Implementing a Blocked Aasen's Algorithm with a Dynamic Scheduler on Multicore Architectures

Grey Ballard*, Dulceneia Becker†, James Demmel*, Jack Dongarra†‡§
Alex Druinsky¶, Inon Peled¶, Oded Schwartz*, Sivan Toledo¶, and Ichitaro Yamazaki†[1]

†*University of Tennessee, Knoxville, USA*
¶*Tel-Aviv University, Tel-Aviv, Israel*
**University of California, Berkeley, Berkeley, USA*
‡*Oak Ridge National Laboratory, Oak Ridge, USA*
§*University of Manchester, Manchester, UK*

*ballard@cs.berkeley.edu, dbecker7@eecs.utk.edu, demmel@cs.berkeley.edu, dongarra@eecs.utk.edu
alexdrui@post.tau.ac.il, inon.peled@gmail.com, odedsc@cs.berkeley.edu, stoledo@tau.ac.il, iyamazak@eecs.utk.edu*

***Abstract*—Factorization of a dense symmetric indefinite matrix is a key computational kernel in many scientific and engineering simulations. However, there is no scalable factorization algorithm that takes advantage of the symmetry and guarantees numerical stability through pivoting at the same time. This is because such an algorithm exhibits many of the fundamental challenges in parallel programming like irregular data accesses and irregular task dependencies. In this paper, we address these challenges in a tiled implementation of a blocked Aasen's algorithm using a dynamic scheduler. To fully exploit the limited parallelism in this left-looking algorithm, we study several performance enhancing techniques; e.g., parallel reduction to update a panel, tall-skinny LU factorization algorithms to factorize the panel, and a parallel implementation of symmetric pivoting. Our performance results on up to 48 AMD Opteron processors demonstrate that our implementation obtains speedups of up to 2.8 over MKL, while losing only one or two digits in the computed residual norms.**

I. Introduction

Many scientific and engineering simulations require the solution of the dense symmetric indefinite linear systems of equations of the form

$$Ax = b, \tag{1}$$

where A is an n-by-n dense symmetric indefinite matrix, b is a given right-hand-side, and x is the solution vector to be computed. Nonetheless, there is no scalable factorization algorithm which takes advantage of the symmetry and has a provable numerical stability. The main reason for this is that stable factorization requires pivoting which is difficult to parallelize efficiently. To address this, in this paper, we develop an efficient implementation of a so-called blocked Aasen's algorithm that is proposed very recently [1].

[1]Corresponding author

To solve (1), Aasen's algorithm [2] factorizes A into an $\mathrm{LTL}^{\mathrm{T}}$ decomposition of the form

$$PAP^T = LTL^T, \tag{2}$$

where P is a permutation matrix, L is a unit lower-triangular matrix, and T is a symmetric tridiagonal matrix. Aasen's algorithm takes advantage of the symmetry in A and performs $\frac{1}{3}n^3 + O(n^2)$ floating-point operations (flops), which is a half of the flops needed for an LU factorization of A. Furthermore, it is backward stable subject to a growth factor. Once the factorization is computed, the solution x is computed by successively solving the linear systems with the matrices L, T, and L^T.

To exploit the memory hierarchy on a modern computer, a partitioned-version of Aasen's algorithm was recently proposed [3]. This algorithm first factorizes a panel in a left-looking fashion, and then uses BLAS-3 operations to update the trailing submatrix in a right-looking way. In comparison with a standard column-wise algorithm, this partitioned algorithm slightly increases the operation count, performing $\frac{1}{3}(1 + \frac{1}{n_b})n^3 + O(n^2 n_b)$ flops with a block size of n_b. However, BLAS-3 can be used to perform most of these flops; i.e., $\frac{1}{3}(1 + \frac{1}{n_b})n^3$ flops. Since the BLAS-3 operations have higher ratios of flop counts over communication volumes than the BLAS-2 or BLAS-1 operations do, this partitioned algorithm is shown to significantly shorten the factorization time on modern computers, where data transfer is much more expensive than floating-point operations [3]. However, the panel factorization is still based on BLAS-1 and BLAS-2 operations. As a result, this panel factorization often obtains only a small fraction of the peak performance on modern computers, and could become the bottleneck, especially in a parallel implementation.

The blocked-version of Aasen's algorithm, which we study in this paper, was proposed to avoid this bottleneck at

the panel factorization. It computes an LTL^T factorization of A, where T is a banded matrix (instead of tridiagonal) with its half-bandwidth being equal to the block size n_b, and then uses a banded matrix solver to compute the solution. In this blocked algorithm, each panel can be factorized using an existing LU factorization algorithm, such as recursive LU [4], [5], [6], [7] and communication-avoiding LU (CALU) [8], [9]. In comparison with the panel factorization algorithm used in the partitioned Aasen's algorithm, these LU factorization algorithms reduce communication, and hence are expected to speed up the whole factorization process.

In this paper, we implement this blocked Aasen's algorithm on multicore architectures, and analyze its parallel performance. Our implementation follows the framework of PLASMA[2] and uses a dynamic scheduler called QUARK[3]. To efficiently utilize a large number of cores in parallel, all the existing factorization routines in PLASMA update the trailing submatrix in right-looking fashion. Hence, our implementation in this paper is not only the first parallel implementation of the blocked Aasen's algorithm, but it is also the first implementation of a left-looking algorithm in PLASMA. In order to fully exploit the limited parallelism in this left-looking algorithm, we study several performance enhancing techniques; e.g., parallel reduction to update the panel, recursive LU and CALU for panel factorization, and parallel symmetric pivoting. Our performance results on up to 48 AMD Opteron processors demonstrate that a left-looking algorithm can be implemented efficiently on a multicore architecture. In addition, our numerical results show that in comparison with the widely-used stable algorithm (the Bunch-Kaufman algorithm of LAPACK), our implementation loses only one or two digits in the computed residual norms when a recursive LU is used on a panel. We also present numerical results to show the different numerical behavior of the algorithm when a CALU is used on the panel. In our concluding remarks, we briefly discuss a potential root of this numerical difference, but more detailed analysis will be in our future reports.

The remainder of the paper is organized as follows: after surveying the related work in Section II, we describe the blocked Aasen's algorithm in Section III. Then, in Sections IV and V, we present our parallel implementation and performance results, respectively. Finally, in Section VI, we conclude with final remarks. Table I summarizes the notation used in this paper. Some notation is reused in column-wise and block-wise algorithms (e.g., n denotes either the number of columns or the number of block-columns in the column-wise or block-wise algorithm, respectively), but the meaning of the notation should be clear from the context. Our discussion here assumes the lower-triangular part of the symmetric matrix A is stored, but it can easily be extended to the case where the upper-triangular part of A is stored.

n	dimension of A (number of diagonal blocks in A)
n_b	block size
a_{ij}	(i,j)-th element of A
A_{ij}	(i,j)-th block of A
$\mathbf{a}_{:,\mathbf{j}}$	j-th column of A
$\mathbf{a}_{\mathbf{i},:}$	i-th row of A
$A_{:,j}$	j-th block-column of A
$A_{i,:}$	i-th block-row of A
$A_{i:m,j:n}$	i-th to m-th (block) rows and j-th to n-th (block) columns
I	identity matrix
e_j	j-th column of I

Table I
NOTATION USED IN THIS PAPER.

II. Related Work

In Section V, we compare the performance and stability of the blocked Aasen's algorithm with those of the following two state-of-the-art factorization algorithms.

A. LAPACK – Bunch-Kaufman algorithm

LAPACK[4] is a set of dense linear algebra routines that is extensively used in many scientific and engineering simulations. For solving a symmetric indefinite system (1), LAPACK implements a partitioned LDL^T factorization with the Bunch-Kaufman algorithm [10], [11] that computes

$$PAP^T = LDL^T, \quad (3)$$

where D is a block diagonal matrix with either 1-by-1 or 2-by-2 diagonal blocks. This algorithm is backward stable subject to growth factors [12], and performs the same number of flops as the column-wise Aasen's algorithm, i.e. $\frac{1}{3}n^3 + O(n^2)$ flops. A serial implementation of the partitioned Aasen's algorithm is shown to be as efficient as the Bunch-Kaufman algorithm of LAPACK on a single core [3].

