Computing and Informatics, Vol. 38, 2019, 197–222, doi: 10.31577/cai_2019_1_197

ASSESSMENT OF TWO TASK FRAMEWORKS WITH DEPENDENCIES FOR MATRIX FACTORIZATIONS ON A MULTICORE ARCHITECTURE

Jarosław Bylina

Marie Curie-Sklodowska University
Institute of Mathematics
Pl. M. Curie-Sklodowskiej 5
20-031 Lublin, Poland
e-mail: `jaroslaw.bylina@umcs.pl`

Abstract. In this study, we evaluate two task frameworks with dependencies for important application kernels coming from the numerical linear algebra. In this approach, the algorithms of the matrix factorization are considered, namely the tiled LU and the WZ factorizations both without pivoting. In tiled algorithms, the operations are represented as a sequence of small tasks which operate on square blocks (tiles) of the data. The dependencies among tasks are expressed as a direct acyclic graph and the runtime system runs the graph on a multicore architecture. The performance of applications based on the task dependencies is related to efficient compilers and the runtime systems. We report the performance and the scalability of two task frameworks with dependencies on the multicore architecture for the matrix factorizations. Namely, we compare OpenMP and Intel Thread Building Blocks. Our results show that the number of tiles in both factorizations always have an impact on the performance and the speedup. Both the frameworks show their suitability for efficient parallelization of such applications, although both have their own merits and flaws.

Keywords: Task parallelism, task dependencies, parallel programming model, runtime system, OpenMP, Intel TBB

Mathematics Subject Classification 2010: 65Y05

1 INTRODUCTION

In recent years task-based parallel programming paradigms became an alternative to classical thread-based paradigms on the shared memory multicore architectures. Such task-based implementations of parallel applications are suitable for multicore architectures. The use of tasks causes higher concurrency and scalability of the implementations.

However, the task parallelism requires appropriate compilers and execution systems to perform efficiently. Such systems must decide about load-balancing, overheads, and task scheduling. It is difficult to evaluate the efficiency of such run-time systems because, for various applications, various criteria will be important. Contemporary architectures which employ shared-memory parallelism produce appearance of a lot of frameworks exploiting them efficiently – such as OpenMP [6], Intel Threading Building Blocks (TBB for short) [14], Cilk Plus [24], OpenCL [25] and others. Thus, choosing a proper framework for a specific problem is not easy.

Different techniques are provided by task-based programming frameworks to the programmer for writing programs. In some approaches, the algorithms are represented as graphs of tasks and the runtime system runs the graph on the target architecture. In the graphs, the nodes are computational tasks performed in kernel subroutines and edges represent the dependencies among them. In particular, in the application connected to the numerical linear algebra, direct acyclic graphs (DAGs or dags) are utilized. A dag is a finite directed graph without cycles. A dag contains a finite number of vertices and edges. Each edge is directed from one vertex to another.

Matrix factorization algorithms are dense linear algebra algorithms used often in many scientific applications. This paper addresses two matrix factorizations. In addition to the well-known LU factorization, we test another form of factorization, namely the WZ factorization. The WZ factorization was introduced in [8, 19]. It was a novel method for solving linear systems in parallel. They both have $O(n^3)$ time complexity using $O(n^2)$ data space. The tiled LU and WZ factorization algorithms use the standard set of Basic Linear Algebra Subprograms (BLAS) [7] and a block array layout for better cache performance. Moreover, the computations on array tiles (square blocks) fit the task-based parallel model well. The tiled LU and WZ algorithms can be represented as a dag where nodes are the executed BLAS routines.

The first contribution of this paper is providing details of implementations of the tiled LU and WZ factorizations in OpenMP and TBB; the second contribution is an analysis of the experimental results of these factorization implementations for two frameworks that support task parallelism with dependencies. In this article, we investigate two task frameworks, namely OpenMP and Intel TBB. We chose these frameworks because they are different in the way of implementation development. OpenMP is an extension to the C/C++ languages and TBB is a C++ library. These frameworks have also been chosen because they are quite popular, work for different architectures and CPUs, they can perform differently on different hardware architectures and they differ in their approaches to tasks. To compare OpenMP and TBB we study the tiled LU factorization without pivoting and the WZ factorization (also without pivoting).

