

# Javelin: A Scalable Implementation for Sparse Incomplete LU Factorization

Joshua Dennis Booth  
*Department of Computer Science*  
*Franklin & Marshall College*  
*Lancaster, USA*  
 joshua.booth@fandm.edu

Gregory Bolet  
*Department of Computer Science*  
*Franklin & Marshall College*  
*Lancaster, USA*  
 gbolet@fandm.edu

**Abstract**—In this work, we present a new scalable incomplete LU factorization framework called Javelin to be used as a preconditioner for solving sparse linear systems with iterative methods. Javelin allows for improved parallel factorization on shared-memory many-core systems by packaging the coefficient matrix into a format that allows for high performance sparse matrix-vector multiplication and sparse triangular solves with minimal overheads. The framework achieves these goals by using a collection of traditional permutations, point-to-point thread synchronizations, tasking, and segmented prefix scans in a conventional compressed sparse row format. Moreover, this framework stresses the importance of co-designing dependent tasks, such as sparse factorization and triangular solves, on highly-threaded architectures. Using these changes, traditional fill-in and drop tolerance methods can be used, while still being able to have observed speedups of up to  $\sim 42\times$  on 68 Intel Knights Landing cores and  $\sim 12\times$  on 14 Intel Haswell cores.

## I. INTRODUCTION

The solution to sparse linear systems of equations dominates the execution time of most parallel scientific simulations. A common approach to solving such systems is using iterative methods, e.g., Conjugate Gradients (CG) or Generalized Minimal Residual Method (GMRES), that rely on sparse matrix-vector multiplication (`spmv`) and sparse triangular solves (`stri`). However, iterative methods may not converge or take a large number of steps depending on the conditioning of the system, and some form of preconditioning is normally used. Incomplete factorization is one common method used to precondition the system in order to improve the iterative method by factorizing the coefficient matrix representing the linear system of equations while controlling the number of nonzero elements calculated during the factorization. This incomplete factorization used as a preconditioner allows for the linear system to be modified to:  $Ax = b \rightarrow M^{-1}Ax = M^{-1}b$ . Because the precondition method tries to limit the number of calculations with  $M$ , the ratio of float-point operations required to construct  $M$  and solve  $M^{-1}b$  is memory-bound making incomplete factorization extremely hard to achieve strong-scaling. One common and useful preconditioner ( $M$ ) is based on the incomplete LU factorization. In this work, we provide a new scalable incomplete LU factorization (ILU) framework called Javelin that is designed for current large

many-core systems, and allows for scalable sparse triangular solves without having to reformat the input matrix like most scalable triangular solve methods [1].

A rich variety of incomplete LU factorization algorithms exist, such that  $A \approx M = LU$  where  $L$  and  $U$  are lower and upper triangular matrices, respectively. In general, these algorithms can be categorized by the method they use to reduce nonzeros due to fill-in: dropping based on numerical value (ILU( $\tau$ )) and dropping based on fill-in level (ILU( $k$ )). The design of Javelin allows for any combination of the two to be used (ILU( $k, \tau$ )) along with modified ILU [2]. However, most of the current packages that can be used for traditional incomplete factorization are serial because of the difficulty of scaling and the overhead that may exist to reformat the output matrix in order to achieve scalable sparse triangular solves. Though parallel ILU packages exist [3]–[6], many rely on complex data-structures or modifications of ILU by using approximations to reduce synchronizations, and may not have scalable `stri`. These packages apply these modifications due to the difficulty in achieving scalable performance with traditional ILU. Javelin is designed to have the robust nature of traditional ILU methods while limiting the use of complex data-structures and still achieving the scalability needed for incomplete factorization, `spmv`, and `stri` on current many-core systems such as Intel Xeon Phi by co-optimizing incomplete factorization and `stri`.

This paper presents Javelin as implemented currently as a threaded templated C++ incomplete LU factorization using OpenMP, and having the following contributions:

- A parallel ILU optimized for parallel `spmv` and `stri` on many-core architectures
- A parallel ILU that requires minimal data preprocessing and permutations
- An empirical evaluation of Javelin on Intel Xeon (Haswell) and Intel Xeon Phi (Knights Landing) for ILU(0) and `stri`
- An evaluation of optional parameters

## II. BACKGROUND

This section provides a brief overview of the foundations and improvements of incomplete LU factorization, i.e.,  $A \approx M = LU$ , and current scalable implementations

of sparse matrix-vector multiplication (`spmv`) and sparse triangular solve (`stri`). Here, `spmv` and `stri` methods are the driving force behind this work as we desire to have a scalable incomplete factorization that is optimized to the state-of-the-art parallel many-core `spmv` and `stri` as they dominate the time of iterative methods.

**Incomplete LU factorization.** Preconditioning based on incomplete LU decomposition is generally regarded as a robust “black-box” method for unstructured systems arising from a wide range of application areas. ILU has been studied extensively. This includes extensive studies on variations of  $ILU(k)$ ,  $ILU(\tau)$ , and multiple levels of these two together [2], [7]. The choice of which combination is highly dependent on the linear system. Variations exist that try to improve serial performance using supernodes [8]. However, unstructured systems with limited fill-in will have limited amount of overlapping sparsity patterns for supernodes.

**Parallel incomplete LU factorization.** The first work on parallel incomplete factorization was for regular structured systems. However, the need for parallel incomplete factorization on unstructured systems resulted in much work throughout the 1980s until today. The traditional approach in dealing with this irregular problem of unstructured systems is to use *level scheduling*. In level scheduling, the coefficient matrix is transformed into a graph, such that rows, columns, or blocks of the coefficient matrix represent vertices and dependencies are represented as edges in a graph. Groups of vertices in the graph that do not have any incoming edges are placed together into a level, and the outgoing edges from these vertices are removed. The rows, columns, or blocks that are in each level can be scheduled to be solved concurrently. In a similar fashion, orderings, such as Coloring and Nested-Dissection, have been used to identify rows, columns, or blocks that can be factorized concurrently while in some cases exposing more concurrency. The disadvantage to all of these reorderings is that they *may* negatively impact the number of iterations needed for the iterative method to converge [9]. We explore this possible negative impact with multiple reordering in section VII. However, the performance of parallel incomplete factorization depends on many more factors than the exposure of the levels of parallelism. These factors include programming model, task size, and data-structure related to the targeted machine. This was first noted in the early 1990s [10], and recently examined on current many-core systems with different factorization codes in [5]. Recently, one version of ILU has been shown to achieve very good performance on many-core and GPU systems [3]. Despite performance, this method may result in an incomplete factorization that is nondeterministic and that challenges traditional dropping or modified incomplete factorization due to race conditions.