To select a pivot at each step of the Bunch-Kaufman algorithm, up to two columns of the trailing submatrix must be scanned, where the index of the second column corresponds to the row index of the element with the maximum modulus in the first column. Since only the lower-triangular part of the submatrix is stored, this leads to irregular data accesses and irregular task dependencies. As a result, it is difficult to develop an efficient parallel implementation of this algorithm. For instance, on multicores, LAPACK obtains its parallelism using threaded BLAS, which leads to an expensive fork-join programming paradigm.

B. PLASMA – Random Butterfly Transformation

PLASMA provides a set of dense linear algebra routines based on tiled algorithms that break a given algorithm into fine-grained computational tasks that operate on small

[2]http://icl.utk.edu/plasma/
[3]http://icl.utk.edu/quark/
[4]http://www.netlib.org/lapack/

square submatrices called tiles. Since each tile is stored contiguously in memory and fits in a local cache memory, this algorithm can take advantage of the hierarchical memory architecture on the modern computer. Furthermore, by dynamically scheduling the tasks as soon as all of their dependencies are resolved, PLASMA can exploit a fine-grained parallelism and utilize a large number of cores.

Randomized algorithms are gaining popularity in linear algebra algorithms since they can often exploit more parallelism than corresponding deterministic algorithms can [13]. To solve (1), PLASMA extends a randomization technique developed for the LU factorization [14] to symmetric indefinite systems [15], [16]. Namely, it uses a multiplicative preconditioner by means of random matrices called recursive butterfly matrices U_k:

$$\left(U_d^T U_{d-1}^T \dots U_1^T\right) A \left(U_1 U_2 \dots U_d\right),$$

where

$$U_k = \begin{pmatrix} B_1 & & \\ & \ddots & \\ & & B_{2^{k-1}} \end{pmatrix}, B_k = \frac{1}{\sqrt{2}} \begin{pmatrix} R_k & S_k \\ R_k & -S_k \end{pmatrix},$$

and R_k and S_k are random diagonal matrices. As a result, the original matrix A is transformed into a matrix that is sufficiently random so that, with a probability close to one, pivoting is not needed.

This random butterfly transformation (RBT) requires only $2dn^2 + O(n)$ flops in comparison with the $\frac{1}{3}n^3 + O(n^2)$ flops required for the factorization. In practice, $d = 2$ achieves satisfying accuracy, and this is the default setup used in PLASMA. Since A is factorized without pivoting after RBT, it allows a scalable factorization of a dense symmetric indefinite matrix.

The main drawback of this method is reliability. There is no theory demonstrating its deterministic stability. Though it may fail in certain cases, numerical tests showed that with iterative refinement, it is reliable for many test cases, including pathological ones [15]. Furthermore, since iterative refinement is required, the failure of the method can be signaled without extra computation.

Besides the dense factorization on multicores, a parallel factorization of a dense symmetric indefinite matrix with pivoting has been studied on distributed-memory systems in [17]. Furthermore, many implementations of the sparse $\mathrm{LDL^T}$ factorization have been proposed on distributed and shared memory architectures (see [18] and the references therein).

III. Variations of Aasen's Algorithm

In this section, we describe column-wise right- and left-looking algorithms to compute the $\mathrm{LTL^T}$ factorization (2) (Sections III-A and III-B), and a blocked left-looking version of the algorithm (Section III-C). To compute the $\mathrm{LTL^T}$ factorization, we use an intermediate Hessenberg matrix H which is defined as $H = TL^T$.

A. Right-looking Algorithm

By the 1-st column of the equation $A = LTL^T$, we have

$$t_{1,1} = a_{1,1} \quad \text{and} \quad \boldsymbol{\ell}_{2:n,2}\ t_{2,1} = \mathbf{v}, \tag{4}$$

where $\mathbf{v} = a_{2:n,1} - \boldsymbol{\ell}_{2:n,1}\ t_{1,1}$. Hence, given the first columns of A and L, we can compute the 1-st column of T and the 2-nd column of L (in our implementation, we let $\boldsymbol{\ell}_{1:n,1} = e_1$). Moreover, from the trailing submatrix of the equation $A = LTL^T$, we have

$$\begin{aligned} A_{2:n,2:n} &= \boldsymbol{\ell}_{2:n,2}\ t_{1,2}\ \boldsymbol{\ell}_{2:n,1}^T + \boldsymbol{\ell}_{2:n,1} t_{1,1} \boldsymbol{\ell}_{2:n,1}^T \\ &\quad + \boldsymbol{\ell}_{2:n,1}\ t_{2,1}\ \boldsymbol{\ell}_{2:n,2}^T + L_{2:n,2:n} T_{2:n,2:n} L_{2:n,2:n}^T. \end{aligned}$$

Hence, if we update the trailing submatrix as

$$\begin{aligned} A_{2:n,2:n} \ -= \ & \boldsymbol{\ell}_{2:n,1}\ t_{2,1}\ \boldsymbol{\ell}_{2:n,2}^T + \boldsymbol{\ell}_{2:n,1} t_{1,1} \boldsymbol{\ell}_{2:n,1}^T \\ & + \boldsymbol{\ell}_{2:n,2}\ t_{2,1}\ \boldsymbol{\ell}_{2:n,1}^T, \end{aligned} \tag{5}$$

then the $\mathrm{LTL^T}$ factorization of A can be computed by recursively computing the $\mathrm{LTL^T}$ factorization of $A_{2:n,2:n}$:

$$A_{2:n,2:n} = L_{2:n,2:n} T_{2:n,2:n} L_{2:n,2:n}^T.$$

It is possible to update the trailing submatrix using two rank-one updates as in

$$A_{2:n,2:n} \ -= \ \boldsymbol{\ell}_{2:n,1} \mathbf{h}_{2:n,1}^T + \boldsymbol{\ell}_{2:n,2}\ t_{2,1}\ \boldsymbol{\ell}_{2:n,1}^T, \tag{6}$$

where $\mathbf{h}_{2:n,1} = \boldsymbol{\ell}_{2:n,2} t_{2,1} + \boldsymbol{\ell}_{2:n,1} t_{1,1}$. Alternatively, it can be updated using a symmetric rank-two update as in

$$A_{2:n,2:n} \ -= \ \mathbf{w}_{2:n,1} \boldsymbol{\ell}_{2:n,1}^T + \boldsymbol{\ell}_{2:n,1} \mathbf{w}_{2:n,1}^T, \tag{7}$$

where $\mathbf{w}_{2:n,1} = \boldsymbol{\ell}_{2:n,2} t_{2,1} + \frac{1}{2} \boldsymbol{\ell}_{2:n,1} t_{1,1}$.

A numerical issue comes when $\mathbf{v}$ is scaled so that $\ell_{2,2}$ is one in (4). An attractive feature of this algorithm is that the numerical instability can be avoided by simply using the element of $\mathbf{v}$ with the largest modulus as the pivot. This right-looking algorithm is equivalent to the Parlett-Reid algorithm [19].

The above algorithm performs a total of $\frac{2}{3}n^3 + O(n^2)$ flops, which is twice as many as the flops needed to compute the $\mathrm{LDL^T}$ factorization (3). This is because in (3), D is block diagonal, and it requires only one rank-one update to update the trailing submatrix using each column of L.

B. Left-looking Algorithm

Given the first j columns of L and the first $j-1$ columns of T, the left-looking Aasen's algorithm first computes the j-th column of H; i.e., from the j-th column of the equation $H = TL^T$, we have, for $k = 1, 2, \dots, j-1$,

$$h_{k,j} = t_{k,k-1} \ell_{j,k-1}^T + t_{k,k} \ell_{j,k}^T + t_{k,k+1} \ell_{j,k+1}^T.$$

```
1:  for j = 1 to n do
2:     Compute H_{1:(j-1),j} and update T_{j,j} (see Figure 2)
3:     if j > 1 then
4:        T_{j-1,j} = T_{j,j-1}^T
5:     end if
6:     T_{j,j} = L_{j,j}^{-1} T_{j,j} L_{j,j}^{-T}
7:     if j < n then
8:        // Compute (j,j)-th block of H
9:        H_{j,j} = T_{j,j} L_{j,j}^T
10:       if j > 1 then
11:          H_{j,j} = H_{j,j} + T_{j,j-1} L_{j,j-1}^T
12:       end if
13:       // Extract L_{:,j+1} of L
14:       L_{(j+1):n,j+1} = A_{(j+1):n,j} - L_{(j+1):n,1:j} H_{1:j,j}
15:       [L_{(j+1):n,j+1}, H_{j+1,j}, P_j] = LU(L_{(j+1):n,j+1})
16:       // Apply pivots to other part of matrices
17:       L_{(j+1):n,1:j} := P_j L_{(j+1):n,1:j}
18:       A_{(j+1):n,(j+1):n} := P_j A_{(j+1):n,(j+1):n} P_j^T
19:       // Extract T_{J+1,J}
20:       T_{j+1,j} = H_{j+1,j} L_{j,j}^{-T}
21:    end if
22: end for
```

Figure 1. Blocked left-looking Aasen's algorithm.

