Symmetric Indefinite Linear Solver using OpenMP Task on Multicore Architectures

Ichitaro Yamazaki, Jakub Kurzak, Panruo Wu, Mawussi Zounon, and Jack Dongarra

Abstract—Recently, the Open Multi-Processing (OpenMP) standard has incorporated task-based programming, where a function call with input and output data is treated as a task. At run time, OpenMP's superscalar scheduler tracks the data dependencies among the tasks and executes the tasks as their dependencies are resolved. On a shared-memory architecture with multiple cores, the independent tasks are executed on different cores in parallel, thereby enabling parallel execution of a seemingly sequential code. With the emergence of many-core architectures, this type of programming paradigm is gaining attention—not only because of its simplicity, but also because it breaks the artificial synchronization points of the program and improves its thread-level parallelization. In this paper, we use these new OpenMP features to develop a portable high-performance implementation of a dense symmetric indefinite linear solver. Obtaining high performance from this kind of solver is a challenge because the symmetric pivoting, which is required to maintain numerical stability, leads to data dependencies that prevent us from using some common performance-improving techniques. To fully utilize a large number of cores through tasking, while conforming to the OpenMP standard, we describe several techniques. Our performance results on current many-core architectures—including Intel's Broadwell, Intel's Knights Landing, IBM's Power8, and Arm's ARMv8—demonstrate the portable and superior performance of our implementation compared with the Linear Algebra PACKage (LAPACK). The resulting solver is now available as a part of the PLASMA software package.

Index Terms—Linear algebra, Symmetric indefinite matrices, Multithreading, Runtime.

1 Introduction

Solving dense symmetric–indefinite linear systems of equations is relevant to many scientific and engineering problems, including physics of structures, acoustics, and electromagnetism. Such linear solvers are also needed for unconstrained or constrained optimization problems or for solving the augmented system for general least squares discretized–incompressible Navier-Stokes equations. A few algorithms have been proposed for solving such linear systems (including the Bunch Kaufman [1], rook pivoting [2], and Aasen's algorithms [3] implemented in the Linear Algebra PACKage [LAPACK] [4]). Compared to the non-symmetric linear solver, the symmetric solver has several advantages in terms of storage and floating point operation (FLOP) count (i.e., only the triangular part of the matrix needs to be stored and computed), spectral properties (e.g., the inertia remains the same under the symmetric transformation), and structure (e.g., the symmetry of the whole matrix is preserved when factorizing the dense diagonal blocks of a dense or sparse symmetric matrix).

However, the symmetric pivoting required to maintain the numerical stability of the solver leads to the data dependencies that prevent us from using some of the standard techniques to enhance the solver performance (e.g., lookahead). As a result, developing a scalable symmetric linear solver still presents a significant problem (e.g., the Scalable Linear Algebra PACKage [ScaLAPACK] still does not support such a solver). At the same time, the demand for a scalable symmetric solver is growing because of emerging hardware like shared memory many–core computers, accelerators, and extreme scale distributed–memory computers. To address this demand, we focused on improving the performance of the solver on the many-core architectures.

In this paper, to provide a portable high-performance of solving a symmetric-indefinite linear system on many-core architectures, we rely on Open Multi Processing's (OpenMP's) new dataflow-based task scheduling, and we implement the "communication-avoiding" variant of Aasen's algorithm [5]. This algorithm can be implemented using the Basic Linear Algebra Subroutines (BLAS) and the LAPACK routines. Hence, to obtain high performance, we do not need to develop new optimized kernels to, for example, factorize a block column of the symmetric indefinite matrix—which could be difficult because of symmetric pivoting—but instead rely on the standard software packages whose high-performance implementations are readily available on many architectures. This decision improves the portability of our solver. Though a similar implementation of the algorithm has been previously developed using the QUeuing And Runtime for Kernels (QUARK) [6] runtime system, this paper contains several new contributions, as listed in Section 3.

The rest of the paper is organized as follows. Sections 2 and 3 list the related works and this paper's contributions, respectively. Section 4 describes our new OpenMP-based PLASMA framework. Section 5 presents the algorithms studied in this paper. Section 6 describes our implementation. Section 7 outlines our experiments' configuration, and Section 8 presents our results. Final remarks are listed in Section 9. Throughout the paper, we use $a_{i,j}$ and $\mathbf{a}_j$ to denote the (i,j)-th entry and the j-th column of the matrix A, while $A_{I,J}$, $A_{:,J}$, and $A_{I,:}$ are the (I,J)-th square block, the J-th block column, and the I-th block row of A,

- *University of Tennessee, Knoxville, Tennessee, U.S.A.*
- *The University of Manchester, Manchester, U.K.*

respectively. In addition, we use n to denote the dimension of the coefficient matrix, while n_b is the block size, and n_t is the number of block columns or rows in the matrix (i.e., $n_t = \frac{n}{n_b}$). We used random matrices for our performance studies and computed the GFlop/s for all the symmetric indefinite factorization using the FLOP count needed for the factorization without pivoting (i.e., $\frac{1}{3}n^3 + \frac{1}{2}n^2 + \frac{1}{6}$ FLOPs).

2 Related Works

2.1 Symmetric Indefinite Solvers

The availability of the symmetric indefinite linear solvers vary in the numerical linear algebra packages that are publicly available today. For solving the symmetric indefinite linear system of equations on the shared-memory computers, LAPACK is still the de facto package, and at the time of preparing this paper, it implements the blocked variants of the Bunch-Kaufman [1], rook pivoting [2], and Aasen's [7] algorithms (see Section 5.1). Among these three algorithms, the Bunch-Kaufman is often the fastest [8] and is used to compare the performance of our solver against.[1] A commercial implementation of LAPACK, such as Intel MKL or IBM ESSL, may provide the vendor-optimized versions of the solvers on the specific architecture.

In contrast, many of the popular open-source linear algebra packages, such as ATLAS [9], OpenBLAS [10], libFLAME [11], and Eigen [12], still do not support the symmetric indefinite solver (with pivoting to ensure the numerical stability). Only exception is Elemental [13] which implements, beside the Bunch-Kaufman algorithm, the unblocked variant of the Bunch-Parlett algorithm [14] (hence it may be more accurate but slower than the Bunch-Kaufman implementation).

2.2 Runtime Systems

The superscalar scheduling technique was pioneered by the software projects at the Barcelona Supercomputer Center. These projects include GridSs, CellSs, OMPSs, and StarSs, where "Ss" stands for the superscalar [15], [16], [17]. The researchers at INRIA and the Uppsala University have been also developing their own superscalar runtimes called StarPU and SuperGlue [18], [19], respectively. Finally, QUARK [20] was developed at the University of Tennessee, mainly for the development of the shared-memory dense linear algebra packages. The effectiveness of the superscalar schedulers to improve the parallel scalability of the algorithms has been demonstrated on many modern architectures including on the manycore architecture. These studies have lead to the recent adaptation of the techniques in the OpenMP standard. In this paper, we test these new OpenMP features for developing the portable high-performance symmetric indefinite linear solver on the manycore architecture.

3 Contributions

Though a similar implementation of the algorithm has been previously developed using the QUeuing And Runtime for Kernels (QUARK) [6] runtime system, this paper contains several new contributions, listed below.

- We developed a production-ready symmetric indefinite linear solver. Previous studies only implemented Aasen's algorithm and relied on LAPACK for the rest of the solver stages. As a result, the previous solver was not production ready and was not publicly released. The new OpenMP-based solver, on the other hand, is now available on our Parallel Linear Algebra Software for Multicore Architectures (PLASMA) project's Bitbucket repository [21].
- We improved the software portability and sustainability by relying on well-established standards like BLAS, LAPACK, and OpenMP instead of proprietary solutions like QUARK.
- We provided a comprehensive description of all the stages of our new implementation. Also, to conform to the OpenMP standard, several changes were made to our new Aasen's implementation compared with our original QUARK-based implementation (e.g., nested parallelization). In addition, for the complete solver to utilize many-core architectures, it needed to be implemented with care (e.g., merging the multiple stages of the solver).
- We tested our solver performance on new many-core architectures, including Intel's Broadwell, Intel's Knights Landing (KNL), IBM's Power8, and Arm's ARMv8. Our experimental results demonstrate the improved portability and superior performance of our implementation compared to vendor-optimized implementations of LAPACK.

4 PLASMA OpenMP Framework

In this section, we provide an overview of our new OpenMP-based PLASMA framework, including details on the tile layout, tile algorithm, and the OpenMP standard.

4.1 Tile Layout

PLASMA uses a special matrix layout to exploit the memory hierarchy of modern computers and to generate enough independent tasks conforming to the OpenMP standard.

4.1.1 General Matrix

While LAPACK stores the matrix in a column-major order, PLASMA stores the matrix in *tiles*, which are small square blocks of the matrix stored in a contiguous memory region. These tiles can be loaded into the cache memory efficiently and operated on with little risk of eviction. Hence, the use of the tile layout minimizes the number of cache misses and of translation lookaside buffer (TLB) misses, reduces false sharing, and maximizes the potential for prefetching.

PLASMA contains parallel and cache-efficient routines that convert LAPACK layouts to PLASMA layouts. These routines are currently implemented in an out-of-place fashion. Therefore, to convert from an LAPACK layout to a PLASMA layout, PLASMA uses a separate memory allocation to store the matrix in tiles—first storing the leading submatrix whose dimension is the largest multiple of the tile size that is less than or equal to n (i.e., the dimension of

1. After the completion of this paper, the latest release of LAPACK, version 3.8.0, now includes the two-stage variant of the Aasen's algorithm studied in this paper (see [8] for the performance comparison).

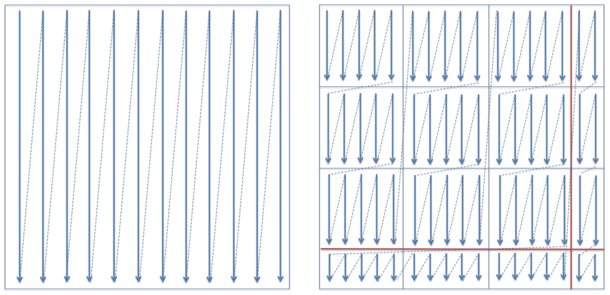


Fig. 1. Matrix layouts used in LAPACK and PLASMA.

