Implementing a blocked Aasen's algorithm with a dynamic scheduler on multicore architectures

Ichitaro Yamazaki*, Dulceneia Becker*, and Jack Dongarra*†‡
*University of Tennessee, Knoxville, TN 37996, USA
†Oak Ridge National Laboratory, Oak Ridge, TN, USA
‡University of Manchester, Manchester, United Kingdom
{iyamazak, dbecker7, dongarra}@eecs.utk.edu

Alex Druinsky, Inon Peled, and Sivan Toledo
Tel-Aviv University, Tel-Aviv 69978, Israel
alexdrui@post.tau.ac.il, inon.peled@gmail.com, stoledo@tau.ac.il

Grey Ballard, James Demmel, and Oded Schwartz
University of California, Berkeley, Berkeley, CA 94704, USA
{ballard, demmel}@cs.berkeley.edu, oded.schwartz@gmail.com

***Abstract*—Factorization of a dense symmetric indefinite matrix is a key computational kernel in many scientific and engineering simulations. However, it is difficult to develop a scalable factorization algorithm that guarantees numerical stability through pivoting and takes advantage of the symmetry 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 left-looking Aasen's algorithm. To exploit parallelism in this left-looking algorithm, we study several performance enhancing techniques; e.g., parallel reduction to update a panel, panel factorization by tall-skinny LU, and parallel symmetric pivoting. Our performance results on up to 48 AMD Opteron processors demonstrate that our implementation with a dynamic scheduler obtains speedups of up to 2.8 over MKL, while losing only one or two digits in the computed residue norms.**

I. Introduction

Many scientific and engineering simulations require the solution of a dense symmetric indefinite linear system of equations,

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

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 left-looking Aasen's algorithm that is proposed this year [1].

To solve (1), the Aasen's algorithm [2] computes an LTL$^{\mathrm{T}}$ factorization 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. Then, the solution x can be computed by successively solving the linear systems with the factored matrices L, T, and L^T. The left-looking Aasen's algorithm performs $1/3\,n^3 + O(n^2)$ floating-point operations (flops) and is component-wise backward stable.

To effectively utilize the memory hierarchy on a modern computer, a partitioned-version of the Aasen's algorithm was recently proposed [3]. This algorithm first factorizes the panel in a left-looking fashion, and then uses BLAS-3 operations to update the trailing submatrix in a right-looking way. In comparison to a standard column-wise algorithm, this partitioned algorithm slightly increases the operation count, performing $(1+1/n_b)n^3+O(n^2)$ flops with a block size of n_b. Nevertheless, it 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, which in comparison to a BLAS-3 operation, have higher ratios of communication volume to flop counts. 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.

To further shorten the factorization time, a blocked-version of the left-looking Aasen's algorithm was proposed this year [1]. It computes an LTL$^{\mathrm{T}}$ factorization, where T is a banded matrix (instead of tridiagonal) with its half-

bandwidth being equal to the block size n_b. In this blocked algorithm, each panel can be factorized using an existing LU factorization, such as recursive LU [4], [5], [6] and communication-avoiding LU (CALU) [7]. In comparison to 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[1] and uses a dynamic scheduler called QUARK[2]. This 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. To efficiently utilize larger numbers of cores in parallel, all the existing factorization routines in PLASMA update the trailing submatrix in right-looking fashion. In order to fully exploit the limited parallelism in the left-looking algorithm, we study several performance enhancing techniques; e.g., parallel reduction to update the panel, panel factorization by tall-skinny LU, and parallel symmetric pivoting. Our experimental results on up to 48 AMD Opteron processors demonstrate that a left-looking implementation can obtain a good parallel performance. We also present numerical results to show that in comparison to the widely-used stable algorithm (Bunch-Kauffman algorithm of LAPACK), our implementation loses only one or two digits in the computed residue norms.

The remainder of the paper is organized as follows: after listing 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. Last, in Section VI, we present our final remarks. Table I summarizes the notations used in this paper. Some notations are reused in column-wise and block-wise algorithms, 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.

II. Related works

In Section V, we compare the performance and accuracy of our blocked Aasen's algorithm with the following state-of-the-art factorization algorithms that can take advantage of symmetry in dense symmetric indefinite matrices on multicore architectures:

[1]http://icl.utk.edu/plasma/

[2]http://http://icl.utk.edu/quark/

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
$\mathbf{a_{:j}}$	j-th column of A
$\mathbf{a_{:i}}$	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
NOTATIONS USED IN THIS PAPER.

A. LAPACK – Bunch-Kauffman algorithm

LAPACK[3] is a set of dense linear algebra routines that is extensively used in many scientific and engineering simulations. For solving symmetric indefinite systems (1), LAPACK implements a partitioned $\mathrm{LDL^T}$ factorization with Bunch-Kauffman algorithm [8], [9] that computes

$$PAP^T = LDL^T,$$

where D is a block diagonal matrix with either 1-by-1 or 2-by-2 diagonal blocks. This algorithm performs the same number of flops as the left-looking column-wise Aasen's algorithm, i.e. $1/3\,n^3 + O(n^2)$ flops, and is norm-wise backward stable. In [3], a serial implementation of the partitioned Aasen's algorithm is shown to be as efficient as this Bunch-Kauffman algorithm of LAPACK on a single core. Unfortunately, at each step of the Bunch-Kauffman algorithm, up to two columns of the trailing submatrix must be scanned, where the index of the second column corresponds to the index of the row with the maximum modulo 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 the algorithms into fine-grained computational tasks that operate on small 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 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.