**Sparse matrix-vector multiplication and triangular solve.** Iterative methods, such as CG and GMRES, spend the vast majority of their time performing the operations

of sparse matrix-vector multiplication (`spmv`) and sparse triangular solve (`stri`). In particular, preconditioned CG using incomplete Cholesky Decomposition, i.e.,  $M = LL^T$ , spends up to 70% of its execution time in forward and backward `stri` [11]. Because these operations dominate the solve time, having an incomplete decomposition that can allow us to make use of these operations efficiently is critically important. There has been a vast amount of research in this area over the past 10 years as large many-core systems are becoming the norm [1], [11]–[13]. Many of these methods require reordering and special data-structures [1], [13]. Therefore, these methods are not ideal for iterative methods as matrices would need to be copied over to these data-structures, and the number of iterations would depend on ordering. Moreover, these orderings may not be ideal for parallel incomplete factorization. Two works stand out that require us to limit reordering and data movement. The first work [13] introduces an efficient storage format for `spmv`, namely CSR5. CSR5 does not require a reordering of sparsity pattern, depends on the common Compressed Sparse Row (CSR) format, and allows for quick segmented scans. The only overhead needed is a little extra storage for tile information to help support segmented scan operations. Segmented scan has been shown to be an ideal way to implement `spmv` and `stri` on vector based machines [14], with most many-core systems having large vector register lanes. The second work [11], [12] focuses on improving `stri` on standard CSR format storage using general level scheduling and sparsifying synchronization. This work demonstrates that many of the overheads dealing with level scheduling could be removed, and allowing `stri` to scale. Javelin uses these works as the foundation to achieve scalable `stri`, with modification that allow incomplete factorization.

### III. A FRAMEWORK FOR ILU

The Javelin framework for parallel incomplete LU factorization depends on predetermining the sparsity pattern and applying an up-looking LU algorithm, i.e., a left-looking algorithm done by rows instead of columns, to the pattern. Both of these operations need to be done in parallel for the factorization code to scale on modern many-core systems. Determining the sparsity pattern in parallel has been studied in the following work [6], and a good implementation exists in [5], [15]. Therefore, we will primarily focus on the second half of implementing the up-looking LU algorithm to the given nonzero pattern. Fig. 1 provides an example of the up-looking LU algorithm using standard MATLAB notation for the sparse matrix  $S$  containing the sparsity pattern, which is used by many incomplete factorization codes. The main reason for the popularity of up-looking LU is because it can be easily modified to be used with dropping based on some tolerance to the pivot value and modified ILU techniques which help to reduce iteration counts [2]. Moreover, up-looking LU allows for local estimates of resilience from soft-

errors and the convergence rate. Like many other incomplete factorizations, Javelin does not allow for pivoting. These algorithms could be applied to other preconditioners and is why we call Javelin a framework.

**Require:**  $S$  contains the sparsity pattern for  $L$  and  $U$ , and  $L$  and  $U$  are stored in  $A$

```

1: for row = 1 to  $n$  do
2:    $s_{row} = S(row, :)$ 
3:   for col in  $s_{row}$  do
4:     if col < row then
5:        $a_{row,col} = a_{row,col} / a_{col,col}$ 
6:        $s_{update} = S(col, :)$ 
7:       for update_col in  $s_{update}$  do
8:         if update_col in  $S(row, :)$  and update_col > col then
9:            $a_{row,update\_col} -= a_{row,col} \times a_{col,update\_col}$ 
10:        end if
11:      end for
12:    end if
13:  end for
14: end for

```

Figure 1. Up-looking Sparse Incomplete LU Factorization Algorithm

In addition, Javelin uses level scheduling as the primary method to apply up-looking LU. Past work related to level scheduling of `stri` shows it is difficult to achieve high performance, with applications scaling to only a few cores. These difficulties were due to the requirement of thread barriers between levels and levels containing too few rows to keep all threads busy. To the best of our knowledge, this is the first time using level scheduling for up-looking LU using `stri` access pattern. Javelin faces these issues head-on by using point-to-point synchronizations between levels, and then using a second stage approach when the size of levels becomes too small for the thread count. Two different methods exist for factoring the second stage, namely Segmented-Row and Even-Row. We will explain both along with their trade offs, and demonstrate their performance differences in Javelin, though Javelin by default will make the choice for the user based on the matrix structure. In order to apply this two-stage structure, we first find the level scheduling order to either  $lower(A)$  or  $lower(A + A^T)$  depending on the user options, where  $lower$  is the sparsity pattern of the lower triangular half of the given matrix. We address the importance of the option between  $A$  and  $A + A^T$  in the subsection related to the second stage factorization method. Though finding the order alone would be enough to schedule the top stage point-to-point factorization method, we permute the nonzeros in the matrix into the level ordering while copying  $A$  into the CSR data-structure of  $L$  and  $U$  in parallel allowing for first-touch.

#### A. Level Scheduling, Upper Stage

During preprocessing, the sparsity pattern of  $A$  with desired fill-in is found and permuted into a level-based ordering as in Fig. 2. Javelin uses a set of user defined

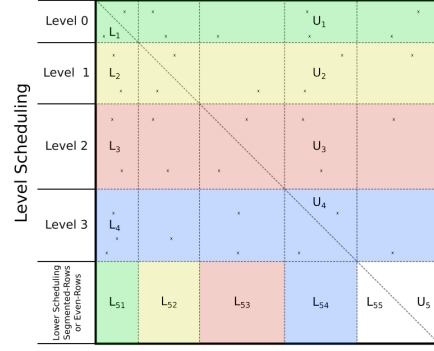


Figure 2. The general structure generated by level scheduling ordering. Levels in the top half can be factored in an up-looking method while point-to-point synchronizations are used to handle dependencies. The last level can be solved using one of two different methods.

options to determine the portion of the matrix to factor using level scheduling. These options include minimal number of rows per level, relative location of level in ordering, and row density, i.e., the number of nonzeros per row. The idea is that Javelin will want to apply a level scheduling to levels that have a very large number of rows so that no thread will run out of work and any imbalance in the amount of work would be amortized across multiple rows. However, if the rows become too dense compared to the relative average density of the matrix or the number of rows in the level is too few, these are moved towards the end to be solved with Javelin's second stage solver. The catch is when there exists a level with very few rows or high row density in the middle of larger level sets, see Fig. 3. In such cases, moving these rows to the second stage would result in moving a lot of work to the second stage (i.e., all the dependent rows). Traditional implementations of level scheduling would use barriers and would have difficulty because of such rows. However, Javelin using point-to-point synchronizations does not suffer in the same way.