Then, the (j,j)-th element of H is computed from the (j,j)-th element of equation $A = LH$,

$$h_{j,j} = \ell_{j,j}^{-1}\left(a_{j,j} - \sum_{k=1}^{j} \ell_{j,k} h_{k,j}\right).$$

Next, we obtain $t_{j,j}$ from the (j,j)-th element of the equation $H = TL^T$, i.e.,

$$t_{j,j} = \left(h_{j,j} - t_{j,j-1}\, \ell_{j,j-1}^T\right) \ell_{j,j}^{-T}.$$

In addition, from the j-th column of the equation $A = LH$, if we let

$$\mathbf{v} = \mathbf{a}_{(j+1):n,j} - \sum_{k=1}^{j} \boldsymbol{\ell}_{(j+1):n,k} h_{k,j},$$

then we can extract the $(j+1,j)$-th element of H and the $(j+1)$-th column of L by

$$h_{j+1,j} = v_1 \quad \text{and} \quad \boldsymbol{\ell}_{(j+1):n,j+1} = \frac{\mathbf{v}}{v_1}. \tag{8}$$

Finally, from the $(j+1,j)$-th element of the equation $H = TL^T$, we get

$$t_{j+1,j} = h_{j+1,j}\ell_{j,j}^{-T}.$$

A numerical issue comes when scaling $\mathbf{v}$ in the second equation of (8). Just like in the right-looking algorithm, numerical stability is maintained by simply using the element of $\mathbf{v}$ with the largest modulus as the pivot. This is the algorithm described by Aasen in [2].

Approach 1:

```
1:  // Compute H_{1:(j-1),j}
2:  for k = 1 to j - 1 do
3:     U_k = T_{k,k} L_{j,k}^T
4:     V_k = T_{k,k+1} L_{j,k+1}^T
5:     H_{k,j} = U_k + V_k
6:     if k > 1 then
7:        H_{k,j} = H_{k,j} + T_{k,k-1} L_{j,k-1}^T
8:     end if
9:  end for
10: // Update T_{j,j}
11: W_{1:(j-1)} = (1/2) U_{1:(j-1)} + V_{1:(j-1)}
12: T_{j,j} = A_{j,j} - L_{j,1:(j-1)} W_{1:(j-1)} - W_{1:(j-1)}^T L_{j,1:(j-1)}^T
```

Approach 2:

```
1:  // Compute H_{1:(j-1),j}
2:  for k = 1 to j - 1 do
3:     H_{k,j} = T_{k,k} L_{j,k}^T
4:     H_{k,j} = H_{k,j} + T_{k,k+1} L_{j,k+1}^T
5:     if k > 1 then
6:        H_{k,j} = H_{k,j} + T_{k,k-1} L_{j,k-1}^T
7:     end if
8:  end for
9:  // Update T_{j,j}
10: T_{j,j} = A_{j,j} - L_{j,1:(j-1)} H_{1:(j-1),j}
11: if j > 1 then
12:    T_{j,j} = T_{j,j} - L_{j,j} T_{j,j-1} L_{j,j-1}^T
13: end if
```

Figure 2. Compute $H_{1:(j-1),j}$ and update $T_{j,j}$.

This left-looking algorithm performs a total of $\frac{1}{3}n^3 + O(n^2)$ flops, hence requiring only half of the flops needed by the right-looking algorithm. This is because the left-looking algorithm takes advantage of the fact that the j-th symmetric pivoting can be applied any time before the trailing submatrix is updated using the j-th column of L. Specifically, the left-looking algorithm applies the pivoting to the original matrix A and then extracts the next column from it. If the symmetric pivoting and the submatrix update must be alternately applied, then the left-looking algorithm is not possible because the j-th update must be applied to the trailing submatrix before applying the $(j+1)$-th symmetric pivot. Furthermore, in order to keep the trailing submatrix symmetric, two rank-one updates (6) or (7) are needed, doubling the number of flops.[5]

[5] A right-looking algorithm could update $A_{2:n,2:n}$ by $A_{2:n,2:n} -= \boldsymbol{\ell}_{2:n,1}\mathbf{h}_{2:n,1}^T$ and compute $A_{2:n,2:n} = L_{2:n,2:n}H_{2:n,2:n}^T$. However, the symmetry is lost in the trailing submatrix, and the whole submatrix including both its upper- and lower-triangular parts must be updated leading to $\frac{2}{3}n^3 + O(n^2)$ flops.

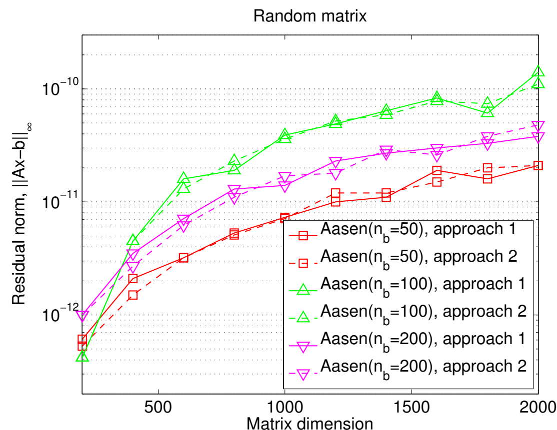


Figure 3. Solution accuracy using two updating schemes in Figure 2.

C. Blocked Left-looking Algorithm

Either the right- or left-looking algorithm above can be extended to a blocked algorithm. Even though the right-looking algorithm exhibits more parallelism, it requires twice as many flops as the left-looking algorithm does. Hence, in this paper, we focus on the left-looking algorithm.

If we replace all the element-wise operations with block-wise operations in the left-looking algorithm in Section III-B, we then have a blocked version of the algorithm: From the j-th block column of $H = TL^T$ and the (j, j)-th block of $A = LH$, we have for $k = 1, 2, \ldots, j-1$,

$$H_{k,j} = T_{k,k-1}L_{j,k-1}^T + T_{k,k}L_{j,k}^T + T_{k,k+1}L_{j,k+1}^T \quad (9)$$

and

$$H_{j,j} = L_{j,j}^{-1}\left(A_{j,j} - \sum_{k=1}^{j} L_{j,k}H_{k,j}\right). \quad (10)$$

Then, we obtain $T_{j,j}$ from the (j, j)-th block of $H = TL^T$, i.e.,

$$T_{j,j} = \left(H_{j,j} - T_{j,j-1}L_{j,j-1}^T\right) L_{j,j}^{-T}. \quad (11)$$

Next from the j-th block column of $A = LH$, we can extract the $(j + 1)$-th block column of L,

$$P_j^T L_{(j+1):n,j+1}H_{j+1,j} = \mathrm{LU}(V),$$

where

$$V = A_{(j+1):n,j} - \sum_{k=1}^{j} L_{(j+1):n,k}H_{k,j},$$

and $L_{(j+1):n,j+1}$ and $H_{j+1,j}$ are the L and U factors of V with the partial pivoting P_j. This partial pivoting is then applied to the corresponding part of the submatrices; i.e.,

$$A_{j+1:n,j+1} := P_j A_{(j+1):n,(j+1):n} P_j^T$$

and

$$L_{(j+1):n,1:j} := P_j L_{(j+1):n,1:j}.$$

Finally, from the $(j + 1, j)$-th block of $H = TL^T$, we have

$$T_{j+1,j} = H_{j+1,j}L_{j,j}^{-T}.$$

Unfortunately, the above procedure is unstable because the symmetric $T_{j,j}$ is computed through a sequence of unsymmetric expressions as in (11). To recover the symmetry, we first substitute $H_{j,j}$ of (10) into (11), and obtain

$$L_{j,j}T_{j,j}L_{j,j}^T = A_{j,j} - \sum_{k=1}^{j-1} L_{j,k}H_{k,j} - L_{j,j}T_{j,j-1}L_{j,j-1}^T. \quad (12)$$