[3]http://www.netlib.org/lapack/

Randomization algorithms can exploit more parallelism than corresponding deterministic algorithms can, and are gaining popularity in linear algebra algorithms [10]. The LDL$^{\mathrm{T}}$ factorization of PLASMA [11], [12] extends a randomization technique developed for the LU factorization [13] to symmetric indefinite systems. Here, 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 randomization technique 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. This random butterfly transformation (RBT) requires only $2dn^2 + O(n)$ flops in comparison to the $1/3n^3 + O(n^2)$ 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 stability. Nonetheless, since iterative refinement is required, the error can be evaluated without extra computation and signal the failure of the method. Numerical tests [11] showed that with iterative refinement, it is accurate for many test cases, including pathological ones, though it may fail in certain cases.

Beside the dense factorization on multicores, a parallel factorization of a dense symmetric indefinite matrix with pivoting has been studied on distributed-memory systems in [14]. Furthermore, many implementations of the sparse LDL$^{\mathrm{T}}$ factorization have been proposed on distributed and shared memory architectures [15], [16], [17].

III. Aasen's algorithms

To compute the LTL^T factorization, the Aasen's algorithm uses an intermediate Hessenberg matrix H which is defined as $H = TL^T$. In this section, we describe column-wise right- and left-looking Aasen's algorithms (Sections III-A and III-B), and a blocked left-looking version of the algorithm (Section III-C).

A. Right-looking Aasen's algorithm

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

$$t_{1,1} = a_{1,1} \quad \text{and} \quad \boldsymbol{\ell}_{2:n,2}\, t_{2,1} = a_{2:n,1} - \boldsymbol{\ell}_{2:n,1}\, t_{1,1}. \quad (3)$$

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

$$\begin{aligned} A_{2:n,2:n} \;-= \; & \boldsymbol{\ell}_{2:n,1}\, t_{2,1}\, \boldsymbol{\ell}_{2:n,2}^T + \boldsymbol{\ell}_{2:n,1} t_{1,1} \boldsymbol{\ell}_{2:n,1}^T \\ & + \boldsymbol{\ell}_{2:n,2}\, t_{2,1}\, \boldsymbol{\ell}_{2:n,1}^T, \end{aligned} \quad (4)$$

then the LTL$^{\mathrm{T}}$ factorization of A can be computed by recursively computing the LTL$^{\mathrm{T}}$ 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-1 updates as in

$$A_{2:n,2:n} \;-= \; \boldsymbol{\ell}_{2:n,1} \mathbf{h}_{2:n,1}^T - \boldsymbol{\ell}_{2:n,2}\, t_{2,1}\, \boldsymbol{\ell}_{2:n,1}^T, \quad (5)$$

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-1 update as in

$$A_{2:n,2:n} \;-= \; \mathbf{w}_{2:n,1} \boldsymbol{\ell}_{2:n,1}^T - \boldsymbol{\ell}_{2:n,1} \mathbf{w}_{2:n,1}^T, \quad (6)$$

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 the right-hand-side of the second equation in (3) is scaled so that $\ell_{2,2}$ is one. To maintain the numerical stability, the element with the largest modulo is used as the pivot. This right-looking Aasen's algorithm is equivalent to the Parllet-Reid algorithm [18].

The above algorithm performs a total of $\frac{2}{3}n^3 + O(n^2)$ flops. On the other hand, in an LDL^T factorization, D is block diagonal, and it requires only one rank-1 update to update the trailing submatrix using each column of L, requiring only half of the flops required by the right-looking Aasen's algorithm.

B. Left-looking Aasen's 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 $k = 1, 2, \dots, j-1$,

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

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

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

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

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

for $j = 1$ to n **do**
  Compute $H_{1:(j-1),j}$ and update $T_{j,j}$ (see Figure 2)
  if $j > 1$ **then**
    $T_{j,j+1} = T_{j,j-1}^T$
  end if
  $T_{j,j} = L_{j,j}^{-1} T_{j,j} L_{j,j}^{-T}$
  // Compute (j,j)-th block of H
  $H_{j,j} = T_{j,j} L_{j,j}^T$
  if $j < n$ **then**
    // Extract $L_{:,j+1}$ of L
    $L_{(j+1):n,j+1} = A_{(j+1):n,j} - L_{(j+1):n,1:j} H_{1:j,j}$
    $[L_{(j+1):n,j+1}, H_{j,j+1}, P_j] = \mathrm{LU}(L_{(j+1):n,j+1})$
    // Apply pivots to other part of matrices
    $L_{(j+1):n,1:j} := P_j L_{(j+1):n,1:j}$
    $A_{(j+1):n,(j+1):n} := P_j A_{(j+1):n,(j+1):n} P_j^T$
    // Extract $T_{J+1,J}$
    $T_{j+1,j} = H_{j+1,j} L_{j,j}^{-T}$
  end if
end for

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

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

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

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

$$h_{j+1,j} = v_1 \quad \text{and} \quad \boldsymbol{\ell}_{(j+1):n,j+1} = \frac{\mathbf{v}}{v_1}. \tag{7}$$

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

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

Again, a numerical issue comes when scaling $\mathbf{v}$ with $\frac{1}{v_1}$ in the second equation of (7). To maintain the numerical stability, the element of $\mathbf{v}$ with the largest modulo is used as the pivot.

