Aasen's Symmetric Indefinite Linear Solvers in LAPACK

Ichitaro Yamazaki and Jack Dongarra

December 20, 2017

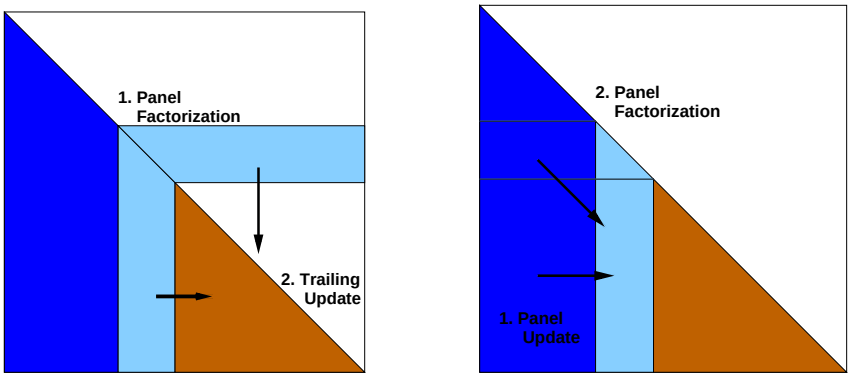
Abstract

Recently, we released two LAPACK subroutines that implement Aasen's algorithms for solving a symmetric indefinite linear system of equations. The first implementation is based on a partitioned right-looking variant of Aasen's algorithm (the column-wise left-looking panel factorization, followed by the right-looking trailing submatrix update using the panel). The second implements the two-stage left-looking variant of the algorithm (the block-wise left-looking algorithm that reduces the matrix to the symmetric band form, followed by the band LU factorization). In this report, we discuss our implementations and present our experimental results to compare the stability and performance of these two new solvers with those of the other two symmetric indefinite solvers in LAPACK (i.e., the Bunch-Kaufman and rook pivoting algorithms).

1 Introduction

Solving the dense symmetric indefinite linear systems of equations is relevant to many scientific and engineering problems. Compared with the non-symmetric linear solver, the symmetric solver has several advantages (e.g., the symmetric factorization not only preserves the structural and spectral properties of the matrix, but also reduces the computational and storage costs of the factorization). However, maintaining both the symmetric structure and the numerical stability of the factorization leads to data access patterns that present significant challenges in the development of a high-performance symmetric indefinite linear solver. In this report, we examine both the performance and stability of the four symmetric indefinite solvers in the current release of LAPACK [2], including the two newly-released subroutines that are based on Aasen's algorithms [1, 10, 6]. Our experimental results show that though the backward errors of the new two-stage Aasen algorithm can be an order of magnitude greater, this two-stage algorithm has a superior thread-parallel scalability for factorizing a large-scale matrix on a multicore architecture.

The rest of the paper is organized as follows. Section 2 introduces LAPACK's symmetric indefinite solvers, while Section 3 describes the two variants of Aasen's algorithm implemented in LAPACK in more detail. Then, Section 4 presents the LAPACK implementations of the two Aasen's algorithms. Finally, Section 5 shows our numerical and performance results. For completeness, we lists the interfaces to the new Aasen linear solvers in the Appendix. In this paper, we use $a_{i,j}$ and $\mathbf{a}_j$ to denote the (i,j)-th entry and the j-th column of the matrix A, respectively, while $A_{i_1:i_2,j_1:j_2}$ is the submatrix consisting of the i_1-th through the i_2-th rows and the j_1-th through the j_2-th columns of A. We also use $A_{I,J}$ and $A_{I_1:I_2,J_1:J_2}$ to denote the (I,J)-th block and the submatrix consisting of the I_1-th through the I_2-th block rows and J_1-th through the J_2-th block columns of A, where n_b is the block size and n_t is the total number of block columns or rows in A of dimension n $\left(\text{i.e., } n_t = \lceil \frac{n}{n_b} \rceil\right)$.



(a) Right-looking. (b) Left-looking.

Figure 1: Two different updating schemes.

2 LAPACK Symmetric Indefinite Solvers

For solving the symmetric indefinite linear system of equations, $A\mathbf{x} = \mathbf{b}$, LAPACK factorizes the dense symmetric indefinite matrix A using either the Bunch-Kaufman [7], rook pivoting [4], or Aasen's [10] algorithm. These algorithms maintain the numerical stability of the symmetric factorization by selecting a diagonal pivot when computing each column and row of the factors. The main difference between the algorithms is the way the pivot is selected. These different pivoting schemes lead to the different forms of the factorization. Specifically, Bunch-Kaufman and rook pivoting compute the LDL^T factorization, where D is a symmetric block diagonal matrix with either 1-by-1 or 2-by-2 diagonal blocks. In contrast, the partitioned or two-stage variants of Aasen's algorithm computes the LTL^T factorization, where T is either a symmetric tridiagonal matrix or a symmetric band matrix, respectively. In all the algorithms, L is a lower-triangular matrix with unit diagonals. After the factorization, the solution $\mathbf{x}$ to the linear system can be computed by first applying the forward substitution with L to the right-hand-side vector $\mathbf{b}$, then solving either the tridiagonal or band linear system with T, and finally performing the backward substitution with L^T (combined with the application of the column and row permutations).

In order to improve the data locality and the performance of the factorization, LAPACK implements a *partitioned* factorization [3]. Namely, LAPACK partitions the matrix into the leading block column, which is referred to as the *panel*, and the trailing submatrix. Then the *right-looking* formulation of the algorithm first factorizes the panel that is then used to update the trailing submatrix (see Figure 1(a) for an illustration). The algorithm is referred to as right-looking because the panel is used to update the trailing submatrix, which is on the right of the panel. This is in contrast to the *left-looking* formulation of the algorithm that updates the panel using all the previous columns of the matrix, which is on the left of the panel, at each step of the factorization (see Figure 1(b)). The same procedure is repeatedly applied to the trailing submatrix to factorize the whole matrix. The partitioned factorization improves the performance because the BLAS-3 matrix-matrix operations can be used for the trailing submatrix or panel update that dominates the number of floating point operations (flops) needed to factorize the matrix. In particular, if n is the number of the columns in A and n_b is the block size, then both Bunch-Kaufman and Aasen's perform a total of $O(n_b n^2)$ flops to factorize the n_t panels of A (i.e., $O(n_t \cdot n_b^2 n)$ flops with $n_t = \frac{n}{n_b}$). This is a lower term in the total number of flops needed to factorize the whole matrix, i.e., total of $\frac{1}{3}(1 + \frac{1}{n_b})n^3 + O(n^2 n_b)$ flops with the Bunch-Kaufman

algorithm or $\frac{1}{3}n^3 + O(n^2 n_b)$ flops with the partitioned Aasen's algorithm (the partitioned Aasen's has the additional $\frac{1}{n_b}$ factor in the leading term of the flop count because it requires an additional rank-one update for each trailing submatrix update).

3 Aasen's Algorithms

We now recall the algorithms to compute the LTL^T factorization of the matrix A,

$$PAP^T = LTL^T \tag{1}$$

where P is a row permutation matrix constructed to maintain the numerical stability of the factorization, T is a symmetric tridigonal matrix, and L is a lower triangular matrix with unit diagonals.

The Parlett-Reid algorithm [9] is a column-wise algorithm (unlike a partitioned algorithm that is block-column wise) to compute the LTL^T factorization (1) in a right-looking fashion. Compared with Bunch-Kaufman, it requires about twice as many flops, i.e., $\frac{2}{3}n^3 + O(n^2)$ flops, because it performs a rank-two update of the trailing submatrix after factorizing each column, compared with Bunch-Kaufman that performs only a rank-one update. The left-looking Aasen algorithm [1] halves the number of flops needed to compute the LTL^T factorization, using an intermediate Hessenberg matrix H which is defined as $H = LT$. In this section, we discuss two variants of Aasen's algorithm that are implemented in LAPACK; the first one is a partitioned right-looking algorithm that generates a symmetric tridiagonal matrix T, while the second one is a two-stage algorithm whose first stage reduces the matrix A into a symmetric band matrix T with the bandwidth equal to the block size n_b.

3.1 Partitioned Right-looking Aasen

To exploit the memory hierarchy on a modern computer, a partitioned variant of Aasen's algorithm was proposed in [10]. It is a right-looking algorithm that first performs the panel factorization and then updates the trailing submatrix using the panel. Here, we describe the algorithm that only accesses the lower-triangular part of the matrix and factorizes a block-column at a time. We can reformulate the algorithm such that it only accesses the upper-triangular part of the matrix by factorizing a block-row at a time.