$n - (n \bmod n_b)$), followed by the leftover tiles (Figure 1b). The matrix entries of each tile are stored in the column-major order, and the tiles in each submatrix are stored in column-major order. For all of our performance results, our input and output matrices are in the LAPACK layout, and—unless otherwise noted—the timing results include the time required for the layout conversion.

4.1.2 Triangular Matrix

For a symmetric matrix, PLASMA only stores the tiles in the triangular part of the matrix, which reduces the memory storage by roughly half of that required to store the full matrix layout (Figure 2).

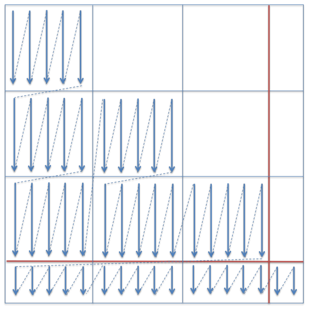


Fig. 2. PLASMA triangular format.

Our OpenMP-based solver often generates nested tasks that operate on a submatrix. This symmetric tile layout makes it easy to pass the trailing submatrix to the nested tasks (e.g., for symmetric pivoting) since the tiles in the submatrix are stored in the contiguous memory regions. On the other hand, since the tiles are stored in the column-major order in the full matrix layout, the tiles in each block column of the trailing submatrix are separated by the tiles on top of the submatrix (see Figure 1b).

4.1.3 Band Matrix

To reduce the storage cost, LAPACK compactly stores a band matrix in a $(k_\ell + k_u + 1)$-by-n array, where k_ℓ and k_u are the numbers of the subdiagonal and superdiagonal entries, respectively, within the band (see Figure 3). For a symmetric matrix, only the upper or lower triangular part of the band matrix is stored.

PLASMA's band matrix layout uses the same band layout as LAPACK, but each element $a_{i,j}$ becomes an n_b-by-n_b tile $A_{I,J}$. To simplify the implementation, the leftover tiles

Full layout

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

Band layout

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

Fig. 3. LAPACK's band matrix layouts.

have the leading dimension of n_d for the band matrix. For instance, for the full matrix, the leading dimension of each tile depends on the block row index, while in the band matrix, it depends on both the block and column indices (i.e., $A_{I,J}$ is stored in the $(\frac{k_u}{n_b}+I-J+1)$-th block row in the band layout, where $\frac{k_u}{n_b}$ is the number of tiles above the diagonal). By having the fixed leading dimension of the tiles in the band matrix, we can simplify our macro that returns the leading dimension, `plasma_tile_mmain(A, i)`, where `A` is a structure and stores the matrix type, and `i` is the block row index.

4.2 Tile Algorithm and Runtime

PLASMA implements LAPACK routines based on tiled algorithms that break a given algorithm into fine-grained computational tasks that operate on tiles. Figure 4 shows the tiled implementation of the matrix-matrix multiply (`ZGEMM`) where each call to the `core_omp_zgemm` routine represents a task. Operations on small tiles create fine-grained parallelism that provides enough independent tasks to keep a large number of cores busy. In addition, each task operates on the tiles in the cache before the tiles are evicted back to the main memory. Hence, this implementation improves overall performance because it maximizes the data reuse—where compared to the arithmetic operations on the tiles, the data movement through the memory hierarchy can be more expensive.

PLASMA relies on a runtime system to dynamically schedule computational tasks on physical cores based on the availability of required data. For this reason, it is also referred to as "data-driven scheduling." The concept of data-driven scheduling, in practice, is often described as expressing the algorithm as a directed acyclic graph (DAG) that consists of the tasks (vertices) and their data-dependencies (edges). At run time, these tasks are executed on different cores as their data dependencies are satisfied (see Section 4.3 for the details of our PLASMA implementation using OpenMP). In contrast, LAPACK is based on fork-join scheduling, where artificial joints expose synchronization points. As a result, with the fork-join scheduling, when each forked computation does not provide enough parallelism to utilize many cores, multiple cores will be idle waiting for the other cores to finish processing the assigned work.

```
for (int i = 0; i < C.mt; i++) {
    int mvci = plasma_tile_mview(C, i);
    int ldci = plasma_tile_mmain(C, i);
    int ldai = plasma_tile_mmain(A, i);
    for (int j = 0; j < C.nt; j++) {
        int nvcj = plasma_tile_nview(C, j);
        for (int k = 0; k < A.nt; k++) {
            int nvak = plasma_tile_nview(A, k);
            int ldbk = plasma_tile_mmain(B, k);
            plasma_complex64_t zbeta = k == 0 ? beta : 1.0;
            core_omp_zgemm(
                transa, transb,
                mvci, nvcn, nvak,
                alpha, A(i, k), ldai,
                       B(k, j), ldbk,
                zbeta, C(i, j), ldci,
                sequence, request);
        }
    }
}
```

Fig. 4. Tile ZGEMM algorithm where `A(i,j)` is a macro returning the starting memory location of the tile $A_{I,J}$. In addition, `plasma_tile_mview(A, i)` and `plasma_tile_nview(A, j)` return the numbers of the rows and columns of a tile in the i-th block row and the j-th block column of A, respectively, while `plasma_tile_mmain(A, i)` returns the leading dimension of a tile in the i-th block row.

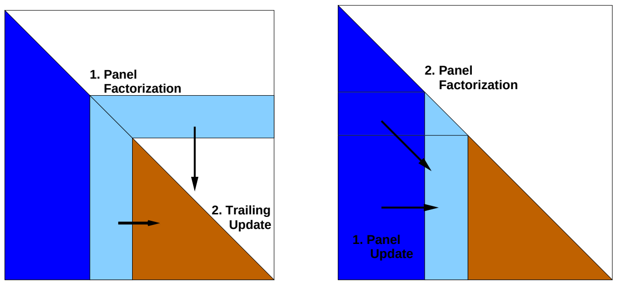
4.3 OpenMP Standard

OpenMP 3.0 introduced tasking that followed the simple Cilk model, which was then extended by OpenMP 4.0 to define the data dependencies among the tasks and enable dataflow-based task scheduling. OpenMP 4.5 further extended the tasking capabilities. For example, OpenMP 4.5 added task priorities that are critical for obtaining high performance using some of our PLASMA routines [22]. These OpenMP standards are supported by popular compilers, including the GNU Compiler Collection (GCC) and the Intel C Compiler (ICC). PLASMA now relies on these OpenMP features for portable performance on a wide range of many-core architectures.

```
#pragma omp task depend(in:A[0:lda*ak]) \
                 depend(in:B[0:ldb*bk]) \
                 depend(inout:C[0:ldc*n])
{
    if (sequence->status == PlasmaSuccess)
        core_zgemm(transa, transb,
                   m, n, k,
                   alpha, A, lda,
                          B, ldb,
                   beta,  C, ldc);
}
```

Fig. 5. OpenMP ZGEMM task `core_omp_zgemm` where `core_zgemm` is simply a wrapper around the BLAS ZGEMM.

Figure 5 shows an OpenMP task for computing `ZGEMM` on tiles. The `#pragma omp task` clause informs the compiler that the following code segment is a task. To define the data dependencies, the task's required data is provided by a memory pointer, an offset, a data size, and a type (e.g., `type:A[offset:size]`), where the data type can be either input, output, or input and output. For instance, the `in` tag specifies that the data is input, and in the DAG, the task becomes a descendant of all the previously inserted tasks that list the data as an output (specified by the `out` tag) or an input and output (specified by the `inout` tag). On the other hand, the `inout` or `out` dependence makes the task a descendant of the previously inserted tasks that lists the data as `in`, `inout`, or `out`.



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

Fig. 6. Two different updating schemes.

Then, at runtime, OpenMP keeps track of the data dependencies among the tasks and dynamically schedules the tasks on the many-core architecture while avoiding data hazards. In other words, a task is not scheduled until all of its ancestor tasks have completed and all of its data dependencies are satisfied. When the task completes, it then releases the data dependencies of its descendant tasks. Note that PLASMA's tile layout enables efficient data dependency tracking through OpenMP, which requires that each datum be stored in a contiguous memory region. It would be difficult to generate fine-grained tasks using the LAPACK column-major layout.

5 Algorithms

In this section, we describe the algorithms that LAPACK and PLASMA implement for solving the symmetric indefinite–linear system of equations.

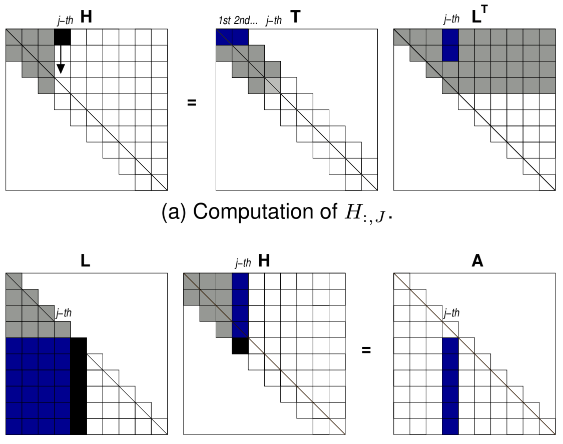
5.1 LAPACK

For solving the symmetric indefinite linear system $A\mathbf{x} = \mathbf{b}$, LAPACK factorizes the dense symmetric matrix A using either the Bunch-Kaufman [1], rook pivoting [2], or Aasen's algorithm [3], [7]. To improve the data locality and the performance of the factorization, the algorithms follow block factorization procedures. Specifically, the algorithms first factorize the leading block column (i.e., the panel), the result of which is then used to update the trailing submatrix. The same procedure is applied repeatedly to the trailing submatrix to factorize the entire matrix. This procedure is referred to as "right looking" because the panel is used to update the trailing submatrix, which is on the right side of the panel (Figure 6a).