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-1 updates (5) or (6) are needed, doubling the number of flops.[4]

Approach 1:

// Compute $H_{1:(j-1),j}$
for $k = 1, 2$ to $j-1$ **do**
  $H_{k,j} = T_{k,k} L_{j,k}^T$
  $H_{k,j} = H_{k,j} + T_{k,k+1} L_{j,k+1}^T$
  if $k > 1$ **then**
    $H_{k,j} = H_{k,j} + T_{k,k-1} L_{j,k-1}^T$
  end if
end for
// Update $T_{j,j}$
$T_{j,j} = A_{j,j} - L_{j,1:(j-1)} H_{1:(j-1),j}$
if $J > 1$ **then**
  $T_{j,j} = T_{j,j} - L_{j,j} T_{j,j-1} L_{j,j-1}^T$
end if

Approach 2:

// Compute $H_{1:(J-1),J}$
for $k = 1$ to $j-1$ **do**
  $U_k = T_{k,k} L_{j,k}^T$
  $V_k = T_{k,k+1} L_{j,k+1)}^T$
  $H_{k,j} = U_k + V_j$
  if $K > 1$ **then**
    $H_{k,j} = H_{k,j} + T_{k,k-1} L_{j,k-1}^T$
  end if
end for
// Update $T_{j,j}$
$W_{1:(j-1)} = \frac{1}{2} V_{1:(j-1)} + U_{1:(j-1)}$
$T_{j,j} = A_{j,j} - L_{j,1:(j-1)} W_{1:(j-1),j}^T$
  $W_{1:(j-1)} L_{j,1:(j-1)}^T$

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

C. Blocked left-looking Aasen's algorithm

Either the right- or left-looking Aasen's algorithm 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

[4]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.

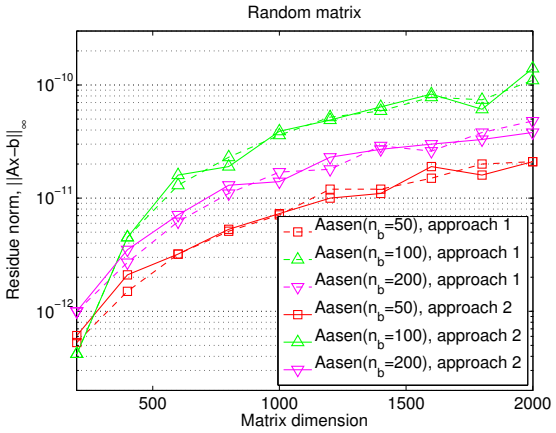


Figure 3. Solution accuracy using two updating schemes in Figure 2.

block of $A = LH$, we have for $k = 1, 2, \ldots, j-1$,

$$H_{k,j} = T_{k,k-1}L_{k-1,j} + T_{k,k}L_{k,j} + T_{k,k+1}L_{k+1,j} \quad (8)$$

and

$$H_{j,j} = L_{j,j}^{-1}\left(A_{j,j} - \sum_{k=1}^{j} L_{j,k}H_{k,j}\right). \quad (9)$$

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

$$T_{j,j} = \left(H_{j,j} - T_{j,j-1}L_{j,j-1}^T\right)L_{j,j}^{-T}. \quad (10)$$

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 not stable in practice because the symmetric $T_{j,j}$ is computed through a sequence of unsymmetric expressions (10). To recover the symmetry, we substitute $H_{j,j}$ of (9) into (10), and obtain

$$L_{j,j}T_{j,j}L_{j,j}^T = A_{j,j} - \sum_{k=1}^{j-1} L_{j,k}H_{k,j} - L_{j,j}T_{j,j-1}L_{j,j-1}^T, \quad (11)$$

Furthermore, replacing $H_{k,j}$ of (11) with that of (8) gives us

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

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

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

Figure 1 shows the pseudocode of this blocked algorithm. We have investigated two approaches to update the diagonal block $T_{j,j}$ (see Figure 2). The second approach considers the symmetry while updating the diagonal block, but requires extra jn_b^2 flops. Figure 3 shows that these two approaches obtain similar stability on random matrices. For the rest of the paper, we focus on the first approach.

In Figure 3, the residue norms increase slightly with the increase in the block size n_b. This agrees with the error bound in [1] that is propotional to both the block size and the number of blocks.