The algorithm first factorizes the leading block column, or the panel, in a left-looking column-wise fashion; it first sets the first column $\boldsymbol{\ell}_1$ of L to be the first column of an identity matrix. Then, for $j = 1, 2, \ldots, n_b$, assuming that the first $(j-1)$ columns of H and the first j columns of L have been computed, the j-th column $\mathbf{h}_{1:j,j}$ of H is computed from the j-th column of the equation $A = HL^T$,

$$\mathbf{h}_{j:n,j}\ell_{j,j}^T := \mathbf{a}_{j:n,j} - H_{j:n,1:j-1}\boldsymbol{\ell}_{j,1:j-1}^T,$$

where $\ell_{j,j}$ is one. Also, from the j-th column of the equation $H = LT$, we have

$$\mathbf{h}_{j:n,j} = \boldsymbol{\ell}_{j:n,j-1}t_{j-1,j} + \boldsymbol{\ell}_{j:n,j}t_{j,j} + \boldsymbol{\ell}_{j:n,j+1}t_{j+1,j}.$$

Hence, if we let $\mathbf{w} = \boldsymbol{\ell}_{j:n,j}t_{j,j} + \boldsymbol{\ell}_{j:n,j+1}t_{j+1,j}$, then we can compute $\mathbf{w}$ by

$$\mathbf{w} := \mathbf{h}_{j:n,j} - \boldsymbol{\ell}_{j:n,j-1}t_{j-1,j},$$

and since $w_1 = \ell_{j,j}t_{j,j} + \ell_{j,j+1}t_{j+1,j}$, and $\ell_{j,j}$ is one and $\ell_{j,j+1}$ is zero, we have

$$t_{j,j} := w_1.$$

Finally, since $\mathbf{w}_{2:n} = \boldsymbol{\ell}_{j+1:n,j} t_{j,j} + \boldsymbol{\ell}_{j+1:n,j+1} t_{j+1,j}$, the $(j+1)$-th column of L can be computed by

$$\boldsymbol{\ell}_{j+1:n,j+1} := \frac{\mathbf{v}}{v_1} \quad \text{and} \quad t_{j+1,j} := v_1,$$

where $\mathbf{v} = \mathbf{w}_{2:n} - \boldsymbol{\ell}_{j+1:n,j} t_{j,j}$. To maintain numerical stability, the element with the largest module in $\mathbf{v}$ is used as the pivot.

After the panel factorization, the trailing submatrix is updated in a right-looking fashion,

$$A^{(2,2)} := A^{(2,2)} - H^{(2,1)} (L^{(2,1)})^T - \boldsymbol{\ell}_{n_b}^{(2,1)} t_{n_b+1,n_b} (\boldsymbol{\ell}_1^{(2,2)})^T, \tag{2}$$

where the matrix A is partitioned as

$$A = \left(\begin{array}{c|c} A^{(1,1)} & A^{(2,1)T} \\ \hline A^{(2,1)} & A^{(2,2)} \end{array} \right)$$

with $A^{(1,1)} = A_{1:n_b,1:n_b}$, and A is factorized as

$$L = \left(\begin{array}{c|c} L^{(1,1)} & \\ \hline L^{(2,1)} & I \end{array} \right) \quad \text{and} \quad T = \left(\begin{array}{c|c} T^{(1,1)} & \\ \hline & A^{(2,2)} \end{array} \right).$$

and $\boldsymbol{\ell}_{n_b}^{(2,1)}$ and $\boldsymbol{\ell}_1^{(2,2)}$ are the last and first columns of $L^{(2,1)}$ and $L^{(2,2)}$, respectively. Then, the same procedure is recursively applied on the trailing submatrix $A^{(2,2)}$. The resulting algorithm is implemented in LAPACK's `xSYTRF_AA` subroutine and released as a part of LAPACK version 3.7.0.

We compare this partitioned Aasen algorithm with the Bunch-Kaufman algorithm (Figure 2 shows the pseudocodes of the partitioned Bunch-Kaufman and Aasen algorithms):

- To compute the $(j+1)$-th column of L, the standard left-looking algorithm updates the current column using the 1-st through the j-th columns. The j-th step of the Aasen panel factorization update $\mathbf{a}_j$ using j `xAXPY`'s to compute $\mathbf{h}_{j+1}$ and then $\mathbf{w}$, but requires one additional `xAXPY` to compute $\mathbf{v}$ and $\boldsymbol{\ell}_{j+1}$. Moreover, compared with Bunch-Kaufman, Aasen's algorithm requires an additional rank-1 update for updating the trailing submatrix. Hence, Aasen's algorithm performs about $\frac{1}{3} n_t n^2$ additional flops for factorizing A, where n_t is the number of block columns $\left(\text{i.e., } n_t = \left\lceil \frac{n}{n_b} \right\rceil\right)$[1].

- To select the j-th pivot, Aasen's algorithm only uses the j-th column of A, while Bunch-Kaufman may access and update two columns for factorizing each column of its panel. If Bunch-Kaufman tests for the 2-by-2 pivots but selects a 1-by-1 pivot, then the computation used to update the second column is wasted. However, if Bunch-Kaufman selects a 2-by-2 pivot, it only swaps one column and row for factorizing these two columns of L.

- Bunch-Kaufman, and rook pivoting, compute the block diagonal D whose diagonal blocks represent either 1-by-1 or 2-by-2 pivots, and this block structure simplifies many of the matrix operations. For instance, during the factorization, updating the trailing submatrix with the tridiagonal matrix T of the partitioned Aasen algorithm requires one additional rank-one update. It is also a challenge to stably factorize the symmetric band matrix T while preserving its band structure because the symmetric pivoting (required to ensure the numerical stability) can completely destroy its band structure. LAPACK currently does

[1] When $n_b = 1$, or equivalently $n_t = n$, the partitioned Aasen algorithm becomes the right looking Parlett-Reid algorithm [9], performing $\frac{2}{3} n^3 + O(n^2)$ flops.

```
1:  α = (1 + √17)/8
2:  j = 1
3:  while j < n do
4:     k = j
5:     {Panel factorization}
6:     while j < k + n_b − 1 do
7:        Update a_j with the previous columns
8:        r = argmax_{i>j} |a_{i,j}| and ω_1 = |a_{r,j}|
9:        if ω_1 > 0 then
10:          if |a_{j,j}| ≥ αω_1 then
11:             s = 1
12:             Use a_{j,j} as a 1 × 1 pivot.
13:          else
14:             Update a_r with the previous
                columns
15:             ω_r = max_{i≥j;i≠r} |a_{i,r}|
16:             if |a_{j,j}|ω_r ≥ αω_1^2 then
17:                s = 1
18:                Use a_{j,j} as a 1 × 1 pivot.
19:             else
20:                if |a_{r,r}| ≥ αω_r then
21:                   s = 1
22:                   Swap rows/columns (j, r)
23:                   Use a_{r,r} as a 1 × 1 pivot.
24:                else
25:                   s = 2
26:                   Swap rows/columns (j + 1, r)
27:                   Use ( a_{j,j}  a_{j,r} )
                           ( a_{r,j}  a_{rr} ) as 2 × 2
                      pivot.
28:                end if
29:             end if
30:          end if
31:       else
32:          s = 1
33:       end if
34:       Scale the pivot columns to extract
          ℓ_{j:j+s−1}
35:       j = j + s
36:    end while
37:    {Trailing submatrix update}
38:    A^{(2,2)} := A^{(2,2)} − L^{(2,1)} D^{(1,1)} (L^{(2,1)})^T
39: end while
```

(a) Bunch-Kaufman algorithm.

```
1:  for J = 0, n_b, 2n_b, ..., n do
2:     for j = J, J + 1, ..., J + n_b − 1 do
3:        h_{j+1:n,j+1} := a_{j+1:n,j+1} − H_{j:n,1:j} ℓ^T_{j+1,J:j}

4:        w := h_{j+1:n,j+1}
5:        if j > 0 then
6:           w := w − ℓ_{j+1:n,j} t_{j,j+1}
7:        end if
8:        t_{j+1,j+1} := w_1
9:        if j < n − 1 then
10:          v := w_{2:n} − ℓ_{j+2:n,j+1} t_{j+1,j+1}
11:          k := arg min |v|_i
12:          swap i-th and k-th rows of L and H
13:          swap i-th and k-th rows and columns
             of A
14:          t_{j+2,j+1} := v_1
15:          ℓ_{j+1:n,j+1} := v/t_{j+2,j+1}
16:       end if
17:    end for
18:    A^{(2,2)} := A^{(2,2)} − H^{(2,1)} (L^{(2,1)})^T
                 − ℓ_{j+1:n,j} t_{j+1,j} ℓ^T_{j+1:n,j+1}
19: end for
```

(b) Aasen's algorithm where $H = LT$ and ℓ_1 is the first column of the identity matrix.