Most of the FLOPS needed to factorize the matrix are performed for the trailing submatrix update where the level 3 BLAS matrix-matrix operations can be used. In particular, if n is the number of columns in A and n_b is the panel width, then the panel factorization of both the Bunch-Kaufman algorithm and Aasen's algorithm need a total of $O(n_b n^2)$ FLOPS, which is the lower term in the total number of FLOPS needed to factorize the whole matrix: $\frac{1}{3}n^3 + O(n^2 n_b)$ FLOPS in the Bunch-Kaufman algorithm and $\frac{1}{3}(1+\frac{1}{n_b})n^3 + O(n^2 n_b)$ FLOPS in Aasen's algorithm. Aasen's algorithm has the additional $\frac{1}{n_b}$ factor in the leading term of the FLOP count because it requires an additional rank-one update for each trailing submatrix update. In order to maintain the symmetry during the trailing submatrix update,

```
for $J = 1, 2, \dots, n_t$ do
    for $I = 2, 3, \dots, J-1$ do
        $H_{I,J} := T_{I,I-1}L^T_{J,I-1} + T_{I,I}L^T_{J,I} + T_{I,I+1}L^T_{J,I+1}$
    end for

    if $J > 2$ then
        $A_{J,J} := A_{J,J} - L_{J,2:J-1}H_{2:J-1,J} - L_{J,J}T_{J,J-1}L^T_{J,J-1}$
    end if
    $T_{J,J} := L^{-1}_{J,J}A_{J,J}L^{-T}_{J,J}$

    if $J < n_t$ then
        if $J > 1$ then
            $H_{J,J} := T_{J,J-1}L^T_{J,J-1} + T_{J,j}L^T_{j,j}$
        end if

        $A_{J+1:n_t,J} := A_{J+1:n_t,J} - L_{J+1:n_t,2:J}H_{2:J,J}$
        $[L_{J+1:n_t,J+1}, H_{J+1,J}, P^{(J)}] :=$ LU$(A_{J+1:n_t,J})$

        $T_{J+1,J} := H_{J+1,J}L^{-T}_{J,J}$

        $L_{J+1:n_t,2:J} := P^{(J)}L_{J+1:n_t,2:J}$
        $A_{J+1:n_t,J+1:n_t} := P^{(J)}A_{J+1:n_t,J+1:n_t}P^{(J)T}$
        $P_{J+1:n_t,1:n_t} := P^{(J)}P_{J+1:n_t,1:n_t}$
    end if
end for
```

Fig. 7. CA Aasen's [5], where the first block column $L_{1:n_t,1}$ is the first n_b columns of the identity matrix and $[L, U, P] = \text{LU}(A_{J+1:n_t,J})$ returns the LU factors of $A_{J+1:n_t,J}$ with partial pivoting such that $LU = PA_{J+1:n_t,J}$.



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

(b) Panel update.

Fig. 8. Illustration of Aasen's algorithm.

LAPACK updates one block column at a time (one column at a time to update the diagonal block). Since LAPACK uses the threaded BLAS to parallelize the factorization, it has the artificial synchronization point at the end of each BLAS call, and its parallelism is limited to that to update each block column.

To factorize each column of the panel, the Bunch-Kaufman algorithm and Aasen's algorithm both select a pivot from the remaining diagonals and then swap the corresponding row and column in a way that maintains the numerical stability and the symmetry of the factors (see Figure 11b). Since any remaining diagonal may be selected as the pivot, the entire trailing submatrix must be updated before the panel factorization can begin. On many-core architectures, the panel factorization could play a significant role in the factorization time since the level 2 BLAS matrix vector (or the level 1 BLAS vector operations) used to factorize the panel often obtains only a small fraction of the performance possible with level 3 BLAS.

5.2 PLASMA

To solve the symmetric indefinite–linear system of equations, PLASMA implements the CA variant of Aasen's algorithm [5], which computes the LTL^T factorization of the coefficient matrix A:

$$PAP^T = LTL^T,$$

where P is a row permutation matrix constructed to maintain the numerical stability of the factorization, L is a lower triangular matrix with unit diagonals, and T is a symmetric band matrix with a band width equal to the block size n_b.

The J-th step of the algorithm computes the j-th block column of an auxiliary Hessenberg matrix, $H = TL^T$, and uses the block column to update the panel:

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

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

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

where $P^{(J)}$ denotes the partial pivoting used for the numerical stability of the factorization. Figure 7 shows the pseudocode of the algorithm, and Figure 8 illustrates the main phases of the algorithm. The algorithm performs roughly the same number of FLOPS as the Bunch-Kaufman algorithm discussed in Section 5.1.

Unlike the right-looking algorithms in LAPACK that update each block column one panel at a time, this is a left-looking algorithm, where each step of the factorization uses level 3 BLAS to update the panel all at once using all of the previous block columns (see Figure 6b). However, the tiled implementation must be carefully designed because of the limited parallelism associated with potential write conflicts when updating each tile of the panel—a problem that we address in the next section.

6 Implementations

In this section, we describe our PLASMA symmetric–indefinite solver that consists of the following stages:

1) Factorization
    a) Aasen's LTL^T factorization (Section 6.2) with LU panel factorization (Section 6.1)
    b) Band LU factorization (Section 6.3)
2) Solve
    a) Forward substitution with the lower-triangular L-factor from Aasen's factorization, interleaved with the row pivoting (Section 6.5)
    b) Forward substitution with the L-factor from the band LU, interleaved with the partial pivoting (Section 6.4)
    c) Backward substitution with the upper-triangular U-factor from the band LU (Section 6.4)
    d) Backward substitution with the U-factor from Aasen's factorization (Section 6.5)
    e) Row interchange of the solution vectors with column pivoting from Aasen's factorization (Section 6.5)

6.1 LU Panel Factorization

One of the critical components of our solver is the LU panel factorization (Line 17 of Figure 7) which has been extensively studied and optimized in the past. The current tile implementation of the LU panel factorization in PLASMA is influenced, in particular, by the parallel cache assignment (PCA) [23] and parallel recursive panel factorization [24]. Some of our earlier work also provides a good overview of different implementations of the LU factorization in general [25].

Our LU panel factorization routine, discussed here, relies on internal blocking and persistent assignment of tiles to threads. Unlike past implementations, our LU panel factorization routine uses blocking instead of the relatively inferior plain recursion. Memory residency provides cache reuse for the factorization of sub-panels, while inner-blocking provides some level of compute intensity for the sub-tile update operations. The result is an implementation that is not memory bound and scales well with the number of cores.

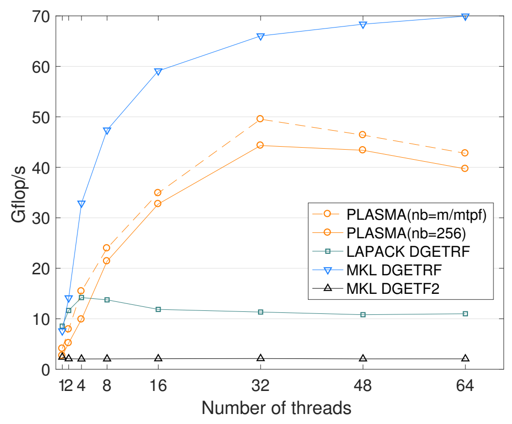


Fig. 9. Performance comparison of LU solver panel, excluding data layer translation ($m = 20000$, $n = 256$, and "mtpf" stands for the maximum number of threads used for the panel factorization and is the same as the number of threads in this figure).

Since panel factorization affects the entire block column, data-dependent tasks are created for the block column operations and not for tile operations. Hence, the dependency tracking is resolved at the granularity of the block columns, not at the granularity of the individual tiles. Because we store the tiles in column-major order, the tiles in each panel are stored contiguously in memory. Nested tasks are then created within each panel factorization and internally synchronized using thread barriers.

To improve the core utilization of symmetric indefinite factorization, we merge Aasen's algorithm with the band LU factorization, both of which use this LU panel factorization routine. Since the LU panel factorization from these two different stages of the algorithm may be executed concurrently, our LU panel routine ensures thread safety by using private workspaces and a private variable for synchronizing its nested tasks.

Figure 9 compares the performance of our LU panel factorization with that of Intel's Math Kernel Library (MKL) on Knights Landing. Though our LU routine obtained lower performance than the vendor-optimized MKL, it was more scalable than LAPACK linked to the threaded BLAS from MKL. Section 7 describes our experimental setups in details.

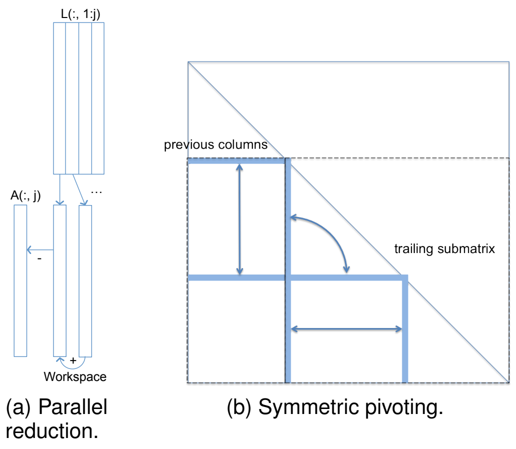
6.2 Aasen's Factorization

The panel update and the panel factorization are the two main computational kernels of Aasen's algorithm (Lines 16 and 17 of Figure 7). For the panel factorization, we used the LU routine from Section 6.1, which takes the tiles in the panel as input and generates the nested tasks. Therefore, when updating the panel, we also generate the nested tasks on the block columns such that we can then define the data dependencies between the panel update and the factorization without using the explicit synchronization with `#taskwait` (the panel factorization task must wait for the completion of all the tasks that update the panel tiles). Figure 10 shows our panel update code.

```
for (int j = 1; j <= k; j++) {
    // update L(:,k+1) using L(:,j) //
    // > block column to update with
    plasma_complex64_t *a1 = L(k, j);
    plasma_complex64_t *a2 = L(A.mt-1, j); // left-over tile
    plasma_complex64_t *b  = H(j, k);
    // > block column to be updated
    plasma_complex64_t *c1_in = A(k, k);
    plasma_complex64_t *c2_in = A(A.mt-1, k); // left-over tile
    plasma_complex64_t *c_out = A(k+1, k);    // panel
    ...

    int mvan = plasma_tile_mview(A, j);
    #pragma omp task depend(in:a1[0:ma1*na])      \
                     depend(in:a2[0:ma2*na])      \
                     depend(in:b[0:A.mb*mvak])    \
                     depend(in:c1_in[0:mc1_in*nc]) \
                     depend(in:c2_in[0:mc2_in*nc]) \
                     depend(out:c_out[0:mc_out*nc])
    {
        for (int i = k+1; i < A.mt; i++) {
            int mvai = plasma_tile_mview(A, i);
            int ldai = plasma_tile_mmain(A, i);
            #pragma omp task
            {
                core_zgemm(
                    PlasmaNoTrans, PlasmaNoTrans,
                    mvai, mvak, mvan,
                    -1.0, L(i, j), ldai,
                          H(j, k), A.mb,
                     1.0, A(i, k), ldai);
            }
        }
        #pragma omp taskwait
    }
}
```

Fig. 10. Left-looking update of $(k+1)$-th panel with nested parallelization.