IV. Implementation

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

A. Tiled implementation

As mentioned in Section II, PLASMA is based on tiled algorithms that breaks the 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 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 details 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 an example of DAG for our blocked Aasen's algorithm. At run time, the runtime system QUARK 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

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($'N', 'T', 1.0, T_{i,i}, L_{i,i}, 0.0, H_{i,j}$)
5: GEMM($'N', 'T', 1.0, T_{i+1,i}, L_{i,i+1}, 1.0, H_{i,j}$)
6: **if** $I > 1$ **then**
7: GEMM($'N', 'T', 1.0, T_{i-1,i}, L_{i,i-1}, 1.0, H_{i,j}$)
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($'N', 'N', 1.0, T_{j,j-1}, L_{j,j-1}, 0.0, W_j$)
17: GEMM($'N', 'N', -1.0, L_{j,j}, W_{j,j}, 1.0, T_{j,j}$)
18: **end if**
19: SYGST($T_{j,j}, L_{j,j}$)
20: *// Compute* (j, j)*-th block of* H
21: GEMM($'N', 'T', 1.0, T_{j,j}, L_{j,j}, 0.0, H_{j,j}$)
22: **if** $j < n$ **then**
23: *// Extract* j*-th block column of* L
24: **for** $i = j + 1$ to n **do**
25: LACPY($A_{i,j}, L_{i,j+1}$)
26: **end for**
27: **for** $k = 1$ to j **do**
28: **for** $i = j + 1$ to n **do**
29: GEMM($'N', 'N', -1.0, L_{i,k}, H_{k,j}, 1.0, L_{k,j+1}$)
30: **end for**
31: **end for**
32: $[L_{(j+1):n,j+1}, H_{j,j+1}, P_j] = \text{LU}(L_{(j+1):n,j+1})$
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: GEMM($'N', 'N', 1.0, H_{j+1,j}, L_{j,j}^{-T}, 0.0, T_{j+1,j}$)
38: **end if**
39: **end for**

Figure 4. Tiled implementation of blocked left-looking Aasen's algorithm.

can be reduced, allowing us to utilize a large number of cores.

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 24 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,

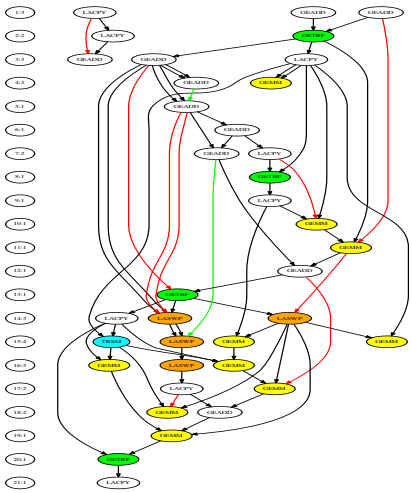


Figure 5. DAG of blocked Aasen's algorithm with $n = 750$ and $n_b = 50$. (need to fix or remove this DAG)

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 algorithm. Specifically, we first use 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 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 to $O(n_b^3)$ 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 errores leads to slightly different factors on different numbers of cores.

C. Parallel panel factorization

When a recursive LU is used for the panel factorization, all the blocks in the panel must be updated before the panel

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

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

factorization can start. Furthermore, even with the parallel reduction described in Section IV-B, the updates on the next panel cannot be started until the current symmetric pivoting is applied to the previous columns of L. Hence, there is 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 panel has only a small number of tiles, some cores are left idle.

This synchronization can be avoided if we use a communication-avoiding LU 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 11 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 cannot start until the yellow blocks after the red block (updating $A_{j:n,j}$ with $L_{j:n,j}$) finishes.[5] Hence, in this left-looking algorithm, we often cannot overlap all of the panel factorization tasks with the tasks to update the panel. As a result, 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. Since only the lower triangular part of the sub-

[5]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 disappeares, 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 cores during the recursive LU.

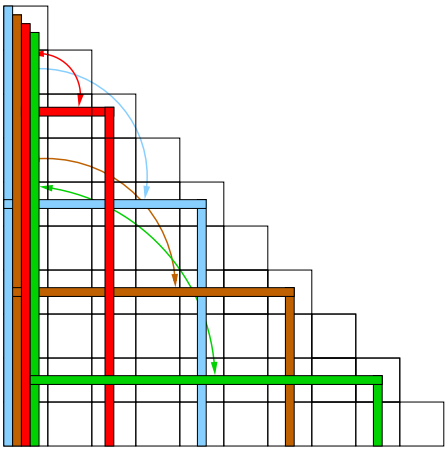


Figure 8. Illustration of symmetric pivoting.