Figure 2: The partitioned Bunch-Kaufman and Aasen algorithms.

not have a subroutine that solves a symmetric indefinite band linear system of equations. Hence, to compute the solution with Aasen's algorithm, our current implementations rely on the non-symmetric tridiagonal or non-symmetric band factorization (`xGTTRF` or `xGBTRF`). Furthermore, since these diagonal blocks of D can be easily factorized and inverted in place, the Bunch-Kauffman, or rook pivoting, algorithm makes it feasible to compute the inverse of A without additional storage. On the other hand, it is more challenging to compute the matrix inverse in place (i.e., using only the lower or upper triangular part of the matrix) when Aasen's algorithm computes T which is either a tridiagonal or a band matrix and

```
 1: for J = 1, 2, ..., n_t do
 2:    for I = 2, 3, ..., J − 1 do
 3:       H_{I,J} := T_{I,I−1} L^T_{J,I−1} + T_{I,I} L^T_{J,I} + T_{I,I+1} L^T_{J,I+1}
 4:    end for
 5:
 6:    if J > 2 then
 7:       A_{J,J} := A_{J,J} − L_{J,2:J−1} H_{2:J−1,J} − L_{J,J} T_{J,J−1} L^T_{J,J−1}
 8:    end if
 9:    T_{J,J} := L^{−1}_{J,J} A_{J,J} L^{−T}_{J,J}
10:
11:    if J < n_t then
12:       if J > 1 then
13:          H_{J,J} := T_{J,J−1} L^T_{J,J−1} + T_{J,j} L^T_{j,j}
14:       end if
15:
16:       A_{J+1:n_t,J} := A_{J+1:n_t,J} − L_{J+1:n_t,2:J} H_{2:J,J}
17:       [L_{J+1:n_t,J+1}, H_{J+1,J}, P^{(J)}] := LU(A_{J+1:n_t,J})
18:
19:       T_{J+1,J} := H_{J+1,J} L^{−T}_{J,J}
20:
21:       L_{J+1:n_t,2:J} := P^{(J)} L_{J+1:n_t,2:J}
22:       A_{J+1:n_t,J+1:n_t} := P^{(J)} A_{J+1:n_t,J+1:n_t} P^{(J)T}
23:       P_{J+1:n_t,1:n_t} := P^{(J)} P_{J+1:n_t,1:n_t}
24:    end if
25: end for
```

Figure 3: First stage of the two-stage Aasen [6], where the first block column $L_{1:n_t,1}$ is the first n_b columns of the identity matrix and $[L, U, P] = \mathrm{LU}(A_{J+1:n_t,J})$ returns the LU factors of $A_{J+1:n_t,J}$ with partial pivoting such that $LU = PA_{J+1:n_t,J}$.

whose inverse is a full matrix.

3.2 Two-stage Left-looking Aasen

LAPACK implements another variant [6] of Aasen's algorithm that exploits the memory hierarchy on a modern computer. Unlike the partitioned algorithm that updates the trailing submatrix using block columns, this variant of the algorithm replaces all the element-wise operations of the left-looking column-wise Aasen algorithm [1] with block-wise operations. This is a two-stage algorithm whose first stage reduces the matrix A into a symmetric band matrix T with the band width equal to n_b. Then, the second stage of the algorithm factorizes the band matrix T.

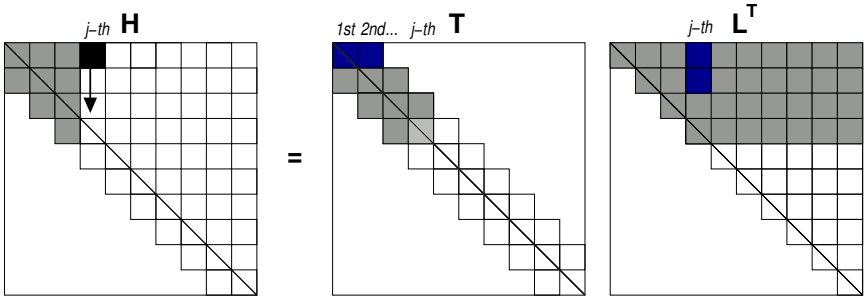
The j-th step of the algorithm computes the j-th block column of $H = TL^T$, and uses the block column to update the panel,

$$A_{j+1:N_t:,j} := A_{J+1:n_t,J} - L_{j+1:n_t,1:j} H_{1:j,j}.$$

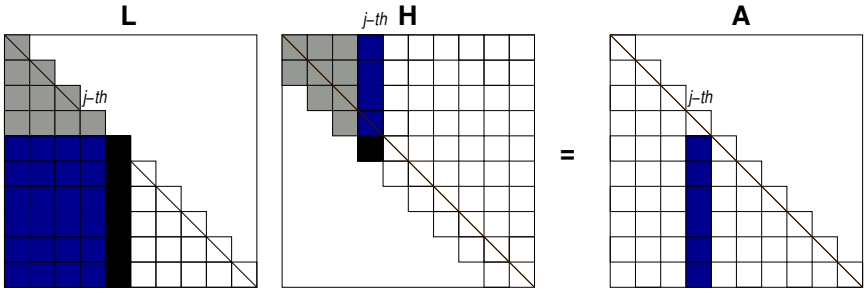
Then, the LU factorization of the panel is computed to generate the $(J+1)$-th block column of L,

$$L_{j+1:n_t,j+1} H_{j+1,j} = P^{(J)} A_{j+1,n_t,j},$$

where $P^{(J)}$ denotes the partial pivoting used for the numerical stability of the factorization. Figure 3 shows the pseudocode of the algorithm, and Figure 4 illustrates the main phases of the algorithm. The algorithm performs about the same number of flops as Bunch-Kaufman. It was



(a) Computation of $H_{:,J}$.



(b) Panel update.

Figure 4: Illustration of Aasen's algorithm.

implemented in LAPACK's `xSYTRF_AA_2STAGE` subroutine and released as a part of LAPACK version 3.7.2.

We compare the right-looking and left-looking algorithms:

- *Pivoting previous columns:* One advantage of the right-looking algorithm is that after updating the trailing submatrix, these previous columns of the factors are not referenced. Hence, the pivots do not have to be applied to these previous columns. In contrast, each step of the left-looking algorithm updates the panel using all of the previous columns, and hence the pivots must be applied to these previous columns.

- *BLAS used for update:* In order to maintain the symmetry during the right-looking trailing submatrix update, LAPACK updates one block column at a time (one column at a time to update the diagonal block). On a multicore system, LAPACK relies on the threaded BLAS to parallelize the factorization, and it has the artificial synchronization point at the end of each BLAS call, and its parallelism is limited to that to update each block column. On the other hand, at each step of the factorization, this left-looking Aasen algorithm uses BLAS-3 to update the panel with all the previous block columns at once (see Figure 1(b)). Though there are write conflicts to update each element of the panel, the update can be performed by single calls to `xSYRK` on the diagonal block and `xGEMM` on the off-diagonal blocks.

4 Implementations

We now describe the LAPACK implementations of the two Aasen algorithms. Both implementations rely on the BLAS and LAPACK as building blocks, and their thread-level parallelization is obtained through the threaded BLAS. The implementation can take either the upper or lower

$$\begin{pmatrix} a_{1,1} & a_{1,2} & & & \\ a_{2,1} & a_{2,2} & a_{2,3} & & \\ a_{3,1} & a_{3,2} & a_{3,3} & a_{3,4} & \\ & a_{4,2} & a_{4,3} & a_{4,4} & a_{4,5} \\ & & a_{5,3} & a_{5,4} & a_{5,5} \end{pmatrix}$$

(a) Full layout.

$$\begin{pmatrix} * & a_{1,2} & a_{2,3} & a_{2,4} & a_{2,5} \\ a_{1,1} & a_{2,2} & a_{3,3} & a_{4,4} & a_{5,5} \\ a_{2,1} & a_{3,2} & a_{4,3} & a_{5,4} & * \\ a_{3,1} & a_{4,2} & a_{5,3} & * & * \end{pmatrix}$$

(b) Band layout.

Figure 5: To reduce the storage cost, LAPACK compactly stores a band matrix in a $(k_\ell + k_u + 1)$-by-n array, where k_ℓ and k_u are the numbers of the subdiagonal and superdiagonal entries within the band ($k_u = 1$ and $k_\ell = 2$ in this illustration). For the two-stage Aasen algorithm, the bandwidth is the block size (i.e., $k_\ell = k_u = n_b$).

triangular part of A, but here, we focus our discussion on the implementation that only accesses the lower-triangular part of the matrix.

4.1 Partitioned Right-looking Aasen

LAPACK's `DSYTRF_AA` implements the partitioned Aasen algorithm, whose interface is shown in Figure 15(a). The subroutine takes the upper or lower triangular part of the symmetric matrix A as an input. Then on exit, the diagonal and subdiagonal entries of the tridiagonal matrix T are stored in the diagonals and the subdiagonals of A while L is stored below (or above) the subdiagonals such that $\mathbf{a}_{j:n,j-1}$ is replaced by $\boldsymbol{\ell}_{j:n,j}$ where $\boldsymbol{\ell}_{:,1}$ is the first column of the identity matrix and is not stored. The only error code that this subroutine may return is for an invalid argument (e.g., a non-positive leading dimension).

This factorization subroutine `xSYTRF_AA` uses workspace of dimension n-by-$(n_b + 1)$ to store the n_b columns of the auxiliary matrix H generated during the panel factorization. Though the default block size is returned by `ILAENV`, the subroutine adjusts the block size if the user did not provide a large enough workspace. Hence, the minimum workspace required is $2n$ (i.e., $n_b = 1$). The default workspace size is returned through the user query based on the default block size, which is 64. The workspace has the extra column for storing $\boldsymbol{\ell}_{n_b}^{(2,1)} t_{n_b+1}$ and updating the trailing submatrix with a single call to `xGEMM` (see Equation (2)). Like the Bunch-Kaufman and the rook pivoting factorization subroutines (`xSYTRF` and `xSYTRF_ROOK`, respectively), during the trailing submatrix update, `xSYTRF_AA` accesses only the triangular part of the matrix and updates one block column at a time (one column at a time to update the diagonal block).