The rest of this paper is organized as follows: Section 2 shows some related works. Section 3 presents the tiled LU factorization without pivoting and tiled WZ factorization without pivoting and shows dags (direct acyclic graphs) for each algorithm. Section 4 describes the details of parallel implementations of the tiled LU and tiled WZ algorithms on multicore, shared-memory machines. One of them relies on the use of the OpenMP `task` directive with the `depend` clause. The second one uses TBB. Section 5 is devoted to the results of numerical experiments carried out on shared memory multicore architectures and to the comparisons of the two task-based frameworks, namely OpenMP standard and TBB. Section 6 shows the conclusions of our research and presents future plans.

2 RELATED WORKS

In this paper, we evaluate two task frameworks with dependencies on the multicore architecture. Similarly, the issue of the comparison of the task parallel frameworks in the multicore environments is considered in the works [16, 20, 21, 22].

In the work [16], the tasks without dependencies are considered. The authors compare OpenMP 3.0 runtimes on unbalanced task graphs against Cilk and Intel TBB. The conclusion of these studies is the fact that the OpenMP task management mechanisms are less optimized than those of the other threading approaches, namely Cilk and Intel TBB.

The evaluation of OpenMP 4.0 tasks with dependencies with the benchmark called KASTORS consisting of small kernels ported to the OpenMP dependent task model is described in the paper [22]. KASTORS uses the OpenMP 4.0 task dependency constructs to extend different applications. One with these kernels is the LU decomposition from the PLASMA library (Parallel Linear Algebra for Scalable Multicore Architectures) framework [1, 13]. The performance of OpenMP applications expressing task dependencies is closely related to how efficiently compilers and runtime systems implement this new feature. The FLAME (Formal Linear Algebra Method) project [11, 15, 18] is another set of high performance libraries. Moreover, it is not only software but rather a formal approach to creating correct, fast and efficient linear algebra algorithms and their implementations.

The author of the work [21] evaluates Intel's C++ Concurrent Collections (CnC) and Threading Building Blocks (TBB) libraries for application coming from numerical linear algebra, namely tiled Gauss–Jordan algorithm. The conclusion of these studies is the fact that CnC is almost as fast as TBB.

The paper [20] aims to evaluate OpenMP, TBB and other ways of parallelization and optimization of computational problems that need task parallelism as well as data parallelism. The examples used there are adaptive Simpson's integration and Belman-Ford algorithm.

3 MATRIX DECOMPOSITION

The matrix decomposition is a factorization of a matrix into a product of matrices. We assume that the decomposed matrix is nonsingular, square, and diagonally dominant (thus, we can use factorization without pivoting). In this section, we describe two tiled matrix decompositions, namely the well-known tiled LU decomposition without pivoting and the tiled WZ decomposition (also without pivoting). Each of the algorithms is expressed in terms of the elementary operations and the graphs.

3.1 Block LU Factorization

Let the dense square ($n \times n$) diagonally dominant matrix $\mathbf{A}$ be partitioned into $q \times q$ tiles of size $t \times t$ ($n = qt$ and $1 \leq t \leq n$) and $\mathbf{A}_{ij}$ is a square tile on row i and column j. The tiled LU factorization algorithm performs the majority of its floating-point operations (flop) using the level 3 BLAS operations.

The tiled algorithm for the LU factorization may base on the following set of elementary operations.

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

Algorithm 1 presents the tiled LU factorization algorithm expressed in terms of elementary operations. The circled numbers shown in Algorithm 1 emphasize the correspondence between the operations and the tasks in Figure 1.

The scheduler executes tasks in any order that respects the dependencies shown in the dag. This approach is presented in [3] for tiled linear algebra algorithms. Figure 1 presents a directed acyclic graph for parallel tile LU factorization of a 4×4 tile matrix. Arrows show dependencies between tasks. The tasks are denoted by circles. The red circles (with the number 1) represent line 2 in Algorithm 1; the magenta circles (with the number 2) – line 4; the green circles (with the number 3) – line 7 and the blue ones (with the number 4) correspond to line 11.

3.2 Block WZ Factorization

The WZ factorization is described in [8, 19, 23]. Let us assume that $\mathbf{A}$ is a square, nonsingular and diagonally dominant matrix of the size $n \times n$ (we consider only even n, for simplicity's sake).