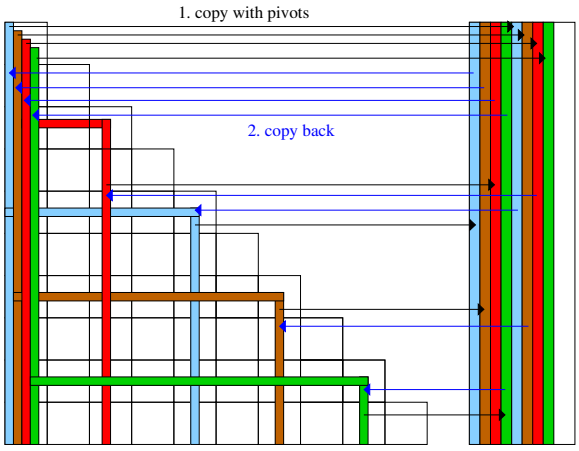
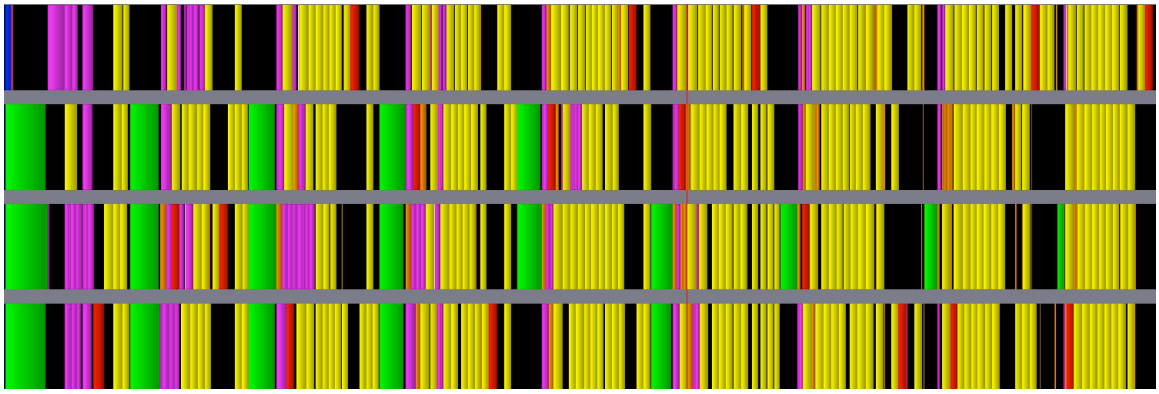


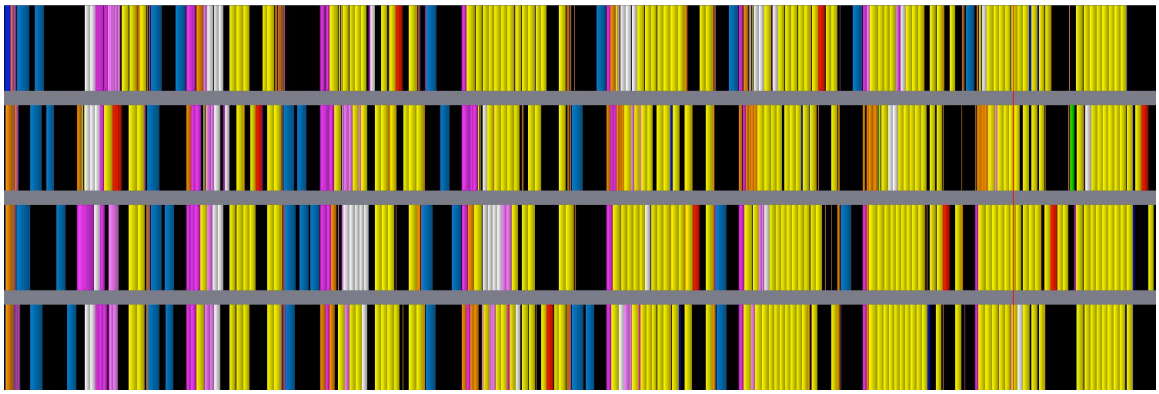
Figure 9. Implementation of parallel symmetric pivoting.

matrix is stored, this symmetric pivoting leads to irregular memory accesses and irregular task dependencies (see Figure 8). On the other hand, in an LU factorization with partial pivoting, only the rows within block columns are swapped, leading to more regular dependencies. Moreover, in the symmetric factorization, only the lower-triangular part of the trailing submatrix is updated, and the cost of applying pivots is more dominant in the total cost of the factorization. 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 lay on the critical path, and must be implemented as efficient as possible.

As illustrated in Figure 9, our symmetric pivoting consists of 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 $\frac{n}{n_b} - j$ tasks, each of which independently copies to the workspace the columns in one of the $\frac{n}{n_b} - j$ block columns of



(a) recursive LU



(b) CALU

Figure 7. Traces of our block-Aasen's algorithm, where green, blue, orange, magenta, red, yellow, white, and black blocks represent recursive LU, CALU, pivoting of L, pivoting of A, SYGST, GEMM, TRSM, and idle time, respectively ($n = 2000$ and $n_b = 200$).

the submatrix. Notice that due to the symmetric storage, 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 memory locations in 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.

