Int. J. Appl. Math. Comput. Sci., 2019, Vol. 29, No. 2, 407–419
DOI: 10.2478/amcs-2019-0030

THE PARALLEL TILED WZ FACTORIZATION ALGORITHM FOR MULTICORE ARCHITECTURES

BEATA BYLINA [a], JAROSŁAW BYLINA [a,*]

[a] Institute of Mathematics
Marie Curie-Skłodowska University, Pl. M. Curie-Skłodowskiej 5, 20-031 Lublin, Poland
e-mail: {beata.bylina,jaroslaw.bylina}@umcs.pl

The aim of this paper is to investigate dense linear algebra algorithms on shared memory multicore architectures. The design and implementation of a parallel tiled WZ factorization algorithm which can fully exploit such architectures are presented. Three parallel implementations of the algorithm are studied. The first one relies only on exploiting multithreaded BLAS (basic linear algebra subprograms) operations. The second implementation, except for BLAS operations, employs the OpenMP standard to use the loop-level parallelism. The third implementation, except for BLAS operations, employs the OpenMP `task` directive with the `depend` clause. We report the computational performance and the speedup of the parallel tiled WZ factorization algorithm on shared memory multicore architectures for dense square diagonally dominant matrices. Then we compare our parallel implementations with the respective LU factorization from a vendor implemented LAPACK library. We also analyze the numerical accuracy. Two of our implementations can be achieved with near maximal theoretical speedup implied by Amdahl's law.

Keywords: tiled algorithm, WZ factorization, solution of linear systems, Amdahl's law, high performance computing, multicore architectures.

1. Introduction

Solving linear systems of the form

$$\mathbf{A}\boldsymbol{x} = \boldsymbol{b}, \tag{1}$$

where $\mathbf{A} \in \mathbb{R}^{n\times n}$, $\boldsymbol{b} \in \mathbb{R}^n$ is a common and crucial problem in engineering and scientific computations. The matrix $\mathbf{A}$ is square, non-singular, dense, and of even size (there is no loss of generality, in our case). Direct methods of solving a dense linear system (1) usually consist in a factorization of the matrix $\mathbf{A}$ into simpler matrices—that is, its decomposition into a product of matrices of a simpler structure or of some specific properties—and then solving two (or more) simpler linear systems. The best known factorization is the LU one. It was implemented in the LAPACK (Linear Algebra PACKage) library (Anderson *et al.*, 1999), which is a standard software library for numerical linear algebra. It was designed to effectively exploit the caches on modern cache-based architectures. It uses the standard set of basic linear algebra subprograms (BLAS) (Dongarra *et al.*, 1990), which are important for efficient performance of many applications. Theoretical and practical studies on the numerical stability of the parallel LU factorization have been conducted in a lot of works (Donfack *et al.*, 2015; Dongarra *et al.*, 2013; Buttari *et al.*, 2009). The LU factorization is numerically stable for strictly diagonally dominant matrices.

In this work, we continue to study another form of factorization, that is, the WZ one. It was introduced by Evans and Hatzopoulos (1979) as a new method for solving linear systems in parallel, for SIMD (*single instruction, multiple data* (Flynn, 1972)) computers. It was originally named the quadrant interlocking factorization (QIF) method. The WZ factorization is designed straight for parallel computers. Its advantage is that it simultaneously evaluates two columns or two rows instead of one column or one row as it happens with the LU factorization. García *et al.* (1990) as well as Yalamov and Evans (1995) showed that the WZ factorization can solve linear systems faster than the well-known LU factorization without pivoting for the diagonally dominant matrices on the specific architectures. It can

*Corresponding author

also be useful as preconditioning (Bylina and Bylina, 2007; 2009). In the works of Evans and Hatzopoulos (1979) as well as Yalamov and Evans (1995), various analyses of the WZ factorization have been made and they claim essentially the same numerical stability as that of Gaussian elimination; in particular, the algorithms are stable if the matrix is symmetric positive definite or diagonally dominant.

Our contribution consists in providing a parallel tiled WZ factorization algorithm with its implementations using multithreaded BLAS operations and the OpenMP standard on shared memory multicore architectures. We also compare our implementations with the LU factorization and study the performance, the speedup, and the numerical accuracy.

The rest of this paper is as follows. Section 2 is devoted to related works. Section 3 recalls the WZ factorization, Section 4 presents the tiled WZ factorization algorithm for a 4×4 matrix and describes elementary operations for the algorithm on small matrices. Section 5 describes the details of three parallel implementations for multicore, shared-memory machines. One of them relies on the use of multithreaded BLAS operations, the second and the third one consist of the OpenMP standard and BLAS operations. The second implementation uses `omp parallel for`. The third implementation uses the OpenMP `task` directive with the `depend` clause. Section 6 is devoted to the results of numerical experiments carried out on shared memory multicore architectures and to the comparisons of the LU and WZ factorizations. Section 7 contains conclusions of our research and presents future plans.

2. Related works

The numerical solution of linear systems on the multicore architecture is an important issues for high performance computing. There is a need for parallel numerical algorithm which would achieve high performance and maintain the accuracy of the numerical algorithm on the multicore architecture. The issues concerning parallel numerical algorithms and particularly Gaussian elimination on multicore architectures were presented, among others, by Dumas *et al.* (2016) and Buttari *et al.* (2009).

Dumas *et al.* (2016) present investigations into the parallelization of sub-cubic Gaussian elimination on shared memory multicore architectures. They focus on the parallelization of three sub-routines, namely, matrix multiplication, triangular system solving and the PLUQ factorization. They investigate two runtime implementations of the OpenMP standard for data flow paradigms. In our implementations of the parallel WZ factorization, we also use the OpenMP standard.

In turn, Buttari *et al.* (2009) present a class of parallel tiled linear algebra algorithms for multicore architectures. In tiled algorithms, the operations are performed on square blocks of the data which are suitable for parallel implementations. There is also an implementation of the parallel tiled LU factorization in the PLASMA (parallel linear algebra for scalable multicore architectures) framework (Agullo *et al.*, 2009; Kurzak *et al.*, 2010). The authors of the PLASMA implementation use a DAG-based scheduler for runtime scheduling of the kernel tasks. The remainder of this paper shows how this approach can be applied for the parallel tile WZ factorization. A methodology similar to PLASMA is used in the MAGMA (matrix algebra on GPU and multicore architectures) framework (Agullo *et al.*, 2009). MAGMA is a set of some kernel functions of linear algebra designed for the use with heterogeneous architectures, namely, for multi-GPUs and multicore systems. The hybridization methodology, in which algorithms are divided into tasks of varying granularity, is used in MAGMA. MAGMA and PLASMA often employ functions of Level 3 BLAS for operations on the square matrices. We too employ functions of Level 3 BLAS in our implementations of the parallel WZ factorization.

FLAME (formal linear algebra method) is another example of numerical libraries that has been designed to achieve high performance on multicore architectures (Marqués *et al.*, 2011). The FLAME library approach is based on fundamental computer science. In FLAME, numerical algorithms of linear algebra are expressed in formal notation, similar to the way algorithms are usually presented. This similarity maintains the clarity of the original algorithm and facilitates software development with the use of modern software engineering principles. In our implementations of the parallel WZ factorization, we also try to use these principles, such as hiding the matrices implementation details.

Another numerical library is the Math Kernel Library (MKL), a closed and proprietary library provided by Intel (2019). It is wide and highly optimized, so that it constitutes one of the best choices for practical use. In the work of Dumas *et al.* (2016), we can see in Fig. 10 that the MKL is better than PLASMA for 32 Sandy Bridge cores (although there is also some performance drop for some matrix sizes). However, now, the PLASMA library undergoes a process of porting from the QUARK task scheduler to the OpenMP task scheduler and that can change the PLASMA performance a little, but the stable version is still based on QUARK. On the other hand, in the work of Yarkhan *et al.* (2017), we can see in Fig. 15 that the QUARK-based PLASMA implementation and its OpenMP version achieve almost identical performance—both somewhat worse than the MKL (on 20 cores of the Haswell processor, which is similar to our environment).

Those results made us decide to compare our

implementations with the MKL, not PLASMA (also since our implementations are based on the OpenMP task scheduler and OpenMP-based PLASMA is still under development).

3. WZ factorization

Here we shortly present the usage of the WZ factorization to solve (1). The WZ factorization is described by Rao (1997), Evans and Hatzopoulos (1979) or Yalamov and Evans (1995). We assume that $\mathbf{A}$ is a square and nonsingular matrix of an even size n. We are to find matrices $\mathbf{W}$ and $\mathbf{Z}$ fulfilling $\mathbf{WZ} = \mathbf{A}$. The matrices $\mathbf{W}$ and $\mathbf{Z}$ have the structure shown in Fig. 1, where gray fields are potential non-zeros. The matrix $\mathbf{W}$ is non-singular with $\det \mathbf{W} = 1$. Thus, $\det \mathbf{A} = \det \mathbf{Z}$.