The left-looking update of the panel dominates the computational cost of Aasen's algorithm. Though all of the panel's tiles can be independently updated in parallel, all the updates on each tile have write conflicts. Hence, its parallelism is inherently limited by the number of tiles in the panel. To increase the parallelism for updating each tile, we apply a parallel reduction; we first accumulate sets of independent updates into separate workspaces, and then—based on the binary-tree reduction—update the panel with the accumulated updates in the workspaces (see Figure 11a). The amount of parallelism that the algorithm can exploit depends on the number of tiles in the panel and the amount of the workspace provided. Our implementation allocates the workspace of size $3n_t n_b^2$ for the parallel reduction by default.



(a) Parallel reduction. (b) Symmetric pivoting.

Fig. 11. Optimization techniques for SYTRF.

After the LU panel factorization, the selected pivots are applied symmetrically to the trailing submatrix. Our symmetric pivoting routine takes the triangular part of the whole trailing submatrix as an input and generates the nested tasks. At each step, each thread is assigned with a subset of tiles in the current column, to which the pivot is applied (see Figure 11b). After each pivot is applied to the trailing submatrix, these nested tasks are synchronized before moving on to the next pivot. Since we store the matrix in the symmetric tile layout as shown in Section 4.1, these trailing submatrix tiles are stored contiguously in the memory. Figure 12 shows the effects of the parallel reduction and the parallel pivot on the factorization performance.

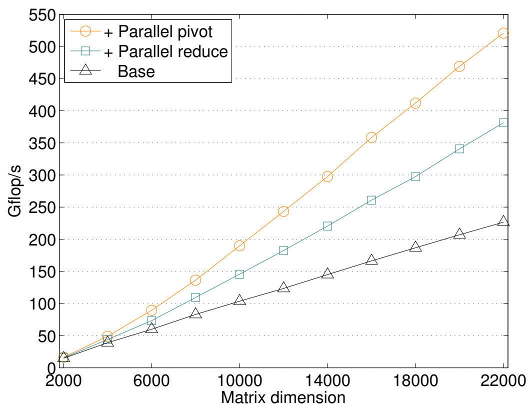


Fig. 12. Effects of parallel reduce and pivoting on DSYTRF.

Figure 13 illustrates the task dependency graph of Aasen's algorithm at the 4-th step of factorizing A with 6-by-6 tiles. The figure shows that the algorithm has several independent tasks, which the OpenMP runtime can schedule in parallel as soon as all of the data dependencies are satisfied. This parallel execution of the independent tasks is critical for utilizing a large number of cores. In contrast, LAPACK's fork-join paradigm relies on multi-threaded BLAS to exploit the parallelism and introduces the explicit join at the end of each BLAS call. This fork-join paradigm creates unnecessary synchronization points and limits the amount of work that many-core architectures can execute at a time,

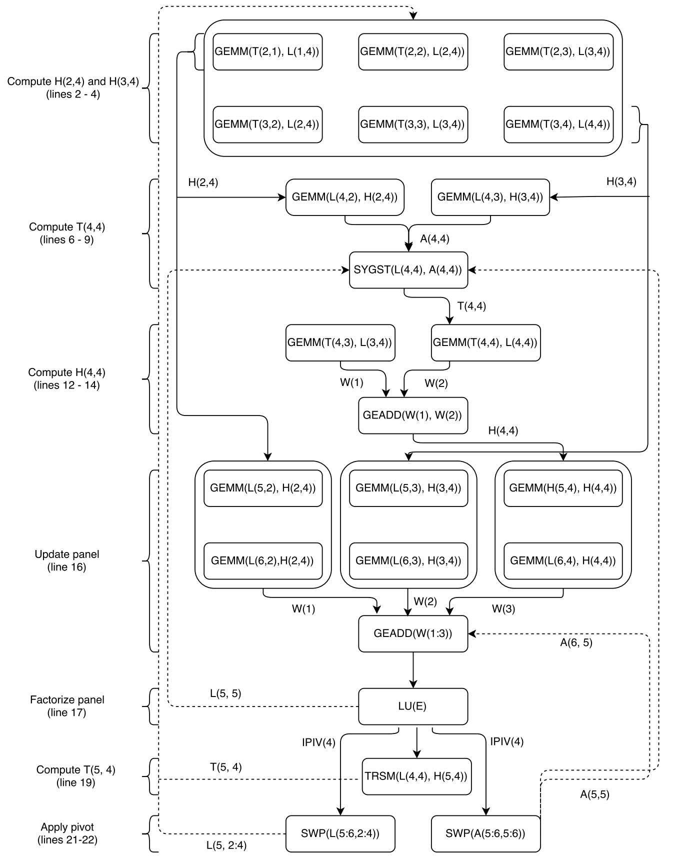
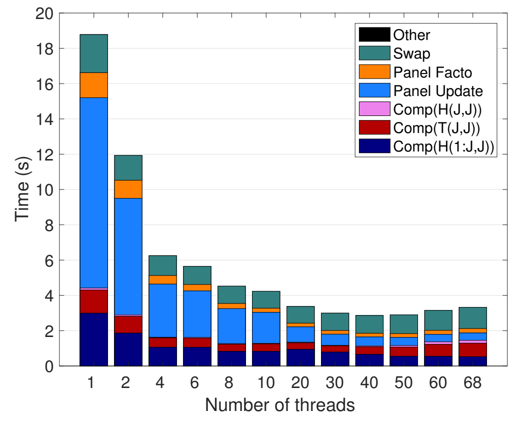


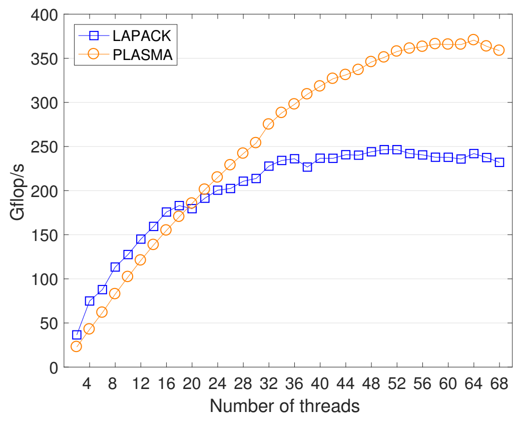
Fig. 13. Task dependency a as a DAG for the 4th step ($J = 4$) and $n_t = 6$ as in Figure 7. Arrow indicates a task dependency. Dashed arrow from bottom to top indicates a cross-iteration dependency; it goes to the same task but in the **next** iteration, which is not shown in the graph.

which then limits the parallel efficiency and scalability of the program. Figure 14 compares the performance of our PLASMA implementation of Aasen's algorithm to our LAPACK implementation of Aasen's algorithm. Though some computational kernels like LU panel factorization from the vendor-optimized LAPACK or BLAS may obtain higher performance, PLASMA implementation was able to obtain superior scalability because some of the small computational kernels, like applying the symmetric pivoting and updating the diagonal block $A_{J,J}$, did not scale well and became the performance bottleneck in the LAPACK implementation. Furthermore, this figure only compares the performance of the first stage of the symmetric solver. The real power of the task-based programming is that it allows us to merge the several stages of the solver, which will be discussed in the following subsections.

Merging the several steps of the symmetric indefinite solver is critical for utilizing a large number of cores because each stage has only limited amount of parallelism. For example, though Aasen's algorithm has a few independent tasks that can be executed in parallel, its data dependencies also prevent us from using some of the standard techniques to enhance the solver performance. Such techniques include "lookahead," one of the critical techniques for obtaining scalable performance with the LU factorization, used in many high-performance software packages, including PLASMA, HPL [26], and MAGMA [27]. The main



(a) Time breakdown of LAPACK.



(b) Performance

Fig. 14. LAPACK and PLASMA implementaions of Aasen's on KNL ($n = 20000$, $n_b = 256$). LAPACK implementation is linked with the threaded MKL.

idea behind the lookahead technique is hiding the panel factorization—based on level 1 and level 2 BLAS—behind the update, which is based on level 3 BLAS. This technique allows us to obtain a factorization performance close to that of level 3 BLAS. To achieve this performance with Aasen's algorithm's left-looking update, we want to start updating the J-th panel while factorizing the $(J-1)$-th panel. Unfortunately, the symmetric pivoting leads to data accesses and dependencies that make the lookahead technique more difficult to execute. In particular, the computation of the J-th block columns $W_{:,J}$ and $H_{:,J}$ has the input dependency on the J-th row $L_{J,:}$ (see Figure 7). However, $L_{J,:}$ becomes available only after applying the symmetric pivoting from the $(J-1)$-th panel factorization to the trailing submatrix—these pivots are applied to bring the pivot rows to the J-th row. These J-th block columns of W and H are then used to update the J-th panel. Hence, there are no other tasks that can be executed during the panel factorization. Since the parallelism of the panel factorization is limited by the number of tiles in the panel, the panel factorization can utilize fewer cores as the factorization proceeds. This lack of lookahead makes it challenging to obtain high performance with the overall LTL^T factorization. We address this lack of parallelism by merging the several steps of our solvers as discussed in the following sections.

6.3 Band LU Factorization

It is a challenge to stably factorize the symmetric band matrix T, which is generated by the first stage of Aasen's algorithm, while preserving its band structure. This is because the symmetric pivoting (required to ensure the numerical stability) can completely destroy its band structure. For example, for this reason, LAPACK currently does not support a symmetric indefinite band solver. Hence, our current implementation stores T as a non-symmetric band matrix and computes the non-symmetric band LU factorization with partial pivoting.