At each step of symmetric pivoting, each tile of the submatrix is read by two tasks (i.e., at the j-th step, $A_{i,k}$ is read by the $(i-j)$-th and $(k-j)$-th tasks). It is possible for each task to only access the tiles in a single block column and copy all the required columns of the block column to the workspace. In this way, each tile is read by only one task, but the k-th task accesses k tiles, leading to a workload imbalance among the tasks. This imbalance can be reduced by generating finer-grained tasks, where each task accesses only a fixed number of tiles in the block column. However, this often adds a significant overhead to the runtime system. Furthermore, this approach often does not improve the memory access pattern because only the rows and columns to be swapped are accessed, and hence the accesses to these columns and rows of the tile are irregular.

Another approach is to let each task gather all the columns to be swapped into a set of tiles in the workspace. This approach provides much finer-grained parallelism than the previous two approaches because as soon as these columns are copied into the tiles, they can be copied back to the submatrix and used to updated the next block column. On the other hand, with the previous approaches, 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 into which columns of the workspace. However, this new approach leads to an irregular memory access to the block column of the trailing submatrix. Furthermore, 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 approach gave the best performance in

many cases, and we use 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 are 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, we assign a higher-priority to apply the symmetric pivoting to the next block column of A since that block column is needed at the next step, while the other block columns 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 we cannot overwrite $A_{:,j+1}$ with $L_{:,j+1}$ at the j-th step. If the diagonal tiles of T are stored in those of A, then only additional memory required to return L and T are those to store the off-diagonal tiles of T (i.e., $(n-1)n_b$-by-n_b storage).

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 small 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 be overlapped, the performance gain obtained using two different workspace for these two stages is often small.

F. Banded solver

Once the matrix is reduced to a banded form, we used the general banded solver GBSV of threaded MKL to solved this banded system. To improve the performance of the solver, we are exploring a few other options (e.g., [19]) to solve this banded system.

V. Performance studies

We now analyse the performance of our blocked Aasen's algorithm on up to eight 6-core 2.8MHz AMD Opteron processors. Our code was compiled with **gcc** compiler and **-O2** optimization flag, and was linked with the version 2011.1.107 MKL library. All the experiments are in real double precision.

Name	
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.

A. Numerical stability

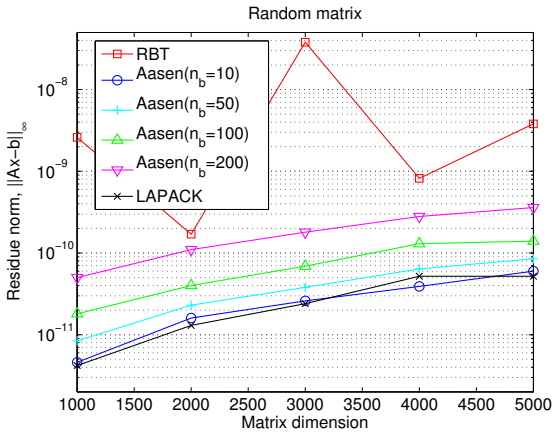
Figures 10 – 12 compare the computed residue norms of our blocked Aasen's algorithm with those of the LDL^T factorization of LAPACK using the Bunch-Kauffman algorithm and with that of PLASMA using RBT. For RBT, we used the default transformation depth of two. The figure clearly shows that the residue norm of the Aasen's algorithm increases with the increase in the value of the block size n_b. However, the residue norm of the Aasen's algorithm with $n_b = 200$ was competitive with or significantly smaller than those of RBT. Furthermore, the factorization with RBT failed for RIS and Sparse matrix with $t = 0.2$. For the Sparse matrix, RBT will be successful if the transformation depth is increased to make the transformed matrix sufficiently dense. With RBT the oscillations of the residue norms are expected, but a few iterations of iterative refinement can smooth out the residue norms.

Figures 10 – 11 show the residue norms when a communication-avoiding LU is used in our block 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 panel rank revealing pivoting [20].

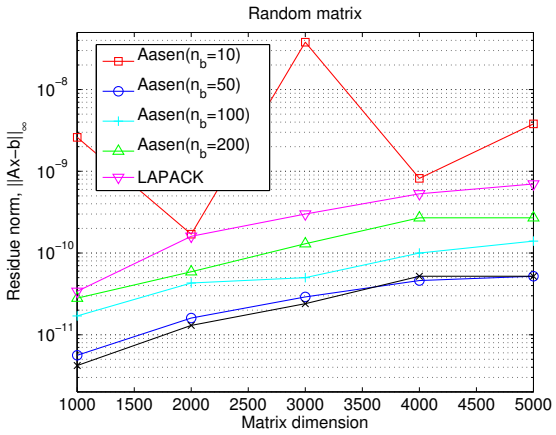
B. Parallel scalability

Figure 14 compares the parallel performance of the Aasen's algorithm with those of RBT and recursive LU of PLASMA. We computed Gflop/s as the number of flops required by the LDL^T factorization of LAPACK (i.e., $\frac{1}{3}n^3 + \frac{3}{2}n^2$ flops) over the time in second required by the algorithm to compute the factorization. As discussed in Section II, in term of factorization time, 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, the Aasen's algorithm stays close to RBT, but due to the combination of its left-looking formulation 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 Aasen's algorithm with recursive LU obtained about 83% and 73% of Gflop/s of RBT, while it obtained speedups of about 1.6 and 1.4 over the recursive LU. Notice that RBT obtains the speedups of about 2.0 over the recursive LU, which is our practical upper bound.