After the factorization, we can solve two linear systems:

$$\begin{aligned} \mathbf{W}\boldsymbol{c} &= \boldsymbol{b}, \\ \mathbf{Z}\boldsymbol{x} &= \boldsymbol{c} \end{aligned} \tag{2}$$

(where $\boldsymbol{c}$ is an auxiliary intermediate vector) instead of one (1). The cost of determining $\boldsymbol{c}$ and $\boldsymbol{x}$ equals $O(n^2)$, similarly to the LU factorization, and is much smaller than the cost of the factorization itself, which equals $O(n^3)$.

That is why, in this paper, we are only interested in obtaining the matrices $\mathbf{Z}$ and $\mathbf{W}$. The first part of the algorithm consists of setting successive parts of columns of the matrix $\mathbf{A}$ to zeros. In the first step, we do that with the elements in the first and n-th columns—from the second row to the $(n-1)$-th row. Next, we update the matrix $\mathbf{A}$. Rao (1997) showed that if the matrix $\mathbf{A}$ is diagonally dominant then it has the WZ factorization which is unique. The WZ factorization algorithm for the matrix $\mathbf{A} \in \mathbb{R}^{n\times n}$ requires

$$C_{WZ}(n) = \frac{2}{3}n^3 - \frac{7}{6}n - 3 \tag{3}$$

floating-point arithmetic operations (Bylina and Bylina, 2015).

4. Tiled WZ factorization

The block WZ factorization is described by Bylina (2018). We present the design of the tiled WZ factorization algorithm, dividing the matrix only into $4 \times 4 = 16$ tiles, and this construction can be easily generalized for r^2 tiles. Let $\mathbf{A}$ be a nonsingular matrix ($\mathbf{A} \in \mathbb{R}^{n\times n}$, n divisible by 4) for which a WZ factorization exists ($\mathbf{A} = \mathbf{WZ}$). The elements of the matrix are divided into 16 square tiles (of the same size) in the following manner:

$$\mathbf{A} = \begin{bmatrix} \mathbf{A}_{11} & \mathbf{A}_{12} & \mathbf{A}_{13} & \mathbf{A}_{14} \\ \mathbf{A}_{21} & \mathbf{A}_{22} & \mathbf{A}_{23} & \mathbf{A}_{24} \\ \mathbf{A}_{31} & \mathbf{A}_{32} & \mathbf{A}_{33} & \mathbf{A}_{34} \\ \mathbf{A}_{41} & \mathbf{A}_{42} & \mathbf{A}_{43} & \mathbf{A}_{44} \end{bmatrix}. \tag{4}$$

The product of the matrices $\mathbf{W}$ and $\mathbf{Z}$ can be written as follows:

$$\mathbf{WZ} = \begin{bmatrix} \mathbf{W}_{11} & \mathbf{0} & \mathbf{0} & \mathbf{W}_{14} \\ \mathbf{W}_{21} & \mathbf{W}_{22} & \mathbf{W}_{23} & \mathbf{W}_{24} \\ \mathbf{W}_{31} & \mathbf{W}_{32} & \mathbf{W}_{33} & \mathbf{W}_{34} \\ \mathbf{W}_{41} & \mathbf{0} & \mathbf{0} & \mathbf{W}_{44} \end{bmatrix} \cdot \begin{bmatrix} \mathbf{Z}_{11} & \mathbf{Z}_{12} & \mathbf{Z}_{13} & \mathbf{Z}_{14} \\ \mathbf{0} & \mathbf{Z}_{22} & \mathbf{Z}_{23} & \mathbf{0} \\ \mathbf{0} & \mathbf{Z}_{32} & \mathbf{Z}_{33} & \mathbf{0} \\ \mathbf{Z}_{41} & \mathbf{Z}_{42} & \mathbf{Z}_{43} & \mathbf{Z}_{44} \end{bmatrix}, \tag{5}$$

where $\mathbf{A}_{ij}$, $\mathbf{W}_{ij}$ and $\mathbf{Z}_{ij}$ are $s \times s$ matrices ($s = n/4$) for $i, j = 1, \ldots, 4$. The matrices $\mathbf{W}$ and $\mathbf{Z}$ are nonsingular, because $\det \mathbf{A} \neq 0$, $\det \mathbf{W} = 1$, which implies $\det \mathbf{Z} \neq 0$.

To compute the matrices $\mathbf{W}$ and $\mathbf{Z}$, we have to proceed in the following four stages.

4.1. Stage 1. We build the matrix

$$\mathbf{B} = \begin{bmatrix} \mathbf{A}_{11} & \mathbf{A}_{14} \\ \mathbf{A}_{41} & \mathbf{A}_{44} \end{bmatrix}. \tag{6}$$

For this $2s \times 2s$ matrix we perform the original WZ factorization and obtain matrices

$$\mathbf{W_B} = \begin{bmatrix} \mathbf{W}_{11} & \mathbf{W}_{14} \\ \mathbf{W}_{41} & \mathbf{W}_{44} \end{bmatrix}, \tag{7}$$

$$\mathbf{Z_B} = \begin{bmatrix} \mathbf{Z}_{11} & \mathbf{Z}_{14} \\ \mathbf{Z}_{41} & \mathbf{Z}_{44} \end{bmatrix}. \tag{8}$$

The matrix $\mathbf{B}$ is invertible. Therefore, we can compute the matrices $\mathbf{W_B}$ and $\mathbf{Z_B}$ from the factorization theorem by Rao (1997).

4.2. Stage 2. We compute the first block column of the matrix $\mathbf{W}$ (that is, the blocks $\mathbf{W}_{21}$ and $\mathbf{W}_{31}$) as well as the fourth block column of the matrix $\mathbf{W}$ (that is, the blocks $\mathbf{W}_{24}$ and $\mathbf{W}_{34}$) from linear systems (which can also be computed in parallel, because these linear systems are independent):

$$\begin{cases} \mathbf{W}_{21}\mathbf{Z}_{11} + \mathbf{W}_{24}\mathbf{Z}_{41} = \mathbf{A}_{21}, \\ \mathbf{W}_{21}\mathbf{Z}_{14} + \mathbf{W}_{24}\mathbf{Z}_{44} = \mathbf{A}_{24} \end{cases} \tag{9}$$

and

$$\begin{cases} \mathbf{W}_{31}\mathbf{Z}_{11} + \mathbf{W}_{34}\mathbf{Z}_{41} = \mathbf{A}_{31}, \\ \mathbf{W}_{31}\mathbf{Z}_{14} + \mathbf{W}_{34}\mathbf{Z}_{44} = \mathbf{A}_{34}. \end{cases} \tag{10}$$

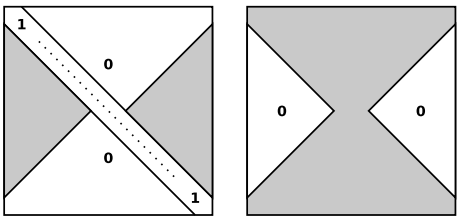


Fig. 1. Forms of the $\mathbf{W}$ (left side) and $\mathbf{Z}$ (right side) matrices.

4.3. Stage 3. We compute the first block row of the matrix $\mathbf{Z}$ (that is, the blocks $\mathbf{Z}_{12}$ and $\mathbf{Z}_{13}$) as well as the fourth block row of the matrix $\mathbf{Z}$ (that is, the tiles $\mathbf{Z}_{42}$ and $\mathbf{Z}_{43}$)—from linear systems (which also can be computed in parallel):

$$\begin{cases} \mathbf{W}_{11}\mathbf{Z}_{12} + \mathbf{W}_{14}\mathbf{Z}_{42} = \mathbf{A}_{12}, \\ \mathbf{W}_{41}\mathbf{Z}_{12} + \mathbf{W}_{44}\mathbf{Z}_{42} = \mathbf{A}_{42} \end{cases} \tag{11}$$

and

$$\begin{cases} \mathbf{W}_{11}\mathbf{Z}_{13} + \mathbf{W}_{14}\mathbf{Z}_{43} = \mathbf{A}_{13}, \\ \mathbf{W}_{41}\mathbf{Z}_{13} + \mathbf{W}_{44}\mathbf{Z}_{43} = \mathbf{A}_{43}. \end{cases} \tag{12}$$

4.4. Stage 4. We compute the blocks $\mathbf{W}_{22}$, $\mathbf{W}_{23}$, $\mathbf{W}_{32}$, $\mathbf{W}_{33}$ and $\mathbf{Z}_{22}$, $\mathbf{Z}_{23}$, $\mathbf{Z}_{32}$, $\mathbf{Z}_{33}$ from

$$\begin{cases} \mathbf{W}_{21}\mathbf{Z}_{12} + \mathbf{W}_{22}\mathbf{Z}_{22} + \mathbf{W}_{23}\mathbf{Z}_{32} + \mathbf{W}_{24}\mathbf{Z}_{42} = \mathbf{A}_{22}, \\ \mathbf{W}_{21}\mathbf{Z}_{13} + \mathbf{W}_{22}\mathbf{Z}_{23} + \mathbf{W}_{23}\mathbf{Z}_{33} + \mathbf{W}_{24}\mathbf{Z}_{43} = \mathbf{A}_{23}, \\ \mathbf{W}_{31}\mathbf{Z}_{12} + \mathbf{W}_{32}\mathbf{Z}_{22} + \mathbf{W}_{33}\mathbf{Z}_{32} + \mathbf{W}_{34}\mathbf{Z}_{42} = \mathbf{A}_{32}, \\ \mathbf{W}_{31}\mathbf{Z}_{13} + \mathbf{W}_{32}\mathbf{Z}_{23} + \mathbf{W}_{33}\mathbf{Z}_{33} + \mathbf{W}_{34}\mathbf{Z}_{43} = \mathbf{A}_{33}, \end{cases} \tag{13}$$