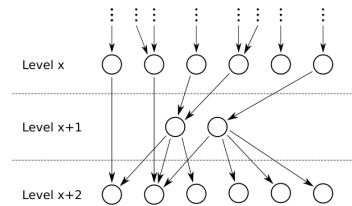


Figure 3. Here we have a large number of tasks in level  $x$  and  $x+2$ . However, between them is a level with few tasks. Therefore, this level may not be ideal to be solved by our second level method.

Point-to-point synchronizations for level scheduling was introduced as an effective way to parallelize sparse triangular solve [11], [12]. Here we use the observation that incomplete factorization done with up-looking LU has the

same dependency structure as sparse triangular solve. The principle of sparsifying point-to-point synchronizations for level scheduling is as follows. The level sets are found as in Fig. 4. Once the sets are found, rows are mapped to threads. This mapping of rows to threads induces an implied ordering, and the implied ordering is used to prune the full set of dependencies. In traditional methods, these dependencies are taken care of by a global barrier between levels or by generating tasks with dependencies in a parallel tasking programming model. Both of these methods have high overheads at runtime. However, point-to-point's implementation relies on inexpensive spinlocks and allows for certain threads to speed ahead of others.

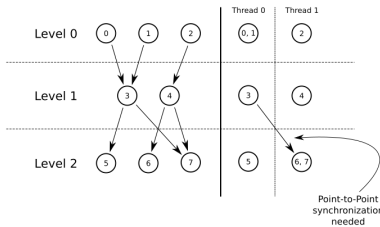


Figure 4. In the left side of the figure, the tasks and their dependencies that are generated by level scheduling. In the right side of the figure, tasks map to threads while imposing an implied ordering of tasks in a level. Using this order, dependencies are pruned.

### B. Lower Stage

At some point, level scheduling will no longer offer the degree of concurrency needed to achieve continued speedup on modern many-core systems. In these cases, Javelin will switch to one of two different methods. We name these methods: Segmented-Rows and Even-Rows.

**Segmented-Rows.** The Segmented-Rows (SR) method is inspired by the segmented scan that achieves cross-platform scalability of `spmv` in CSR5 [13]. The goal is to have a structure that can be factored efficiently while already being setup for a `spmv`-like update needed by `stri`. In preprocessing, the level ordering of  $\text{lower}(A + A^T)$  is found, and the rows identified to have SR applied to them are permuted to the end of the matrix during the copy-fill-in phase of building a CSR data-structure. Fig. 5 provides a visual example of this format, where the index of each nonzero element in a row is numbered starting from 1. Each subblock ( $L_{k,i}$ ) is defined by the level scheduling using in the upper stage. Inside each subblock, nonzeros are grouped together in a contiguous manner, as displayed in Fig. 5 by different colors. Given a subblock ( $L_{k,i}$ ) formed based on the level scheduling ordering, we note that there are no dependencies between the columns within the same subblock, i.e.,  $L_{k,i}(:, x)$  does not depend on  $L_{k,i}(:, y)$  where  $y > x$ . This fact is due to using the level ordering for  $\text{lower}(A + A^T)$  in place of  $\text{lower}(A)$ , which the latter does not guarantee the above observation. This observation

does allow for tiles to be created within each level of the subblock in a similar manner to CSR5 that only requires a small additional array of pointers into the block. These tiles build the foundation to the SR algorithm. A tasking model is used to apply factorization to each level, then to apply the needed update to other tiles in  $L$  and  $U$ , and allowing the next subblock's set of tiles to be factored. In Javelin, OpenMP tasking is used as the programming model, and tile size options are made available to the user. Fig. 6 provides the algorithm used, and for simplicity we do not explicitly state the dependencies.

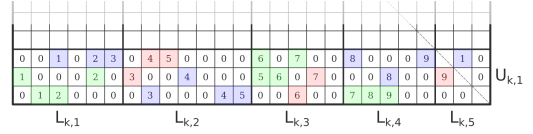


Figure 5. Layout of blocks and tiles in SR method. Tiles are laid out in each block ( $L_{k,i}/U_{k,1}$ ) independently of each other, and can span multiple rows. Nonzero elements in each row are indexed starting from 1. Contiguous nonzeros that are grouped together are colored similarly.

**Require:** Let  $k$  be the level to complete SR.

```

for  $i = 0$  to  $\text{num\_lvl}$  do
  spawn all tiles ( $\text{DIVIDE\_COLUMNS}(L_{k,i})$ )
  for  $j = i$  to  $\text{num\_lvl}$  do
    spawn all tiles ( $\text{UPDATE\_BLOCK}(L_{k,i}, L_{k,j})$ )
  end for
end for
spawn all tiles ( $\text{FACTOR\_LU}()$ )

```

Figure 6. Segmented-Row Method (SR) Algorithm

In Fig. 6, `DIVIDE_COLUMNS` takes the diagonal value in  $U$  corresponding to the used columns in the tile and divides the entries in the tile. The `UPDATE_BLOCK` method applies the multiplication-subtraction updates to nonzeros in tiles in later levels, similar to the inner-most line in Fig. 1. Lastly, `FACTOR_LU` factors the tiles in the blocks  $L_{k,k}$  and  $U_{k,1}$ .

SR is useful in several different situations. These situations are when there exist fewer rows that are excluded from the level scheduling method than threads running on the system and the number of nonzeros in the rows are highly imbalanced. However, SR suffers from the need to use a tasking programming model and the overheads that come with it. Moreover, the fixed tile size of SR lends itself to using optimized vector operations (e.g., AVX). Many new hardware such as Intel Xeon Phi have large vector lanes that need to be used to achieve top performance.