We then substitute $H_{k,j}$ of (9) into (12), and compute $T_{j,j}$ as in

$$L_{j,j}T_{j,j}L_{j,j}^T = A_{j,j} - \sum_{k=1}^{j-1} L_{j,k}W_{j,k} - \sum_{k=1}^{j-1} W_{j,k}^T L_{j,k}^T, \quad (13)$$

where $W_{j,k} = \frac{1}{2}U_{j,k} + V_{j,k}$, $U_{j,k} = T_{k,k}L_{j,k}^T$ and $V_{j,k} = T_{k,k+1}L_{j,k+1}^T$. Finally, from the (j, j)-th block of $H = TL^T$, we compute $H_{j,j}$ by

$$H_{j,j} = T_{j,j}L_{j,j}^T + T_{j,j-1}L_{j,j-1}^T.$$

Figure 1 shows the pseudocode of this blocked algorithm, which was proposed in [1]. We have investigated two approaches to update the diagonal block $T_{j,j}$ (see Figure 2). The second approach does not consider the symmetry while updating the diagonal block, and requires jn_b^2 less flops. Figure 3 shows that these two approaches obtain similar stability on random matrices.[6] For the rest of the paper, we focus on the second approach.

In Figure 3, the norm of the residual increases slightly with the increase in the block size n_b. This agrees with the error bound in [1] that is proportional to both the block size and the number of blocks. We provide more numerical results in Section V.

IV. Implementation

We now describe our implementation of this blocked Aasen's algorithm on multicores. This is done within the framework of PLASMA using the QUARK runtime system to dynamically schedule our computational tasks.

A. Tiled Implementation

As mentioned in Section II, PLASMA is based on tiled algorithms that break a given algorithm into fine-grained computational tasks that operate on small square tiles. Figure 4 shows the pseudocode of our tiled Aasen's algorithm, where most of the computational tasks are performed

[6] The non-symmetric banded solver of LAPACK is used to solve the banded system.

1: **for** $j = 1$ to n **do**
2:   *// Compute off-diagonal blocks of* $H_{:,j}$
3:   **for** $i = 1$ to $j-1$ **do**
4:     GEMM($`N', `T', 1.0, T_{i,i}, L_{j,i}, 0.0, H_{i,j}$)
5:     GEMM($`T', `T', 1.0, T_{i+1,i}, L_{j,i+1}, 1.0, H_{i,j}$)
6:     **if** $i > 1$ **then**
7:       GEMM($`N', `T', 1.0, T_{i,i-1}, L_{j,i-1}, 1.0, H_{i,j}$)
8:     **end if**
9:   **end for**
10:   *// Compute* (j, j)*-th block of* T
11:   LACPY($A_{j,j}, T_{j,j}$)
12:   **for** $i = 1$ to $j-1$ **do**
13:     GEMM($`N', `N', -1.0, L_{j,i}, H_{i,j}, 1.0, T_{j,j}$)
14:   **end for**
15:   **if** $j > 1$ **then**
16:     GEMM($`N', `N', 1.0, L_{j,j}, T_{j,j-1}, 0.0, W_j$)
17:     GEMM($`N', `T', -1.0, W_j, L_{j,j-1}, 1.0, T_{j,j}$)
18:   **end if**
19:   SYGST($T_{j,j}, L_{j,j}$)
20:   **if** $j < n$ **then**
21:     *// Compute* (j, j)*-th block of* H
22:     GEMM($`N', `T', 1.0, T_{j,j}, L_{j,j}, 0.0, H_{j,j}$)
23:     **if** $i > 1$ **then**
24:       GEMM($`N', `T', 1.0, T_{j,j-1}, L_{j,j-1}, 1.0, H_{j,j}$)
25:     **end if**
26:     *// Extract* $(j+1)$*-th block column of* L
27:     **for** $k = 1$ to j **do**
28:       **for** $i = j+1$ to n **do**
29:         GEMM($`N', `N', -1.0, L_{i,k}, H_{k,j}, 1.0, A_{k,j}$)
30:       **end for**
31:     **end for**
32:     $[L_{(j+1):n,j+1}, H_{j+1,j}, P_j] = \text{LU}(A_{(j+1):n,j})$
33:     *// Apply pivots to other part of matrices*
34:     $L_{(j+1):n,1:j} := P_j L_{(j+1):n,1:j}$
35:     $A_{(j+1):n,(j+1):n} := P_j A_{(j+1):n,(j+1):n} P_j^T$
36:     *// Extract* $(j+1, j)$*-th block of* T
37:     LACPY($H_{j+1,j}, T_{j+1,j}$)
38:     TRSM($`R', `T', L_{j,j}, T_{j+1,j}$)
39:   **end if**
40: **end for**

Figure 4. Tiled implementation of blocked left-looking Aasen's algorithm. Here, GEMM($T_A, T_B, \alpha, A, B, \beta, C$) computes $C := \alpha * \text{op}(A) * \text{op}(B) + \beta C$, where $\text{op}(A) = A$ or A^T with $T_A = `N'$ or $`T'$, respectively, and $\text{op}(B)$ is similarly defined with T_B, LACPY(A, B) copies A to B, SYGST(A, L) computes $A := L^{-1} A L^{-T}$, and TRSM($`R', `T', L, A$) computes $A := AL^{-T}$.

using BLAS-3 routines. The only exceptions are the LU panel factorization (Line 32) and the application of the pivots (Lines 33 through 35), which we will discuss in more detail in Sections IV-C and IV-D, respectively.

In the above tiled algorithm, the dependencies among the computational tasks can be represented as a Directed Acyclic Graph (DAG), where each node represents a computational task, and edges between the nodes represent the dependencies among them. Figure 5 shows a DAG for our blocked Aasen's algorithm. At run time, the QUARK runtime system uses this DAG to schedule the tasks as soon as all of their dependencies are resolved. This not only allows us to exploit the fine-grained parallelism of the algorithm, but in many cases, this also results in out-of-order execution of tasks, scheduling the independent tasks from the different stages of factorization at the same time (e.g., computation of $T_{j,j}$ and $L_{(j+1):n,j+1}$). As a result, the idle time of cores can be reduced, allowing us to utilize a large number of cores.

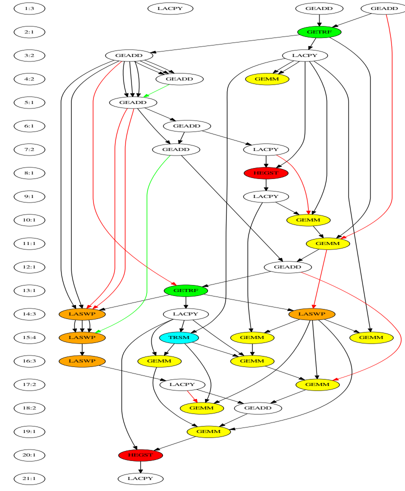


Figure 5. DAG of blocked Aasen's algorithm ($n = 750$ and $n_b = 250$).

B. Parallel Reduction

At the j-th step of the left-looking algorithm, the j-th block column $A_{j:n,j}$ is updated with the previously-factorized block columns (Lines 11 through 18 and Lines 21 through 31 of Figure 4). There is a limited parallelism because multiple updates cannot be accumulated onto the same block at the same time (e.g., $T_{j,j} = A_{j,j} - L_{j,1:(j-1)} H_{1:(j-1),j}$). Furthermore, as j increases, each block must be updated with more previous block columns, while the number of blocks in the current block column decreases. Hence, it is critical that we exploit as much parallelism as possible when updating each block.