Though we operate on tiles, our implementation of the band LU solver follows that of LAPACK (i.e., a band LU factorization [`xGBTRF`] and a solver [`xGBTRS`]) and exhibits the same numerical behavior. For the band LU factorization with the partial pivoting, the main concern is that when the row pivot is applied to the previous columns of L, it could destroy the entire band structure. Accommodating this worst-case scenario requires the full storage for L. To avoid this additional storage, LAPACK performs a right-looking update and only applies the pivot to the trailing submatrix, thereby maintaining the band structure of L. With this scheme, LAPACK only needs to store $1 + k_u + 2k_\ell$ numerical values for each column of the factors and only introduces—at most—k_l fills per column of U.

Like our Aasen's algorithm implementation, our band LU implementation factorizes the panel using the LU panel factorization from Section 6.1. After factorizing each column of the panel, the panel factorization routine applies the pivot to the previous columns of the panel. For this reason, it could introduce fills in the last tile of the band panel, thereby destroying the lower-triangular structure of the tile. Hence, compared with LAPACK, our band LU requires additional storage to accommodate up to $n_b - k$ fills in the k-th column of the panel.[2]

After the panel factorization, the panel is then used to update the trailing submatrix in a right-looking fashion. Similar to how our Aasen's algorithm works (described in Section 6.2), our band LU implementation generates nested tasks for updating each block column of the trailing submatrix to avoid `#taskwait` between the panel factorization and the trailing submatrix update. In addition, during the right-looking update, a higher priority is assigned for updating the next panel so that the panel factorization may be started while the rest of the block columns are still being updated.

Figure 15 compares the performance of our band LU implementation to the performance of MKL and LAPACK running on KNL. Our band LU was slower on a single core, which may be because BLAS obtained lower performance on tiles compared to the larger blocks used by LAPACK. However, our implementation scaled better and obtained similar performance on a larger number of cores. We also tested tracking the fills in U and skipping the updates with empty blocks. However, for the random matrices used in this experiment, a significant amount of fills were introduced, and this did not significantly improve the perfor-

2. It is possible to apply pivoting during the layout translation to have this additional storage only during the tiled factorization. However, PLASMA does not implement this.

mance. By default, our routine updates all the tiles that hold the potential fills, even though they might be updated with empty tiles.

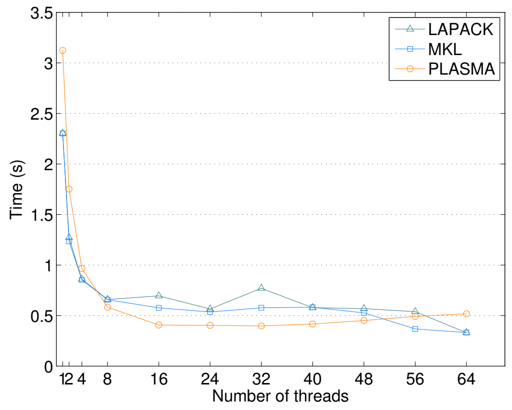


Fig. 15. Performance comparison of band LU factorization DGBTRF ($n = 10000$ and $k_u = k_\ell = 1000$. LAPACK is linked to the threaded BLAS from MKL.

When the band LU factorization is used alongside Aasen's factorization, there is the potential to merge these two algorithms, i.e., xGBTRF can start factorizing $T_{:,J}$ once it is computed by Aasen's algorithm. For the symmetric-indefinite solver, merging the two algorithms is critical for improving the parallel performance because the tiled Aasen's algorithm generates the band matrix with the band width equal to the tile size (i.e., $k_u = k_\ell = n_b$). Hence, there are only three tiles in each block column of T, which leads to severely limited parallelism for the band LU. To merge these two algorithms, the thread-safe LU panel from Section 6.1 is critical since the panel factorization of Aasen's algorithm and the band LU algorithm may execute concurrently.

To merge Aasen's factorization and the band LU factorization, we must resolve one data dependency: while our implementation stores T in a separate allocation to be factorized using xGBTRF—with additional space for the fills—the J-th step of Aasen's algorithm reads $T_{1:J-1,:}$ to compute the J-th columns of W and H. In order to merge these two algorithms, we need to store a copy of T to complete the LTL^T factorization, while xGBTRF starts factorizing T. Though our current implementation uses a separate workspace to store a copy of T, we can avoid this additional storage through careful planning. 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 may store the $(J+1)$-th block column of L in the J-th block column of A. 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., $T_{J,J}$ can be stored in $A_{J,J}$, and $T_{J+1,J}$ can be stored in the upper-triangular part of $A_{J+1,J}$).

6.4 Band LU Solver

Our forward substitution of the band LU solver, xGBTRS, also generates the nested tasks (Figure 16). The nested parallelization is needed because at each J-th step, the pivoting is applied over the multiple tiles of the right-hand side, $B_{J:m_t,:}$, and the xGEMM updates on all these tiles must be completed before the pivoting is applied. Aside from nested parallelism, the standard tiled substitution algorithm also enjoys the pipelining effects, i.e., as soon as the J-th step updates the tile in the I-th block row of B, the $(J+1)$-th step can start its update on the tile. Though our nested parallelization prevents this pipelining, the pivoting at each step does not allow this pipelining. Since there is no explicit synchronization between xGBTRF and xGBTRS, the band LU factorization and the forward substitution algorithms may be merged. Namely, xGBTRF does not apply the pivots to the previous columns of L. Hence, after the J-th step, $L_{:,J}$ is not modified, and the forward substitution with $L_{:,J}$ can begin.

```
for (int j = 0; j < B.nt; j++) {
    a00 = A(k, k);
    a10 = A(imin(k+A.klt, A.mt)-1, k);
    b00 = B(k, j);
    b10 = B(A.mt-1, j);
    b11 = B(k1, j);

    plasma_desc_t view = plasma_desc_view(B,
                                          0, j*A.nb,
                                          A.m, nvbj);
    view.type = PlasmaGeneral;

    #pragma omp task depend (inout:b00[0:mb00*nb])  \
                     depend (inout:b10[0:ldb10*nb]) \
                     depend (in:ipiv[k*A.nb:k*A.nb+mvbk])
    {
        core_zgeswp(PlasmaRowwise, view,
            k*A.nb+1, k*A.nb+mvbk, ipiv, 1);
    }

    #pragma omp task depend (in:a00[0:ma00*na])  \
                     depend (in:a10[0:ma10*na])  \
                     depend (inout:b00[0:mb00*nb])  \
                     depend (inout:b10[0:ldb10*nb]) \
                     depend (inout:b11[0:mb11*nb])
    {
        core_ztrsm(
            side, uplo, trans, diag,
            mvbk, nvbj,
            lalpha, A(k, k), ldak,
                    B(k, j), ldbk);

        for (int i = k+1; i < imin(k+A.klt, A.mt); i++) {
            int mvbi = plasma_tile_mview(B, i);
            int ldai = plasma_tile_mmain(A, i);
            int ldbi = plasma_tile_mmain(B, i);
            #pragma omp task
            {
                core_zgemm(
                    PlasmaNoTrans, PlasmaNoTrans,
                    mvbi, nvbj, B.mb,
                    -1.0,   A(i, k), ldai,
                            B(k, j), ldbk,
                    lalpha, B(i, j), ldbi);
            }
        }
        #pragma omp taskwait
    }
}
```

Fig. 16. Forward substitution for band LU solver.

There is an implicit synchronization between the forward and backward substitutions since the last tile of the solution vectors updated by the forward substitution is needed by the first task of the backward substitution. Also, the pivoting is not applied during the backward substitutions, and the nested parallelization is not needed. This is why we explicitly synchronize before the backward substitution and use a standard tiled algorithm that allows the pipelining of the substitution tasks. Figure 17 compares the performance of the DGBSV band LU solver. Our implementation obtained higher performance than MKL even on a single core. This may be because, though LAPACK's band layout reduces the memory requirement (up to $n_b - 1$ less numerical values per column than our tile layout), it requires level 2 BLAS operations. On the other hand, PLASMA's tile layout allows us to use level 3 BLAS kernels, which are often better optimized. We see in this experiment that merging the factorization and forward substitution did not significantly improve the performance—either because there were not enough cores or not enough work for the solver. However, merging the two algorithms can be more significant in the context of the symmetric indefinite solver, where each block column of the band matrix has only three blocks, leading to severely limited parallelisms in both factorization and substitution algorithms.

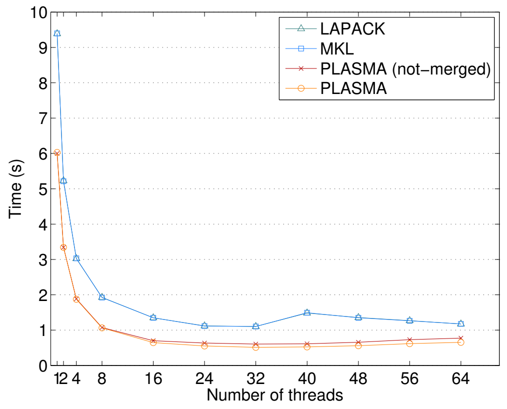


Fig. 17. Performance comparison of band LU solver DGBSV ($n = 10000$ and $k_u = k_\ell = 1000$).

6.5 Solver

When the matrix and the right-hand sides are available at the same time, an opportunity exists to merge Aasen's factorization and the forward substitution with L. However, there are data dependencies that prevent a simple merge. Specifically, Aasen's algorithm applies the pivoting to the previous columns of L, and if the solver tasks are inserted after the factorization tasks, the solve tasks cannot start until all the pivoting is applied to the previous columns. As a result, merging the algorithms must be done manually.

To merge the two algorithms: at the J-th step of Aasen's, we inserted the forward substitution with $L_{:,J}$ placed after the right-looking update of the panel, but before the panel factorization, to compute $L_{:,J+1}$. The goal is to overlap the forward substitution with the panel factorization since the panel factorization often lacks the parallelism required to occupy a large number of cores. Unfortunately, this may not lead to successful overlap, because the substitution tasks can be scheduled much sooner. To enforce the overlap, we set the data dependency of $L_{:,J}$ for the right-looking update to be `inout` instead of `in`.