We are to find matrices $\mathbf{W}$ and $\mathbf{Z}$ that fulfill $\mathbf{WZ} = \mathbf{A}$. The main diagonal of the matrix $\mathbf{W}$ consists only of ones. The second diagonal consists of zeros. These

Algorithm 1 Tiled LU factorization

```
Require: $\mathbf{A}$, $q$
Ensure: $\mathbf{L}$, $\mathbf{U}$
 1: for  $k \leftarrow 1, q$ do
 2:   LU($\mathbf{A}_{kk}, \mathbf{L}_{kk}, \mathbf{U}_{kk}$)                        ①
 3:   for $i \leftarrow k+1, q$ do
 4:     DTRSM(nonu, up, q, $\mathbf{U}_{kk}$, $\mathbf{L}_{ik}$, $\mathbf{A}_{ik}$)      ②
 5:   end for
 6:   for $j \leftarrow k+1, q$ do
 7:     DTRSM(u, lo, l, $\mathbf{L}_{kk}$, $\mathbf{U}_{kj}$, $\mathbf{A}_{kj}$)         ③
 8:   end for
 9:   for $i \leftarrow k+1, q$ do
10:     for $j \leftarrow k+1, q$ do
11:       DGEMM($\mathbf{A}_{ij}$, $\mathbf{L}_{ik}$, $\mathbf{U}_{kj}$)               ④
12:     end for
13:   end for
14: end for
```

diagonals divide the matrix into four triangles. The left and right triangles contain non-zeros, and the top and bottom ones contain only zeros. The matrix $\mathbf{Z}$ has non-zeros where the matrix $\mathbf{W}$ has zeros or ones – and vice versa. The first part of the WZ factorization 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 1st and nth columns – from the 2nd row to the $(n-1)$th row. Next, we update the inner submatrix of $\mathbf{A}$ of the size $(n-2) \times (n-2)$ and for $k = 2, \ldots, \frac{n}{2}$ we zero elements in the kth and $(n-k+1)$st columns – from the $(k+1)$st row to the $(n-k)$th row and we update the inner submatrix.

The tiled WZ factorization algorithm [5] performs the majority of its floating-point operations (flop) using the level 3 BLAS operations. We assume that $\mathbf{A}$ is a square nonsingular matrix of an even size n and it is partitioned on $r \times r$ (r is also even) parts (r of each side – rows and columns). The tiled WZ algorithm consists of four repeating stages $r/2$ times. Stage 1 (line 3 in Algorithm 2) comprises the WZ factorization of a matrix built from four corner blocks of the input matrix. Stage 2 (lines 4–11 in Algorithm 2) computes $2s$ (where $s = \frac{n}{r}$) columns of the matrix $\mathbf{W}$ – s right columns and s left columns. Stage 3 (lines 12–19 in Algorithm 2) computes $2s$ rows of the matrix $\mathbf{Z}$ – s bottom rows and s top rows. Stage 4 (lines 20–25 in Algorithm 2) updates the inner submatrix of $\mathbf{A}$ – that is, $\mathbf{A}$ without outer $2s$ columns and $2s$ rows. In the next step, the algorithm is repeated for this inner matrix. The tiled algorithm for the WZ factorization will be based on the following set of elementary operations.

- `DTRSM(u/nonu, up/lo, l/r, A, X, B)`. This BLAS subroutine is used to compute $\mathbf{X} = \mathbf{A}^{-1} \cdot \mathbf{B}$ (denoted by `l`), or $\mathbf{X} = \mathbf{B} \cdot \mathbf{A}^{-1}$ (denoted by `r`), where $\mathbf{X}$