Thus

$$\begin{cases} \mathbf{W}_{22}\mathbf{Z}_{22} + \mathbf{W}_{23}\mathbf{Z}_{32} = \mathbf{A}_{22} - \mathbf{W}_{21}\mathbf{Z}_{12} - \mathbf{W}_{24}\mathbf{Z}_{42}, \\ \mathbf{W}_{22}\mathbf{Z}_{23} + \mathbf{W}_{23}\mathbf{Z}_{33} = \mathbf{A}_{23} - \mathbf{W}_{21}\mathbf{Z}_{13} - \mathbf{W}_{24}\mathbf{Z}_{43}, \\ \mathbf{W}_{32}\mathbf{Z}_{22} + \mathbf{W}_{33}\mathbf{Z}_{32} = \mathbf{A}_{32} - \mathbf{W}_{31}\mathbf{Z}_{12} - \mathbf{W}_{34}\mathbf{Z}_{42}, \\ \mathbf{W}_{32}\mathbf{Z}_{23} + \mathbf{W}_{33}\mathbf{Z}_{33} = \mathbf{A}_{33} - \mathbf{W}_{31}\mathbf{Z}_{13} - \mathbf{W}_{34}\mathbf{Z}_{43}. \end{cases} \tag{14}$$

On the left-hand side of (14), we have a matrix which is to be factorized into the WZ factors and thus we repeat the procedure from the beginning, but for a new matrix of a smaller size:

$$\begin{cases} \mathbf{A}_{22}^{\text{new}} = \mathbf{A}_{22} - \mathbf{W}_{21}\mathbf{Z}_{12} - \mathbf{W}_{24}\mathbf{Z}_{42}, \\ \mathbf{A}_{23}^{\text{new}} = \mathbf{A}_{23} - \mathbf{W}_{21}\mathbf{Z}_{13} - \mathbf{W}_{24}\mathbf{Z}_{43}, \\ \mathbf{A}_{32}^{\text{new}} = \mathbf{A}_{32} - \mathbf{W}_{31}\mathbf{Z}_{12} - \mathbf{W}_{34}\mathbf{Z}_{42}, \\ \mathbf{A}_{33}^{\text{new}} = \mathbf{A}_{33} - \mathbf{W}_{31}\mathbf{Z}_{13} - \mathbf{W}_{34}\mathbf{Z}_{43}. \end{cases} \tag{15}$$

The new $(n-2s) \times (n-2s)$ matrix $\mathbf{A}^{\times new}$ can be written as

$$\mathbf{A}^{\text{new}} = \begin{bmatrix} \mathbf{A}_{22}^{\text{new}} & \mathbf{A}_{23}^{\text{new}} \\ \mathbf{A}_{32}^{\text{new}} & \mathbf{A}_{33}^{\text{new}} \end{bmatrix}. \tag{16}$$

Thus we perform only Stage 1, but for the matrix $\mathbf{A}^{\text{new}}$. Figure 2 shows the tiles labeled with the numbers of the stage numbers in which the tiles are computed. For even $r > 4$, these stages are repeated $r/2$ times, for a smaller and smaller matrix in every step. Algorithm 1 presents the tiled WZ factorization as a list of steps for a nonsingular matrix $\mathbf{A}$ divided into r^2 tiles, where r is an even number. There is no loss of generality—we can easily extend the matrix of an odd size by one column and one row (filled with ones with a one on the diagonal), then we perform a WZ factorization, and next, we cut off the last column and the last row. As a result, we get matrices $\mathbf{W}$ and $\mathbf{Z}$. $\mathbf{A}_{ij}$, $\mathbf{W}_{ij}$, $\mathbf{Z}_{ij}$ are $s \times s$ matrices (for $i, j = 1, \ldots, r$ and $n = r \times s$).

4.5. Elementary operations for the tiled WZ factorization. The tiled WZ factorization algorithm performs most of its floating-point arithmetic operations using Level 3 BLAS operations. Thus, the implementations of the algorithm will be based on the following set of elementary operations:

`WZ(B, W, Z)`. This subroutine performs a sequential WZ factorization for matrix $\mathbf{B}$.

`DTRSM(u/nonu, up/lo, l/r, A, X, B)`. This BLAS subroutine is used to compute $\mathbf{X} = \mathbf{A}^{-1}\mathbf{B}$ (denoted by `l`) or $\mathbf{X} = \mathbf{B}\mathbf{A}^{-1}$ (denoted by `r`), where $\mathbf{X}$ and $\mathbf{B}$ are $s \times s$ matrices, $\mathbf{A}$ is a unit (`u`) or non-unit (`nonu`), upper (`up`) or lower (`lo`) triangular matrix.

`DGEMM(A, B, C)`. This BLAS subroutine is used to compute $\mathbf{A} = -\mathbf{B}\mathbf{C} + \mathbf{A}$, where $\mathbf{A}$, $\mathbf{B}$, and $\mathbf{C}$ are $s \times s$ matrices.

`DGEMM_copy(A, B, C, D)`. This BLAS subroutine is used to compute $\mathbf{A} = -\mathbf{B}\mathbf{C} + \mathbf{D}$, where $\mathbf{A}$, $\mathbf{B}$, $\mathbf{C}$, and $\mathbf{D}$ are $s \times s$ matrices.

Note that no extra storage is required for the matrices $\mathbf{W}_{ij}$ and $\mathbf{Z}_{ij}$ since they can overwrite the corresponding tiles $\mathbf{A}_{ij}$ of the original matrix $\mathbf{A}$.

Algorithm 2 presents the tiled WZ factorization algorithm, which is expressed in terms of elementary operations `WZ`, `DTRSM`, `DGEMM`, and `DGEMM_copy` for $\mathbf{A} \in \mathbb{R}^{n \times n}$.

4.6. Graph-driven asynchronous execution. Generally, algorithms (including linear algebra algorithms) can be pictured as directed acyclic graphs (DAGs). A directed acyclic graph is a finite directed graph without cycles. It contains a finite number of vertices and edges. Each edge is directed from one vertex to another.

DAGs are useful in different models of parallel programming, namely, in the task-based programming model, where nodes are computational tasks performed in kernel subroutines and where edges represent the dependencies among them. The scheduler may execute tasks in any order that respects the dependencies shown in the DAG. This approach is presented by Buttari *et al.*

1	3	3	1
2	4	4	2
2	4	4	2
1	3	3	1

Fig. 2. Order of computing tiles in TWZ for 4×4 tiles.

Algorithm 1. Tiled WZ factorization for even r.