Though it does not need to, `xSYTRF_AA` applies the pivoting to the previous columns of L so that its solver `xSYTRS_AA` can use the standard triangular solver `xTRSM` for the forward and backward substitutions with L. LAPACK version 3.7.0 released the new rook pivoting subroutine `xSYTRF_RK` that also applies the pivoting to the previous columns so that it can use the BLAS-3 based solver `xSYTRS_3`. To solve the linear system with the tridiagonal matrix T, `xSYTRS_AA` calls `xGTSV`. Hence, the subroutine requires workspace of size $3n - 2$ to store the tridiagonal matrix T. Figure 16(a) shows the interface to the triangular solver `xSYTRS_AA` that relies on the partitioned Aasen factorization computed by `xSYTRF_AA`.

4.2 Two-stage Left-looking Aasen

LAPACK's `xSYTRF_AA_2STAGE` implements the two-stage Aasen algorithm, exclusively using BLAS and LAPACK. For instance, `xSYGST` is used to symmetrically apply L^{-1} to compute $T_{J,J}$ on

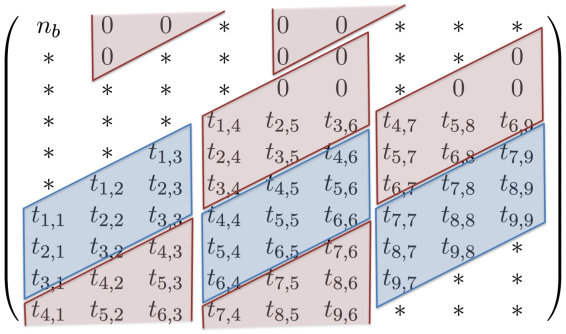


Figure 6: Illustration of T stored in the band format with $n_b = 3$, diagonal and off-diagonal blocks are colored in blue and red, respectively.

line 9, while `xGETRF` is used for the LU panel factorization on line 17.[2] Then, the symmetric band matrix T is factorized using the band LU factorization subroutine `xGBTRF` of LAPACK.

Figure 15(b) shows the interface to `xSYTRF_AA_2STAGE`. Similar to `xSYTRF_AA`, `xSYTRF_AA_2STAGE` stores the lower-triangular matrix L below (or above) the subdiagonal blocks such that $A_{J:n_t,J-1}$ is replaced by $L_{J:n_t,J}$, where $L_{:,1}$ is the first n_b columns of the identity matrix and is not stored. We could store the band matrix T in the diagonal and the first subdiagonal blocks of A such that the lower-triangular part of $A_{J,J}$ and the upper-triangular part of $A_{J+1,J}$ are replaced with those of $T_{J,J}$ and $T_{J+1,J}$, respectively. However, to compute the LU factorization of T, `xGBTRF` requires the additional storage for the fills. Hence, if T is stored on the diagonals and subdiagonals of A, then it needs to be copied to separate workspace before computing its LU factorization. To avoid the data copy, during the first stage of factorization, we directly store T in separate storage using LAPACK band layout (see Figure 5). To store the band matrix T of bandwidth n_b, while leaving n_b space for the fills during its LU factorization, the workspace `TB` of size $(3n_b + 1)n$ is used. Though the default block size is returned by `ILAENV`, the subroutine adjusts the block size if the user did not provide large enough workspace. Hence, the minimum size of `TB` is $4n$ (i.e., $n_b = 1$). The workspace size is returned through the user query based on the default block size, which is 192. The subroutine also takes workspace of dimension $n \cdot n_b$ to store the current block column of H.

For each column of T, we leave n_b space on the top for `xGBTRF` to store the fills (see Figure 6). Hence, we can call a BLAS subroutine on the blocks of T stored in band layout by shifting the leading dimension during the first stage of the factorization. For example, we can copy $A_{J,J}$ in the full layout into $T_{J,J}$ in the band layout by

```
      CALL DLACPY( 'Full', KB, KB, A( J*NB+1, J*NB+1 ), LDA,
     $                  TB( TD+1 + (J*NB+1)*LDTB ), LDTB-1 )
```

where the leading dimension `LDTB` of `TB` is set to be $3n_b + 1$, and `TD` is set to be $2n_b$ and identifies the position of the diagonal entries in the band layout.

The $(J+1)$-th block column of the L is stored in $A_{J:n_t,J-1}$. Instead of using `xSYRK` and `xGEMM` to separately update the diagonal and off-diagonal blocks of $L_{J:n_t,J}$, we use a single `xGEMM` call to

[2]For the symmetric complex factorization by `ZSYTRF_AA_2STAGE`, LAPACK does not provide `ZSYGST`. Hence, we expand the diagonal block $A_{J,J}$ to a full block in $T_{J,J}$ and then symmetrically apply the triangular solves by calling `ZTRSM` twice to compute $T_{J,J} := L_{J,J}^{-1} T_{J,J} L_{J,J}^{-T}$. Finally, the block $T_{J,J}$ is expanded to a symmetric full matrix ensuring the symmetry of the diagonal block. Also when the upper triangular part of A is stored, to factor the panel, we transpose and copy the block row $A_{J,J+1:n_t}$ into a workspace before calling `xGETRF`.

$$A := \begin{pmatrix} 0 & & & & & 2 \\ & n & n & & & \\ & n & 0 & n-1 & & \\ & & n-1 & 0 & \ddots & \\ & & & \ddots & \ddots & 3 \\ 2 & & & & 3 & 0 \end{pmatrix}$$

(a) `synth-1`

$$\Psi := \left(\begin{array}{ccccc|ccc} \psi_1 & & & & & & & \\ \psi_2 & \psi_1 & & & & & & \\ \psi_2 & \psi_2 & \psi_1 & & & & & \\ \vdots & \vdots & \ddots & \ddots & & & & \\ \psi_2 & \psi_2 & \cdots & \cdots & \psi_1 & & & \\ \hline \psi_2 & \psi_2 & \psi_2 & \cdots & \psi_2 & \psi_3 & & \\ \vdots & \vdots & \vdots & & \vdots & 1 & \psi_3 & \\ & & & & & 1 & 1 & \psi_3 \\ & & & & & \vdots & \vdots & \ddots \end{array}\right)$$

(b) `synth-2` with $A := C \cdot \Psi$ where $\cdot$ is the element-wise multiply, C is `random`, and Ψ is generated using $\psi_1 = 10^{-18}$, $\psi_2 = 10^{-10}$, $\psi_3 = 10^{-1}$, and the leading diagonal block is of dimension 50×50.

Figure 7: Synthetic test matrices.

update the whole panel (both the diagonal and off-diagonal blocks) at once.

There is a variant of the formulation to update $A_{J,J}$, i.e.,

$$A_{J,J} = A_{J,J} - L_{J,1:(J-1)} W_{1:(J-1)} - W_{1:(J-1)}^T L_{J,1:(J-1)}, \tag{3}$$

where $W_K = \frac{1}{2} T_{K,K} L_{J,K}^T + T_{K,K+1} L_{J,K+1}^T$ [6]. This variant forms the symmetric diagonal block $A_{J,J}$ using the symmetric update `xSYRK`, while `xSYTRF_AA_2STAGE` performs the unsymmetric update using `xGEMM` (line 7 of Figure 3). In our previous studies [5], both variants obtain about the same backward errors, while the variant (3) requires slightly more flops (with `xGEADD`, a none-standard BLAS operation) and the additional n-by-n_b workspace.

Figure 16(b) shows the interface to the triangular solver `xSYTRS_AA_2STAGE` based on the two-stage Aasen factorization computed by `xSYTRF_AA_2STAGE`.

5 Experimental Results

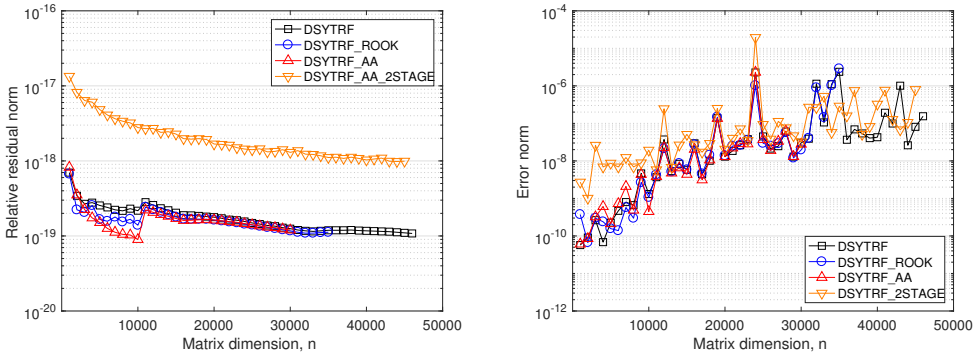
We now compare the stability and performance of the symmetric indefinite solvers in LAPACK.

5.1 Experiment Setups

We conducted all the experiments on the following two computers:

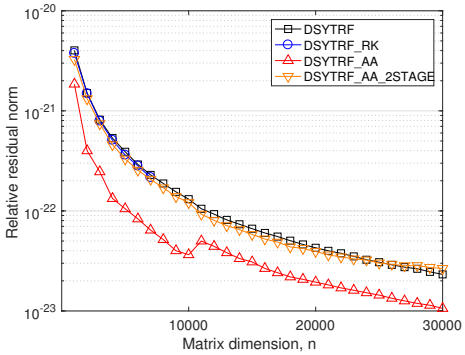
- **Haswell**: This machine has two ten-core Intel Xeon E5-2650 v3 (Haswell) CPUs with 2.30GHz, and 64GB of main memory. The codes are compiled using the Intel `icc` version 17.0.4 compiler with the `-O3` optimization flag and linked with LAPACK version 3.8.0 and the threaded BLAS of the Intel Math Kernel Library (MKL) version 17.0.4.