**Even-Rows.** The Even-Rows (ER) method is a much simpler method than SR. ER depends on the number of rows excluded from the level scheduling method being greater than the number of desired threads to be used on the system. With the number of rows being greater than the number of threads, each thread can factor multiple rows and any imbalance can be averaged over all thread's

rows. The excluded rows are reordered to the end of the matrix similar to SR after finding the level order. Here, either  $\text{lower}(A + A^T)$  or  $\text{lower}(A)$  could be used to find the level order. While  $\text{lower}(A)$  generally offers more and larger levels, this does not seem to have much difference in performance, see section VII. Options exist to allow in preprocessing to form tiles similar to SR to be added for `stri`, but are not needed for the incomplete factorization. In Fig. 7, each thread gets a row and factors  $L$  up until the corner piece (i.e.,  $L_{k,2}$  and  $U_{k,1}$ ) while making the needed updates to the corner piece's  $L_{k,2}$  and  $U_{k,1}$ . Fig. 8 provides an overview of the ER method, where `FACTOR_L` does the factorization of each row up until the corner piece. When all the threads are done with their rows, the corner piece can be factored with a call to `FACTOR_LU`. The factorization of the corner can be done in serial or parallel. However, for most matrices, serial seems to be good enough.

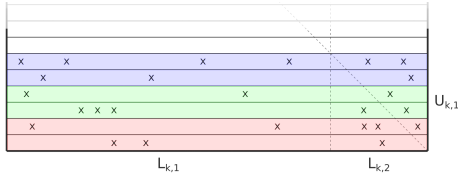


Figure 7. Layout of blocks in the ER method. Each thread will get a number of rows. They will complete an up-looking incomplete LU on  $L_{k,1}$  while making the needed updates to the values in  $L_{k,2}$  and  $U_{k,1}$ .

```

Require: Let  $k$  be the level to complete ER.
for  $\text{row}$  in  $\text{lower}$  in parallel do
  FACTOR_L(row)
end for
spawn all tiles (FACTOR_LU( $L_{k,2}$ ,  $U_{k,1}$ ))

```

Figure 8. Even-Rows Method (ER) Algorithm

#### IV. EXPERIMENTAL SETUP

Here we provide the setup for the next three sections to evaluate the performance of incomplete LU factorization, sparse triangular solve, and the sensitivity of Javelin to input.

**Test systems.** We test our framework on two systems, i.e., Bridges at PSC and Stampede2 at TACC. The first system contains two Intel Xeon E5-2695 v3 (Haswell) processors each with 14 cores and having 128GB DDR4. The second system contains an Intel Xeon Phi Knights Landing 7250 (KNL) each with 68 cores and having 96GB DDR4. The KNL is setup in cache mode with 32KB L1, 1MB L2 per two-core tile, and 16GB direct-mapped L3. Though KNL has a variety of different ways to setup its fast memory, such as cache mode and flat, we test in cache mode as this is the mode recommended by TACC to most MPI+X scientific applications. Therefore, we wanted to test in the mode likely to be used by applications that will use Javelin. All codes are compiled with Intel Compiler 17.4 / 17.3 with

Table I  
TEST SUITE. N IS THE MATRIX DIMENSION, NNZ IS THE NUMBER OF NONZERO, RD IS NONZEROS / N, SP IS IF THE SYMBOLIC PATTERN OF THE MATRIX IN NATURAL ORDER IS SYMMETRIC, AND LVL IS THE NUMBER OF LEVELS FOUND IN THE LEVEL SCHEDULING.

Matrix	N	Nnz	RD	SP	Lvl	
wang3	26064	177168	6.8	yes	10	B
TSOPF_RS_b300_c2	28338	2943887	103.88	no	180	B
3D_28984_Tetra	28984	285092	9.84	no	34	B
ibm_matrix_2	51448	537038	10.44	no	29	B
fem_filter	74062	1731206	23.38	yes	554	B
trans4	116835	749800	6.42	no	20	B
scircuit	170998	958936	5.61	yes	34	B
transient	178866	961368	5.37	yes	16	B
offshore	259789	4242673	16.33	yes	74	A
ASIC_320ks	321671	1316085	4.09	yes	16	B
af_shell3	504855	1.756e7	34.79	yes	630	A
parabolic_fem	525825	3674625	6.99	yes	28	A
ASIC_680ks	682712	1693767	2.48	yes	21	B
apache2	715176	4817870	6.74	yes	13	A
tmt_sym	726713	5080961	6.99	yes	28	B
ecology2	999999	4995991	5	yes	13	A
thermal2	1.2e6	8580313	6.99	yes	27	A
G3_circuit	1.5e6	7660826	4.83	yes	13	B

-O3 options. We use OpenMP with the DYNAMIC scheduling and `CHUNK_SIZE=1` for all our tests, though ER may benefit from different scheduling and chunk size options. This decision was made to limit the number of possible combinations in finding the best overall performance.

**Matrices.** We select a large array of different matrices from the SuiteSparse collection [16]. Table I provides the details for each. Our test suite of matrices is broken into two pieces (labeled A and B). The first group (A) are commonly tested for ILU with regard to convergence [3], [8], despite being symmetric positive definite. The goal of group A is to demonstrate the effect our structure and preordering described in this section have on the convergence. Though previous studies have looked at the effect of preordering on convergence, no study has looked at our particular ordering combination of level set. This is presented in section VII. The second group (B) contains a wide array of matrices from multiple areas. While much previous work focuses on only providing empirical evaluation of matrices from areas coming from the discretization of partial differential equations (PDEs), there is a growing need for iterative methods in other areas that have very irregular matrices, such as certain stages of circuit simulation [17]. Therefore, to have a thorough examination, we include a wide range of matrices with different numeric patterns and row densities, i.e., average number of nonzeros per row.

**Incomplete levels.** We will only consider  $\text{ILU}(k)$  with  $k = 0$ , though Javelin supports other levels as implemented by other work [5], [6], [15] and commonly used in iterative solvers. As  $k$  increases, additional fill-in is allowed into the sparsity pattern. The placement and amount of this would depend on the ordering of the matrix. Therefore,



comparing performance as  $k$  varies would not provide a deep understanding of the scalability of Javelin without knowing where and how much fill-in was produced. It is therefore common to compare scalability primarily with ILU(0) [3], [5] over a large test suite of matrices with different sparsity pattern and row density in order to estimate how well the implementation scales with fill-in as we do in this paper.