Finally, like in the band LU solver, we use a standard tiled algorithm for the backward substitution with the U-factor from Aasen's algorithm. This allows the pipelining of the substitution tasks and also the two backward substitution algorithms to merge: one with the U-factor from the band LU and the other with the U-factor from Aasen's algorithm. The backward substitution ends with the top tiles of the right-hand side, which are passed to the tasks that then apply the pivoting from Aasen's algorithm to the corresponding block columns of the right-hand sides. The resulting execution trace and performance are shown in Figures 18 and 19. Merging the algorithms brought a greater performance improvement as the number of threads increases, garnering up to a 25% reduction in time to solution.

7 Experimental Setups

This section describes the hardware and setups used for our performance studies. For our experiments, we used all the available cores of the machine, launching one thread per core. We did not enable hyper-threading since our solvers are mostly compute bound.

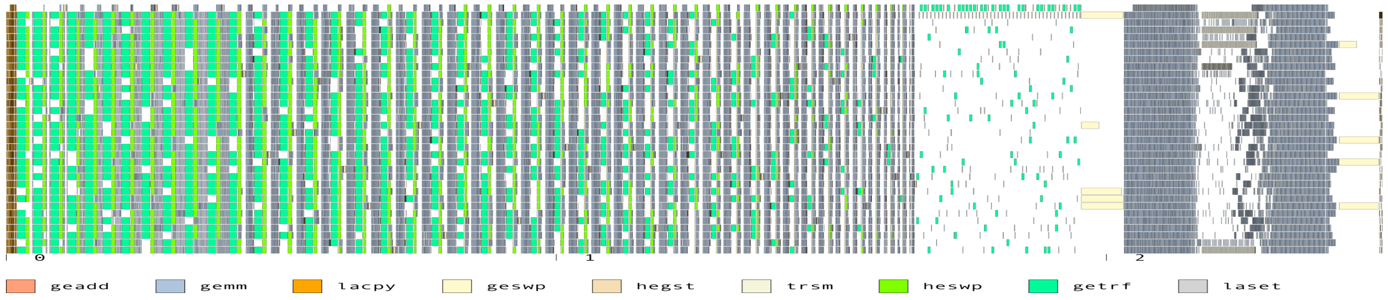
7.1 Broadwell

Broadwell is the codename of Intel's 14 nm die shrink of the Haswell CPU microarchitecture. While there is a wide variety of Broadwell chips with different configurations, the experiments in this work used a dual-socket motherboard with two Intel Xeon E5-2690v4 CPUs (Broadwell, 14 cores per socket, 28 cores total). Each core has 64 kB of L1 cache, 32 kB of L1 data cache, 32 kB of L1 instruction cache, 256 kB of L2 cache, and 35 MB of L3 cache that is shared between 14 cores (1 socket). The memory is shared across the whole node, but 64 GB is placed on each non-uniform memory access (NUMA) island, and the memory controller on each NUMA node has 4 channels. The base frequency of each core is 2.5 GHz and can be increased up to 3.8 GHz using "Turbo Boost." However, setting all of the cores to use Turbo Boost limits the frequency of each core to 2.9 GHz. In terms of instruction set, the microarchitecture implements advanced vector extensions (AVX2) and fused multiply–add (FMA3), which enables each core to achieve up to 16 FLOPS/cycle in double-precision arithmetic. Consequently, in double precision, the theoretical peak performance of the whole node (28 cores) is 1.20 TFlop/s when set in base frequency mode, and 1.30 TFlop/s when using Turbo Boost.

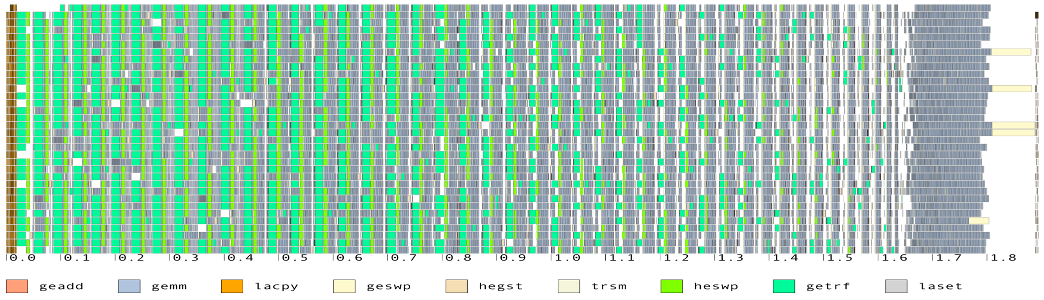
We compiled PLASMA using GCC 6.3 and linked to MKL 2017 for its single-thread BLAS functions. For a performance baseline, we used LAPACK 3.7.0 downloaded from Netlib, compiled using GCC 6.3, and linked to MKL 2017 for its multi-threaded BLAS functions. For the sake of comparison, we also report the performance of MKL's DSYTRF/DSYSV routines.

7.2 Knights Landing

KNL, also known as Knights Landing, is the codename for the second generation of Intel's Xeon Phi Many Integrated Core (MIC) Architecture. The 7230 and 7250 KNL variants utilizes 14 nm lithography similar to the Broadwell Xeon E5 and E7 server processors to achieve 3.05 TFlop/s of peak performance in double precision. The 7250 KNL chip's 68 cores are also based on the out-of-order Silvermont microarchitecture and support four execution threads. Both the 7230 and 7250 KNL variants have two AVX512 vector processing



(a) before merging algorithms.



(b) after merging algorithms.

Fig. 18. Execution trace of `DSYSV` with 34 threads.

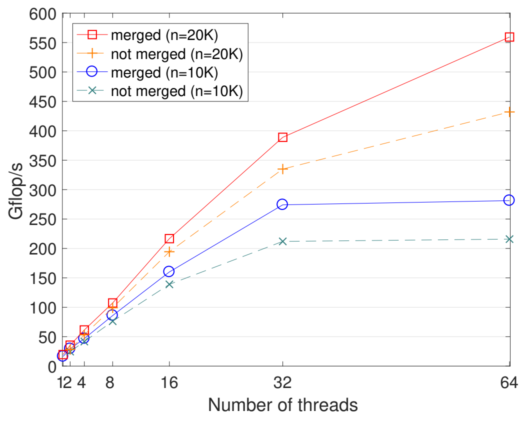


Fig. 19. Effects of merging algorithms on `DSYSV`. Half of the available threads were used for the panel factorization.

units per core. There are two types of memory used in KNL: a larger 96 GB of DDR4-2400 memory providing up to 115.2 GB/s of bandwidth and a small 16 GB of MCDRAM providing up to 490 GB/s of sustained bandwidth through the 2-D mesh interconnect. The memory bandwidth per KNL core is approximately 11 GB/sec for small thread counts.

For our experiments, we used the 16 GB high speed MCDRAM as a NUMA node (flat mode) instead of transparent cache for DRAM (cache mode). We use Quadrant cluster mode. PLASMA is compiled using GCC 7.0.1 and linked with MKL (2017.2.174) for its single-thread BLAS functions. For the performance baseline, we downloaded LAPACK 3.7.0 from Netlib and linked with MKL for its multi-threaded BLAS functions. We show the LAPACK's `DSYTRF`/`DSYSV` performance, compiled using both GCC 7.0.1 and ICC 16.0.3 since they obtained higher performance using ICC. We also show the performance of MKL's `DSYTRF`/`DSYSV` routines.

7.3 POWER8

IBM's POWER8 [28] is a reduced instruction set computer (RISC) fabricated using 22 nm technology that features up to 12 cores per socket with 8 threads per core. For memory, the POWER8 boasts (per core) 32 KB of instruction cache, 64 KB of L1 data cache, 512 KB of L2 cache, and 8 MB of L3 cache. Each core can issue up to 10 instructions per cycle and complete 8 instructions per cycle. The POWER8 has two independent fixed-point units (FXU), two independent load-store units (LSU) plus two more load units and two independent floating-point vector/scalar units (VSUs). The maximum double-precision floating-point issue rate is 4 FMAs per cycle. The maximum single-precision floating-point issue rate is 8 FMAs per cycle. The single instruction, multiple data (SIMD) width is 2 for double precision and 4 for single precision. Thus, when the peak performance is computed as the frequency $\times$ 8 $\times$ number of cores, the peak performance in double precision is 560 GFlop/s for a 3.5 GHz frequency and 20-core node.

We used a dual-socket IBM POWER8 compute node with 20 cores running at a roughly 3.5 GHz clock speed. We compiled PLASMA using GCC 6.3.1, linked with Engineering and Scientific Subroutine Library (ESSL) 5.5.0 for its single-thread BLAS functions. We show the results of LAPACK 3.7.0 from Netlib, linked with ESSL 5.5.0 for its multi-threaded BLAS functions. We show the LAPACK results using both IBM's XL compiler (version 20161123) and GCC 6.3.1 because LAPACK obtained higher performance using the XL compiler. The baseline ESSL runs used the multi-threaded `DSYTRF` and `DSYSV` directly from ESSL 5.5.0. For PLASMA runs, we used environment variables to peg each thread to one physical CPU core. We only used one

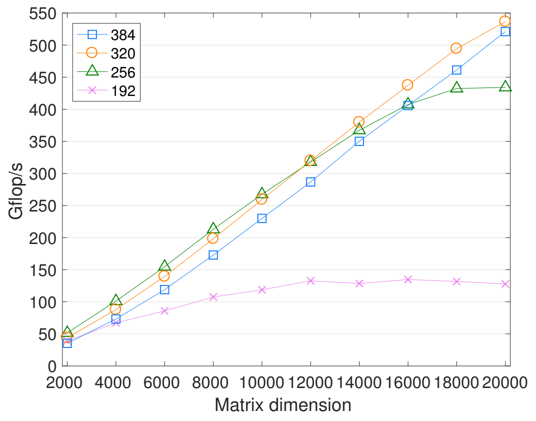


Fig. 20. Effects of block size on the performance of DSYSV on KNL.

thread out of the 8 available threads per core because one thread was able to fully utilize the floating point units and maximize the memory bandwidth while also providing the best performance.

