from top to bottom, e.g., $L_{11}U_{11}$, L_{31}, then L_{61}. Each node is processed by a thread independently. Separator nodes matrices, i.e., level 1-3 in Figure 2(b), are computed by processing submatrices corresponding the separator node from level 1 up to the separator node. For example, at separator 1 *Basker* computes $U_{1,3}$ & $U_{2,3}$ in level 1, then $L_{33,}$ & $U_{3,3}$ in level 2. For separator 0, level 1 computes $U_{1,7}$, $U_{2,7}$, $U_{4,7}$, & $U_{5,7}$, level 2 computes $U_{3,7}$ & $U_{6,7}$, and finally level 3 computes $L_{7,7}U_{7,7}$.

We now provide a more an in-depth examination of Algorithm. 4. The leaf node matrices are handled first. This is done by each thread first using Gilbert-Peierls algorithm to factor A_{ii} with a call to GP_FULL_BLK (Line 6). The lower diagonal submatrices corresponding to the leaf node diagonal submatrices are factored using the U_{ii} just found using a call to GP_LOWER_UPDATE (Line 7). This call translates to finding $L_{ji}(k) = A_{ji}(k) \setminus U_{ii}(k)$. This diagonal block factor and off-diagonal factors are done column-by-column to reuse the found U. After all the submatrices are factored in level 0, all threads are synced as indicated by the ALLSYNC. More details on all syncronization made in *Basker* in Section IV.

The factorization of separator node matrices are more complex. We provide Figure 3 to better illustrate the process. Figure 3 illustrates the computation of the first 4 threads when

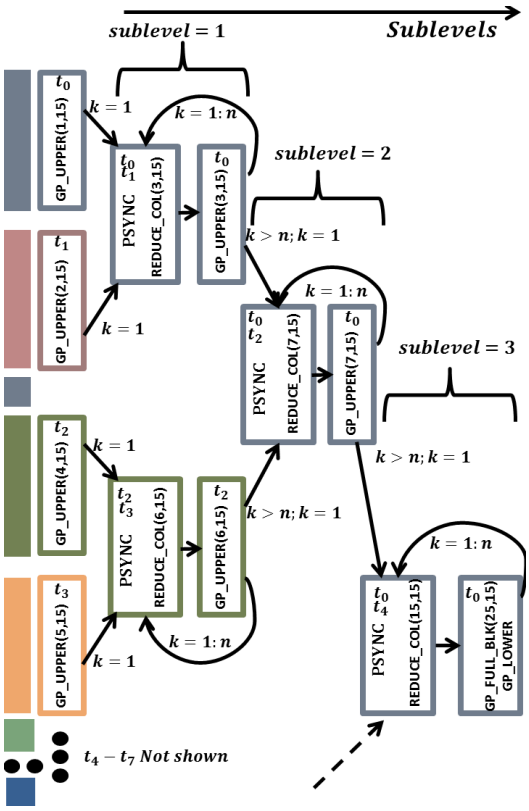


Fig. 3. Diagram corresponding to loop on lines 11-34 in Algorithm 4 when level = $log_2(8)$, showing the execution pattern of the first 4 threads (t_0-t_3). The blocks are colored to correspond to the primary thread executing the operation based on the assignment in Figure 2(a). Lines 16 maps to $sublelve = 0$, lines 18-24 map to $sublevel = 1, 2$, and lines 27-31 map to $sublevel = 3$.

level=$log_2(p)$ in Algorithm 4 where $p = 8$, i.e., the last column of block submatrices in Figure 2(a). Each thread (t_p) first factors U_{ij} where i corresponds the submatrix on the diagonal in level 0.

After, a reduction-factor method is used to computed the submatrices on the remaining levels, called sublevels, up to the root. A point-to-point synchronize, i.e., PSYNC, of threads is used to combine the solutions from a sublevel for the next sublevel. We will refer to this combining as reduction, i.e., `REDUCE_COL` (Line 21), and has the form: $\hat{A}_{l,j}(k) = A_{l,j}(k) - \sum_s L_{l,s}U_{s,L}(k)$ where L is the index based on level, l is the index based on sublevel, s is the index of all submatrices below l in the modified tree, and k is the column in the submatrix. Each thread computes their LU update and subtracts it from the column. For example, t_0 and t_1 synchronizes in sublevel 1 to form $\hat{A}_{3,15} = A_{3,15} - L_{3,1}U_{1,15} - L_{3,2}U_{2,15}$. Once formed, the updated matrix is factored using a call to `GP_UPPER` (Line 23), and repeated for all the columns in the block. The example in Figure 3 repeats this method for sublevel 1 and 2. Thread teams fit naturally to this type of reduction were threads have to work on shared portions of data.