**Preordering.** It is common to permute the nonzeros of a sparse matrix before applying an iterative method. These permutations allow for better memory access patterns and reductions to iteration counts [9]. Javelin currently only has a parallel level-set ordering built-in that is needed for the framework. Therefore, we assume that the given matrix is already ordered in some manner that is ideal for the given solver method by the user. For the performance evaluation, we assume that the given matrix has been ordered in the following manner. A Dulmage-Mendelsohn ordering is used to move nonzeros to the diagonal of the matrix, then Nested-Dissection (ND) from METIS is performed. This ordering was chosen because ND is commonly applied to coefficient matrices for parallel factorization [15]. We consider the sensitivity of this choice in section VII, and compare against the use of ordering with Reverse Cuthill McKee (RCM) ordering to using ND.

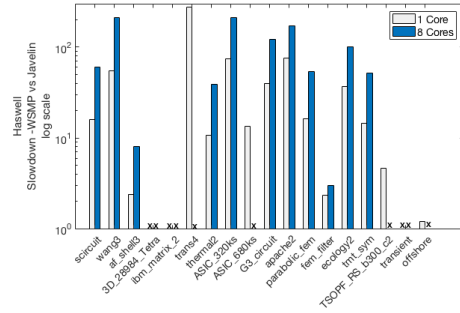
## V. EVALUATION OF INCOMPLETE LU

We first considered the performance of Javelin to other packages. We note that there does not exist many threaded incomplete factorization packages. Many popular packages (even ones in distributed memory), such as SuperLU [8] and Trilinos, only offer a serial option, and we found that Javelin in serial was either faster than (15 matrices) or within 10% (3 matrices) of these serial packages. Additionally most packages that offer traditional ILU use only ILU( $k, \tau$ ) and does not provide an interface to ILU( $k$ ). Therefore, we will use this double method to compare to the commercial package Watson Sparse Matrix Package (WSMP) [18], but will use ILU( $k$ ) for scalable performance analysis for reasons stated before. WSMP is given the matrix in the level ordering used by Javelin with  $k=0$  and  $\tau$  is set to be a value so that nonzeros are similar to that of ILU(0), and we do not allow pivoting. Here, only Javelin with levels will be used, as we compare the tradeoff of using lower stage later. We will only consider the time to perform the numeric factorization (i.e.,  $time(matrix, p)$ ) for a particular matrix using  $p$  cores. Fig 9 presents the slowdown of WSMP vs Javelin, i.e.,

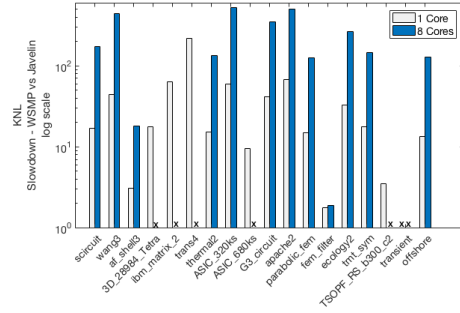
$$slowdown(matrix, p) = \frac{time(WSMP, matrix, p)}{time(Javelin, matrix, p)}.$$

In many cases, WSMP failed due to numerical constraints placed in part by the internal structure and required re-ordering, and we place 'x' on such columns. Moreover, we only present up to 8 cores as WSMP does not scale past this point on either system. Note that Javelin is multiple

magnitudes faster than WSMP in both serial and in parallel. This multiple magnitude difference is due to using very light weight data structures that leads to no overhead. WSMP (along with many other packages) uses supernode or supernode-like data structures that are too big for very sparse incomplete factorization as it does too many data movement operations per float-point operation. As core counts increase, other packages try to reduce synchronizations by grouping like nonzero structures into these data structures, and thus reducing the serial fraction that limits strong scaling. However, if there does not exist many similarities in nonzero structure (as in ILU), the number of reductions is very low. In contrast, Javelin reduces the number of synchronizations by locality in memory and by using point-to-point synchronizations.



(a) Slowdown on Haswell



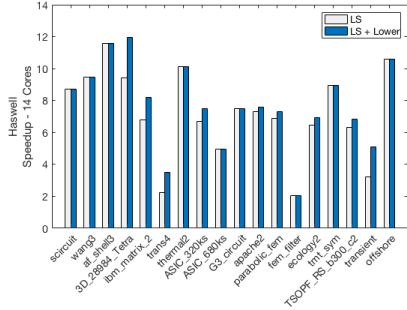
(b) Slowdown on KNL

Figure 9. Slowdown of WSMP compared to Javelin on Haswell and KNL. WSMP does not have any real scaling behavior after 8 cores. 'x' represents where WSMP failed due to internal issues.

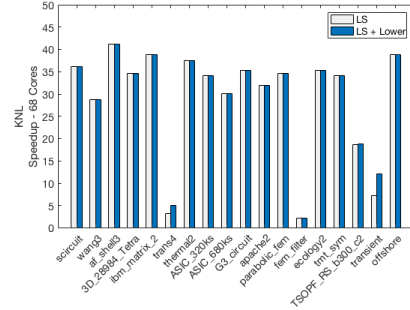
**Scalability of Javelin.** We will define

$$speedup(matrix, p) = \frac{time(matrix, 1)}{time(matrix, p)}$$

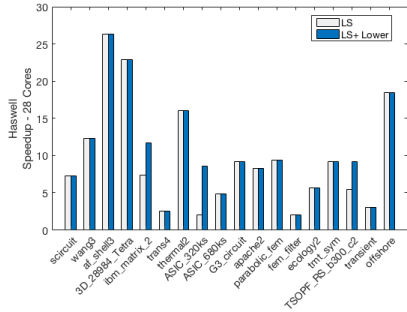
for any particular system for the remainder of this section. We currently do not consider setup time as this is merely finding the level scheduling and copying to CSR which is done in parallel, and would be reused by other methods such as `spmv` and `stri`. Moreover, almost all solver packages require some overhead to copy to structure and preorder, and we do both our copies and level-set ordering in parallel. Javelin is  $\sim 10\times$  faster than WSMP in this stage.



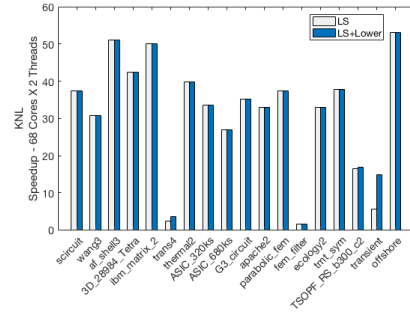
(a) 14 Haswell Cores (1 Socket)



(a) 68 KNL Cores (1 Socket)



(b) 28 Haswell Cores (2 Sockets)



(b) 68 KNL Cores X 2 Threads (1 Socket)

Figure 10. Speedup on Intel Haswell. The LS bars are speedup due only to level scheduling with point-to-point synchronization and the LS+Lower bars are using both level scheduling and best lower method.