Require: $\mathbf{A}$, r
Ensure: $\mathbf{W}$, $\mathbf{Z}$

for $k \leftarrow 1, r/2-1$ **do**
  $k_2 \leftarrow r-k+1$
  The WZ factorization for the corner blocks of $\mathbf{A}$*: STAGE 1*
  $\mathbf{B} \leftarrow \begin{bmatrix} \mathbf{A}_{kk} & \mathbf{A}_{kk_2} \\ \mathbf{A}_{k_2k} & \mathbf{A}_{k_2k_2} \end{bmatrix}$
  $[\mathbf{W_B}, \mathbf{Z_B}] \leftarrow wz(\mathbf{B})$
  $\begin{bmatrix} \mathbf{W}_{kk} & \mathbf{W}_{kk_2} \\ \mathbf{W}_{k_2k} & \mathbf{W}_{k_2k_2} \end{bmatrix} \leftarrow \mathbf{W_B}$,
  $\begin{bmatrix} \mathbf{Z}_{kk} & \mathbf{Z}_{kk_2} \\ \mathbf{Z}_{k_2k} & \mathbf{Z}_{k_2k_2} \end{bmatrix} \leftarrow \mathbf{Z_B}$
  Computing the k*-th and* k_2*-nd columns of* $\mathbf{W}$*: STAGE 2*
  $\mathbf{Z}_{kk}\mathbf{D} \leftarrow \mathbf{Z}_{kk_2}$
  Computing a lower triangular $\mathbf{E}$
  $\mathbf{E} \leftarrow -\mathbf{Z}_{k_2k}\mathbf{D} + \mathbf{Z}_{k_2k_2}$
  for $i \leftarrow k+1, k_2-1$ **do**
    $\mathbf{W}_{ik_2}\mathbf{E} = -\mathbf{A}_{ik}\mathbf{D} + \mathbf{A}_{ik_2}$
    $\mathbf{W}_{ik}\mathbf{Z}_{kk} = -\mathbf{W}_{ik_2}\mathbf{Z}_{k_2k} + \mathbf{A}_{ik}$
  end for
  Computing the k*-th and* k_2*nd rows of* $\mathbf{Z}$*: STAGE 3*
  $\mathbf{D}\mathbf{W}_{kk} \leftarrow \mathbf{W}_{k_2k}$
  Computing an upper triangular $\mathbf{E}$ *(with 1s on its diagonal)*
  $\mathbf{E} \leftarrow -\mathbf{D}\mathbf{W}_{kk_2} + \mathbf{W}_{k_2k_2}$
  for $i \leftarrow k+1, k_2-1$ **do**
    $\mathbf{E}\mathbf{Z}_{k_2i} = -\mathbf{D}\mathbf{A}_{ki} + \mathbf{A}_{k_2i}$
    $\mathbf{W}_{kk}\mathbf{Z}_{ki} = -\mathbf{W}_{kk_2}\mathbf{Z}_{k_2i} + \mathbf{A}_{ki}$
  end for
  The update of the matrix $\mathbf{A}$*: STAGE 4*
  for $j \leftarrow k+1, k_2-1$ **do**
    for $i \leftarrow k+1, k_2-1$ **do**
      $\mathbf{A}_{ij} = -\mathbf{W}_{ik}\mathbf{Z}_{kj} - \mathbf{W}_{ik_2}\mathbf{Z}_{k_2j} + \mathbf{A}_{ij}$
    end for
  end for
end for
$\mathbf{B} \leftarrow \begin{bmatrix} \mathbf{A}_{\frac{r}{2},\frac{r}{2}} & \mathbf{A}_{\frac{r}{2},\frac{r}{2}+1} \\ \mathbf{A}_{\frac{r}{2}+1,\frac{r}{2}} & \mathbf{A}_{\frac{r}{2}+1,\frac{r}{2}+1} \end{bmatrix}$
$[\mathbf{W_B}, \mathbf{Z_B}] \leftarrow wz(\mathbf{B})$
$\begin{bmatrix} \mathbf{W}_{\frac{r}{2},\frac{r}{2}} & \mathbf{W}_{\frac{r}{2},\frac{r}{2}+1} \\ \mathbf{W}_{\frac{r}{2}+1,\frac{r}{2}} & \mathbf{W}_{\frac{r}{2}+1,\frac{r}{2}+1} \end{bmatrix} \leftarrow \mathbf{W_B}$,
$\begin{bmatrix} \mathbf{Z}_{\frac{r}{2},\frac{r}{2}} & \mathbf{Z}_{\frac{r}{2},\frac{r}{2}+1} \\ \mathbf{Z}_{\frac{r}{2}+1,\frac{r}{2}} & \mathbf{Z}_{\frac{r}{2}+1,\frac{r}{2}+1} \end{bmatrix} \leftarrow \mathbf{Z_B}$

(2009) for tiled linear algebra algorithms. A critical path can be identified in the DAG as the one that connects all the nodes which must be carried out sequentially, allowing all possible parallelism. Algorithm 2 can be represented as a DAG.

DAGs are independent of hardware architectures and can be implemented with the use of different technologies.

Algorithm 2. Tiled WZ factorization algorithm for even r based on four elementary operations.

Require: $\mathbf{A}$, r
Ensure: $\mathbf{W}$, $\mathbf{Z}$

for $k \leftarrow 1, r/2-1$ **do**
  $k_2 \leftarrow r-k+1$
  The WZ factorization for the corner blocks of $\mathbf{A}$*: STAGE 1*
  $\mathbf{B} \leftarrow \begin{bmatrix} \mathbf{A}_{kk} & \mathbf{A}_{kk_2} \\ \mathbf{A}_{k_2k} & \mathbf{A}_{k_2k_2} \end{bmatrix}$
  WZ($\mathbf{B}$, $\mathbf{W_B}$, $\mathbf{Z_B}$)
  $\begin{bmatrix} \mathbf{W}_{kk} & \mathbf{W}_{kk_2} \\ \mathbf{W}_{k_2k} & \mathbf{W}_{k_2k_2} \end{bmatrix} \leftarrow \mathbf{W_B}$,
  $\begin{bmatrix} \mathbf{Z}_{kk} & \mathbf{Z}_{kk_2} \\ \mathbf{Z}_{k_2k} & \mathbf{Z}_{k_2k_2} \end{bmatrix} \leftarrow \mathbf{Z_B}$
  Computing the k*-th and* k_2*-nd columns of* $\mathbf{W}$*: STAGE 2*
  DTRSM(nonu, up, l, $\mathbf{Z}_{kk}$, $\mathbf{D}$, $\mathbf{Z}_{kk_2}$)
  Computing a lower triangular $\mathbf{E}$
  DGEMM_copy($\mathbf{E}$, $\mathbf{Z}_{k_2k}$, $\mathbf{D}$, $\mathbf{Z}_{k_2k_2}$)
  for $i \leftarrow k+1, k_2-1$ **do**
    DGEMM($\mathbf{A}_{ik_2}$, $\mathbf{A}_{ik}$, $\mathbf{D}$);
    DTRSM(nonu, lo, r, $\mathbf{E}$, $\mathbf{W}_{ik_2}$, $\mathbf{A}_{ik_2}$)
    DGEMM($\mathbf{A}_{ik}$, $\mathbf{W}_{ik_2}$, $\mathbf{Z}_{k_2k}$);
    DTRSM(nonu, up, r, $\mathbf{Z}_{kk}$, $\mathbf{W}_{ik}$, $\mathbf{A}_{ik}$)
  end for
  Computing the k*-th and* k_2*-nd rows of* $\mathbf{Z}$*: STAGE 3*
  DTRSM(u, lo, r, $\mathbf{W}_{kk}$, $\mathbf{D}$, $\mathbf{W}_{k_2k}$)
  Computing an upper triangular $\mathbf{E}$ *(with 1s on its diagonal)*
  DGEMM_copy($\mathbf{E}$, $\mathbf{D}$, $\mathbf{W}_{kk_2}$, $\mathbf{W}_{k_2k_2}$)
  for $i \leftarrow k+1, k_2-1$ **do**
    DGEMM($\mathbf{A}_{k_2i}$, $\mathbf{D}$, $\mathbf{A}_{ki}$);
    DTRSM(u, up, l, $\mathbf{E}$, $\mathbf{Z}_{k_2i}$, $\mathbf{A}_{k_2i}$)
    DGEMM($\mathbf{A}_{ki}$, $\mathbf{W}_{kk_2}$, $\mathbf{Z}_{k_2i}$);
    DTRSM(u, lo, l, $\mathbf{W}_{kk}$, $\mathbf{Z}_{ki}$, $\mathbf{A}_{ki}$)
  end for
  The update of the matrix $\mathbf{A}$*: STAGE 4*
  for $j \leftarrow k+1, k_2-1$ **do**
    for $i \leftarrow k+1, k_2-1$ **do**
      DGEMM($\mathbf{A}_{ij}$, $\mathbf{W}_{ik}$, $\mathbf{Z}_{kj}$);
      DGEMM($\mathbf{A}_{ij}$, $\mathbf{W}_{ik_2}$, $\mathbf{Z}_{k_2j}$)
    end for
  end for
end for
$\mathbf{B} \leftarrow \begin{bmatrix} \mathbf{A}_{\frac{r}{2},\frac{r}{2}} & \mathbf{A}_{\frac{r}{2},\frac{r}{2}+1} \\ \mathbf{A}_{\frac{r}{2}+1,\frac{r}{2}} & \mathbf{A}_{\frac{r}{2}+1,\frac{r}{2}+1} \end{bmatrix}$
WZ($\mathbf{B}$, $\mathbf{W_B}$, $\mathbf{Z_B}$)
$\begin{bmatrix} \mathbf{W}_{\frac{r}{2},\frac{r}{2}} & \mathbf{W}_{\frac{r}{2},\frac{r}{2}+1} \\ \mathbf{W}_{\frac{r}{2}+1,\frac{r}{2}} & \mathbf{W}_{\frac{r}{2}+1,\frac{r}{2}+1} \end{bmatrix} \leftarrow \mathbf{W_B}$,
$\begin{bmatrix} \mathbf{Z}_{\frac{r}{2},\frac{r}{2}} & \mathbf{Z}_{\frac{r}{2},\frac{r}{2}+1} \\ \mathbf{Z}_{\frac{r}{2}+1,\frac{r}{2}} & \mathbf{Z}_{\frac{r}{2}+1,\frac{r}{2}+1} \end{bmatrix} \leftarrow \mathbf{Z_B}$

Figure 3 shows the DAG for the tiled WZ factorization when Algorithm 2 is executed for a 4×4 tiled matrix (see Fig. 2). The bold lines denote one of critical paths.