Once all sublevels $<$ level are handled, the submatrix on the diagonal that requires full LU factorization, e.g., A_{33} in

Algorithm 4 `NFACT_LARGEBLK`

```
1: //Note, all function calls are made column-by-column to limit workspace
   and enhance reuse of U
2:
3: //Factor all leaf nodes (Domains)
4: for all p threads do
5:     Map p → i where i is a leaf node
6:     CALL GP_FULL_BLK → L_ii, U_ii
7:     CALL GP_LOWER_UPDATE → L_ki ∀k
8: end for
9: **ALLSYNC**
10: //Factor all parent nodes (Separators)
11: for all level = 1 : log2(p) do
12:     Map level → j
13:     //Factor Domain Us
14:     for all p threads do
15:         Map p → i where i is a leaf node
16:         CALL GP_UPPER → U_ij
17:         //Update Parent Separators In Same Column
18:         for all sublevel = 1 : level − 1 do
19:             Map sublevel → l
20:             **PSYNC**
21:             CALL REDUCE_COL → Â_lj
22:             **PSYNC**
23:             CALL GP_UPPER → U_lj
24:         end for
25:         //Factor Separator Node
26:         **PSYNC**
27:         CALL REDUCE_COL → Â_jj
28:         **PSYNC**
29:         CALL GP_FULL_BLK → L_jj, Ujj
30:         **PSYNC**
31:         CALL GP_LOWER_UPDATE → L_kj ∀k
32:     end for
33:     **ALLSYNC**
34: end for
```

Figure 2(a), can be updated based on the submatrices in that iteration same loop of level (Line 11). The forming of this submatrix is similar to that used for sublevels. A PSYNC is used and the reduction is formed. The difference is that now a full LU factorization is called on the block using a call to `GP_FULL_BLK` (Line 31). Off-diagonal L submatrices may need to be computed after the factorization, e.g., $L_{7,3}$ after $L_{3,3}U_{3,3}$ in Figure 2(a). This is the case when $level < log_2(p)$. When multiple L submatrices need to be computed with calls to `GP_LOWER_UPDATE` (Line 31), a PSYNC needs to be called so that multiple threads can compute these submatrices in parallel.

IV. Basker Implementation

In this section, we describe how *Basker* was implemented and decisions that impact its performance. Particularly, we focus the implementation of data layout, Kokkos, and barriers.

A. Two-Dimensional Layout

Basker uses a hierarchy of two-dimensional sparse matrix blocks to store both the original matrix and LU factors. The 2D structure is composed of multiple compressed sparse column (CSC) format matrices. Parallelism must be extracted from between blocks in the BTF structure and within large blocks in order to achieve speedup on low fill-in matrices. A one-dimensional sparse block layout would be sufficient in extracting parallelism between blocks in the BTF structure,

but is not enough within the large blocks. In particular, a hierarchical structure needs to be exploited to reveal more parallelism. Additionally, this also breaks the problem into fine-grain data structures that better fit the structure of modern many-core processor nodes. *Basker* implements this by building this structure during the symbolic factorization after applying the aforementioned orderings. Since the structure is built before numeric factorization, the algorithm determines a static assignment of threads to sparse submatrices.

B. Kokkos

Basker is implemented using Kokkos [7] package in Trilinos. Kokkos is a C++11 package that allows coding portable algorithms on different many-core devices. Using Kokkos, traditional arrays are replaced with views that both pad data to better fit cache lines and promote vectorization. Parallel regions are than executed using a data-parallel method, such as parallel for. Kokkos allows for teams of threads to be launched together to work on regions, and members of a team can synchronize using barriers.

C. Barriers

Light weight synchronizations are needed to allow multiple threads to work on a single column in *Basker* . There are multiple places where these synchronizations need to happen in *Basker* , and they are marked in Algorithm 4 either as ALLSYNC or PSYNC. ALLSYNC indicate that all threads need to synchronize and PSYNC indicate that only select threads need to synchronize. However, the number of threads that need to synchronize depends on location and iteration in the algorithm. For instance, all threads need to synchronize moving from factoring leaf nodes and parent nodes, but only two threads need to synchronize in columns related to first level separators.