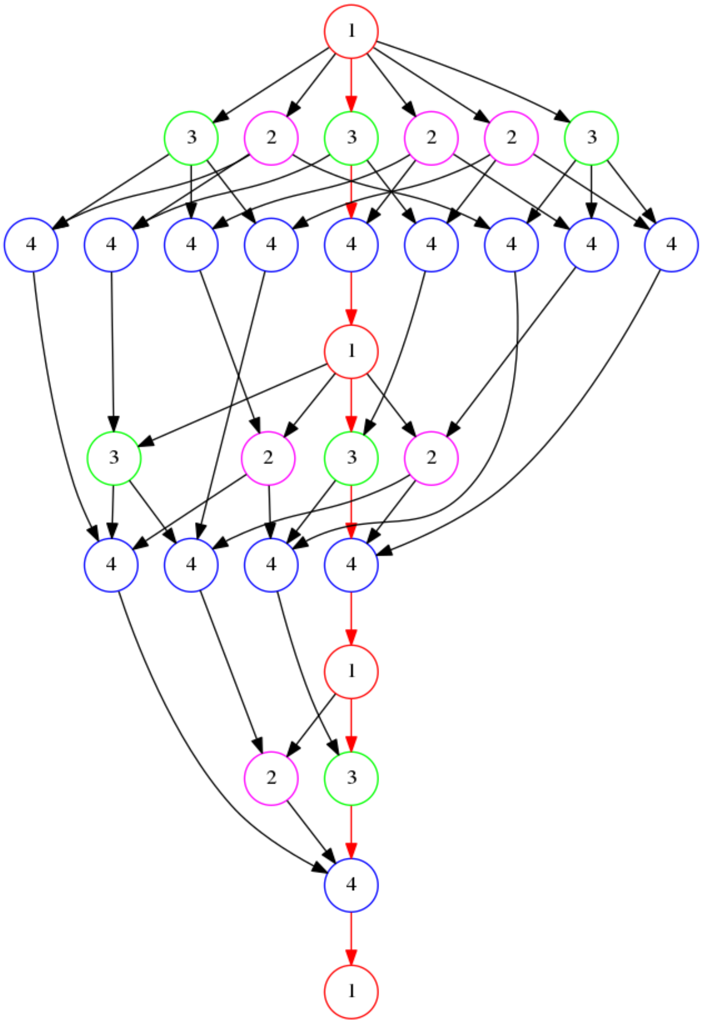

Figure 1. A directed acyclic graph for the tile LU factorization of a 4×4 tiled matrix

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 $\mathbf{A} = -\mathbf{B} \cdot \mathbf{C} + \mathbf{A}$, 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 $\mathbf{A} = -\mathbf{B} \cdot \mathbf{C} + \mathbf{D}$, where $\mathbf{A}$, $\mathbf{B}$, $\mathbf{C}$, and $\mathbf{D}$ are $s \times s$ matrices.

Algorithm 2 presents the tiled WZ factorization algorithm expressed with the above-mentioned operations (`DTRSM`, `DGEMM`, `DGEMM_copy`) for a nonsingular matrix $\mathbf{A}$ partitioned into $r \times r$ blocks. The matrices $\mathbf{W}$ and $\mathbf{Z}$ are the results of this algorithm. Again, the circled numbers in Algorithm 2 show which operations belong to respective tasks in Figure 2.