5. Parallel implementations

5.1. Data structure. The MKL library implements routines from LAPACK and uses the standard interface for this package. In LAPACK, the matrices are stored as one-dimensional arrays. Two methods of storing matrices in memory are considered, namely, the column major order and the row major order. Because we want to compare our implementations of the tiled WZ factorization without pivoting with the implementation of the LU factorization from LAPACK, we chose the column major order (denoted by `LAPACK_COL_MAJOR`) for our two implementations. Additionally, most users store their matrices in the standard column major layout, common to Fortran and LAPACK.

However, the block technique can improve the performance if the data are organized in square blocks (tiles), the fact first featured in the work of Gustavson (1997), where the layout is referred to as a square block format. In our work, we refer to it as tiled layout, similarly as Buttari *et al*. (2009). In the tile layout, the matrices are represented as small square tiles of data contiguous in memory so that each core operates on individual tile independently. Any tile can be cached and fully processed, which helps to minimize the number of cache misses. We use the task-based model available in the OpenMP standard, i.e., we employ the tile layout.

5.2. Implementation using only multithreaded BLAS routines (TWZ(r)). One of the ways to parallelize the code is to use a multithreaded optimized library such as the MKL . The application of standardized functions makes the code more portable. In this implementation (which is denoted by TWZ(r)) of Algorithm 2, we use multithreaded functions from the MKL, namely, Level 3 BLAS routines, which perform matrix-matrix operations. This methodology implies a fork-join model. We use two routines: `cblas_dgemm`, which computes a scalar-matrix-matrix product, and adds the result to a scalar-matrix product and `cblas_dtrsm`, which solves a triangular matrix equation. These multithreaded Level 3 BLAS routines use pThreads and the OpenMP standard for parallelization.

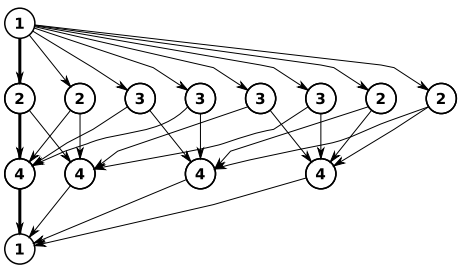


Fig. 3. DAG for the parallel tiled WZ factorization of a 4×4 tiled matrix. The bold lines denote a critical path.

In this implementation, each tile is supported by a number of threads set in the `OMP_NUM_THREADS` environment variable. In each tile, we can see the fork-join model. The end of each executed BLAS routine becomes an implicit synchronization point. If we use only multithreaded Level 3 BLAS functions from the MKL, we come across the problem of excess synchronization as in LAPACK. This implementation uses the LAPACK layout as the data structure for matrices.

5.3. Implementation using BLAS routines and the OpenMP standard (TWZ(r)-fork-join). To better show the profit of tiling, we modify this implementation. We additionally employ the OpenMP standard with the fork-join model using loop-level parallelism and the BLAS routines for matrices' operations. We call this implementation TWZ(r)-fork-join.

The inner loops in Algorithm 2 (starting in lines 10 and 17) have no dependencies. The result of one iteration does not depend on the result of any other. This means that two different iterations could be executed simultaneously by two different processors. We parallelize these loops using the directive `#pragma omp parallel for`. OpenMP is responsible for distributing the iteration between threads. Each thread works on a different part of the matrix, and this does not lead to a race condition. For our implementation of the tiled WZ factorization, we choose the `dynamic` loop scheduling. Loop iterations are divided into chunks of size 1. When a thread finishes, it is dynamically assigned to another piece. The dynamic scheduling allows achieving a balanced load on each processor and keeps location data so as to minimize the overhead associated with the collection of the data. OpenMP allows using nested parallelism. In lines 22 and 23 we have nested loops and we collapse inner loops using the OpenMP clause `collapse(2)`.

Each tile is supported by exactly one thread, but one thread can perform the calculation for more than one tile. This solution improves the location of data between threads. The disadvantage of this implementation is a barrier after each loop and the fact that it takes a lot of the time to create, run and manage threads.

Despite the choice of using the OpenMP standard, the presented approach may also be implemented using other technologies, for example, the pThreads standard. This implementation uses the LAPACK layout as the data structure for matrices.

5.4. Implementation using BLAS routines and a task (TWZ(r)-task). To better show the profit of tiling, we modify our implementation further. We employ the OpenMP task directive and the BLAS routines for matrix operations. We call this implementation TWZ(r)-task. The use of tasks with a depend clause instead of the fork-join model from a OpenMP standard (since standard 4.0) should delete a big number of unnecessary synchronizations and make the implementation more asynchronous. Moreover, the use of tasks will cause higher concurrency and scalability. Such an implementation will be more suitable for multicore architectures.

Attempts have been made to rewrite the implementation which simply consists of `omp parallel for` into the implementation including the tasks. However, tasks are not independent, and the DAG of the algorithm describes their dependencies. These have to be defined in an OpenMP application with the use of the `depend` clause. Namely, the `#pragma omp task depend` clause is used to announce the compiler so that the next code block is executed as a new task. This task is bound to a thread from the current team of threads. The execution of the new task can be instant or delayed according to the task schedule and availability of threads. The `depend` clause enforces additional limitations concerning task scheduling. The OpenMP runtime provides a dynamic scheduler of the tasks while avoiding data hazards by keeping track of dependencies. The dynamic scheduler means that the tasks are queueing and executed as quickly as possible. This implementation uses the tile layout as the data structure for matrices.

5.5. Theoretical speedup. We compute a maximal theoretical speedup of the algorithm from Amdahl's law. First, we have to compute the cost of Algorithm 1.

Stage 1 is a WZ factorization of a square matrix of $2s$ rows. The computational cost of Stage 1 is (cf. (3))

$$C_1(s) = C_{WZ}(2s) = \frac{16}{3}s^3 - \frac{7}{3}s - 3. \tag{17}$$

Stage 2 is finding some blocks of the matrix $\mathbf{W}$. Here we have a couple of operations: a matrix-matrix multiplication and finding an inverse of the matrix (by solving triangular systems). The cost of Stage 2 in the k-th step is ($n = rs$, here and below)

$$\begin{aligned} &C_2(k,r,s) \\ &\quad = 3s^3 + s^2 + \sum_{i=k+1}^{r-k} (6s^3 + 2s^2) \\ &\quad = 3s^3 + s^2 + (6s^3 + 2s^2)(r-2k). \end{aligned} \tag{18}$$

The number of operations for Stage 3 (in the k-th step) is the same as for Stage 2.

Stage 4 comprises two matrix-matrix multiplications and two matrix-matrix additions, for $(r-2k)^2$ blocks. The cost of Stage 4 in the k-th step is

$$\begin{aligned} &C_4(k,r,s) \\ &\quad = \sum_{i=k+1}^{r-k} \sum_{j=k+1}^{r-k} (4s^3 + 2s^2) \\ &\quad = (4s^3 + 2s^2)(r-2k)^2. \end{aligned} \tag{19}$$

The number of floating-point arithmetic operations for the tiled WZ factorization Algorithm 1 is

$$\begin{aligned} &C(n,s) \\ &\quad = \sum_{k=1}^{\frac{r}{2}} (C_1(s) + 2C_2(k,r,s) + C_4(k,r,s)) \\ &\quad = \frac{n^3(4s+2) + 6n^2s^2 + n(6s^3 - 2s^2 - 7s - 9)}{6s}. \end{aligned} \tag{20}$$

The complexity of the tiled WZ factorization is greater than that of the non-tiled WZ factorization. It depends on both the matrix size (n) and the tile size (s). The smallest complexity of the WZ factorization is for $s = n/2$, and it is exactly the same as for the non-tiled WZ factorization (3). The greatest complexity is $n^3 + O(n^2)$ for $s = 1$, which means that it is about $\frac{1}{3}n^3$ (50%) bigger than the complexity of the sequential WZ factorization (3). In this case, the extra cost comes from Stage 4, where we perform operations on 1×1 matrices (in line 24 of the algorithms) in the tiled algorithm, which increases the number of operations from 1 to 2. In other cases, where $1 < s < n/2$, we can estimate $\frac{1}{3}n^3 < C(n,s) < n^3$ and note $C(n,s) = O(n^3)$. We have to carefully choose the correct values for s because too high or too small values may deteriorate the performance of the level 3 BLAS operations used in Algorithm 2.

To foresee the maximum theoretical speedup of the parallel tiled WZ factorization algorithm with the use of p threads, we apply Amdahl's law. Let P_S denote the percentage of the sequential part of the whole algorithm—that is, the part which is not parallelized. In Algorithm 1, only Stage 1 is not parallel. The number of floating-point arithmetic operations of Stage 1 is given by the formula (17). Stage 1 is executed $r/2$ times, and the cost of the whole Algorithm 1 is given by (20). Thus

$$\begin{aligned} &P_S(n,s) \\ &= \frac{\frac{r}{2} \times C_1(s)}{C(n,s)} \\ &= \frac{16s^3 - 7s - 9}{n^2(4s+2) + 6ns^2 + 6s^3 - 2s^2 - 7s - 9}. \end{aligned} \tag{21}$$

Let $P_R(n,s)$ denote the percentage of the parallelized part of the algorithm. Obviously, $P_R(n,s) = 1 - P_S(n,s)$.