A traditional data-parallel approach launches `parallel_for` over a set of threads, and these threads rejoin the master only after the end of the loop. However, if synchronization takes place between all threads at every level, the overhead would be too high. In particular, the total time spent for synchronization for factorizing matrix G2 Circuit with 8 cores is 11% of total. Thread teams can be used to synchronize a smaller set of threads that are launched together, and they are supported by both OpenMP and Kokkos. If thread teams are used to reduce synchronization overhead, the total time spent for synchronization for matrix G2 Circuit with 8 cores is 4.5% of total.

However, neither of these methods are ideal for *Basker* , because which threads that need to synchronize varies even within a column corresponding to a separator. For example, thread 0 and 1 need to sync for sublevel 1 and thread 0 and 2 need to sync for sublevel 2 in Figure 3. Therefore, *Basker* uses a different mechanism to synchronize between threads. This mechanism is a point-to-point synchronization that utilizes writing to a volatile where synchronization only happens between two threads that have a dependency. A detailed explanation of how point-to-point synchronization can speedup sparse triangular solve is outlined in work by Park et al. [16]. Using this method, *Basker* is able to reduce synchronization overhead to 2.3% of total for Matrix G2 Circuit, and is able to apply run on higher core counts.

V. Empirical Evaluation

We now evaluate *Basker* against the solver packages of Pardiso MKL and KLU on a set of spare matrices from circuit and powergrid simulations. We use the Pardiso package from MKL 11.2.2 (PMKL), and note that this not an exact copy of the standalone Pardiso package.

A. Experimental Setup

System Setup. Two systems are used for evaluation. The first system contains two eight-core Xeon E5-2670 running at 2.6GHz, and we denote this system as SandyBridge. The two processors are interconnected using Intel's QuickPath Interconnect (QPI), and share 24GB of DRAM. The second system is a Intel Xeon Phi ran in native mode with 61 cores running at 1.238GHz and 16GB of memory. All codes are compiled using Intel 15.2 with -03 optimization.

Test Suite. *Basker* is evaluated over a test suite of circuit and powergrid matrices taken from Xyce [] and the University of Florida Sparse Matrix Collection [17]. These matrices vary in size, sparsity pattern, and number of BTF blocks. Additionally, these matrices vary in fill-in density, i.e., $\frac{|L+U|}{|A|}$ where $|A|$ is the number of nonzeros in A. In Davis and Natarajan [5], coefficient matrices coming from circuit simulation generally have lower fill-in density than those coming from two an three dimensional problems, i.e., $\frac{|L+U|}{|A|} < 4.0$. Matrices with lower fill-in tend to perform better using a Gilbert-Peierls algorithm than a supernodal approach. For fairness, we include seven matrices with fill-in density larger than 4.0. Table I V-A provides a list of all matrices sorted by increasing fill-in density measured using KLU. The dimension of A is denote as n, the percent of matrix composed of small independent diagonal submatrices, i.e., `SMALLBLK`, is denoted by BTF%, and the total number of diagonal submatrices that exist in BTF form is denoted by BTF blocks. Matrices internal to Sandia/Xyce are marked with "*" and those from powergrids marked with "+". A double line divides matrices with fill-in density higher than 4.0. Additionally, the number of nonzeros in $L + U$ is included for KLU, PMKL, and *Basker* . The nonzeros reported for PMKL and Basker from using 8 cores on SandyBridge, and we note that this number varies for Basker depending on number of cores. We bold the entry for nonzero in L+U that is the smallest between PMKL and *Basker* .

B. Fill-In

We first consider the memory needed by *Basker* and PMKL. Though the number of processing threads are increasing on many-core systems, the size of memory unfortunately is not increasing at the same rate. Therefore, memory is very important on a shared-memory node. Here, we regard memory in terms of nonzero entries in the factored matrix. Table V-A provides the nonzero for the factored matrix for both *Basker* and

TABLE I

MATRIX TEST SUITE. n REPRESENTS DIMENSION OF MATRIX, $|.|$ IS THE NUMBER OF NONZEROS IN THE MATRIX. THE NUMBER OF NONZEROS IN THE FACTORIZATION OF A MATRIX IS BOLDED IF MINIMUM BETWEEN BASKER AND PMKL. * INDICATES SANDIA/XYCE MATRICES, + INDICATES POWERGRIDS.