7.4 ARMv8

For this experiment, we used Cavium's dual-socket, 96-core ThunderX compute node. ThunderX, a 64-bit ARMv8 processor, features up to 48 cores with a 2.0 GHz clock rate and is fabricated using a 28 nm process. The double-precision FMA performance is around 4 GFlop/s per core and the peak double precision performance is 384 GFlop/s on the node.

We compiled PLASMA using GCC 6.1.0 and linked with OpenBLAS [10] pre-0.2.20 for its single-threaded BLAS functions. We downloaded LAPACK 3.7.0 directly from Netlib, compiled using GCC 6.1.0, and linked with multi-threaded OpenBLAS. The vendor optimized BLAS/LAPACK package LibSci 17.03 is also used for its DSYTRF and DSYSV functions. Due to a thread safety bug in serial BLAS functions of LibSci, we can only get the multi-threaded DSYTRF/DSYSV performance using the Cray compilers 8.5.8. For all of our measurements, we used the OpenMP environment variables to bind each thread to each physical core.

7.5 Block size

Block size is a critical parameter that must be tuned to obtain optimum performance for PLASMA. A smaller block size increases the parallelism, but it lowers the performance of the computational kernels that each task invokes (e.g., sequential BLAS). For example, Figure 20 shows the effects of the block size on the performance of DSYSV on KNL. Obviously, the block size needs to be chosen carefully. We show the optimal block sizes with the results in Section 8, but to keep things simple, our results in both Sections 6 and 8 only show PLASMA performance with near-optimal, properly tuned blocksizes.

8 Experimental Results

Figure 21 shows the performance of PLASMA's DSYTRF factorization routine (Aasen's followed by band LU) on four different architectures and compares it against the performance of the LAPACK reference implementation linked with the vendor-provided BLAS and also against the performance of the vendor-provided LAPACK. On Broadwell, compared to the reference LAPACK implementation, PLASMA was able to more effectively utilize the 28 cores even with a relatively small matrix, while LAPACK needed larger matrices to fully utilize the cores. We see that Intel has optimized MKL and obtains much higher performance, especially for small matrices, compared to LAPACK. However, even with a relatively small matrix, PLASMA came very close to the performance of the vendor-optimized routine. PLASMA's performance advantage over MKL significantly widened as we increased the matrix size. This performance advantage was facilitated by a combination of the algorithmic redesign and its careful implementation within PLASMA.

On KNL and ARMv8, even PLASMA required bigger matrices to fully utilize the large number of cores, and its performance continued to increase as the matrices grew in size. However, in contrast to the Broadwell results, as the matrix size grew, PLASMA obtained greater speedups than LAPACK (up to 4.8 and $5.2\times$ respectively) on ARMv8 and KNL. On POWER8, PLASMA requires larger matrices to surpass the performance of ESSL/LAPACK. Also, the performance of ESSL and LAPACK plateaued at the matrix sizes of around 7,000 and larger, which we suspect is because of the large and deep four level on–chip cache system, which boasts 8MB of L3 cache per physical core and 16MB of L4 cache per memory controller, and possibly because of the NUMA system in the main memory. PLASMA, owing to its tile data layout, seems to be better suited to exploit complicated cache systems.

Figure 22 compares the performance of the DSYSV symmetric solver. The results are similar to those of the factorization shown in Figure 21, but PLASMA obtained greater speedups by merging different stages of the solver.

9 Conclusion

In this work, we developed a dense symmetric–indefinite solver for many-core architectures. To fully utilize a large number of cores and to obtain portable performance, we relied on OpenMP's task-based programming and on its runtime system. OpenMP breaks the artificial synchronization points and allows independent tasks from different stages of the solver to be executed in parallel. Our performance studies of current many-core architectures, including KNL, POWER8, and ARMv8, demonstrated the superior portable performance of our PLASMA solver when compared to LAPACK.

Moving forward, we are investigating different implementations of the algorithms to further improve their performance and portability. For instance, LU panel factorization is a critical component that requires additional development to ensure synchronization among the nested tasks. We would also like to see if the solver can be ported to run on accelerators and on distributed-memory computers.

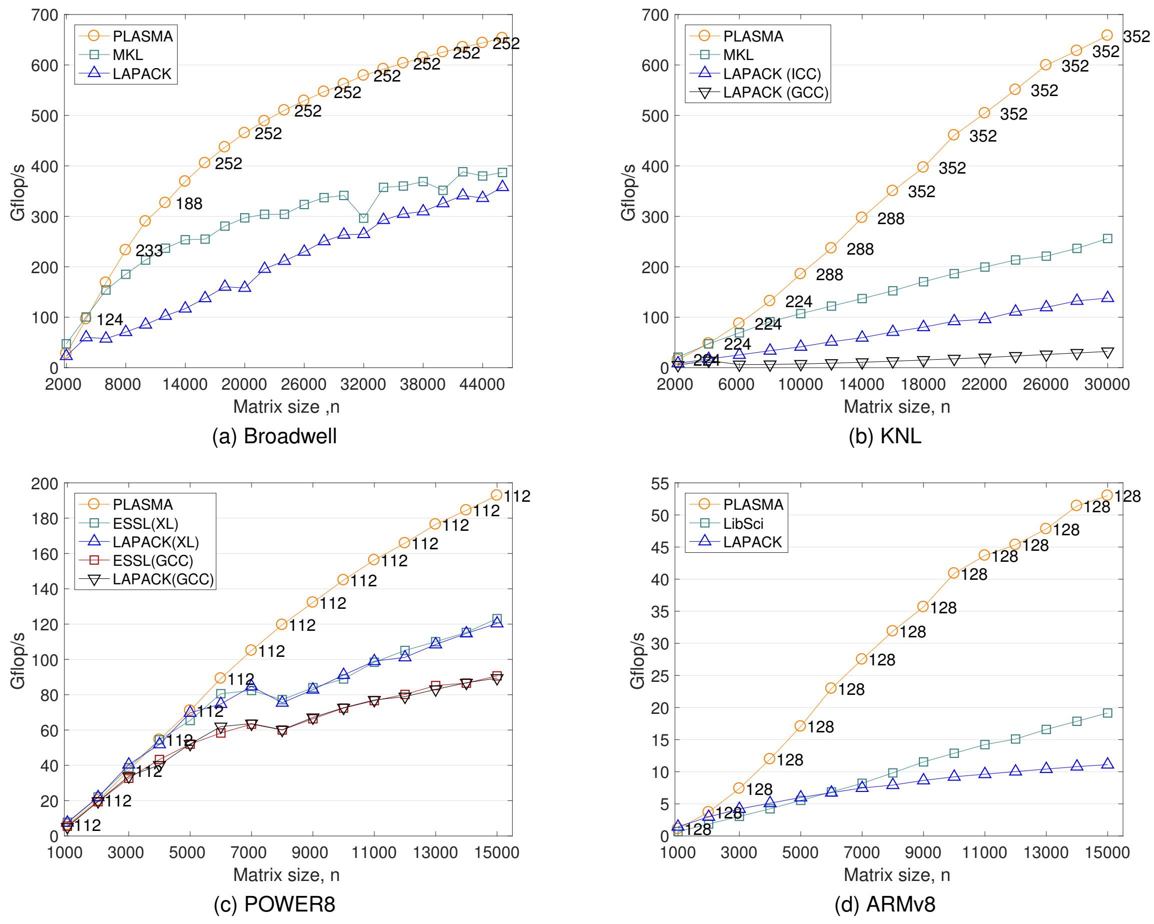


Fig. 21. Performance of DSYTRF where the numbers by the PLASMA markers represent the block sizes.

10 Future Work

The Aasen's algorithms described in this paper is our choice of the symmetric indefinite solver for the SLATE package (*Software for Linear Algebra Targeting Exascale*), developed as part of the *Exascale Computing Project* (ECP). ECP is a collaborative effort of the *U.S. Department of Energy Office of Science* (DOE-SC) and the *National Nuclear Security Administration* (NNSA), established with the goals of maximizing the benefits of high performance computing for the United States and accelerating the development of a capable exascale computing ecosystem.

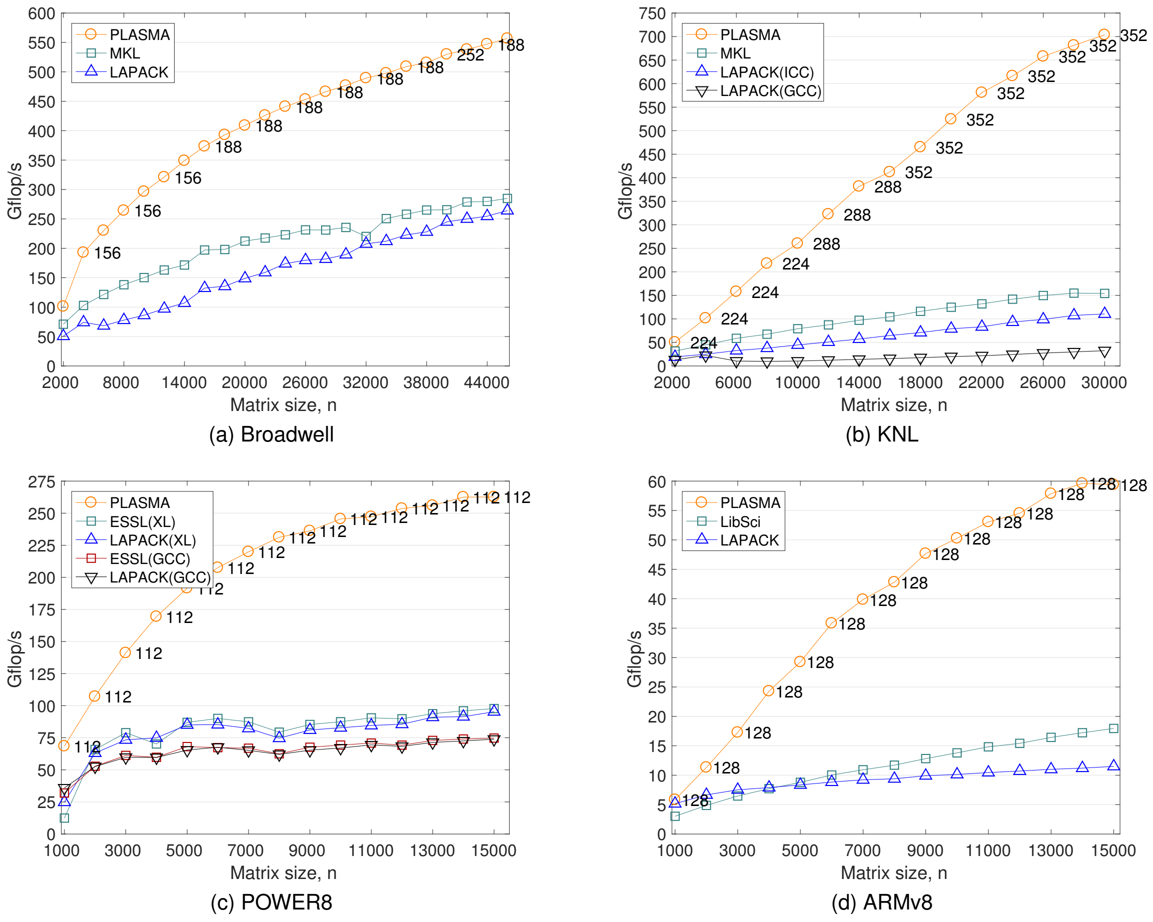
The objective of SLATE is to serve as a replacement for ScaLAPACK for the next generation of DOE's machines, and, ultimately, to target exascale systems [29]. The symmetric indefinite solver is scheduled for a release in the third quarter of 2018, alongside the non-symmetric and symmetric positive definite solvers, and will support distributed memory execution with multi-GPU acceleration. The implementation described in this paper is uniquely suitable for SLATE, as SLATE relies on storing the matrix by tiles and dynamic scheduling of operations [30].