Updating a single block with multiple blocks can be considered as a reduction operation, and to exploit the parallelism, we can apply a parallel reduction scheme. Specifically, we first use a separate workspace to accumulate sets of independent updates to a block, and then use binary reduction to accumulate them into the first block of the workspace. Finally, the accumulated update is added to the corresponding destination block. Figure 6 shows the

1: $m_j = \min(j, \frac{m}{n-j})$ *// number of workspaces per block*
2: **for** $k = 1$ to j **do**
3: **for** $i = j+1$ to n **do**
4: $c = \text{mod}(k, m_j) + 1$
5: **if** $k < m_j$ **then**
6: $\beta = 0.0$
7: **else**
8: $\beta = 1.0$
9: **end if**
10: GEMM($`N'$, $`N'$, $-1.0, L_{i,k}, H_{k,j}, \beta, W_{i,c}$)
11: **end for**
12: **end for**
13: *// Binary reduction of workspace into* $A_{(j+1):n,j}$

Figure 6. Left-looking update with a binary-tree reduction, where m is the number of n_b-by-n_b workspaces.

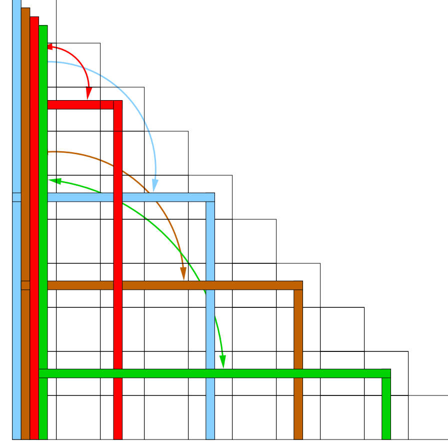


Figure 8. Illustration of symmetric pivoting.

pseudocode of our left-looking update algorithm. This not only exploits the parallelism to accumulate the independent updates onto a single block, but it also allows us to start computing the updates before the destination block is ready. Specifically, while applying the pivots to $A_{j:n,j}$, idle cores can accumulate the updates in the workspace.

The accumulation of two updates using GEADD requires only $O(n_b^2)$ flops in comparison with $O(n_b^3)$ flops needed for computing the update with GEMM. Since scheduling these relatively small accumulation tasks can add significant overhead to the runtime system, we group a set of accumulation tasks into a single task. Also, this parallel reduction is invoked only when the number of tiles in the block column is less than the number of available cores. Hence, round-off errors lead to slightly different factors on different numbers of cores.

C. Parallel Panel Factorization

When a recursive LU is used for the panel factorization, we must complete all the updates on all the blocks in the panel before starting the panel factorization. Furthermore, even with the parallel reduction described in Section IV-B, the updates on the next panel cannot start until the symmetric pivoting from the current panel factorization is applied to the previous columns of L. Hence, there are no other tasks that can be scheduled during the panel factorization. The trace in Figure 7(a) shows that there is synchronization before and after the panel factorization, and that when the number of tiles in the panel is less than the number of cores, some cores are left idle.

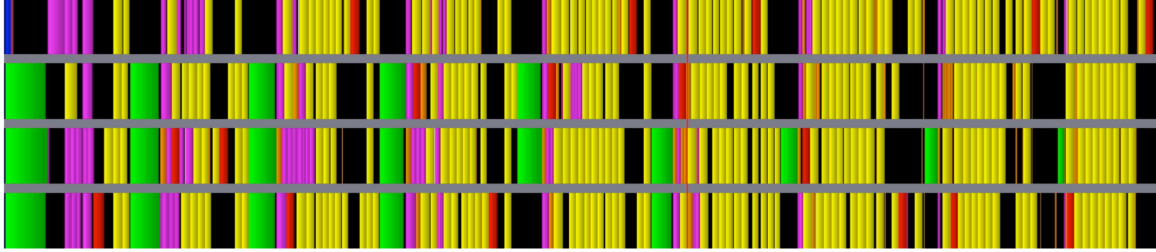
This synchronization can be avoided if we use a CALU on the panel. In this algorithm, as soon as all the updates are applied to a pair of tiles, we can start the LU factorization of the pair. Unfortunately, updating $A_{j:n,j}$ with $L_{j:n,j}$ requires $H_{j,j}$ (see Line 14 of Figure 1), whose computation is often on the critical path of the algorithm. This can be seen in the trace in Figure 7(b), where the panel factorization can start only after the yellow blocks (updating $A_{j:n,j}$ with $L_{j:n,j}$) finish, which in turn can start only after the red block (symmetrically solving $L_{j,j}^{-1} T_{j,j} L_{j,j}^{-T}$) finishes. Hence, in this left-looking algorithm, we often cannot completely overlap the panel factorization with the tasks to update the panel.[7] As a result, though we used the priority and locality features of QUARK to improve its performance, the total factorization time is often slower using CALU than that using the recursive LU (see Section V). We are examining if we can improve the performance of CALU in this left-looking algorithm by reducing more than two tiles at each step of tournament, or by using a static scheduling scheme.

D. Parallel Symmetric Pivoting

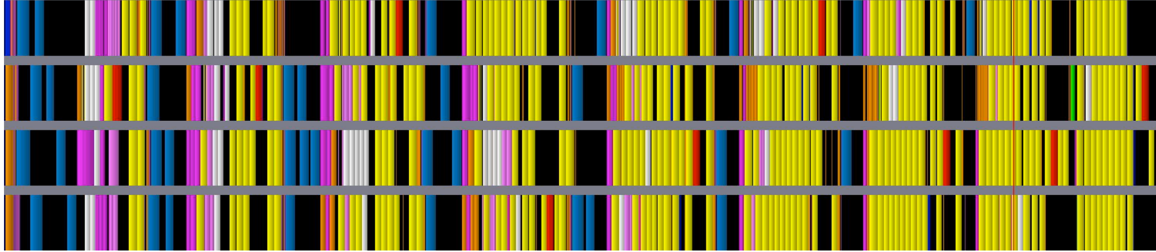
To maintain the symmetry of the trailing submatrix, pivoting must be applied symmetrically to both rows and columns of the submatrix. Since only the lower triangular part of the submatrix is stored, this symmetric pivoting leads to irregular memory accesses and irregular task dependencies (see Figure 8). In comparison, an LU factorization with partial pivoting only swaps the rows, leading to both a fewer applications of a pivot and more regular dependencies. Finally, as described in Section IV-C, in our left-looking algorithm, the pivots must be applied before the updates can be accumulated onto the next panel. Hence, the application of the pivots can lie on the critical path, and must be implemented as efficiently as possible.

As illustrated in Figure 9, we apply symmetric pivoting in two steps. The first step copies all the columns of the trailing submatrix, which need to be swapped, into an n-by-$2n_b$ workspace. At the j-th step, this is done by generating

[7] These traces use a relatively small n for illustration. With a larger n, GEMM becomes more dominant. As a result, the idle time during the updates disappears, but that for the panel factorization tends to stay. For instance, with CALU, many of the panel factorization tasks can be overlapped with updates, but not all. We assign four tiles on each core during the recursive LU.



(a) recursive LU



(b) CALU

Figure 7. Traces of our block-Aasen's algorithm with $n = 2000$ and $n_b = 200$ on four cores. The x-axis represents the run time which is broken down into subroutines: recursive LU (green,) CALU (blue), pivoting of L (orange), pivoting of A (magenta), SYGST (red), GEMM (yellow), TRSM (white), and idle time (black).

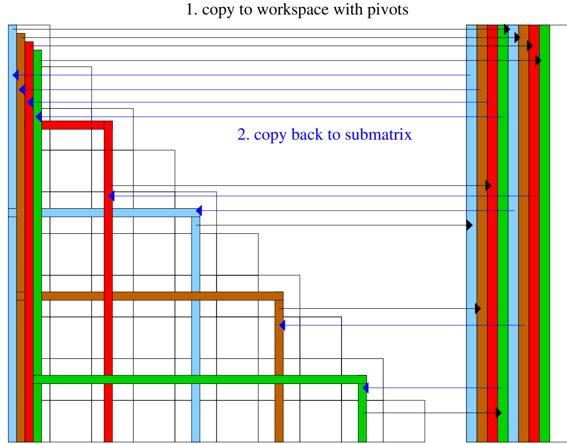


Figure 9. Implementation of parallel symmetric pivoting.

$\frac{n}{n_b} - j$ tasks, each of which independently copies the columns in one of the $\frac{n}{n_b} - j$ block columns of the trailing submatrix. Here, due to the symmetry, the k-th block column consists of the blocks in the k-th block row and those in the k-th block column (i.e., $A_{k,j:k}$ and $A_{(k+1):n,k}$). Then, in the second step, we generate another set of $\frac{n}{n_b} - j$ tasks, each of which copies the columns of the workspace back to a block column of the submatrix after the column pivoting is applied. While the columns are copied into the workspace, we use a global permutation array to apply the row pivoting to each column. This leads to irregular accesses to the workspace. As a result, the first step of copying the columns into the workspace is often slower than the second step of copying them back to the submatrix.