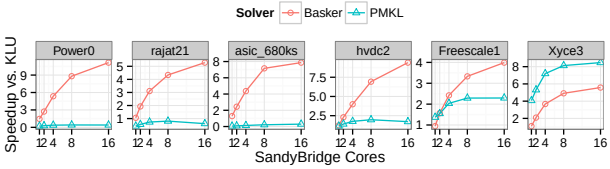
Matrix	n	$\|A\|$	KLU $\|L+U\|$	Pardiso $\|L+U\|$	Basker $\|L+U\|$	BTF %	BTF blocks	KLU $\frac{\|L+U\|}{\|A\|}$
RS_b39c30+	6.0E4	1.1E6	6.9E5	6.3E6	**6.9E5**	100	3E3	0.6
RS_b678c2+	3.6E4	8.8E6	5.8E6	5.9E7	**5.8E6**	100	271	0.7
Power0*+	9.8E4	4.8E5	6.4E5	9.1E5	**6.4E5**	100	7.7E3	1.3
Circuit5M	5.6E6	6.0E7	6.8E7	3.1E8	**7.4E7**	0	1	1.3
memplus	1.2E4	9.9E4	1.4E5	**1.3E5**	1.4E5	0.1	23	1.4
rajat21	4.1E5	1.9E6	2.8E6	4.9E6	**2.8E6**	2	5.9E3	1.5
trans5	1.2E5	7.5E5	1.2E6	1.3E6	**1.2E6**	0	1	1.6
circuit_4	8.0E4	3.1E5	5.0E5	5.8E5	**5.1E5**	34.8	2.8E4	1.6
Xyce0*	6.8E5	3.9E6	4.7E6	3.8E7	**4.8E6**	85	5.8E5	1.8
Xyce4*	6.2E6	7.3E7	4.5E7	5.0E7	**4.5E7**	12	7.5E5	2.0
Xyce1*	4.3E5	2.4E6	5.1E6	5.6E6	**5.1E6**	21	9.9E4	2.4
asic_680ks	6.8E5	1.7E6	4.5E6	2.9E7	**4.5E6**	86	5.8E5	2.6
bcircuit	6.9E4	3.8E5	1.1E6	1.1E6	**1.1E6**	0	1	2.8
scircuit	1.7E5	9.6E5	2.7E6	2.7E6	**2.7E6**	0.3	48	2.8
hvdc2+	1.9E5	1.3E6	3.8E6	**3.0E6**	3.8E6	100	67	2.8
Freescale1	3.4E6	1.7E7	7.1E7	**5.6E7**	6.8E7	0	1	4.1
hcircuit	1.1E5	5.1E5	7.3E5	**6.7E5**	7.1E5	13	1.4E3	6.9
Xyce3*	1.9E6	9.5E6	7.6E7	**4.3E7**	7.7E7	20	4.0E5	9.2
memchip	2.7E6	1.3E7	1.3E8	**6.5E7**	9.4E7	0	1	9.9
G2_Circuit	1.5E5	7.3E5	2.0E7	**1.3E7**	2.0E7	0	1	27.7
twotone	1.2E5	1.2E6	4.8E7	**2.7E7**	4.7E7	0	5	39.9
onetone1	3.6E4	3.4E5	1.4E7	**4.3E6**	1.2E7	1.1	203	40.8

MKL. The solver using the fewest nonzeros is bolded. We observe that for matrices with fill-in density matrices $<$ .4 *Basker* provides factors that have less nonzero entries. This reduce can be as high as an order of magnitude for the matrix RS_b678c2+, and is the result of using a structure that avoids unnecessary factoring and uses fill reduced ordering on subblocks. However, PMKL uses slightly less memory on matrix with fill-in density $>$ .4. than *Basker* . The additional memory used by *Basker* is far less than the additional used by PMKL on the first group of matrices. One reason for additional nonzero entries in *Basker* is the restrictions placed on constrained AMD in order to allow pivoting within a local submatrix.

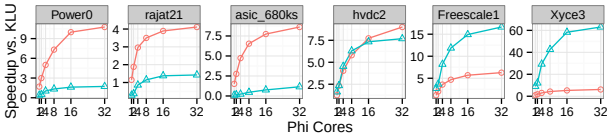
C. Performance

We now examine the speedup of the numeric factorization phase of *Basker* and PMK. Only the numeric factorization phase is consider, since the symbolic factorization of both *Basker* and PMKL is limited by finding the ND ordering. Finding such solvers an ordering in parallel in shared-memory is an expensive problem that does not scale well. In order to better compare, we use the relative speedup to KLU, i.e., $Speedup(matrix, solver, p) = \frac{Time(matrix,KLU,1)}{Time(matrix,solver,p)}$, where $Time$ is the time of the numeric factorization phase, $matrix$ is the input matrix, $solver$ is either *Basker* or PMKL, and p is the number of cores.