In consequence, according to Amdahl's law (Amdahl, 1967), the best theoretical speedup for the parallel tiled WZ factorization algorithm (Algorithm 1) with the use of p threads is

$$S(p;n,s) = \frac{1}{P_S(n,s) + \frac{P_R(n,s)}{p}}, \tag{22}$$

where $n \times n$ is the matrix size.

6. Numerical experiments

In this section, we test the performance and the speedup of the parallel tiled WZ factorization algorithm presented in Section 4. Our intention was to investigate the properties of the implementations of the parallel tiled WZ factorization described in Section 5 and to check the parallel behavior of the WZ factorization against the LU one.

In this paper, we tested the multithreaded `LAPACKE_mkl_dgetrfnpi` and `LAPACK_dgetrf` routines from the MKL. The former computes the complete LU factorization of a general matrix without pivoting, and the latter computes the complete LU factorization of a general matrix with partial pivoting. In our case, the matrices are square of the size $n \times n$. In the implementation of the `LAPACKE_mkl_dgetrfnpi` routine, the panel factorization (factorization of a block of columns of LAPACK) is used along with the Level 3 BLAS routines `DTRSM` and `DGEMM` and the OpenMP fork-join model.

We compared five applications for shared memory multicore architectures. These applications are as follows:

- the parallel tiled WZ factorization algorithm with use of the multithreaded Level 3 BLAS (denoted by TWZ(r));
- the parallel tiled WZ factorization algorithm with the BLAS operations and the fork-join model with the OpenMP standard with dynamic scheduling (denoted by 'TWZ(r)-fork-join');
- the parallel tiled WZ factorization algorithm with BLAS operations and the task with the OpenMP standard with dynamic scheduling (denoted by 'TWZ(r)-task');
- the `dgetrfnpi` routine from the MKL which is a multithreaded implementation of the LU factorization without pivoting (in the sequel, it is denoted by 'LU no-pivoting');
- the `dgetrf` routine from the MKL, which is a multithreaded implementation of the LU factorization with partial pivoting (in what follows, it is denoted by 'LU part. piv.').

Table 1. Hardware used in the experiments.

CPU:	Intel® Xeon E5-2670 v.3 (Haswell)
# sockets × # cores × # threads:	2 × 12 × 2
Clock speed:	2.30 GHz
Level 1 instruction cache:	32 kB per core
Level 1 data cache:	32 kB per core
Level 2 cache:	256 kB per core
Level 3 cache:	30 MB
Host memory:	128 GB
Compiler:	Intel icc 16.0.0
BLAS library, LAPACK:	MKL 2016.0.109

Table 1 shows the hardware specification used in the numerical experiment. We also conducted some tests with a newer ICC compiler and MKL (both in version 19), and the results were practically the same. All floating point calculations were performed in double precision. The input matrices were generated (by the authors). They were dense, random matrices, with a dominant diagonal of an even size of 128, 256, 512, $1024m$ (for $m \in \{1, \ldots, 14\}$). Various numbers of tiles were tested; namely, each matrix was divided into $r = 16, 32, 64, 128$ tiles for each side (both for the rows and the columns). The performance times were measured with the use of a standard function, namely, `dsecnd()`. We set the number of OpenMP threads using the `OMP_NUM_THREADS` environmental variable.

6.1. Performance. In our experiments, as a metric, we use the number of floating-point operations per second (flops). The number of floating point operations for both the LU factorization and the WZ factorization 3 of the matrix of the size $n \times n$ is $\frac{2}{3}n^3 + O(n^2)$, so it approximately equals $\frac{2}{3}n^3$.

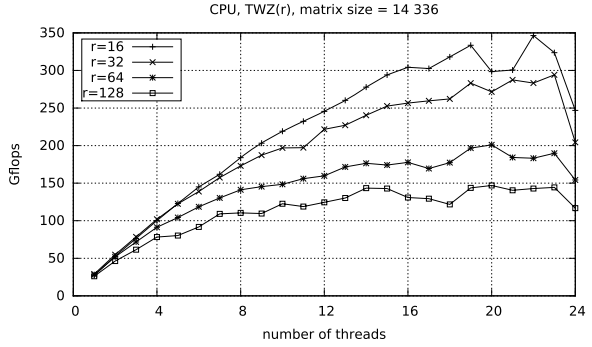
Thus, to obtain the metric in Gflops ($= 10^9$ flops), we use the following formula:

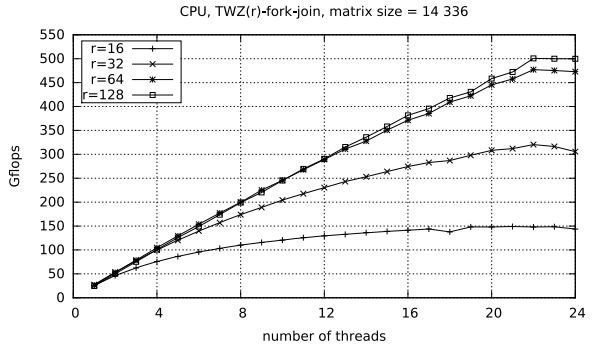
$$\frac{2n^3}{3 \times T \times 10^9}, \tag{23}$$

where T is the execution time of a measured implementation. This metric allows comparing all implementations with the same measure.

Figure 4 presents the performance (in Gflops) of TWZ(r), TWZ(r)-fork-join and TWZ(r)-task for a matrix of the size 14 336 for four different numbers of tiles ($r = 16, 32, 64, 128$) as a function of the number of threads. Figure 5 presents the performance (in Gflops) of TWZ(r), TWZ(r)-fork-join and TWZ(r)-task for 24 threads for four different numbers of tiles ($r = 16, 32, 64, 128$) as a function of the matrix size.

Figures 4 and 5 provide roughly the same information. For TWZ(r) implementation we achieve the best performance for $r = 16$, regardless of the number of threads and the matrix size. This is caused by the fact





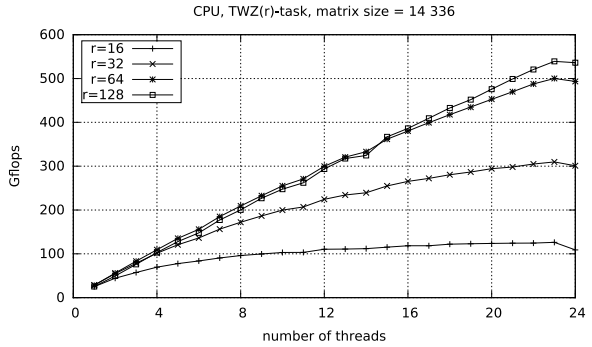
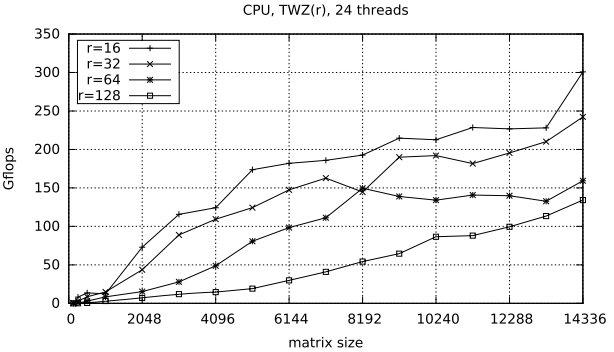
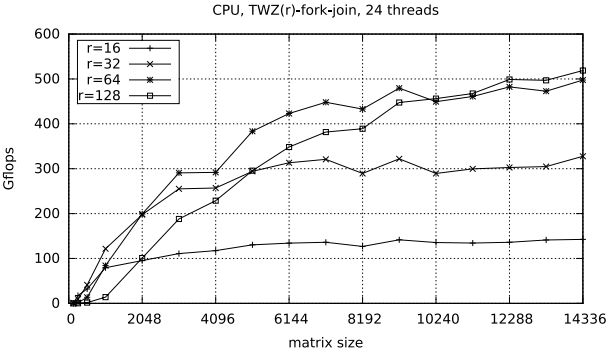


Fig. 4. Performance in Gflops of the parallel tiled WZ factorization algorithms (TWZ(r), TWZ(r)-fork-join and TWZ(r)-task) for a matrix of size 14 336 for four different r (16, 32, 64, 128) as a function of the number of threads.