- **Knights Landing**: This machine has the 7250 variant of 68 cores Intel Xeon Phi Many Integrated Core (MIC) CPUs. There are two types of memory used in KNL: a larger 96 GB of DDR4-2400 memory providing up to 115.2 GB/s of bandwidth and a small 16 GB of MCDRAM providing up to 490 GB/s of sustained bandwidth through the 2-D mesh interconnect. The memory bandwidth per KNL core is approximately 11 GB/sec for small thread counts. In these experiments we used the 16 GB high speed MCDRAM as a NUMA

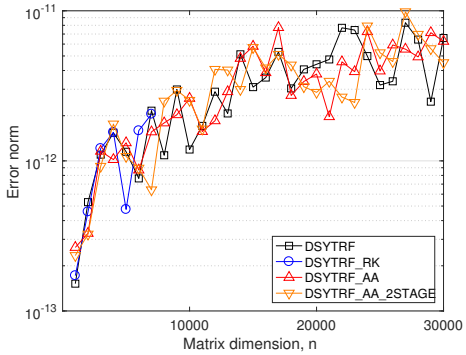


(a) Backward errors with `random`.

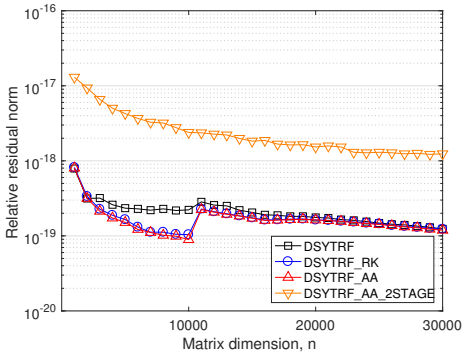
(b) Forward errors with `random`.



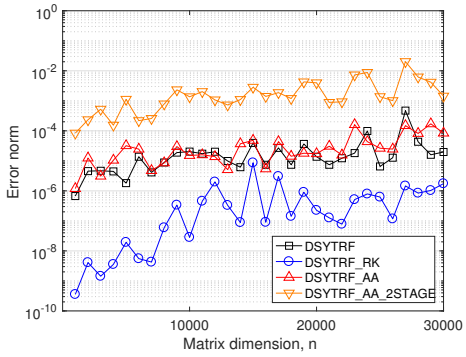
(c) Backward errors with `synth-1`.



(d) Forward errors with `synth-1`.



(e) Backward errors with `synth-2`.

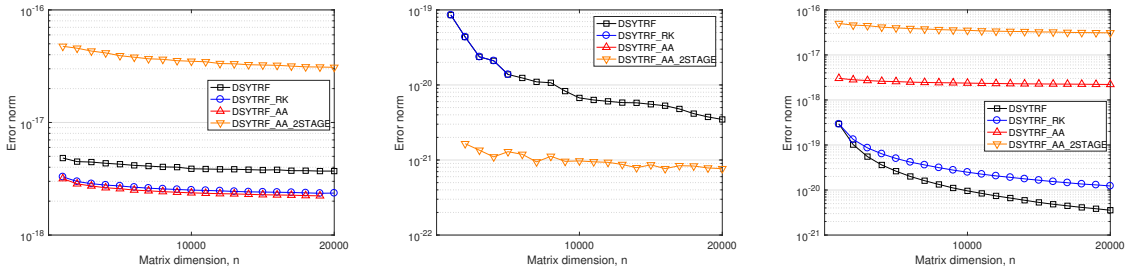


(f) Forward errors with `synth-2`.

Figure 8: Backward errors $\|\mathbf{b} - A\widehat{\mathbf{x}}\|/(n\|A\|\|\widehat{\mathbf{x}}\|)$ and forward errors $\|\mathbf{x} - \widehat{\mathbf{x}}\|$ with the computed and exact solution vectors $\widehat{\mathbf{x}}$ and $\mathbf{x}$ (default block sizes of 192 for two-stage Aasen's and 64 for the rest).

node (flat mode) instead of transparent cache for DRAM (cache mode). The code is compiled with `gcc` version 7.0.1 with the `-O3` optimization flag and linked with LAPACK version 3.8.0 and threaded BLAS of MKL version 17.2.174.

All the results are in real double precision, and in this paper, we focus on the following three types of matrices to demonstrate the stability and performance of the solvers:



(a) `random`. (b) `synth-1`. (c) `synth-2`.

Figure 9: Factorization errors $\|PAP^T - LDL^T\|/(n\|A\|)$ (default block sizes of 192 for two-stage Aasen's and 64 for the rest).

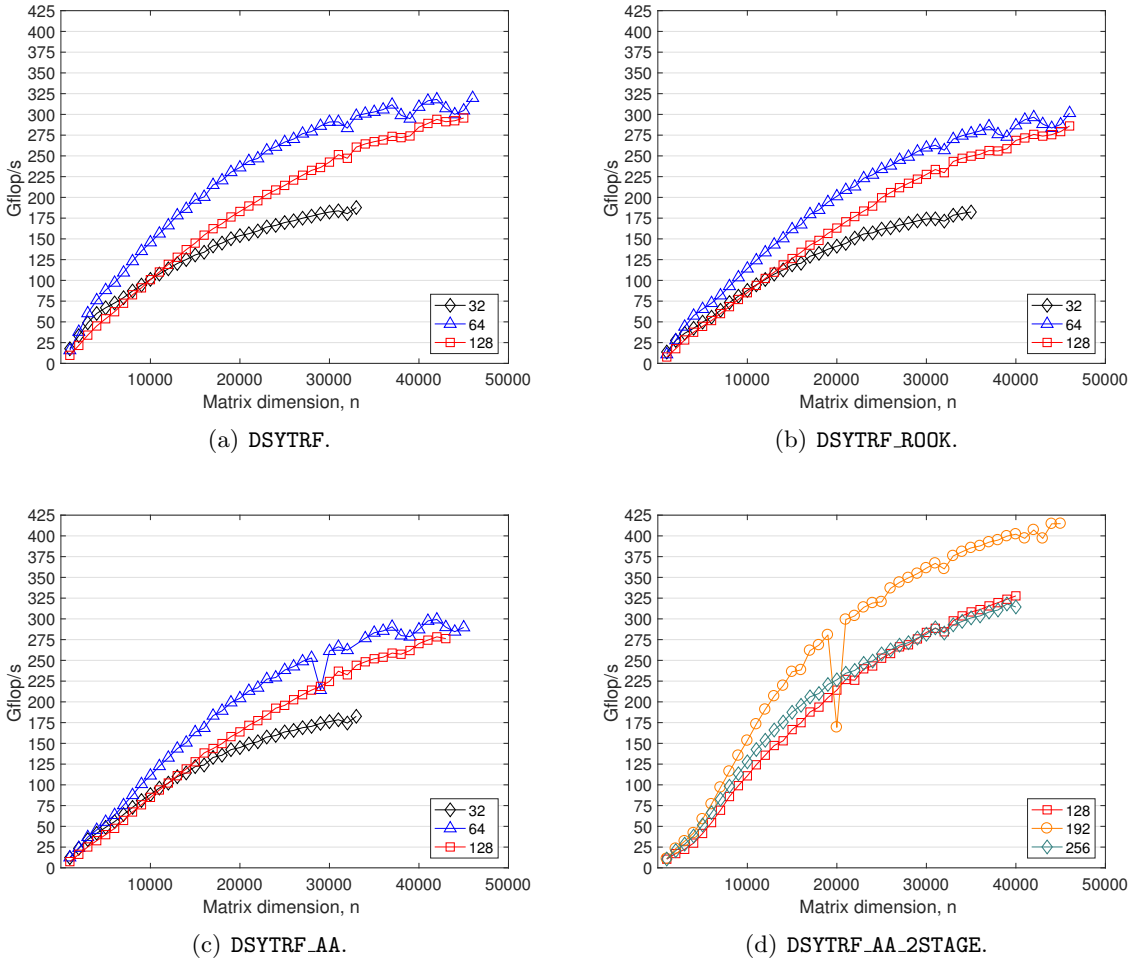
- `random`: the random matrices are generated using LAPACK `DLARNV` with uniform distribution in the interval $(0, 1)$.
- `synth-1`: These matrices, shown in Figure 7(a), are designed to stress the performance of rook pivoting and lead to the full scan of the trailing submatrix at each step of the rook pivoting [4].
- `synth-2`: The matrices, shown in Figure 7(b), are designed to stress the numerical stability of the Bunch-Kaufman algorithm, generating large growth factors in L [4].

Both the performance and numerical results (e.g., Gflop/s and error norm) are the maximum of three separate runs.