In Figure 4(a), the speedup of six matrices on SandyBridge is given. These six matrices where selected due to their fill-in density, and are ordered from least to greatest fill-in density. In these graphs, we observe that *Basker* can achieve up to $11.15\times$



(a) SandyBridge speedup comparison of Basker and PMKL relative to KLU.



(b) Phi speedup comparison of Basker and PMKL relative to KLU.

Fig. 4. Speedup of Basker and PMKL relative to KLU. The

speedup and outperform PMKL in all but one case, i.e., Xyce3 with a high fill-density of 9.2. Moreover, we observe that PMKL has a value less than 1 in serial for four problem demonstrating the inefficiency of a supernodal algorithm to a Gilbert-Peierls algorithm for matrices with low fill-in density. By adding more cores, PMKL is not able to recover from this inefficiency and reach a max speedup of $2.34\times$ on the first four problems. The poor performance of PMKL is due to a semi-dense columns that that *Basker* is able to avoid factoring. However, PMKL does factor Xyce3 faster with its high fill-in density, but *Basker* is still able to have a similar speedup curve.

The relative speedup of the same six matrices on the Intel Xeon Phi are given in Figure 4(b). Here, *Basker* is able to out perform PMKL on four out of the six matrix. *Basker* now achieves a $10.76\times$ max speedup on these six matrices and PMKL achieves $63\times$. We first observe that any overhead from using a Gilbert-Peierls algorithm algorithm on a matrix with high fill-in density is magnified by the Intel Phi. This magnification is exposed and seen in both Freescale1 and Xyce3. One possible reason for this is that the submatrices in the lowest level of the hierarchical structure are too large to fit into a cores's personal L2 cache, i.e., $512KB$. *Basker* currently only has an option to make the submatrices as large as possible to allow for more pivoting. However, *Basker* still achieves speedups higher than PMKL on the four matrices with low fill-in density.

Now we compare the performance on the whole test suite. On SandyBridge, the geometric mean of speedup with *Basker* is $5.91\times$ and with PMKL is $1.5\times$ using 16 cores. On 16 cores, *Basker* is faster than PMKL on 17 matrices, and all the ones PMKL is faster have a high fill-in density. On Phi, the geometric mean speedup with *Basker* is $7.4\times$ and with PMKL is $5.78\times$ using 32 cores. On 32 cores, *Basker* is faster than MKL on 16 matrices. This include the same matrices as on the Sandybridge plus now Freescale1. The reason for such a high speedup for PMKL on Phi is again its higher performance on high fill-in density matrices. PMKL range of speedups on

low density matrices is only slightly better, i.e., a maximum increase of $3.12\times$ for on particular matrix (hvdc2).

Now, we consider a performance profile to gain an understanding of the performance over the test suite. The performance profile measures the relative time of a solver on a given matrix to the best solver. After, these values are plotted for all matrices in a graph with an x-axis of time relative to best time and a y-axis of fraction of matrices. The result is a figure where a point(x,y) is plotted if a solver takes no more than x times the runtime of of the fastest solver for y problems.

We first provide the performance profile of *Basker* , PMKL, KLU in serial on SandyBridge in Figure 5(a). This profile provides a baseline of how well each method is without threads. We observe that *Basker* is better on over 77% of the problems, while the supernodal method of PMKL is within $5\times$ of the the best solver, i.e, *Basker* , for 77%. However, PMKL is only better than under 34% of the problems. Despite have very similar algorithms, *Basker* is able to slightly beat KLU. This slight difference may be caused by Kokkos memory padding, and by slightly different AMD calls being used.

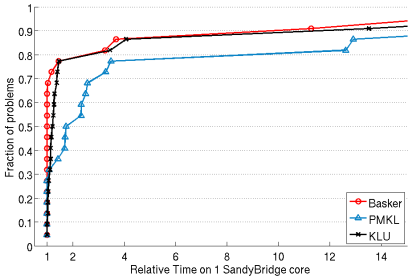
The performance profile for SandyBridge with 16 cores is given in Figure 5(b). KLU is now not include as it is serial. We first observe that *Basker* is the best solver for 80% of the matrices, and PMKL is within $15\times$ slower than *Basker* on 80% of the matrices. PMKL is only the best solver for under 30% of the matrices, which correspond to matrices with high fill-in density. This demonstrates that a supernodal method on SandyBridge does not scale well on low fill-in density matrices.