Figure 11. Speedup on Intel KNL. The LS bars are speedup only due to level scheduling with point-to-point synchronization and the LS+Lower bars are using both level scheduling and best lower method.

In Fig. 10, we present the speedup for Javelin over all matrices on Haswell for 14 and 28 cores. We denote the speedup using only the level scheduling with point-to-point communications as LS, and we denote using best combination of level scheduling and lower as LS+Lower. The geometric mean of speedup for selecting the best mixture of all methods and threads is only  $9.45\times$  (not pictured), however we can see this value does not really reflect the level of speedup possible by Javelin. In general, we observe most matrices can achieve around an  $8\times$  speedup using just level scheduling with point-to-point synchronizations on 14-cores. Several matrices are unable to reach this standard of performance. For *ibm\_matrix\_2*, *trans4*, and *transient*, our lower method is able to boost performance closer to the level of the other matrices. The *fem\_filter* is unable to be helped by these methods, and this is due to the numerical pattern not allowing to find many large levels, see section VII. As we cross socket from 14 cores to 28 cores, we notice a much larger gap in performance of difference matrices. Matrices such as *af\_shell3* and *3D\_28984\_Tetra* are able to perform well over socket, but most cannot. This is due to the non-uniform memory access (NUMA) between the sockets. Currently, Javelin does not account for these types of accesses. The level scheduling depends on light weight point-to-point synchronizations using locks that are not ideal across sockets due to their required memory load.

The SR lower scheduling method does very poorly across sockets. This is due to having no locality in the tasks as they are taken from the tasking queue, and results in fewer matrices being boosted by using a lower method. With the current OpenMP Scheduling, ER also has trouble with memory accesses across sockets. ER could be improved with a more static scheduling or NUMA-aware blocking of the distribution of the lower rows. Despite this, no performance on 28 cores is worse than on 14 cores, and performance of *ibm\_matrix\_2* and *TSPF\_RS\_b300\_c2* is able to be boosted by the lower methods.

In Fig. 11, we present the speedup performance of Javelin on KNL. KNL is a many-core system that is designed for very high levels of concurrent operations. Each core is relatively slow compared to the fast massively out-of-order cores on Haswell. Therefore any inefficiency in parallelizing ILU will be noticeable. In this case, the geometric mean of the best speedup selecting the best parameters tested results in  $25.1\times$ . Again this value is not representative of the performance we observe. We present the speedup on 68 cores, which is all the cores in one socket with one thread in the first figure. We see that for most matrices the speedup is around  $30\times$  using only level scheduling and point-to-point synchronizations. As a fraction of total system use, this is less than that is achieved on Haswell. The main reason for this is the required level of concurrency may not be able to

be extracted from the matrices. The lower methods now only help in two cases, and only by a small fraction. This seems to be due to the tasking overhead of using an OpenMP queue with so many tasks on this number of threads as examined using Intel VTune. A specialized light weight tasking library is currently being constructed in Javelin for this reason. Despite this, we will observe the importance of the lower stage for *stri* in the next section. In the next figure, we test using 136 threads (i.e., 68 cores with 2 threads each) to examine the ability of using multiple threads per core. While minor performance can be gained by some matrices, we see over-subscribing cores does not yield high results with Javelin due to pressure on the memory system. However, the performance does not generally degrade.

## VI. EVALUATION OF TRIANGULAR SOLVE

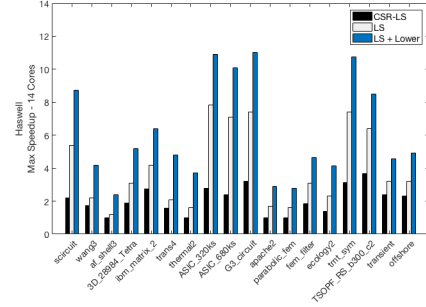
This section provides an evaluation of sparse triangular solve (*stri*) inside of Javelin. Recall that the objective of Javelin was to implement a package that provides a scalable incomplete factorization that could be used as a preconditioner for an iterative solver with the structural needs to make *stri* and *spmv* scale. In standard execution, the incomplete factorization may only be formed once, but *stri* may be called thousands of times. Exact iteration counts for a subset of matrices in our test suite can be found in Table II. We will only consider *stri* here as this is the primary call needed for methods like GMRES that use ILU, and will leave *spmv* for future work that addresses preconditioners that use *spmv* like successive over-relaxation. We will compare the performance to that of a standard implementation of *stri* in CSR format using OpenMP and barriers between levels in a level set ordering as done in previous works [11], [12], and label it as CSR-LS. WSMP is not reported due to its lack of performance and its inability to factor many matrices. Using CSR-LS as the base of comparison, performance is:

$$\text{maxspeedup}(m, mat, p) = \frac{\text{time}(\text{CSR-LS}, mat, 1)}{\max_{i=1}^p \{\text{time}(m, mat, i)\}}$$

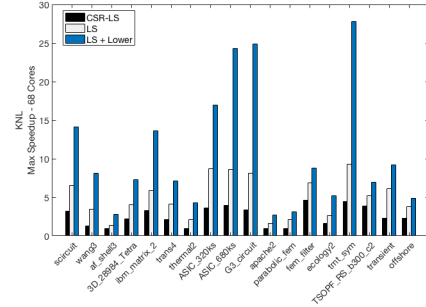
where  $m$  is the method,  $mat$  is the matrix, and  $p$  is the number of cores. The two methods compared from Javelin are using only the level scheduling stage (LS) and using both level scheduling with the lower stage blocking that is automatically picked by Javelin (LS + Lower).

In Fig. 12, we provide the maximal speedup for *stri* on one socket of Intel Haswell and Intel KNL. As is known, the traditional method of using level-sets with barriers does not scale well. Using only the level scheduling method with pruning in Javelin we are able to now have much better scaling. However, in some cases, such as *wang3* on Haswell cores, the difference between the base case can be very small. The lower stage blocks help performance of *stri* on all matrices, and increase performance of *wang3*. In particular, this is seen the most on Intel KNL where

the lower stage does not seem to help speedup incomplete factorization as much as on Haswell cores. We believe that this behavior is currently due to the parameters that judge between ER and SR, lower stage size, and subblocking have been tuned with more weight to *stri* than ILU. This weighting is due to an iterative method's time dominated by *stri* over ILU. Further investigation is needed to provide better balance for different matrices, but may require some estimate of conditioning of the linear system.