Algorithm 2 Tiled WZ factorization

```
Require: $\mathbf{A}$, $r$
Ensure: $\mathbf{W}$, $\mathbf{Z}$
 1: for $k \leftarrow 1, r/2$ do
 2:   $k_2 \leftarrow r - k + 1$
 3:   WZ($\begin{bmatrix} \mathbf{A}_{kk} & \mathbf{A}_{kk_2} \\ \mathbf{A}_{k_2k} & \mathbf{A}_{k_2k_2} \end{bmatrix}$, $\begin{bmatrix} \mathbf{W}_{kk} & \mathbf{W}_{kk_2} \\ \mathbf{W}_{k_2k} & \mathbf{W}_{k_2k_2} \end{bmatrix}$, $\begin{bmatrix} \mathbf{Z}_{kk} & \mathbf{Z}_{kk_2} \\ \mathbf{Z}_{k_2k} & \mathbf{Z}_{k_2k_2} \end{bmatrix}$) ①
 4:   DTRSM(nonu, up, l, $\mathbf{Z}_{kk}$, $\mathbf{D_1}$, $\mathbf{Z}_{kk_2}$)   ①
 5:   DGEMM_copy($\mathbf{E_1}$, $\mathbf{Z}_{k_2k}$, $\mathbf{D_1}$, $\mathbf{Z}_{k_2k_2}$)   ①
 6:   DTRSM(u, lo, r, $\mathbf{W}_{kk}$, $\mathbf{D_2}$, $\mathbf{W}_{k_2k}$)   ①
 7:   DGEMM_copy($\mathbf{E_2}$, $\mathbf{D_2}$,$\mathbf{W}_{kk_2}$,$\mathbf{W}_{k_2k_2}$)   ①
 8:   for $i \leftarrow k + 1, k_2 - 1$ do
 9:     DGEMM($\mathbf{A}_{ik_2}$, $\mathbf{A}_{ik}$, $\mathbf{D_1}$)   ②
10:     DTRSM(nonu, lo, r, $\mathbf{E_1}$, $\mathbf{W}_{ik_2}$, $\mathbf{A}_{ik_2}$)   ②
11:     DGEMM($\mathbf{A}_{ik}$, $\mathbf{W}_{ik_2}$, $\mathbf{Z}_{k_2k}$)   ②
12:     DTRSM(nonu, up, r, $\mathbf{Z}_{kk}$, $\mathbf{W}_{ik}$, $\mathbf{A}_{ik}$)   ②
13:   end for
14:   for $i \leftarrow k + 1, k_2 - 1$ do
15:     DGEMM($\mathbf{A}_{k_2i}$, $\mathbf{D_2}$, $\mathbf{A}_{ki}$)   ③
16:     DTRSM(u, up, l, $\mathbf{E_2}$, $\mathbf{Z}_{k_2i}$, $\mathbf{A}_{k_2i}$)   ③
17:     DGEMM($\mathbf{A}_{ki}$, $\mathbf{W}_{kk_2}$, $\mathbf{Z}_{k_2i}$)   ③
18:     DTRSM(u, lo, l, $\mathbf{W}_{kk}$, $\mathbf{Z}_{ki}$, $\mathbf{A}_{ki}$)   ③
19:   end for
20:   for $j \leftarrow k + 1, k_2 - 1$ do
21:     for $i \leftarrow k + 1, k_2 - 1$ do
22:       DGEMM($\mathbf{A}_{ij}$, $\mathbf{W}_{ik}$, $\mathbf{Z}_{kj}$)   ④
23:       DGEMM($\mathbf{A}_{ij}$, $\mathbf{W}_{ik_2}$, $\mathbf{Z}_{k_2j}$)   ④
24:     end for
25:   end for
26: end for
```

Algorithm 2 can be represented as a dag. Figure 2 shows such a dag for the tiled WZ factorization when Algorithm 2 is executed for a 4×4 tiled matrix. This figure also corresponds to the lines in Algorithm 2. The tasks are denoted by circles and the red circles (with the number 1) represent lines 3–7; the magenta circles (with the number 2) – lines 9–12; the green circle (with the number 3) – lines 15–18 and the blue ones (with the number 4) correspond to lines 22–23.

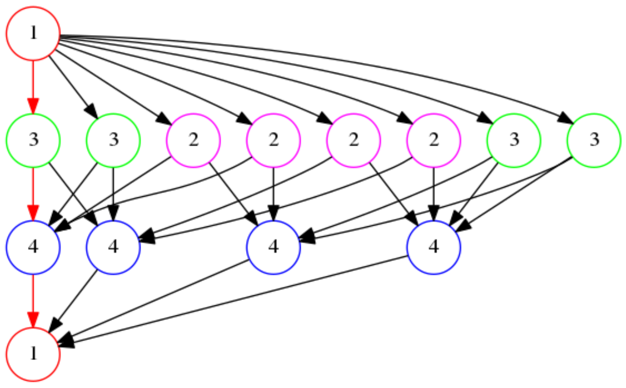

Figure 2. A dag for the tiled WZ factorization of a 4×4 tiled matrix

3.3 Theoretical Speedup – Amdahl's Law

We compute a maximal theoretical speedup of our algorithms from Amdahl's law, using the cost of the sequential traditional versions, that is:

$$C_{LU}(n) = \frac{2}{3}n^3 - \frac{1}{2}n^2 - \frac{1}{6}n,$$
$$C_{WZ}(n) = \frac{2}{3}n^3 - \frac{7}{3}n - 3.$$