Figure 15 shows the Gflop/s of the various factorization algorithms with different matrix dimensions on 24 and 48

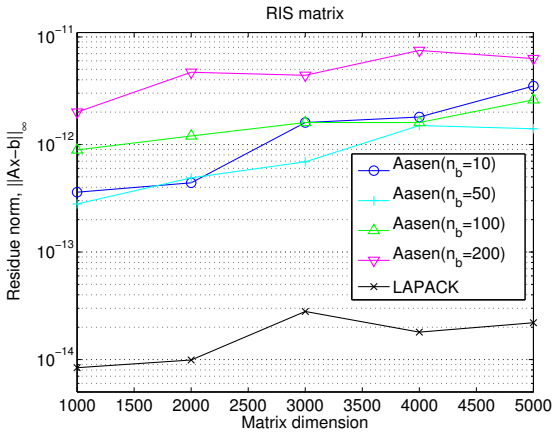


(a) recursive LU

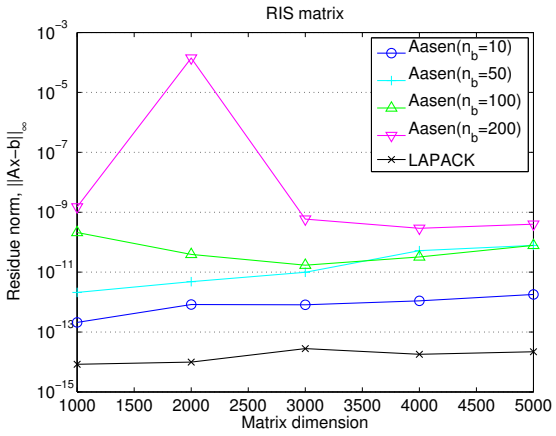


(b) CALU

Figure 10. Solution accuracy of blocked left-looking Aasen's algorithm on Random matrix.

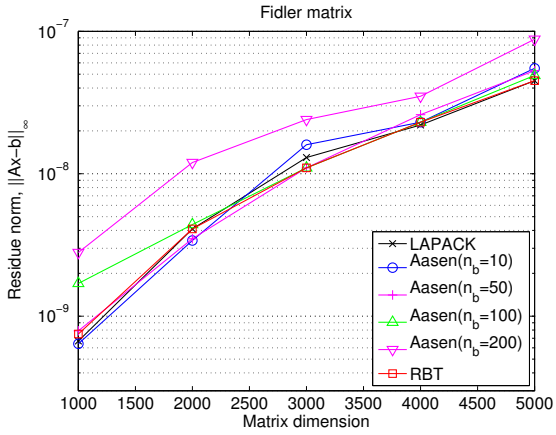


(a) recursive LU



(b) CALU

Figure 11. Solution accuracy of blocked left-looking Aasen's algorithm on RIS matrix (default RBT failed).



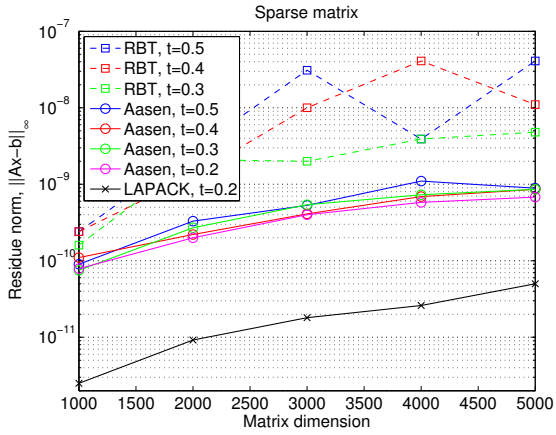


Figure 12. Solution accuracy of blocked left-looking Aasen's algorithm using recursive LU on Fidler and Sparse matrices (default RBT failed on Sparse $t = 0.2$).