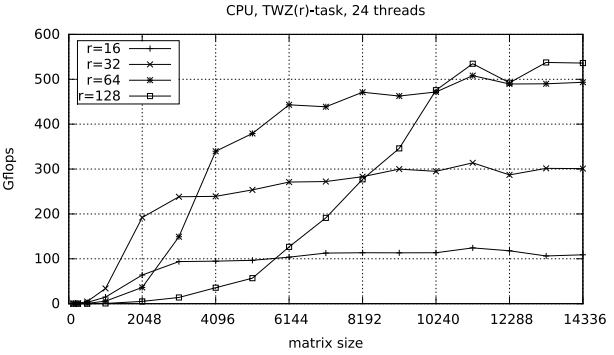


Fig. 5. Performance in Gflops of the parallel tiled WZ factorization algorithms (TWZ(r), TWZ(r)-fork-join and TWZ(r)-task) for 24 threads for four different r (16, 32, 64, 128) as a function of the matrix size.

that, every time we call any Level 3 BLAS routine, the operating system creates and manages threads, and at the end of each routine, there must be a barrier. It causes a big overhead. Thus, the lower the value of r, the smaller the overhead caused by managing the threads, and the parallel computations are executed on bigger data portions at once, because submatrices are bigger.

Conversely, for the second (that is, TWZ(r)-fork-join) and the third implementation (that is, TWZ(r)-task), we achieve the best performance for $r = 64$ or $r = 128$, depending on the number of threads and the matrix size. The threshold for the matrix size equals 10 240. Such behavior is caused by the parallelization of the loops with the use of OpenMP using the fork-join model or tasks. Thus, the bigger the value of r, the smaller the overhead caused by managing the threads.

The TWZ(r)-task implementation achieves better performance (more than 500 Gflops) than TWZ(r)-fork-join (500 Gflops) and TWZ(r) (only about 350 Gflops).

Figure 6 compares the performance of the best cases of all implementations of the parallel tiled WZ factorization algorithm (TWZ(16), TWZ(64/128)-fork-join and TWZ(128)-task), and the LU factorization without pivoting and with partial pivoting (the LAPACK block algorithms from the MKL).

The graph on the left-hand side reports the impact of the number of threads on the performance for a fixed total problem size. The graph on the right-hand side shows the performance using the maximum numbers of cores available on the system (24) with respect to the problem size.

The best performance is achieved for the TWZ(r)-task implementation for $r = 128$, and it is over 500 Gflops. The worst performance is achieved for the TWZ(16) implementation. All our implementations scale well with the number of threads and somewhat worse with the matrix size. Using tasks becomes usable for large size matrices and a big number of threads.

The LU implementations (both with partial pivoting and without pivoting) from the MKL are somewhat sensitive to the number of threads and the matrix size. They scale very well up to 23 threads for the matrix size 14 336. For the number of threads from 1 to 23, the performance of the LU implementations from the MKL is higher than in the case of our three WZ implementations (Fig. 6).

The issue with 24 threads is exceptional because of the fact that the machine frequency is not always the same (thanks to turbo boost mode it is higher when the machine is less loaded and lower when it is more loaded). Also, it can seen that the breakdown should be somewhat earlier, but we do not know the real frequencies used (apart from the fact that they are between a producer-specified minimum and maximum). It is also worth noting that the TWZ(r) implementations perform poorly for 24 threads, and the loss is dramatic, However, for TWZ(r)-fork-join and TWZ(r)-task, the drop is small or even none (besides 23 threads).

Also, for smaller matrices and 24 threads, the MKL implementations of the LU factorization with pivoting and without it provide better results than our implementations of the WZ factorization. For bigger problems, LU without pivoting has a performance comparable with our implementations.

Both the implementations (with partial pivoting and without pivoting) of the LU factorization from the MKL produce similar results because the diagonally dominant matrices were tested, which means that the pivoting was not done even once. At every step, the pivot was searched, which slightly influenced the performance time of the LU factorization.

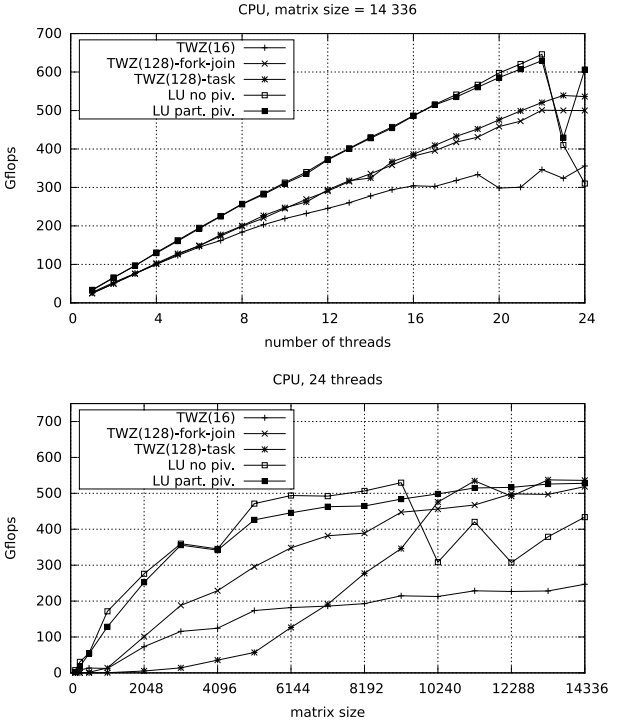


Fig. 6. Performance of all the implementations of the parallel tiled WZ factorization algorithm (TWZ(16), TWZ(128)-fork-join and TWZ(128)-task) and of both the implementations of the LU factorization (with partial pivoting and without pivoting—the LAPACK block algorithms from the MKL).

6.2. Speedup. In our proposed implementations only Stage 1 is not parallelized. In this section, we investigate the influence of this sequential part on the speedup possibilities.

Let T_p be the time to perform the computation using p threads. Speedup for p threads is defined as

$$S_p = \frac{T_1}{T_p}. \tag{24}$$

Figures 7–8 show the theoretical and experimental speedup as a function of the number of threads (1–24 threads) for a matrix of the size 14 336. Figure 7 reports the theoretical speedup from Amdahl's law (22) for different values of r. Figure 8 presents the speedup of the TWZ(16), TWZ(128)-fork-join and TWZ(128)-task implementations (the best ones from all kinds of implementations), both implementations of the LU factorization with partial pivoting and without pivoting and the best theoretical speedup (that is, for $r = 128$).

The theoretical speedup grows with the growth of r. The best experimental speedup is achieved for TWZ(128)-task and TWZ(128)-fork-join, and it is very close to the theoretical speedup. The speedup of the MKL LU implementations is not regular and depends significantly on the matrix size and the number of threads. Moreover, for a bigger number of threads, it is lower than the speedup of our implementations.

6.3. Numerical accuracy. The purpose of this section is not to perform a full study of the numerical stability and accuracy of the tiled WZ algorithm, but only to justify that the tile WZ algorithm can be used in practice. To study the numerical behavior of our implementations, we conducted numerical experiments. We used all the earlier matrices (we analyzed only dense square diagonally dominant matrices, as earlier), but here we show only the results for $n = 4096$, which represent the numerical performance trends for all other matrix sizes.

Table 2 shows the absolute error norms of our implementations which were computed as $||\mathbf{A} - \mathbf{WZ}||_\infty$. However, the matrices $\mathbf{W}$ and $\mathbf{Z}$ were computed with the use of the respective manners:

- $\mathbf{W}_s$ and $\mathbf{Z}_s$ are matrices from the sequential factorization described in Section 3;

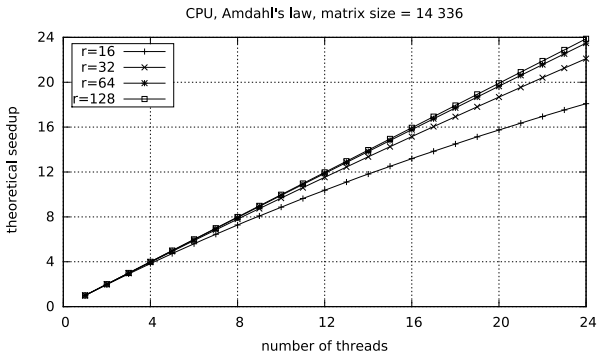


Fig. 7. Theoretical speedup according to Amdahl's law as a function of the number of threads for different values of r.

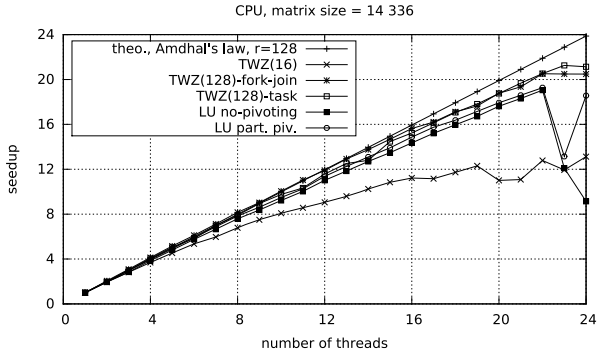


Fig. 8. Speedup of the parallel tiled WZ factorization algorithms (TWZ(r), TWZ(r)-fork-join and TWZ(r)-task) and both implementations of the LU factorization (with partial pivoting and without pivoting) (the LAPACK block algorithm from the MKL) compared with the best theoretical speedup for a matrix of size 14 336 for various numbers of threads.

Table 2. Numerical accuracy of parallel implementations of the tiled WZ factorization algorithms for $n = 4096$.