(a) 14 Haswell Cores (1 Socket)



(b) 68 KNL Cores (1 Socket)

Figure 12. The maximal speedup of *stri* using Javelin. The CRS-LS bars are the speedup of the standard level-set *stri* implement with OpenMP. The LS bars are due only to using the level scheduling stages of Javelin, and LS + Lower is the speedup using both stages.

## VII. SENSITIVITY ANALYSIS

In this section, we will examine options that can be modified in Javelin, and their general effects. We will examine the choice of ordering on iteration count and the distribution of levels for different choices of the lower stage.

**Iteration count.** As previously stated, the number of iterations needed to converge to a solution will depend not only on the conditioning of the system, but the ordering of the sparse coefficient matrix. Though some work has been done to better understand this phenomenon, it is still not fully understood why the order of the sparse coefficient matrix has such a strong impact on the convergence rate. It is known that Coloring and ND orderings in general increase iteration count, and orders like RCM decrease iteration count. In Table II, we provide the number of iterations needed to converge to a solution with relative



error of  $1e-6$  for each of the matrices in group A and using an array of different orders. We consider SYMAMD (AMD) due to a recommendation in [9], Reverse Cuthill-McKee (RCM), ND, and the natural order (NAT). Moreover, we consider the level scheduling ordering imposed on top of coefficient matrices preordered with RCM (LS-RCM) and ND (LS-ND) in order to compare how the level set ordering done in Javelin will affect iteration count. Coloring is not considered as it is known to be worse in terms of iteration than any other ordering considered here. We note that in most cases, RCM and NAT have the fewest iterations. However, LS-RCM is close or even better than RCM in several cases. In Fig. 13, we present the speedup of group A matrices using only level scheduling with point-to-point synchronizations that are initially ordered using RCM. The speedup is calculated with the relative base being serial with ND ordering. We see that the speedup is comparable to those in section V. Therefore, this makes either method a candidate for increased performance using Javelin. However, the speedup relative to itself is slightly less than that with ND ordering. The largest factor in this is the size of level sets found. We also note that the speedup of `stri` is proportional to what can be achieved by ILU. As a result, the choice will all be up to the user to determine if the speedup will offset the number of iterations. For example, the first 3 matrices in Fig. 13 would most likely be best preordered with ND, since they have good speedups and few iterations.

Table II  
THE NUMBER OF ITERATIONS NEED TO CONVERGE BASED ON ORDER FOR GROUP A MATRICES.

Matrix	AMD	RCM	ND	NAT	LS-RCM	LS-ND
offshore	11	10	11	9	11	11
parabolic_fem	300	116	270	288	113	270
af_shell3	420	381	648	284	350	390
thermal2	764	385	750	813	540	753
ecology2	1532	889	1152	881	868	1503
apache2	597	270	597	275	487	603

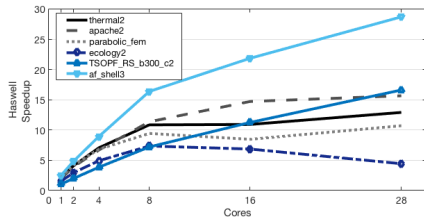


Figure 13. The relative speedup of group A matrices on Intel Haswell when the input matrix is first order with RCM.

**Levels and lower size.** Here we will provide an analysis of the number of levels selected for rows to be solved using one of the two lower methods. In Table III, we provide the statistics related to the level selection when considering the pattern of  $\text{lower}(A + A^T)$ . We note that there are other factors we consider in addition to the minimal row size in

selecting which rows are to be permuted to the end. These factors include the row density and the relative location.

Table III  
LEVEL SET INFORMATION OF  $\text{lower}(A + A^T)$  PATTERN. LVL IS THE NUMBER OF LEVELS FOUND, M IS THE MINIMAL, MAX IS THE MAXIMUM, MED IS THE MEDIAN NUMBER OF ROWS IN A LEVEL. R-A IS THE NUMBER OF ROWS MOVED TO THE END OF THE COEFFICIENT MATRIX, WHERE A IS A SENSITIVITY PARAMETER.

Matrix	Lvl	M	Max	Med	R-16	R-24	R-32
wang3	10	5	1.11e4	582	5	21	21
TSOPF_RS...	180	1	4741	109	0	430	529
3D_28984...	34	1	9054	383	24	24	48
ibm_mat...	29	4	3.02e4	1332	20	41	41
fem_filter	554	2	9003	3	1792	1810	1862
trans4	20	1	8.57e4	84	13	13	13
scircuit	34	1	6.23e4	117	53	87	117
transient	16	1	7.82e4	24	2	47	88
offshore	74	1	3.04e4	2724	58	76	529
ASIC_320ks	16	3	2.23e5	2991	3	3	3
af_shell3	630	1	2.31e4	5	1751	2154	3682
parabolic_fem	28	1	1.02e5	5068	16	33	33
ASIC_680ks	21	1	5.84e5	1316	26	26	50
apache2	13	3	3.22e5	2961	3	19	19
tmt_sym	28	1	1.78e5	7393	18	18	47
ecology2	13	1	4.42e5	189	24	24	24
thermal2	27	1	2.72e5	16951	7	7	31
G3_circuit	13	2	6.19e5	4109	11	11	38

From Table III, we can observe that most matrices have only tens to hundreds of levels. Even matrices with high row density, such as `TSOPF_RS_b300_c3` and `offshore`, fit into this range of levels. We would like to see fewer levels because: 1. reduces the preprocessing time; 2. the level sets tend to be larger allowing for more concurrency. Because of the level size distribution, the median becomes the best possible estimator of possible parallel performance. We see that most matrices can support hundreds of concurrent threads based off this median value. The exception to this is `fem_filter`, `trans4`, and `af_shell3`. While `af_shell3` only has a median value of 5, we discover that level scheduling still does a good job with performance as good or better than other matrices. However, the other two do not do well with level scheduling. On a varying number of cores, the lower method is able to help boost `trans4`, and is not able to for `fem_filter`. We can observe that `trans4` has relatively few rows that ever get permuted to the end (i.e., 13). However, using SR, we are able to improve the speedup by a difference of 1-2 $\times$  on socket. We believe that we could improve selecting a better tile size and changing the tasking system to reduce overhead in the future. Moreover, `transient` also suffers from a small median value. However, the lower method is able to improve the performance on socket by a factor of  $\sim 2.3\times$  on Haswell and by a factor of  $\sim 1.6\times$  on KNL, despite the current overheads in the lower method.