On IPhi with 32 cores, the performance profile is slightly different, and given in Figure 5(c). *Basker* now is the best solver for 70% matrices, and PMKL is within $6\times$ of *Basker* for 70% of matrices. The point that *Basker* and PMKL performance cross is at about 80%, and these 20% are matrices with high fill-in. However, PMKL is now able to scale better on low fill-in density matrices that have very small dense block. Additionally, *Basker* now scales less well on high fill-in density matrices. A reason for not scaling as well is not having a large shared $L3$ of the SandyBridge to share data needed during `REDUCE_COL`.

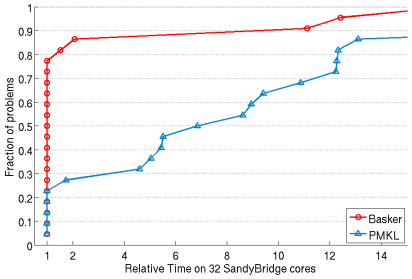
D. Comparison on ideal matrices

Next, we analyze how well *Basker* scales on low fill-in density matrices, compared to how well the supernodal solver PMKL scales on 2D and 3D mesh problems. This comparison allows us to better understand if *Basker* achieves speedup for its ideal input similar to PMKL on its ideal input. PMKL has been fined tuned to be a state-of-the-art supernodal solver, and to achieve a similar speedups would indicate our solver method is as well. We use a second test suite of matrices comprised for PMKL that come from 2D and 3D mesh problems in Table V-D. These performance of PMKL on these matrices will be compared to the performance of *Basker* on the six matrices of our primary test suite with the lowest fill-in density.

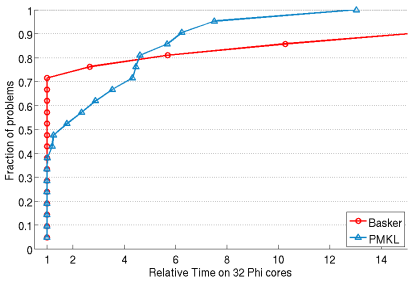
In Figure 6(a), we provide a scatter plot of the speedup for each solver relative to itself over its ideal six matrices.



(a) Performance Profile on 1 SandyBridge Core. *Basker* is the best solver for almost 70% of the matrices and PMKL is only the best solver for about 30%. Even in serial, PMKL is at least $4\times$ worse on 20% of the problems.



(b) Performance Profile on 16 SandyBridge Cores. *Basker* is the best solver for than almost 80% of the matrices, while PMKL is the best solver for only slightly more than 20%. Addtionall, PMKL is at least $8\times$ worse on 50% of the matrices.



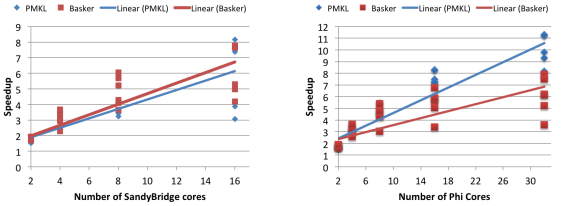
(c) Performance Profile on 32 Phi Cores. *Basker* is the best solver for over 70% of the matrices, while PMKL is the best solver slightly under 40% of the matrices.

Fig. 5. Performance Profiles of *Basker* and PMKL on Intel SandyBridge and Phi. The x-axis represents the relatve time compared to the best solver for a given matrix. The y-axis represents the percent of problems. A pont (x,y) represents that the fraction y of the test problem you are within x$\times$ of the best solver.

Additionally, an linear trend line is plotted through each set of solver speedups. We observe that *Basker* achieve a similar speedup trend as PMKL on SandyBridge. This demonstrates that on systems with a large cache hierarchy *Basker* is able to achieve so called state-of-the-art performance on low fill-in density matrices. In Figure 6(b), a similar plot is given for our Intel Phi system. This time *Basker* has a slightly lower trend line starting at 16 cores. This difference demonstrate that *Basker* does not scale as well as it does on the SandyBridge system. We suspect several reasons for this difference and we discussion future plans for improvement in Section VI.

TABLE II
2/3D MESH PROBLEMS TO TEST BEST PERFORMANCE OF MKL TO COMPARE AGAINST BEST PERFORMANCE OF *Basker* .

Matrix	n	$\|A\|$	$\|L+U\|$	Description
pwtk	2.2E5	1.2E7	9.7E7	Wind tunnel stiffness matrix
ecology	1.0E6	5.0E6	7.1E7	5 pt stencil model movement
apache2	7.2E5	4.8E6	2.8E8	finite difference 3D
bmwcra1	1.5E5	1.1E7	1.4E8	stiffness matrix
parabolic_fem	5.3E5	3.7E6	5.2E7	Parabolic finite element
helm2d03	3.9E5	2.7E6	3.7E7	Helmholtz on square



(a) SandyBridge, Basker and PMKL speedups on ideal inputs. Basker is able to achieve a similar speedup curve as PMKL on 2/3D mesh problems.