In our implementation of the symmetric pivoting above, each of the tiles in the trailing submatrix is read by two tasks (e.g., at the j-th step, $A_{i,k}$ is read by the $(i-j)$-th and $(k-j)$-th tasks). To reduce the number of accesses to the tiles at the j-th step, the k-th task could access only the k-th block row $A_{k,j:k}$, and copy to the workspace all the required columns from both $A_{k,j:k}$ and $A_{j:k,k}$. In this way, each tile is only read by one task. However, this approach often does not improve the memory access because only the rows and columns to be swapped are accessed, and the accesses to these columns and rows of the tile are irregular and may not exploit any cache reuse. Furthermore, the k-th task processes $k-j$ tiles, leading to a workload imbalance among the tasks. This imbalance can be reduced by generating finer-grained tasks, where each task processes only a fixed number of tiles in the block row. However, this often adds a significant overhead to the runtime system.

In these two implementations of the symmetric pivoting, we must wait for all the columns to be copied into the workspace before copying back to the submatrix because we do not know which tasks are writing to which columns of the workspace. However, this synchronization turned out to be not a significant drawback because as we will describe in the next paragraph, the runtime system can use idle cores during the symmetric pivoting to accumulate some updates for the next block column. At the end, the first implementation gave better performances in many cases, and we used that for our performance studies.

Since the previous block columns of L are needed for the parallel reduction described in Section IV-B, the rows of these block columns should be swapped as soon as possible. Hence, we apply the pivoting to these block columns separately from the application of the symmetric pivoting to the trailing submatrix. This is done by letting each task swap the rows in a previous block column of L. The pivoting of these block columns is scheduled before the symmetric pivoting of the trailing submatrix such that the parallel reduction can start as soon as possible, and is executed on the idle cores while the symmetric pivoting is being applied. In addition, since only the next block column of A is needed at the next step, we prioritize the symmetric pivoting of this next block column over the other block columns that are not needed until the proceeding steps.

E. Storage Requirement

Since the first block column of L is the first n_b columns of the identity matrix, they do not have to be stored. Hence, we store the $(j+1)$-th block column of L in the j-th block column of A. Recall that at the j-th step, we compute the $(j+1)$-th block column of L from the j-th block column of A. Hence, $A_{:,j+1}$ is needed at the $(j+1)$-th step, and it cannot be overwritten with $L_{:,j+1}$ at the end of the j-th step. Then, the banded matrix T can be stored in the main diagonal blocks of A and in the first diagonal blocks below them (i.e., $T_{j,j}$ and $T_{j+1,j}$ can be stored in $A_{j,j}$ and $A_{j+1,j}$, respectively).

Since only the j-th block column of H is needed at the j-th step, we reuse an n-by-n_b workspace to store $H_{j:n,j}$ at each step. In addition, we require $2n_b$-by-n workspace for the symmetric pivoting, and cn_b-by-n workspace for the parallel reduction operation, where c is a user-specified constant (in our experiments, we used $c = 2$). It is possible to use the same workspace for the symmetric pivoting and for the parallel reduction. Since only a small number of tasks from these two different stages of the algorithm can overlap, the performance gain obtained using two different workspaces for these two stages is often small. At the end, the algorithm requires the total workspace of size $O(nn_b)$.

F. Banded Solver

Once the matrix is reduced to a banded form, we use the non-symmetric banded solver of threaded MKL to solved this banded system. This routine performs $O(nn_b^2)$ flops, which is much less than $O(n^3)$ flops needed for the factorization. To improve the parallel performance of the solver and maintain symmetry of the complete factorization, we will explore other options (e.g., [20]) to solve this banded system. By maintaining symmetry, the factorization will compute the inertia of the original matrix (i.e., the number of positive, negative, and zero eigenvalues).

Name	for $j \le i$,
Random	$a_{i,j} = 2 \times rand(n,n)$
Sparse(t)	$a_{i,j} = 2 \times rand(n,n)$, $\frac{nnz}{n^2} = t$
Fiedler	$a_{i,j} = \|i-j\|$
RIS	$a_{i,j} = \frac{1}{2(n-i-j+1.5)}$

Figure 13. Test matrices.

V. Performance Studies

We now analyze the performance of our blocked Aasen's algorithm on up to eight 6-core 2.8MHz AMD Opteron processors. Our code was compiled with the **gcc** 4.1.2 compiler and **-O2** optimization flag, and was linked with the MKL 2011.1.107 library. We have experimented using test matrices from various applications, but in this paper, we present results of the test matrices in Table 13, which demonstrate different numerical behaviors of the algorithms. All of the experiments are in real double precision.

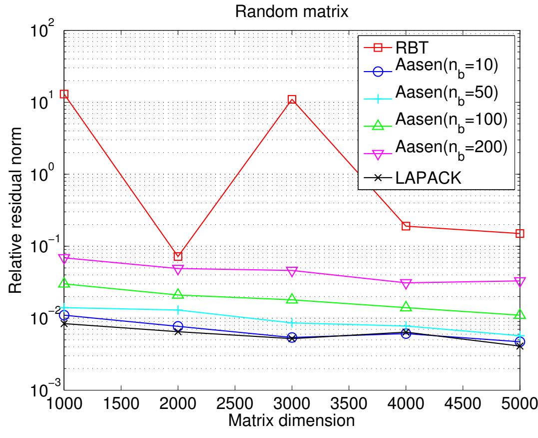
A. Numerical Stability

Figures 10(a), 11(a), and 12 compare the relative residual norms, $\|A\widehat{x} - b\|_\infty / (n\epsilon(\|A\|_\infty \|\widehat{x}\|_\infty + \|b\|_\infty))$, of our blocked Aasen's algorithm using the recursive LU with those of the LDL^T factorization of LAPACK using the Bunch-Kaufman algorithm and with those of PLASMA using RBT. For RBT, we used the default transformation depth of two. The figures show that the residual norm of Aasen's algorithm increases slightly as the block size n_b increases. However, the residual norm of Aasen's algorithm with $n_b = 200$ was competitive with or significantly smaller than that of RBT. Furthermore, the factorization with RBT failed for RIS and Sparse matrix with $t = 0.2$. For the Sparse matrix, RBT will succeed if the transformation depth is increased to make the transformed matrix sufficiently dense, or if the diagonal elements are set to be nonzeros. For RIS, RBT failed even with a greater transformation depth. The oscillation of the residual norm with RBT is expected, but a few iterations of iterative refinement can smooth out the residual norm. After a few iterative refinements, the residual norms of Aasen's algorithm should be as small as that computed by LAPACK.

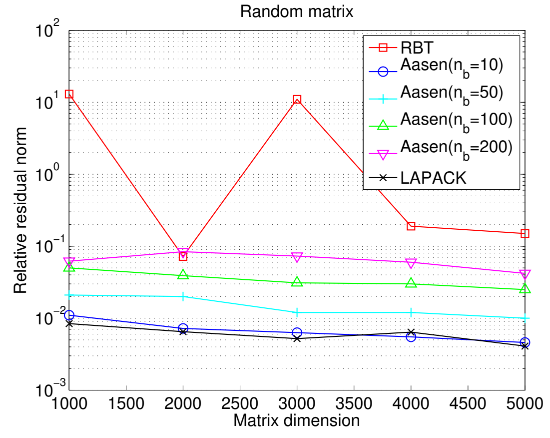
Figures 10(b) and 11(b) show the residual norms when the CALU is used in our blocked Aasen's algorithm. We found that the factorization can become unstable using a large block size (e.g., RIS and Fiedler). We are examining if this instability can be avoided using an LU factorization with more stable panel rank revealing pivoting [21]. In Section VI, we briefly discuss our current work to recover from this numerical difficulties, but more detailed analysis will be in our future reports.

B. Parallel Scalability

In the top plot of Figure 14, we compare the parallel performance of blocked Aasen's algorithm with that of RBT.