Additionally, we look at similar statistics for the  $\text{lower}(A)$  pattern in Table IV. We note that we could only use this pattern if we consider just level scheduling or level

Table IV

LEVEL SET INFORMATION OF  $\text{lower}(A)$  PATTERN. MIN IS THE MINIMAL AND MAX IS THE MAXIMUM NUMBER OF ROWS IN A LEVEL.

Matrix	Min	Max	Median
TSOPF_RS_b330_c2	1	4753	122
3D_28984_Tetra	4	9059	543
ibm_matrix_2	1	30259	1361
trans4	1	8549	493

scheduling with Even-Rows, but not with the Segmented-Row method. We note that the median number of nonzeros does increase in all matrices as we would expect. However, the increase is very small except in a few cases such as `trans4`. Despite this increase, we could not get any real additional speedup performance for `trans4` using the  $\text{lower}(A)$  pattern. Therefore, we by default always recommend using the  $\text{lower}(A+A^T)$  pattern as it will allow for SR to be run and enable tiling for `stri`.

### VIII. CONCLUSIONS

This paper presented a new scalable framework for incomplete factorization, called Javelin, aimed at allowing traditional thresholding and fill-level options while being optimized for sparse matrix-vector multiplication and sparse triangular solves on current many-core systems. This framework is designed using level scheduling of up-looking LU factorization that is implemented with light weight point-to-point synchronizations and then using a second (i.e., lower) level stage when level scheduling will no longer provide the needed degree of concurrency. This lower stage is designed by co-optimizing the incomplete factorization and `stri` after factorization. This is achieved through reducing the serial portion and imbalance to a minimal through design, and thus helping to achieve strong scalability. We demonstrated that Javelin could achieve good speedups on socket for both Intel Haswell and Intel KNL systems for a wide array of coefficient matrices for both ILU and triangular solve. Moreover, we examined the effect that requiring reordering would have on the iteration count of several test systems. This demonstration shows that, not only can we scale incomplete factorization, we can leave the system in an order that has desirable properties.

### ACKNOWLEDGMENTS

We thank Andrew Bradley and Siva Rajamanickam for helpful discussions while at Sandia National Laboratories.

### REFERENCES

- [1] H. Kabir, J. D. Booth, G. Aupy, A. Benoit, Y. Robert, and P. Raghavan, “STS-k: a multilevel sparse triangular solution scheme for NUMA multicores,” in *Proceedings of SuperComputing SC 2015*, 2015, pp. 55:1–55:11.
- [2] S. MacLachlan, D. Osei-Kuffuor, and Y. Saad, “Modification and compensation strategies for threshold-based incomplete factorizations,” *SIAM SISC*, vol. 34, no. 1, pp. 48–75, 2012.
- [3] E. Chow and A. Patel, “Fine-grained parallel incomplete LU factorization,” *SIAM SISC*, vol. 37, no. 2, pp. 169–193, 2015.
- [4] P. Hénon and Y. Saad, “A parallel multistage ILU factorization based on a hierarchical graph decomposition,” *SIAM SISC*, vol. 28, no. 6, pp. 2266–2293, Jan 2006.
- [5] J. D. Booth, K. Kim, and S. Rajamanickam, “A comparison of high-level programming choices for incomplete sparse factorization across different architectures,” in *2016 IEEE IPDPSW*. IEEE, May 2016.
- [6] D. Hysom and A. Pothen, “A scalable parallel algorithm for incomplete factor preconditioning,” *SIAM SISC*, vol. 22, no. 6, pp. 2194–2215, Jan 2001.
- [7] E. Chow and Y. Saad, “Experimental study of ILU preconditioners for indefinite matrices,” *J. of Comput. and Applied Mathematics*, vol. 86, no. 2, pp. 387–414, Dec 1997.
- [8] X. S. Li and M. Shao, “A supernodal approach to incomplete LU factorization with partial pivoting,” *ACM TOMS*, vol. 37, no. 4, pp. 1–20, Feb 2011.
- [9] M. Benzi, D. B. Szyld, and A. van Duin, “Orderings for incomplete factorization preconditioning of nonsymmetric problems,” *SIAM SISC*, vol. 20, no. 5, pp. 1652–1670, 1999.
- [10] S. W. Hammond and R. Schreiber, “Efficient ICCG on a shared memory multiprocessor,” *International Journal of High Speed Computing*, vol. 04, no. 01, pp. 1–21, Mar 1992.
- [11] J. Park, M. Smelyanskiy, N. Sundaram, and P. Dubey, “Spar-sifying synchronization for high-performance shared-memory sparse triangular solver,” in *Proceedings of ISC*, 2014, pp. 124–140.
- [12] A. M. Bradley, “A hybrid multithreaded direct sparse triangular solver,” in *2016 Proceedings SIAM CSC*. SIAM, 2016, pp. 13–22.
- [13] W. Liu and B. Vinter, “CSR5,” in *International Conference on Supercomputing ICS’15*. ACM Press, 2015.
- [14] G. E. Blueloch, M. A. Heroux, and M. Zagha, “Segmented operations for sparse matrix computation on vector multiprocessors,” Pittsburgh, PA, USA, Tech. Rep., 1993.
- [15] K. Kim, Sivasankaran, G. Stelle, H. C. Edwards, and S. Olivier, “Task parallel incomplete cholesky factorization using 2d partitioned-block layout,” published on arXiv.
- [16] T. A. Davis and Y. Hu, “The university of florida sparse matrix collection,” *ACM TOM*, vol. 38, no. 1, pp. 1:1–1:25, 2011.
- [17] W. H. A. Schilders, “Iterative solution of linear systems in circuit simulation,” in *Progress in Industrial Mathematics at ECMI 2000*. Springer Berlin Heidelberg, 2002, pp. 272–277.
- [18] A. Gupta, “Enhancing performance and robustness of ILU preconditioners by blocking and selective transposition,” *SIAM SISC*, vol. 39, no. 1, pp. A303–A332, 2017.