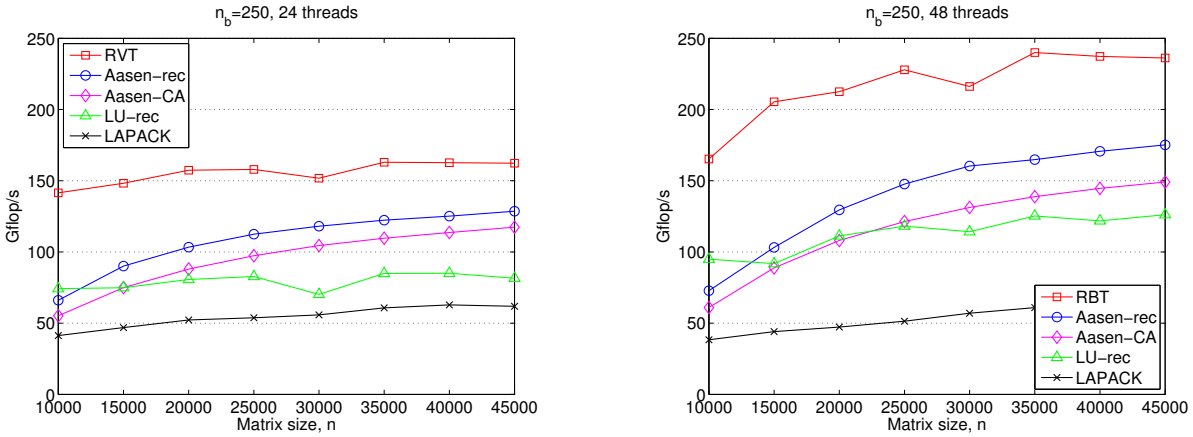


Figure 15. Performance comparisons on Random matrix.

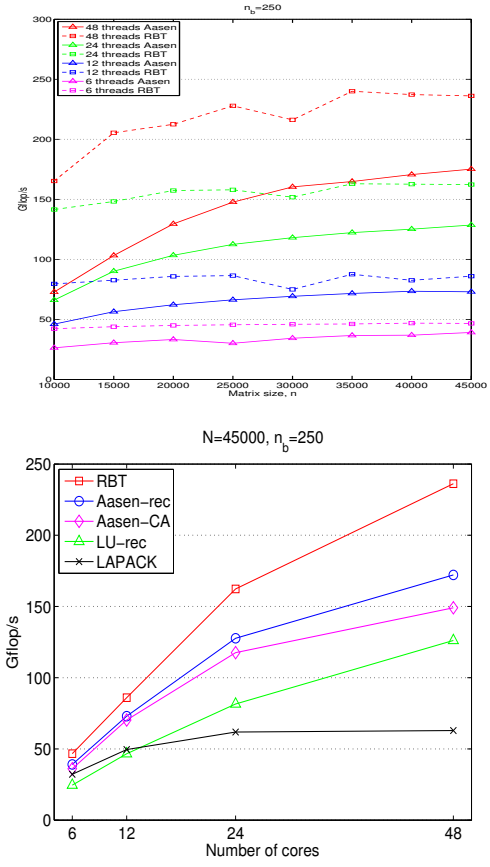


Figure 14. Parallel performance comparison on Random matrix.

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

VI. Conclusion

We analyzed the parallel performance of a blocked left-looking Aasen's algorithm on multicore architectures. This is not only the first implementation of the algorithm, but it is also the first implementation of a left-looking algorithm in PLASMA. The numerical results have shown that in comparison to the Bunch-Kauffman algorithm of LAPACK, our implementation loses only one or two digits in the computed residue 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 the speedup of 1.4 over the state-of-the-art recursive LU algorithm, while it obtained the speedup of 2.8 over the LDL 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 block Aasen's algorithm combined with communication-avoiding algorithms. During our numerical experiments, we have encountered test matrices, where the numerical property 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 if we could take advantage of this numerical low-rank

properties by a mean similar to hierarchically semiseparatble factorization (e.g., [21]). Finally, we will explore more scalable banded solver to improve the performance of the solver.

ACKNOWLEDGMENTS

This research was supported in part by NSF CCF-1117062 and CNS-0905188, and 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." We are grateful to all the researchers at the Innovative Computing Laboratory (ICL) of the University of Tennessee, Knoxville for helpful discussions.

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 (2011) –.

[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, Compute Science department (2011).

[7] J. Demmel, L. Grigori, M. Hoemmen, J. Langou, Communication-optimal parallel and sequential QR and LU factorizations, Tech. Rep. UCB/EECS-2008-89, EECS Department, University of California, Berkeley (2008).

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

[9] 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.

[10] D. S. Parker, Random butterfly transformations with applications in computational linear algebra, Technical Report CSD-950023, Computer Science Department, UCLA (1995).

[11] 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.

[12] M. Baboulin, D. Becker, J. Dongarra, A parallel tiled solver for dense symmetric indefinite systems on multicore architectures, Proceedings of IPDPS 2012 (to appear)LAPACK Working Note 261.

[13] M. Baboulin, J. Dongarra, J. Herrmann, S. Tomov, Accelerating linear system solutions using randomization techniques, ACM Transactions on Mathematical Software (to appear)LAPACK Working Note 246.

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

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

[16] P. Hénon, P. Ramet, J. Roman, On using an hybrid MPI-Thread programming for the implementation of a parallel sparse direct solver on a network of SMP nodes, In PPMA'05, *LNCS* 3911 (2005) 1050–1057.

[17] O. Schenk, K. Gärtner, On fast factorization pivoting methods for symmetric indefinite systems, Elec. Trans. Numer. Anal. 23 (2006) 158–179.

[18] 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.

[19] L. Kaufman, The retraction algorithm for factoring banded symmetric matrices, Numer. Linear Algebra Appl. 14 (2007) 237254.

[20] A. Khabou, J. Demmel, L. Grigori, M. Gu, LU factorization with panel rank revealing pivoting and its communication avoiding version, submitted to SIAM J. Matrix Anal. Appl.

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