5.2 Numerical Results

Figure 8 shows the backward and forward errors of different LAPACK symmetric indefinite solvers. The backward errors of two-stage Aasen depend linearly on the block size [6], and with the block size of $n_b = 192$, the backward errors were about one order of magnitude greater than those of the other algorithms for `random` and `synth-2` matrices. For `synth-1` matrices, two-stage Aasen obtained similar backward errors as Bunch-Kaufman or rook pivoting, while the partitioned Aasen algorithm obtained slightly smaller errors. Overall, all the algorithms obtained small backward errors (i.e., $\|A\widehat{x} - b\| = O(n\epsilon\|A\|\|\widehat{x}\|)$). For both `random` and `synth-1`, all the algorithms obtained similar forward errors. For `synth-2`, rook pivoting obtained smaller forward errors compared with Bunch-Kaufman or the partitioned Aasen, whose forward errors were smaller than those of the two-stage Aasen algorithm.

Figure 9 shows the factorization errors of the different algorithms. For two-stage Aasen, we computed the factorization error from the first stage of the factorization i.e., $\frac{\|PAP^T - LTL^T\|}{n\|A\|}$. There are variation in the relative sizes of the factorization errors using different algorithms. However, the factorization errors of two-stage Aasen were often greater than those of the other algorithms, except for the special cases with the `synth-1` matrices, where due to the particular sparsity structure of the matrices (see Figure 7(a)), two-stage Aasen leads to the empty panels, hence leading to the lower-triangular factor L that is an identity matrix, except for the first panel. For this reason, the partitioned Aasen algorithm obtained zero factorization errors for the `synth-1` matrices. Overall, all the algorithms obtain small factorization errors. Extensive theoretical discussion on these algorithms can be found in [8, 10, 6].



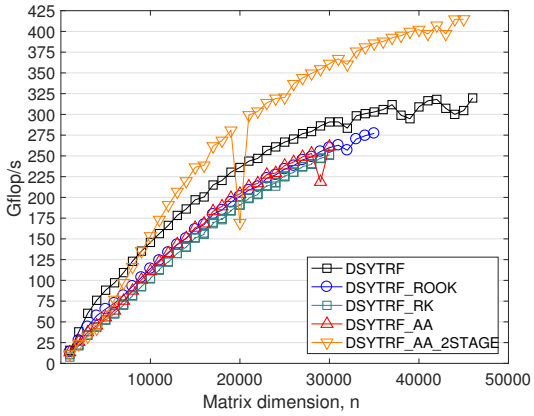
(a) DSYTRF. (b) DSYTRF_ROOK.

(c) DSYTRF_AA. (d) DSYTRF_AA_2STAGE.

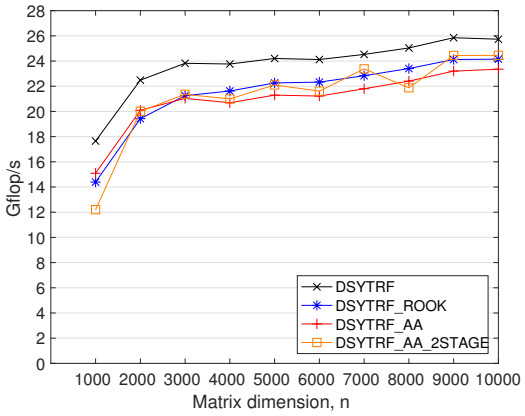
Figure 10: Effects of block size on the factorization performance on Haswell, using the default block sizes of 192 for xSYTRF_AA_2STAGE and 64 for the rest.

5.3 Performance Results

We focus on random matrices which provide representative performance of the subroutines. Figure 10 shows the performance of four LAPACK symmetric indefinite factorization subroutines with different block sizes n_b on our Haswell testbed. The three algorithms, Bunch-Kaufman, rook pivoting, and partitioned Aasen, perform well with the default block size 64. On the other hand, two-stage Aasen algorithm prefers a larger block size (e.g., $n_b = 192$). These are the default block sizes returned by ILAENV (i.e., 192 for two-stage Aasen, and 64 for the rest). Figure 11 then compares the performance of the factorization using the default block sizes. Bunch-Kaufman implemented by DSYTRF often performs best (due to the simple updating scheme and the fewer row and column swaps due to 2-by-2 pivots). Only when the matrix sizes are large enough and with multiple threads, the new two-stage DSYTRF_AA_2STAGE shows the performance improvement. Figure 12 shows similar results for two synthetic matrices. For synth-1, the rook pivoting obtains much lower performance because the matrix is designed to cause the algorithm to scan the whole trailing submatrix to look for each pivot. On the other hand, for synth-1, two-stage Aasen obtains higher performance than the other algorithms because its panels are empty except

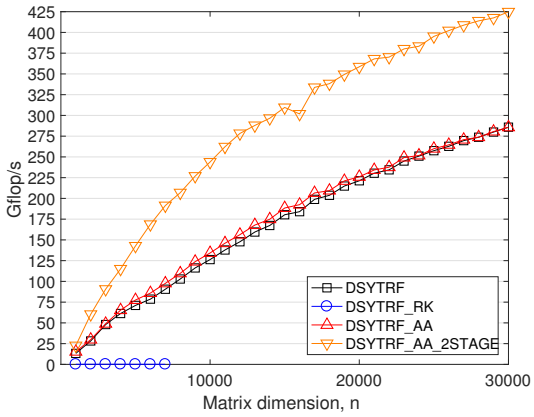


(a) `random` with 20 threads.

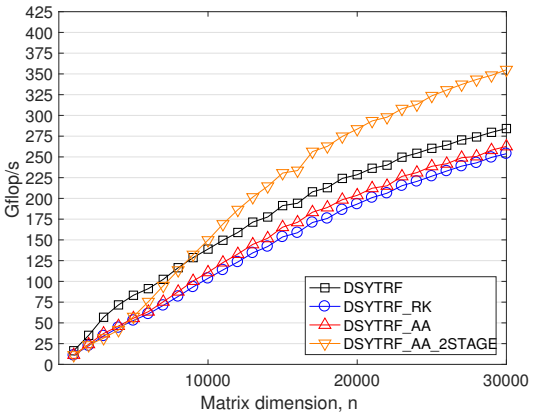


(b) `random` with 1 thread.

Figure 11: Performance on Haswell (default $n_b = 192$ for `DSYTRF_AA_2STAGE` or 64 for the rest).



(a) `synth-1` with 20 threads.

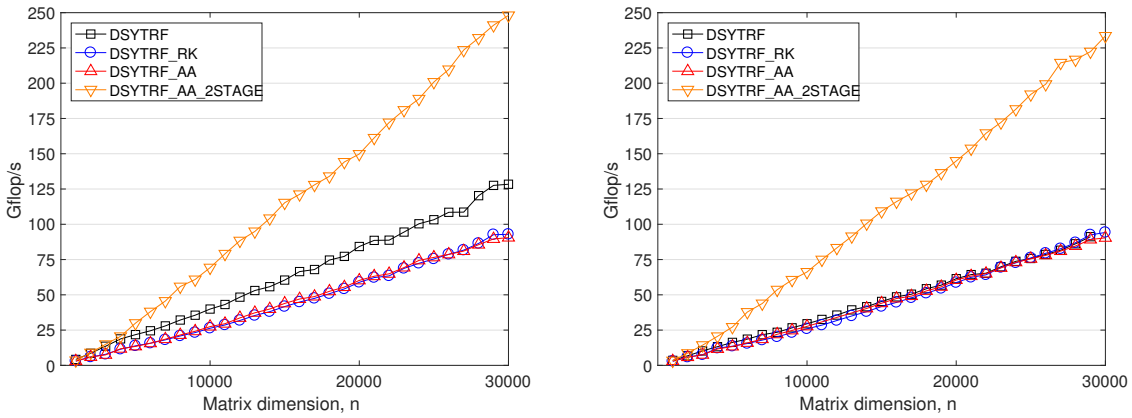


(b) `synth-2` with 20 threads.

Figure 12: Performance on Haswell (default $n_b = 192$ for `DSYTRF_AA_2STAGE` or 64 for the rest).

for the first panel, and the first stage of the algorithm performs almost no floating operations. Figure 13 shows the performance on our KNL testbed, where the two-stage algorithm obtained greater speedups, utilizing the manycore architecture more effectively.

Figure 14 shows the breakdown of the symmetric indefinite solvers. The solution time is dominated by the trailing submatrix update and panel factorization. Compared with the other two subroutines, `DSYTRF_RK` and `DSYTRF_AA` spend more time swapping rows because these two subroutines apply the pivots to all the previous columns. `DSYTRF_AA_2STAGE` reduces both the submatrix update and panel factorization time, and obtains the performance improvement. For the factorization time by `DSYTRF_AA_2STAGE`, "other" is mostly the time spent computing the auxiliary matrix H and the block tridiagonal matrix T.



(a) `DSYTRF` with 68 threads.

(b) `DSYSV` with 68 threads.

Figure 13: Performance on KNL (default $n_b = 192$ for `DSYTRF_AA_2STAGE` or 64 for the rest).