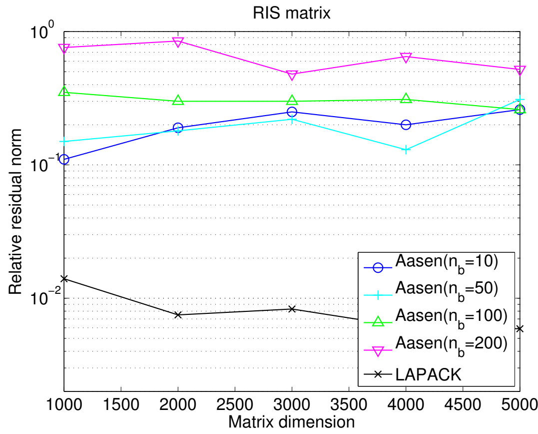


(a) recursive LU

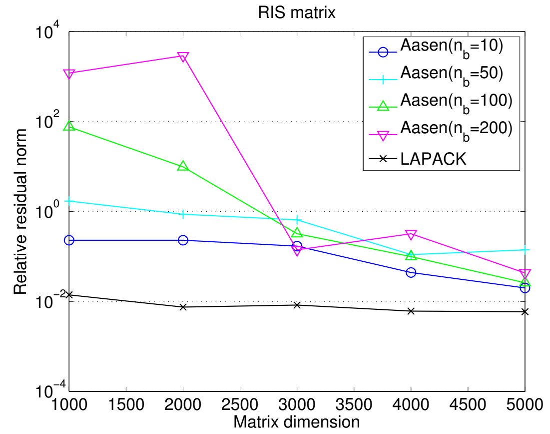


(b) CALU

Figure 10. Residual norms of blocked Aasen's algorithm on Random matrix.

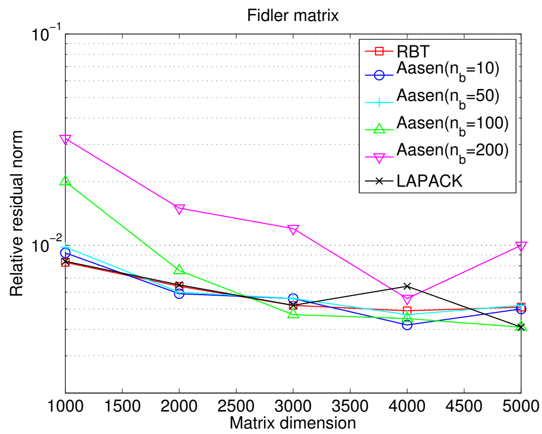


(a) recursive LU



(b) CALU

Figure 11. Residual norms of blocked Aasen's algorithm on RIS matrix (RBT failed).



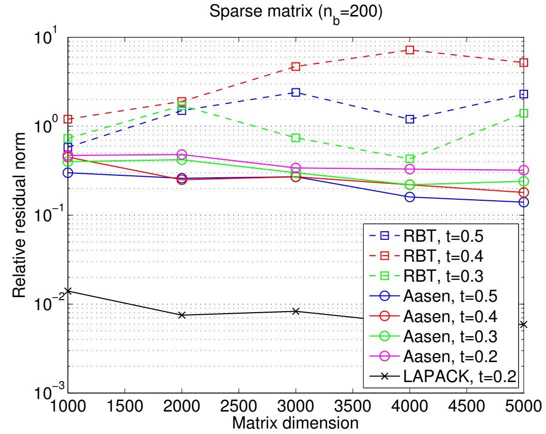
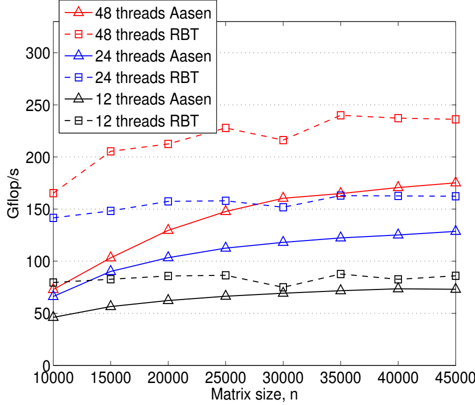
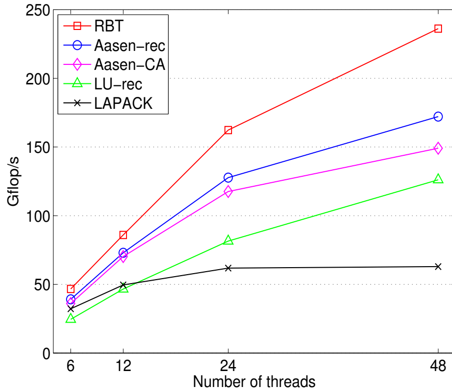


Figure 12. Residual norms of blocked Aasen's algorithm using recursive LU on Fiedler and Sparse matrices (default RBT failed on Sparse $t = 0.2$).



(a) Performance with different thread counts



(b) Parallel performance with $N = 45000$

Figure 14. Parallel performance on Random matrix with $n_b = 250$.

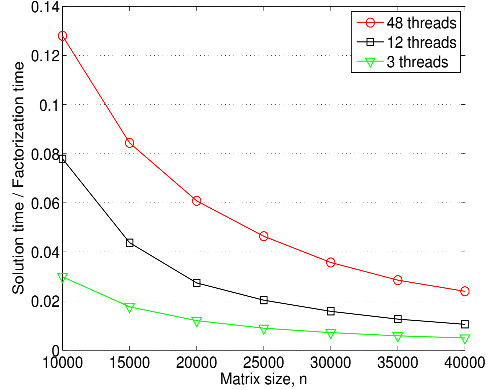


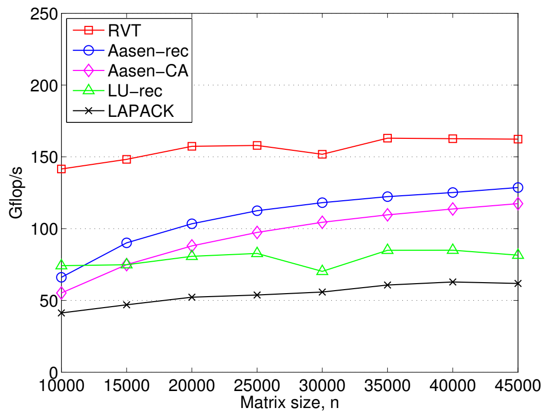
Figure 16. Performance of solver on Random matrix with $n_b = 250$.

For all the experiments in this subsection, we used the block size of 250 (i.e., $b = 250$).[8] In the bottom figure, we also compare the performance with that of the LDL$^{\mathrm{T}}$ factorization of threaded MKL, and with that of recursive LU of PLASMA [6] to compute an LU factorization of A. We computed Gflop/s as the number of flops required by the LDL$^{\mathrm{T}}$ factorization of LAPACK (i.e., $\frac{1}{3}n^3 + \frac{3}{2}n^2$ flops) over the factorization time in second. As discussed in Section II, RBT obtains a Gflop/s that is close to that of Cholesky factorization, and provides our practical upper-bound on the achievable Gflop/s. We see that on a medium number of cores, Aasen's algorithm stays close to RBT, but due to the combination of its left-looking updates and use of the recursive LU on the panel, it does not scale as well as the right-looking algorithms. On 6 and 48 cores, respectively, the Gflop/s of Aasen's algorithm with recursive LU were about 83% and 73% of those of RBT, while it obtained speedups of about 1.6 and 1.4 over the recursive LU. Notice that RBT obtained the speedups of about 1.9 over the recursive LU, which is our practical upper bound.

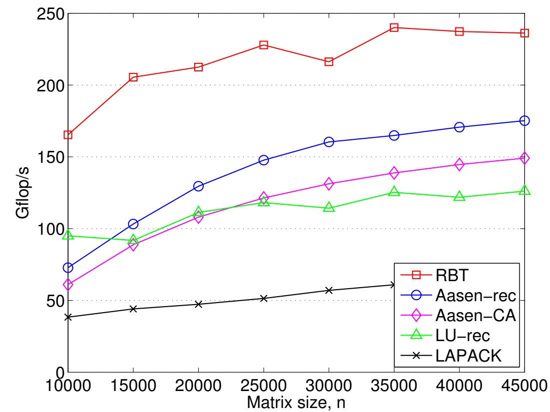
[8]Aasen, RBT, and LU algorithms of PLASMA obtain near-optimal performance using the block size of $n_b = 250$.