$\|\|\mathbf{A} - \mathbf{W}_s\mathbf{Z}_s\|\|_\infty$	—	$3.58e-06$
$\|\|\mathbf{A} - \mathbf{W}_{T(r)}\mathbf{Z}_{T(r)}\|\|_\infty$	$r = 16$	$4.58e-06$
	$r = 32$	$4.68e-06$
	$r = 64$	$4.35e-06$
	$r = 128$	$4.37e-06$
$\|\|\mathbf{A} - \mathbf{W}_{PT(r)}\mathbf{Z}_{PT(r)}\|\|_\infty$	$r = 16$	$4.58e-06$
	$r = 32$	$4.68e-06$
	$r = 64$	$4.35e-06$
	$r = 128$	$4.37e-06$
$\|\|\mathbf{A} - \mathbf{LU}\|\|_\infty$	—	$1.22e-05$

- $\mathbf{W}_{T(r)}$ and $\mathbf{Z}_{T(r)}$ are matrices from the tiled factorization described in Section 4 in the implementation shown in Section 5.2 (the norm does not depend on the number of threads);
- $\mathbf{W}_{PT(r)}$ and $\mathbf{Z}_{PT(r)}$ are matrices from the tiled factorization described in Section 4 in the implementation shown in Section 5.4 (the norm does not depend on the number of threads);
- $\mathbf{L}$ and $\mathbf{U}$ are matrices from the LU factorization.

For the numerical results presented here, we used the number of tiles described in Table 2. All the results are for double real precision.

The number of threads does not influence the accuracy, and neither does the implementation. The factorization seems stable. The norm is the same for the sequential implementation, and various tiled implementations so the tiled WZ algorithm sustains the accuracy of the numerical algorithm. It seems that the WZ factorization is a little more accurate than the LU one.

7. Conclusion

In this article, we studied the decomposition of a matrix $\mathbf{A}$ into factors $\mathbf{W}$ and $\mathbf{Z}$, for a nonsingular, dense, square matrix of an even size. Such a problem has a computationally intensive nature. To reduce the computing time significantly and to use contemporary computer architectures, we considered a partition of the matrix $\mathbf{A}$ into r^2 tiles. Such a tiled algorithm is implemented on a shared memory multicore system. Our implementations used a tiled WZ factorization algorithm. They employ Level 3 BLAS routines and the OpenMP standard with dynamic scheduling using both approaches, namely the fork-join model and the task-based model. Our implementations scale well with the growth of the number of threads and with the problem size. It is important that the parameter r influences the performance. The TWZ(r)-task implementation achieves a little higher performance than the TWZ(r)-fork-join implementation for bigger matrix sizes and a bigger number of threads.

The TWZ(r)-task implementation is better scalable than the `dgetrfnpi` routine (the LU factorization without pivoting) and the `dgetrf` one (the LU factorization with partial pivoting) from the MKL. A similar result for `dgetrf` from the MKL is presented by Dumas *et al*. (2016).

In future works, we plan to investigate a recursive version of the tiled WZ factorization (similarly as Dongarra *et al*. (2013)) algorithm and to use the tile layout (Gustavson, 1997) and the OpenMP runtime system for data flow paradigms on different multicore and manycore architectures. Moreover, we plan to implement the WZ factorization usable for practical problems, particularly for sparse matrices, which often come from real-life applications.

References

Agullo, E., Demmel, J., Dongarra, J., Hadri, B., Kurzak, J., Langou, J., Ltaief, H., Luszczek, P. and Tomov, S. (2009). Numerical linear algebra on emerging architectures: The PLASMA and MAGMA projects, *Journal of Physics: Conference Series* **180**(1): 012037.

Amdahl, G.M. (1967). Validity of the single processor approach to achieving large scale computing capabilities, *Proceedings of the Spring Joint Computer Conference, AFIPS'67 (Spring), Atlantic City, NJ, USA*, pp. 483–485.

Anderson, E., Bai, Z., Bischof, C., Blackford, S., Demmel, J., Dongarra, J., Du Croz, J., Greenbaum, A., Hammarling, S., McKenney, A. and Sorensen, D. (1999). *LAPACK Users' Guide*, 3rd Edn., SIAM, Philadelphia, PA.

Buttari, A., Langou, J., Kurzak, J. and Dongarra, J. (2009). A class of parallel tiled linear algebra algorithms for multicore architectures, *Parallel Computing* **35**(1): 38–53.

Bylina, B. (2018). The block WZ factorization, *Journal of Computational and Applied Mathematics* **331**(C): 119–132.

Bylina, B. and Bylina, J. (2007). Incomplete WZ factorization as an alternative method of preconditioning for solving Markov chains, *in* R. Wyrzykowski *et al.* (Eds.), *PPAM*, Lecture Notes in Computer Science, Vol. 4967, Springer, Berlin/Heidelberg, pp. 99–107.

Bylina, B. and Bylina, J. (2009). Influence of preconditioning and blocking on accuracy in solving Markovian models, *International Journal of Applied Mathematics and Computer Science* **19**(2): 207–217, DOI: 10.2478/v10006-009-0017-3.

Bylina, B. and Bylina, J. (2015). Strategies of parallelizing nested loops on the multicore architectures on the example of the WZ factorization for the dense matrices, *in* M. Ganzha *et al.* (Eds.), *Proceedings of the 2015 Federated Conference on Computer Science and Information Systems*, Annals of Computer Science and Information Systems, Vol. 5, IEEE, Piscataway, NJ, pp. 629–639.

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

Dongarra, J., DuCroz, J., Duff, I.S. and Hammarling, S. (1990). A set of level-3 basic linear algebra subprograms, *ACM Transactions on Mathematics Software* **16**(1): 1–17.

Dongarra, J.J., Faverge, M., Ltaief, H. and Luszczek, P. (2013). Achieving numerical accuracy and high performance using recursive tile LU factorization, *Concurrency and Computation: Practice and Experience* **26**(6): 1408–1431.

Dumas, J.G., Gautier, T., Pernet, C., Roch, J.L. and Sultan, Z. (2016). Recursion based parallelization of exact dense linear algebra routines for Gaussian elimination, *Parallel Computing* **57**: 235–249.

Evans, D. and Hatzopoulos, M. (1979). A parallel linear system solver, *International Journal of Computer Mathematics* **7**(3): 227–238.

Flynn, M.J. (1972). Some computer organizations and their effectiveness, *IEEE Transactions on Computers* **21**(9): 948–960.

García, I., Merelo, J., Bruguera, J. and Zapata, E. (1990). Parallel quadrant interlocking factorization on hypercube computers, *Parallel Computing* **15**(1–3): 87–100.

Gustavson, F.G. (1997). Recursion leads to automatic variable blocking for dense linear-algebra algorithms, *IBM Journal of Research and Development* **41**(6): 737–756.

Intel (2019). Math Kernel Library, `https://software.intel.com/en-us/mkl`.

Kurzak, J., Langou, J., Langou, C.D.J., Ltaief, H., Luszczek, P., Yarkhan, A., Haidar, A., Hoffman, J., Agullo, P.D.E., Buttari, A. and Hadri, B. (2010). PLASMA *Users' Guide*: *Parallel Linear Algebra Software for Multicore Architectures, Version 2.3.*, `http://icl.cs.utk.edu/projectsfiles/plasma/pdf/users_guide.pdf`.

Marqués, M., Quintana-Ortí, G., Quintana-Ortí, E.S. and van de Geijn, R.A. (2011). Using desktop computers to solve large-scale dense linear algebra problems, *The Journal of Supercomputing* **58**(2): 145–150.

Rao, S.C.S. (1997). Existence and uniqueness of WZ factorization, *Parallel Computing* **23**(8): 1129–1139.

Yalamov, P. and Evans, D. (1995). The WZ matrix factorisation method, *Parallel Computing* **21**(7): 1111–1120.

Yarkhan, A., Kurzak, J., Luszczek, P. and Dongarra, J. (2017). Porting the PLASMA numerical library to the OpenMP standard, *International Journal of Parallel Programming* **45**(3): 612–633, DOI:10.1007/s10766-016-0441-6.

Beata Bylina is a mathematics graduate (1998) and obtained her PhD degree in computer science in 2005 at the Institute of Theoretical and Applied Informatics of the Polish Academy of Sciences in Gliwice (Poland). She has been working in the Institute of Mathematics of Marie Curie-Skłodowska University in Lublin (Poland) since 1998, now as an assistant professor. Her research interests include numerical methods for linear equation systems and for ordinary differential equations, scientific and parallel computing, and mathematical software.

Jarosław Bylina is a mathematics graduate (1998) and holds a PhD in computer science obtained from the Silesian University of Technology in Gliwice (Poland) in 2006. He has been working at the Institute of Mathemtaics of Marie Curie-Skłodowska University in Lublin (Poland) since 1998, now as an assistant professor. He is interested in numerical methods for Markov chains, modelling of teleinformatic systems, programming languages, as well as parallel and distributed computing and processing.

Received: 8 September 2018
Revised: 12 January 2019
Accepted: 2 March 2019