6 Conclusion

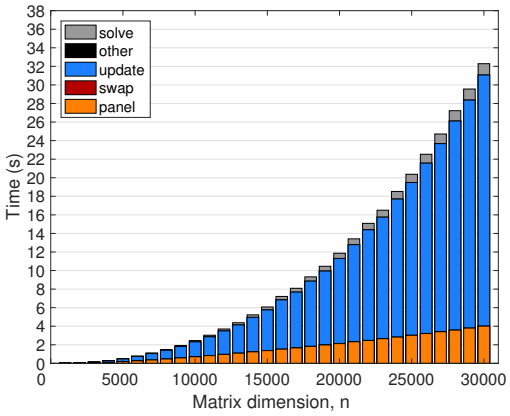
We described two new symmetric indefinite linear solvers in LAPACK. The solvers are based on two variants of Aasen's algorithm: the partitioned right-looking and two-stage left-looking algorithms. Our performance results show that while Bunch-Kaufman is often the fastest in many cases, the two-stage variant of Aasen's algorithm may provide a performance improvement for a large-scale matrix on a multicore system.

The backward error of the two-stage Aasen algorithm linearly depends on the block size. Since the optimal performance of the two-stage algorithm is often obtained using a relatively large block size (e.g, $n_b = 192$), its backward error can be an order of magnitude greater than that of Bunch-Kaufman. An iterative refinement may be an option to recover an equivalent backward error from the two-stage algorithm.

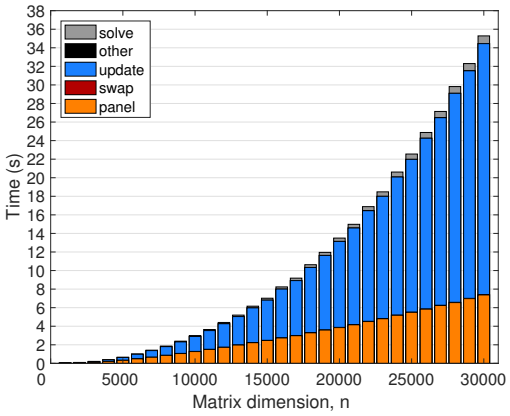
The tiled implementation of the two-stage Aasen algorithm has been recently released through the PLASMA library that uses OpenMP tasks [11]. Compared with the LAPACK implementation, the task-based implementation of PLASMA may be more effectively utilizing the manycore architecture because the different phases of the solver may be executed in parallel on different cores (e.g., the first and second stages of the factorization, and the forward substitution, can be executed in parallel).

Appendix

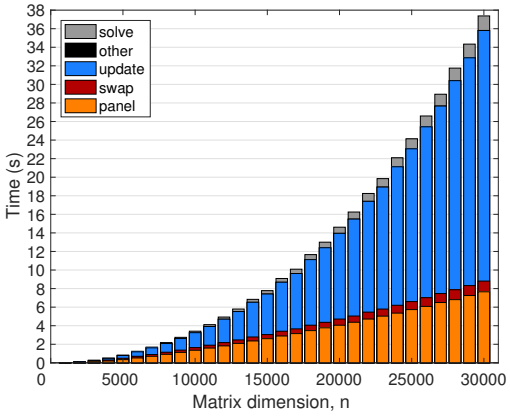
Figure 15(a) shows the interface to LAPACK's `DSYTRF_AA` subroutine that implements the partitioned Aasen algorithm. Figure 15(b) shows the interface to `xSYTRF_AA_2STAGE`. Figure 16(a) shows the interface to the solver based on the partitioned Aasen factorization computed by `xSYTRF_AA`, while Figure 16(b) shows the interface to the solver based on the two-stage Aasen factorization computed by `xSYTRF_AA_2STAGE`.



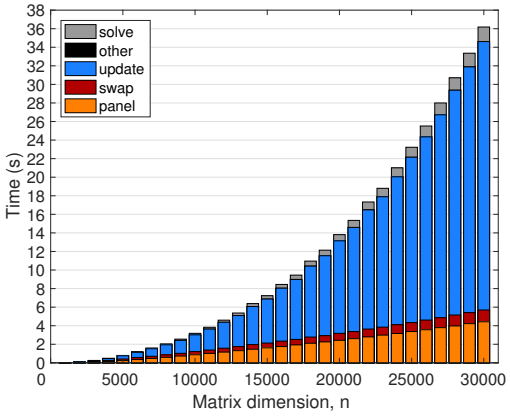
(a) `DSYTRF`.



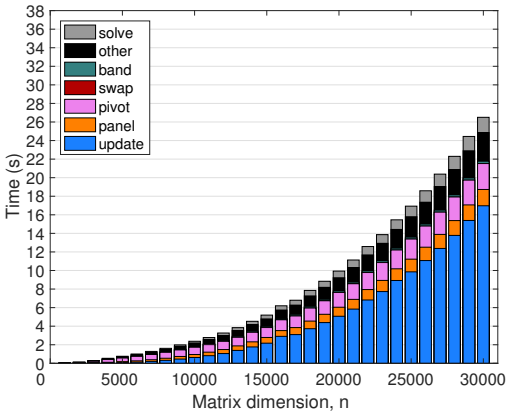
(b) `DSYTRF_ROOK`.



(c) `DSYTRF_RK`.



(d) `DSYTRF_AA`.



(e) `DSYTRF_AA_2STAGE`.

Figure 14: Breakdown of factorization time on Haswell (using the default block sizes).

Acknowledgment

The authors would like to thank the members of the LAPACK project, especially, James Demmel and Igor Kozachenko at the University of California, Berkeley, and Sven Hammarling at the

```
*       SUBROUTINE DSYTRF_AA( UPLO, N, A, LDA, IPIV, WORK, LWORK, INFO )
*
*       .. Scalar Arguments ..
*       CHARACTER          UPLO
*       INTEGER            N, LDA, LWORK, INFO
*       ..
*       .. Array Arguments ..
*       INTEGER            IPIV( * )
*       DOUBLE PRECISION   A( LDA, * ), WORK( * )
*       ..
*
*  Arguments:
*  ==========
*
*> \param[in] UPLO
*>   UPLO is CHARACTER*1
*>   = 'U':  Upper triangle of A is stored;
*>   = 'L':  Lower triangle of A is stored.
*>
*> \param[in] N
*>   N is INTEGER
*>   The order of the matrix A.  N >= 0.
*>
*> \param[in,out] A
*>   A is DOUBLE PRECISION array, dimension (LDA,N)
*>   On entry, the symmetric matrix A.  If UPLO = 'U', the leading
*>   N-by-N upper triangular part of A contains the upper
*>   triangular part of the matrix A, and the strictly lower
*>   triangular part of A is not referenced.  If UPLO = 'L', the
*>   leading N-by-N lower triangular part of A contains the lower
*>   triangular part of the matrix A, and the strictly upper
*>   triangular part of A is not referenced.
*>
*>   On exit, the tridiagonal matrix is stored in the diagonals
*>   and the subdiagonals of A just below (or above) the diagonals,
*>   and L is stored below (or above) the subdiaonals, when UPLO
*>   is 'L' (or 'U').
*>
*> \param[in] LDA
*>   LDA is INTEGER
*>   The leading dimension of the array A.  LDA >= max(1,N).
*>
*> \param[out] IPIV
*>   IPIV is INTEGER array, dimension (N)
*>   On exit, it contains the details of the interchanges, i.e.,
*>   the row and column k of A were interchanged with the
*>   row and column IPIV(k).
*>
*> \param[out] WORK
*>   WORK is DOUBLE PRECISION array, dimension (MAX(1,LWORK))
*>   On exit, if INFO = 0, WORK(1) returns the optimal LWORK.
*>
*> \param[in] LWORK
*>   LWORK is INTEGER
*>   The length of WORK.  LWORK >= MAX(1,2*N). For optimum performance
*>   LWORK >= N*(1+NB), where NB is the optimal blocksize.
*>
*>   If LWORK = -1, then a workspace query is assumed; the routine
*>   only calculates the optimal size of the WORK array, returns
*>   this value as the first entry of the WORK array, and no error
*>   message related to LWORK is issued by XERBLA.
*>
*> \param[out] INFO
*>   INFO is INTEGER
*>   = 0:  successful exit
*>   < 0:  if INFO = -i, the i-th argument had an illegal value.
```