Figure 15 shows the Gflop/s of the various factorization algorithms with different matrix dimensions on 24 and 48 cores. Again, due to the limited parallelism in the left-looking algorithm, our implementation cannot obtain the parallel efficiency of the other right-looking algorithms. However, as the matrix dimension increases, our implementation could exploit an increasing amount of parallelism. By comparing against the Gflop/s of MKL, we clearly see that both RBT and recursive LU of PLASMA obtain excellent parallel performance. For all the cases, the blocked Aasen's algorithm was significantly faster than MKL, obtaining speedups of 2.1 and 2.8 on 24 and 48 cores, respectively.

Finally, Figure 16 shows the ratio of the total solution time over the factorization time of our blocked Aassen's algorithm using the recursive LU on the panels. Here, the solution time includes both the time needed for the forward and backward substitutions with the triangular matrices L and L^T, respectively, and the time needed to solve the banded system with T using the threaded MKL. Even though the ratio is relatively small, it increases with the increase in the number of threads, indicating that the solver does not scale as well as the factorization routine does. For example, in our experiments, the banded solver of the sequential MKL obtained about the same performance as that of the threaded MKL. Using either 48 or 12 threads, though the percentage decreases as the matrix dimension increases, about $60\% - 90\%$ of the solution time is spent in the banded solver. Even when CALU is used for the panel factorization instead of the recursive LU, the same solver routine can be used, requiring about the same amount of the solution time. With RBT, the solver requires about $2\% - 8\%$ or $1\% - 3\%$ of the factorization time using 48 or 12 threads, respectively.



(a) Performance using 24 threads



(b) Performance using 48 threads

Figure 15. Performance comparisons on Random matrix with $n_b = 250$.

VI. Conclusion

We analyzed the parallel performance of a blocked Aasen's algorithm on multicore architectures. The numerical results have shown that in comparison with the Bunch-Kaufman algorithm of LAPACK, our implementation loses only one or two digits in the computed residual norms. Furthermore, it is more robust than a randomization approach, being able to solve a wider range of problems. On 48 AMD Opteron processors, it obtained a speedup of 1.4 over a state-of-the-art recursive LU algorithm, while it obtained a speedup of 2.8 over the LDL^T factorization of MKL. These results demonstrate that this algorithm has the potential of becoming the first scalable algorithm that can take advantage of the symmetry and has a provable stability for solving symmetric indefinite problems.

We are currently studying the cause of the increasing numerical instability with respect to the increase in the block size, and also examining the numerical behavior of the blocked Aasen's algorithm combined with communication-avoiding algorithms. During our numerical experiments, we have encountered test matrices, where the numerical stability of the blocked Aasen's algorithm using CALU was not as good as that using recursive LU. This might be related to the fact that these test matrices lead to small pivots during the LU factorization of off-diagonal blocks. We will investigate whether we can recover or take advantage of these detected numerical low-rank properties by more stable rank revealing pivoting [21] or by means similar to hierarchically semiseparable factorization (e.g., [22]). We are also interested in extending theoretical communication cost analysis to the block-Aasen factorization by establishing communication lower bounds and proving that sequential and parallel versions of the algorithm attain the lower bounds. Finally, we will explore more scalable banded solvers to improve the parallel performance of the solver and an implementation of the algorithm on a distributed memory architecture.

Acknowledgments

This research was supported in part by NSF CCF-1117062 and CNS-0905188, Microsoft Corporation Research Project Description "Exploring Novel Approaches for Achieving Scalable Performance on Emerging Hybrid and Multi-Core Architectures for Linear Algebra Algorithms and Software," grant 1045/09 from the Israel Science Foundation (founded by the Israel Academy of Sciences and Humanities), and grant 2010231 from the US-Israel Bi-National Science Foundation, and by Microsoft (Award #024263) and Intel (Award #024894), and matching funding by U.C. Discovery (Award #DIG07-10227). Additional support comes from ParLab affiliates National Instruments, Nokia, NVIDIA, Oracle and Samsung, as well as MathWorks. Research is also supported by DOE grants DE-SC0004938, DE-SC0005136, DE-SC0003959, DE-SC0008700, and AC02-05CH11231, and DARPA grant HR0011-12-2-0016.

References

[1] A. Druinsky, I. Peled, S. Toledo, G. Ballard, J. Demmel, O. Schwartz, A communication avoiding symmetric indefinite factorization, presented at the SIAM conference on parallel processing for scientific computing, manuscript in preparation (2012).

[2] J. Aasen, On the reduction of a symmetric matrix to tridiagonal form, BIT 11 (1971) 233–242.

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

[4] A. Castaldo, R. Whaley, Scaling LAPACK panel operations using parallel cache assignment, in: Proceedings of the 15th AGM SIGPLAN symposium on principle and practice of parallel programming, 2010, pp. 223–232.

[5] F. Gustavson, Recursive leads to automatic variable blocking for dense linear-algebra algorithms, IBM Journal of Research and Development 41 (1997) 737–755.

[6] J. Dongarra, M. Faverge, H. Ltaief, P. Luszczek, Achieving numerical accuracy and high performance using recursive tile LU factorization, Tech. Rep. ICL-UT-11-08, University of Tennesse, Knoxville, Compute Science Department (2011).

[7] S. Toledo, Locality of reference in LU decomposition with partial pivoting, SIAM J. Matrix Anal. Appl. 18 (4) (1997) 1065–1081.

[8] L. Grigori, J. Demmel, H. Xiang, CALU: a communication optimal LU factorization algorithm, SIAM. J. Matrix Anal. Appl. 32 (4) (2011) 1317–1350.

[9] J. Demmel, L. Grigori, M. Hoemmen, J. Langou, Communication-optimal parallel and sequential QR and LU factorizations, SIAM J. Sci. Comput. 34 (2012) A206–A239, , also available as EECS Department, University of California, Berkeley, Technical report (UCB/EECS-2008-89).

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

[11] E. Anderson, J. Dongarra, Evaluating block algorithm variants in LAPACK, in: Proceedings of the 4th Conference on Parallel Processing for Scientific Computing, 1989, pp. 3–8.

[12] C. Ashcraft, R. Grimes, J. Lewis, Accurate symmetric indefinite linear equation solvers, SIAM J. Matrix Anal. Appl. 20 (1998) 513–561.

[13] D. Parker, Random butterfly transformations with applications in computational linear algebra, Tech. Rep. CSD-950023, University of California, Los Angeles, Computer Science Department (1995).

[14] M. Baboulin, J. Dongarra, J. Herrmann, S. Tomov, Accelerating linear system solutions using randomization techniques, to appear in ACM Transactions on Mathematical Software, also available as LAPACK Working Note 246.

[15] D. Becker, M. Baboulin, J. Dongarra, Reducing the amount of pivoting in symmetric indefinite systems, in: R. Wyrzykowski et. al. (Ed.), 9th International Conference on Parallel Processing and Applied Mathematics (PPAM 2011), Vol. 7203 of Lecture Notes in Computer Science, Springer-Verlag, Heidelberg, 2012, pp. 133–142.

[16] M. Baboulin, D. Becker, J. Dongarra, A parallel tiled solver for dense symmetric indefinite systems on multicore architectures, in: Proceedings of Parallel and Distributed Processing Symposium (IPDPS), 2012, also available as LAPACK Working Note 261.

[17] P. Strazdins, A dense complex symmetric indefinite solver for the Fujitsu AP3000, Technical Report TR-CS-99-01, The Australian National University (1999).

[18] N. Gould, J. Scott, Y. Hu, A numerical evaluation of sparse solvers for symmetric systems, ACM Trans. Math. Softw. 33 (2) (2007) 10:1–10:32.

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

[20] L. Kaufman, The retraction algorithm for factoring banded symmetric matrices, Numer. Linear Algebra Appl. 14 (2007) 237–254.

[21] A. Khabou, J. Demmel, L. Grigori, M. Gu, LU factorization with panel rank revealing pivoting and its communication avoiding version, Tech. Rep. LAPACK working note (LAWN263), submitted to SIAM J. Matrix Anal. Appl. (2012).

[22] J. Xia, S. Chandrasekaram, M. Gu, X. Li, Fast algorithms for hierarchically semiseparable matrices, Numer. Linear Algebra Appl. 17 (2010) 953–976.