(b) Phi, Basker and PMKL speedup on ideal inputs. Basker has a similar plot up to 16 cores as PMKL. Fine-grain access cause imblance at 32 cores.

Fig. 6. *Basker* and PMKL with on 6 ideal input matrices. In order to understand the performance of *Basker* , and if solver for low fill-in density matrices could ever achieve the performance of a supernodal solver on 2/3D mesh problems using multithreaded BLAS.

E. Xyce

Next, we consider the use of *Basker* on a sequence of matrices generated during the transient analysis of a circuit. Xyce is a transistor-level simulator that performs a SPICE-style simulation of circuits, where devices and their interconnectivity are transformed via modified nodal analysis (MNA) into a set of nonlinear differential algebraic equations (DAEs). During transient analysis, these nonlinear DAEs are solved implicitly through numerical integration methods. Any numerical integration method requires the solution to a sequence of nonlinear equations, which in-turn generates a sequence of linear systems. A transient analysis can generate millions of coefficient matrices with the same structure and significantly different values. Each factorization may require a different permuation due to pivoting for this reason. For very large circuits, this results in the numeric factorization being the limiting factor of the simulations overall time and scalability. Furthermore, a solver package must reuse the symbolic factorization for all matrices in the sequence as repeating symbolic factorization would dramatically affect performance.

For this experiment, we chose a sequence from the circuit that generated Xyce1. This circuit is of particular interest because it has been used in prior studies [18] to illustrate the ineffectiveness of preconditioned iterative methods and direct solvers other than KLU. In practice, KLU is the direct solver that has been used to perform the transient simulation of this circuit, as it was the fastest direct solver that would enable the simulation to complete. Attempts to use the PMKL solver had either been met with solver failure or simulation failure until recently. Therefore, we wish to see how well *Basker* performs on a sequence of these matrices (1000 matrices) which represent 10% of the desired transient length.

Over the sequence of 1000 matrices, *Basker* took 175.21 seconds, KLU took 914.77 seconds, and PMKL took 951.34 seconds. This is a speedup of $5.43\times$ when using *Basker* instead of PMKL and $5.22\times$ when using *Basker* instead of KLU. The scalable simulation of this circuit was previously limited by the serial bottleneck produced by using KLU as the direct solver, which is justified due to its performance compared to PMKL. *Basker* provides significant speedup compared to either KLU or PMKL, and will finally provide a scalable direct solver to Xyce for performing the transient analysis of this circuit.

VI. FUTURE WORK

First, we wish to allow for additional levels of hierarchical parallelism. This can be done by first introducing supernodes as a level of hierarchy within sparse blocks with high fill-in density. For example, the matrix memchip has a number of blocks with both high and low fill-in density. It would be easy to think of supernodes within the high fill-in density blocks as one more level of parallelism that could be exploited.

Second, we want to improve scalability across platforms. Therefore, we want our package to scale well on a number of platforms, such as future Intel Phi and IBM Power series. This requires several changes. The first is to modify the data structure in the lowest level of the hierarchy structure to explicitly fit the cache structure. The second change will be adopting an asynchronous tasking system. Asynchronous tasking would allow jobs to be launched and synchronize upon return, and would allow for possible out-of-order jobs when their exist imbalance.

VII. CONCLUSIONS

We introduced a new multithreaded sparse LU factorization, *Basker* , that uses hierarchical parallelism and data layouts. *Basker* provides a valuable alternative to traditional solver that use one-dimensional layout with BLAS. In particular, this is valuable to coefficient matrices with hierarchies of structure that can be exploited. Within this structure, we provide the first parallel implementation of Gilbert-Peierls algorithm to allow multiple threads to work on a single column. Performance results show that *Basker* is able to scale well for matrices with low fill-in density resulting in a speedup of $5.91\times$ (geometric mean) over the test suite on 16 SandyBridge cores and 7.5 over the test suit on 32 Intel Phi cores relative to KLU. Particularly, *Basker* can have speedups on these matrices similar to a supernodal solver on 2D and 3D problems, while reducing the time spent solving a sequence of circuit problems from Xyce by $5.43\times$ for 16 SandyBridge cores. *Basker* shows that in order to speedup sparse factorization on many-core node, solvers must leverage all available parallelism and may do so by using a hierarchy structure.