(a) DSYTRF_AA.

```
      SUBROUTINE DSYTRF_AA_2STAGE( UPLO, N, A, LDA, TB, LTB, IPIV,
     $                             IPIV2, WORK, LWORK, INFO )
*
*     .. Scalar Arguments ..
      CHARACTER          UPLO
      INTEGER            FLAG, N, LDA, LDTB, LWORK, INFO
*     ..
*     .. Array Arguments ..
      INTEGER            IPIV( * ), IPIV2( * )
      DOUBLE PRECISION   A( LDA, * ), TB( LDTB, *), WORK( * )
*
*  Arguments:
*  ==========
*
*> \param[in] UPLO
*>   UPLO is CHARACTER*1
*>   = 'U':  Upper triangle of A is stored;
*>   = 'L':  Lower triangle of A is stored.
*>
*> \param[in] N
*>   N is INTEGER
*>   The order of the matrix A.  N >= 0.
*>
*> \param[in,out] A
*>   A is DOUBLE PRECISION array, dimension (LDA,N)
*>   On entry, the symmetric matrix A.  If UPLO = 'U', the leading
*>   N-by-N upper triangular part of A contains the upper
*>   triangular part of the matrix A, and the strictly lower
*>   triangular part of A is not referenced.  If UPLO = 'L', the
*>   leading N-by-N lower triangular part of A contains the lower
*>   triangular part of the matrix A, and the strictly upper
*>   triangular part of A is not referenced.
*>
*>   On exit, L is stored below (or above) the subdiaonal blocks,
*>   when UPLO  is 'L' (or 'U').
*>
*> \param[in] LDA
*>   LDA is INTEGER
*>   The leading dimension of the array A.  LDA >= max(1,N).
*>
*> \param[out] TB
*>   TB is DOUBLE PRECISION array, dimension (LTB)
*>   On exit, details of the LU factorization of the band matrix.
*>
*> \param[in] LDTB
*>   The leading dimension of the array TB. LTB >= (3*NB+1)*N.
*>
*> \param[out] IPIV
*>   IPIV is INTEGER array, dimension (N)
*>   On exit, it contains the details of the interchanges, i.e.,
*>   the row and column k of A were interchanged with the
*>   row and column IPIV(k).
*>
*> \param[out] IPIV2
*>   IPIV is INTEGER array, dimension (N)
*>   On exit, it contains the details of the interchanges, i.e.,
*>   the row and column k of T were interchanged with the
*>   row and column IPIV(k).
*>
*> \param[out] WORK
*>          WORK is DOUBLE PRECISION workspace of size LWORK
*>
*> \param[in] LWORK
*>          The size of WORK. LWORK >= N*NB.
*>
*> \param[out] INFO
*>   INFO is INTEGER
*>   = 0:  successful exit
*>   < 0:  if INFO = -i, the i-th argument had an illegal value.
```

(b) DSYTRF_AA_2STAGE.

Figure 15: The definitions of Aasen's factorization subroutines.

University of Manchester for the valuable comments.

References

[1] J. Aasen. On the reduction of a symmetric matrix to tridiagonal form. *BIT*, 11:233–242, 1971.

```
*       SUBROUTINE DSYTRS_AA( UPLO, N, NRHS, A, LDA, IPIV, B, LDB,
*                             WORK, LWORK, INFO )
*
*       .. Scalar Arguments ..
*       CHARACTER          UPLO
*       INTEGER            N, NRHS, LDA, LDB, LWORK, INFO
*       ..
*       .. Array Arguments ..
*       INTEGER            IPIV( * )
*       DOUBLE PRECISION   A( LDA, * ), B( LDB, * ), WORK( * )
*       ..
*
*  Arguments:
*  ==========
*
*> \param[in] UPLO
*>          UPLO is CHARACTER*1
*>          Specifies whether the details of the factorization
*>          are stored as an upper or lower triangular matrix.
*>          = 'U':  Upper triangular, form is A = U*T*U**T;
*>          = 'L':  Lower triangular, form is A = L*T*L**T.
*>
*> \param[in] N
*>          N is INTEGER
*>          The order of the matrix A.  N >= 0.
*>
*> \param[in] NRHS
*>          NRHS is INTEGER
*>          The number of right hand sides, i.e., the number of
*>          columns of the matrix B.  NRHS >= 0.
*>
*> \param[in] A
*>          A is DOUBLE PRECISION array, dimension (LDA,N)
*>          Details of factors computed by DSYTRF_AA.
*>
*> \param[in] LDA
*>          LDA is INTEGER
*>          The leading dimension of the array A.  LDA >= max(1,N).
*>
*> \param[in] IPIV
*>          IPIV is INTEGER array, dimension (N)
*>          Details of the interchanges as computed by DSYTRF_AA.
*>
*> \param[in,out] B
*>          B is DOUBLE PRECISION array, dimension (LDB,NRHS)
*>          On entry, the right hand side matrix B.
*>          On exit, the solution matrix X.
*>
*> \param[in] LDB
*>          LDB is INTEGER
*>          The leading dimension of the array B.  LDB >= max(1,N).
*>
*> \param[in] WORK
*>          WORK is DOUBLE array, dimension (MAX(1,LWORK))
*>
*> \param[in] LWORK
*>          LWORK is INTEGER, LWORK >= MAX(1,3*N-2).
*>
*> \param[out] INFO
*>          INFO is INTEGER
*>          = 0:  successful exit
*>          < 0:  if INFO = -i, the i-th argument had an illegal
*>                value
```

(a) `DSYTRS_AA`.

```
*       SUBROUTINE DSYTRS_AA_2STAGE( UPLO, N, NRHS, A, LDA, TB, LTB,
*                                    IPIV, IPIV2, B, LDB, INFO )
*
*       .. Scalar Arguments ..
*       CHARACTER          UPLO
*       INTEGER            N, NRHS, LDA, LTB, LDB, INFO
*       ..
*       .. Array Arguments ..
*       INTEGER            IPIV( * ), IPIV2( * )
*       DOUBLE PRECISION   A( LDA, * ), TB( * ), B( LDB, * )
*       ..
*
*  Arguments:
*  ==========
*
*> \param[in] UPLO
*>          UPLO is CHARACTER*1
*>          Specifies whether the details of the factorization
*>          are stored as an upper or lower triangular matrix.
*>          = 'U':  Upper triangular, form is A = U*T*U**T;
*>          = 'L':  Lower triangular, form is A = L*T*L**T.
*>
*> \param[in] N
*>          N is INTEGER
*>          The order of the matrix A.  N >= 0.
*>
*> \param[in] NRHS
*>          NRHS is INTEGER
*>          The number of right hand sides, i.e., the number of
*>          columns of the matrix B.  NRHS >= 0.
*>
*> \param[in] A
*>          A is DOUBLE PRECISION array, dimension (LDA,N)
*>          Details of factors computed by DSYTRF_AA_2STAGE.
*>
*> \param[in] LDA
*>          LDA is INTEGER
*>          The leading dimension of the array A.  LDA >= max(1,N).
*>
*> \param[out] TB
*>          TB is DOUBLE PRECISION array, dimension (LTB)
*>          Details of factors computed by DSYTRF_AA_2STAGE.
*>
*> \param[in] LTB
*>          The size of the array TB. LTB >= 4*N.
*>
*> \param[in] IPIV
*>          IPIV is INTEGER array, dimension (N)
*>          Details of the interchanges as computed by
*>          DSYTRF_AA_2STAGE.
*>
*> \param[in] IPIV2
*>          IPIV2 is INTEGER array, dimension (N)
*>          Details of the interchanges as computed by
*>          DSYTRF_AA_2STAGE.
*>
*> \param[in,out] B
*>          B is DOUBLE PRECISION array, dimension (LDB,NRHS)
*>          On entry, the right hand side matrix B.
*>          On exit, the solution matrix X.
*>
*> \param[in] LDB
*>          LDB is INTEGER
*>          The leading dimension of the array B.  LDB >= max(1,N).
*>
*> \param[out] INFO
*>          INFO is INTEGER
*>          = 0:  successful exit
*>          < 0:  if INFO = -i, the i-th argument had an illegal
*>                value
```

(b) `DSYTRS_AA_2STAGE`.

Figure 16: The definitions of Aasen's solver subroutines.

[2] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. D. Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. *LAPACK Users' Guide*. Society for Industrial and Applied Mathematics, 3rd edition, 1999. LAPACK software is available at `http://www.netlib.org/lapack` and `https://github.com/Reference-LAPACK`.

[3] E. Anderson and J. Dongarra. Evaluating block algorithm variants in LAPACK. In *Proceedingsof the 4th Conference on Parallel Processing for Scientific Computing*, pages 3–8,

1989.

[4] C. Ashcraft, R. Grimes, and J. Lewis. Accurate symmetric indefinite linear equation solvers. *SIAM J. Matrix Anal. Appl.*, 20:513–561, 1998.

[5] G. Ballard, D. Becker, J. Demmel, J. Dongarra, A. Druinsky, I. Peled, O. Schwartz, S. Toledo, and I. Yamazaki. Implementing a blocked Aasen's algorithm with a dynamic scheduler on multicore architectures. In *Proceedings of the IEEE International Symposium on Parallel and Distributed Processing*, pages 895–907, 2013.

[6] G. Ballard, D. Becker, J. Demmel, J. Dongarra, A. Druinsky, I. Peled, O. Schwartz, S. Toledo, and I. Yamazaki. A communication avoiding symmetric indefinite factorization. *SIAM. J. Matrix Anal. Appl.*, 35(4):1364–1406, 2014.

[7] J. Bunch and L. Kaufman. Some stable methods for calculating inertia and solving symmetric linear systems. *Mathematics of Computation*, 31(137):163–179, 1977.

[8] N. Higham. *Accuracy and Stability of Numerical Algorithms*. Society for Industrial and Applied Mathematics, 2nd edition, 2002.

[9] B. Parlett and J. Reid. On the solution of a system of linear equations whose matrix is symmetric but not definite. *BIT*, 10:386–397, 1970.

[10] M. Rozložník, G. Shklarski, and S. Toledo. Partitioned triangular tridiagonalization. *ACM Trans. Math. Softw.*, 37(4):1–16, 2011.

[11] I. Yamazaki, J. Kurzak, P. Wu, M. Zounon, and J. Dongarra. Symmetric indefinite linear solver using OpenMP task on manycore architectures. *IEEE Transactions on Parallel and Distributed Systems*, 2016, submitted. PLASMA software is available at `https://bitbucket.org/icl/plasma`.