Acknowledgments

The authors would like to thank the members of the PLASMA project for the valuable discussions. We also thank the Intel Corporation for their generous hardware donation and continuous financial support and the Oak Ridge Leadership Computing Facility for providing access to the ARMv8 and POWER8 systems. This research was supported by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration, under prime contract #DE-AC05-00OR22725, and UT Battelle subaward #4000152412.

References

[1] J. Bunch, L. Kaufman, Some stable methods for calculating inertia and solving symmetric linear systems, Mathematics of Computation 31 (137) (1977) 163–179.
[2] C. Ashcraft, R. Grimes, J. Lewis, Accurate symmetric indefinite linear equation solvers, SIAM J. Matrix Anal. Appl. 20 (1998) 513–561.
[3] J. Aasen, On the reduction of a symmetric matrix to tridiagonal form, BIT 11 (1971) 233–242.
[4] LAPACK, http://www.netlib.org/lapack.
[5] G. Ballard, D. Becker, J. Demmel, J. Dongarra, A. Druinsky, I. Peled, O. Schwartz, S. Toledo, I. Yamazaki, A communication avoiding symmetric indefinite factorization, SIAM. J. Matrix Anal. Appl. 35 (4) (2014) 1364–1406.
[6] G. Ballard, D. Becker, J. Demmel, J. Dongarra, A. Druinsky, I. Peled, O. Schwartz, S. Toledo, I. Yamazaki, Implementing a blocked Aasen's algorithm with a dynamic scheduler on multicore architectures, in: Proceedings of the IEEE International Symposium on Parallel and Distributed Processing, 2013, pp. 895–907.
[7] M. Rozložník, G. Shklarski, S. Toledo, Partitioned triangular tridiagonalization, ACM Trans. Math. Softw. 37 (4) (2011) 1–16.



(a) Broadwell (b) KNL (c) POWER8 (d) ARMv8

Fig. 22. Performance of DSYSV where the numbers by the PLASMA markers represent the block sizes.

[8] I. Yamazaki, J. Dongarra, Aasen's symmetric indefinite solvers in LAPACK, LAPACK Working Note 294 , ICL-UT-17-13, University of Tennessee (2017).

[9] R. C. Whaley, A. Petitet, Minimizing development and maintenance costs in supporting persistently optimized BLAS, Software: Practice and Experience 35 (2) (2005) 101–121.

[10] Q. Wang, X. Zhang, Y. Zhang, Q. Yi, AUGEM: automatically generate high performance dense linear algebra kernels on x86 CPUs, in: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, ACM, 2013, p. 25.

[11] libFLAME, https://www.cs.utexas.edu/~flame.

[12] Eigen, http://eigen.tuxfamily.org.

[13] Elemental, http://libelemental.org.

[14] J. R. Bunch, B. N. Parlett, Direct methods for solving symmetric indefinite systems of linear equations, SIAM Journal on Numerical Analysis 8 (4) (1971) 639–655.

[15] R. M. Badia, J. R. Herrero, J. Labarta, J. M. Pérez, E. S. Quintana-Ortí, G. Quintana-Ortí, Parallelizing dense and banded linear algebra libraries using SMPSs, Concurrency and Computation: Practice and Experience 21 (2009) 2438–2456.

[16] A. Duran, E. Ayguadé, R. M. Badia, J. Labarta, L. Martinell, X. Martorell, J. Planas, OMPSS: a proposal for programming heterogeneous multi-core architectures, Parallel Processing Letter 21 (2011) 173–193.

[17] J. M. Pérez, P. Bellens, R. M. Badia, J. Labarta, CellSs: making it easier to program the cell broadband engine processor, IBM Journal of Research and Development 51 (2007) 593–604.

[18] C. Augonnet, S. Thibault, R. Namyst, P.-A. Wacrenier, StarPU: a unified platform for task scheduling on heterogeneous multicore architectures, Concurrency and Computation: Practice and Experience 23 (2011) 187–198.

[19] M. Tillenius, SuperGlue: a shared memory framework using data versioning for dependency-aware task-based parallelization, SIAM J. Sci. Comput. (2015) C617–C642.

[20] A. YarKhan, Dynamic task execution on shared and distributed memory architectures, Ph.D. thesis, University of Tennessee (2012).

[21] PLASMA, http://icl.utk.edu/plasma.

[22] A. YarKhan, J. Kurzak, P. Luszczek, J. Dongarra, Porting the PLASMA numerical library to the OpenMP standard, International Journal of Parallel Programming 45 (2016) 1–22.

[23] A. Castaldo, C. Whaley, Scaling LAPACK panel operations using parallel cache assignment, in: ACM Sigplan Notices, Vol. 45, 2010, pp. 223–232.

[24] J. Dongarra, M. Faverge, H. Ltaief, P. Luszczek, Achieving numerical accuracy and high performance using recursive tile LU factorization with partial pivoting, Concurrency and Computation: Practice and Experience 26 (7) (2014) 1408–1431.

[25] S. Donfack, J. Dongarra, M. Faverge, M. Gates, J. Kurzak, P. Luszczek, I. Yamazaki, A survey of recent developments in parallel implementations of Gaussian elimination, Concurrency and Computation: Practice and Experience 27 (5) (2015) 1292–1309.

[26] HPL, http://icl.utk.edu/hpl.

[27] MAGMA, http://icl.utk.edu/magma.

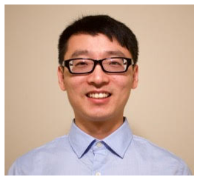
[28] B. Sinharoy, J. Van Norstrand, R. J. Eickemeyer, H. Q. Le, J. Leenstra, D. Q. Nguyen, B. Konigsburg, K. Ward, M. Brown, J. E. Moreira, et al., IBM POWER8 processor core microarchitecture, IBM Journal of Research and Development 59 (1) (2015) 2–1.

[29] A. Abdelfattah, H. Anzt, A. Bouteiller, A. Danalis, J. Dongarra, M. Gates, A. Haidar, J. Kurzak, P. Luszczek, S. Tomov, S. Wood, P. Wu, I. Yamazaki, A. YarKhan, Roadmap for the development of a linear algebra library for exascale computing: SLATE: Software for linear algebra targeting exascale, SLATE Working Note 1, Innovative Computing Laboratory, University of Tennessee (2017).

[30] J. Kurzak, P. Wu, M. Gates, I. Yamazaki, P. Luszczek, G. Ragghianti, J. Dongarra, Designing SLATE: Software for linear algebra targeting exascale, SLATE Working Note 3, Innovative Computing Laboratory, University of Tennessee (2017).

Ichitaro Yamazaki received his PhD degree in Computer Science from the University of California at Davis in 2008. He is currently a research scientist in the Innovative Computing Laboratory at the University of Tennessee at Knoxville, where his interests lie in high-performance computing, especially for linear algebra and scientific computing. He has also worked in Scientific Computing Group at Lawrence Berkeley National Laboratory from 2008 to 2011, as a Postdoctoral Researcher.

Jakub Kurzak received his PhD degree in computer science from the University of Houston. He is a Research Assistant Professor in the Department of Electrical Engineering and Computer Science, University of Tennessee. The main focus of his work is numerical software for solving large scale problems in technical and scientific computing.



Panruo Wu is an assistant professor in the Computer Science department of University of Houston. He received his PhD from University of California Riverside in 2016. He worked as a postdoctoral research associate at the Innovative Computing Laboratory, UTK during 2017. His research interests Include high performance computing, numerical methods and software, and high performance data analytics.

Mawussi Zounon is a Research Associate in the Numerical Linear Algebra group at the University of Manchester. He received a PhD in computer science and applied mathematics from the University of Bordeaux for his contribution to numerical fault tolerant strategies for large sparse linear algebra solvers with a special focus on Krylov subspace methods. His research interests are in parallel algorithms, numerical algorithms in linear algebra, computer architures, and fault tolerance.

Jack Dongarra holds an appointment at the University of Tennessee, Oak Ridge National Laboratory, and the University of Manchester. He specializes in numerical algorithms in linear algebra, parallel computing, use of advanced-computer architectures, programming meth- odology, and tools for parallel computers. He was awarded the IEEE Sid Fernbach Award in 2004; in 2008 he was the recipient of the first IEEE Medal of Excellence in Scalable Computing; in 2010 he was the first recipient of the SIAM Special Interest Group on Supercomputings award for Career Achievement; in 2011 he was the recipient of the IEEE IPDPS Charles Babbage Award; and in 2013 he received the ACM/ IEEE Ken Kennedy Award. He is a Fellow of the AAAS, ACM, IEEE, and SIAM and a member of the National Academy of Engineering.