ACKNOWLEDGMENT

We would like to thank Erik Boman, Andrew Bradley, and Kyungjoo Kim for algorithm insight and being a sounding board. Also, like to H.C. Edwards, Christian Trott, and Simmon Hammond for help with Kokkos on manycore systems. 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.

REFERENCES

[1] J. W. Demmel, S. C. Eisenstat, J. R. Gilbert, X. S. Li, and J. W. H. Liu, "A supernodal approach to sparse partial pivoting," *SIAM J. Matrix Anal. Appl.*, vol. 20, no. 3, pp. 720–755, May 1999.

[2] P. R. Amestoy, I. S. Duff, J.-Y. LExcellent, and J. Koster, "Mumps: a general purpose distributed memory sparse solver," in *Applied Parallel Computing. New Paradigms for HPC in Industry and Academia*. Springer, 2001, pp. 121–130.

[3] L. W. Nagel, "Spice 2, a computer program to simulate semiconductor circuits," Tech. Rep. Memorandum ERL-M250, 1975.

[4] O. Schenk, K. Gärtner, W. Fichtner, and A. Stricker, "Pardiso: A high-performance serial and parallel sparse linear solver in semiconductor device simulation," *Future Gener. Comput. Syst.*, vol. 18, no. 1, pp. 69–78, Sep. 2001.

[5] T. A. Davis and E. Palamadai Natarajan, "Algorithm 907: Klu, a direct sparse solver for circuit simulation problems," *ACM Trans. Math. Softw.*, vol. 37, no. 3, pp. 36:1–36:17, Sep. 2010.

[6] A. George, "Nested dissection of a regular finite element mesh," *SIAM J. Numerical Analysis*, vol. 10, no. 2, pp. 24–45, April 1973.

[7] H. C. Edwards, C. R. Trott, and D. Sunderland, "Kokkos: Enabling manycore performance portability through polymorphic memory access patterns," *Journal of Parallel and Distributed Computing*, vol. 74, no. 12, pp. 3202 – 3216, 2014, domain-Specific Languages and High-Level Frameworks for High-Performance Computing.

[8] P. R. Amestoy, T. A. Davis, and I. S. Duff, "An approximate minimum degree ordering algorithm," *SIAM J. Matrix Anal. Appl.*, vol. 17, no. 4, pp. 886–905, Oct. 1996.

[9] H. M. Markowitz, "The elimination form of the inverse and its application to linear programming," *Management Science*, vol. 3, no. 3, pp. pp. 255–269, 1957.

[10] A. Pothen and C.-J. Fan, "Computing the block triangular form of a sparse matrix," *ACM Trans. Math. Softw.*, vol. 16, no. 4, pp. 303–324, Dec. 1990.

[11] X. S. Li and J. W. Demmel, "Superlu dist: A scalable distributed-memory sparse direct solver for unsymmetric linear systems," *ACM Trans. Math. Softw.*, vol. 29, no. 2, pp. 110–140, Jun. 2003.

[12] I. S. Duff and J. Koster, "On algorithms for permuting large entries to the diagonal of a sparse matrix," *SIAM J. Matrix Anal. Appl.*, vol. 22, no. 4, pp. 973–996, Jul. 2000.

[13] J. W. Demmel, J. R. Gilbert, and X. S. Li, "An asynchronous parallel supernodal algorithm for sparse gaussian elimination," *SIAM J. Matrix Anal. Appl.*, vol. 20, no. 4, pp. 915–952, Jul. 1999.

[14] S. Rajamanickam, E. Boman, and M. Heroux, "Shylu: A hybrid-hybrid solver for multicore platforms," in *Parallel Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International*, 2012, pp. 631–643.

[15] T. A. Davis, *Direct Methods for Sparse Linear Systems (Fundamentals of Algorithms 2)*. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 2006.

[16] 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*, ser. ISC 2014. New York, NY, USA: Springer-Verlag New York, Inc., 2014, pp. 124–140.

[17] T. A. Davis and Y. Hu, "The university of florida sparse matrix collection." [Online]. Available: http://www.cise.ufl.edu/research/sparse/matrices

[18] H. K. Thornquist, E. R. Keiter, R. J. Hoekstra, D. M. Day, and E. G. Boman, "A parallel preconditioning strategy for efficient transistor-level circuit simulation," in *ICCAD '09: Proceedings of the 2009 International Conference on Computer-Aided Design*. New York, NY, USA: ACM, 2009, pp. 410–417.