Let us compute the cost of Algorithm 2, namely $C_{WZ}(n, s)$ (as it depends not only on n but also on the size s of the tile). To achieve this, we are to compute costs of the particular stages 1–4 which we denote C_{WZ1}, C_{WZ2}, C_{WZ3} and C_{WZ4}, respectively ($n = r \cdot s$):

$$C_{WZ1}(s) = C_{WZ}(2s) = \frac{16}{3}s^3 - \frac{7}{3}s - 3,$$
$$\begin{aligned} C_{WZ2}(k, r, s) = C_{WZ3}(k, r, s) &= 3s^3 + s^2 + \sum_{i=k+1}^{r-k} (6s^3 + 2s^2) \\ &= 3s^3 + s^2 + (6s^3 + 2s^2)(r - 2k), \end{aligned}$$
$$C_{WZ4}(k, r, s) = \sum_{i=k+1}^{r-k} \sum_{j=k+1}^{r-k} (4s^3 + 2s^2) = (4s^3 + 2s^2)(r - 2k)^2.$$

Thus, the number of floating-point arithmetic operations for the tiled WZ factorization algorithm (Algorithm 2) is:

$$\begin{aligned} C_{WZ}(n,s) &= \sum_{k=1}^{\frac{r}{2}} (C_{WZ1}(s) + 2C_{WZ2}(k,r,s) + C_{WZ4}(k,r,s)) \\ &= \frac{n^3(4s+2) + 6n^2s^2 + n(6s^3 - 2s^2 - 7s - 9)}{6s}. \end{aligned}$$

Analogously, we can obtain the cost of the tiled LU factorization (here, $n = q \cdot t$ and the size of the block is t; we present only formulas necessary for further considerations), namely:

$$\begin{aligned} C_{LU1}(t) &= \frac{2}{3}t^3 - \frac{1}{2}t^2 - \frac{1}{6}t, \\ C_{LU}(n,t) &= \sum_{k=1}^{q} (C_{LU1}(t) + 2C_{LU2}(k,q,t) + C_{LU4}(k,q,t)) \\ &= \frac{n^3(4t+2) - 3n^2t - n(2t^2 + t)}{6t}. \end{aligned}$$

The maximal theoretical speedup for p threads can be estimated from Amdahl's law. To use this law we must determine which part must be executed sequentially, and which part can be executed in parallel. In our algorithms, the only parts which have to be executed sequentially are the first stages (denoted with $\textcircled{1}$).

Thus, let P_{WZseq} be the relative cost of this sequential part of Algorithm 2. The cost of one execution of stage 1 is C_{WZ1}, but it is executed $\frac{r}{2}$ times. So:

$$P_{WZseq}(n,s) = \frac{\frac{r}{2} \cdot C_{WZ1}(s)}{C_{WZ}(n,s)} = \frac{16s^3 - 7s - 9}{n^2(4s+2) + 6ns^2 + 6s^3 - 2s^2 - 7s - 9}.$$

According to Amdahl's law [10], the best theoretical speedup for the parallel tiled WZ factorization algorithm (for p threads, $n \times n$ matrix and $s \times s$ tile) is:

$$S_{WZ}(p;n,s) = \frac{1}{P_{WZseq}(n,s) + \frac{1 - P_{WZseq}(n,s)}{p}}.$$

Analogously, we can obtain similar formulas for the tiled LU factorization (here, $n = q \cdot t$ and the size of the block is t; we present only formulas necessary for further considerations), namely:

$$
\begin{aligned}
C_{LU1}(t) &= \frac{2}{3}t^3 - \frac{1}{2}t^2 - \frac{1}{6}t,\\
C_{LU}(n,t) &= \sum_{k=1}^{q}(C_{LU1}(t) + 2C_{LU2}(k,q,t) + C_{LU4}(k,q,t)\\
&= \frac{n^3(4t+2) - 3n^2t - n(2t^2+t)}{6t},\\
P_{LUseq}(n,s) &= \frac{q\cdot C_{LU1}(t)}{C_{LU}(n,t)} = \frac{4t^3-3t^2-t}{n^2(4t+2)-3nt-2t^2-t},\\
S_{LU}(p;n,t) &= \frac{1}{P_{LUseq}(n,t) + \frac{1-P_{LUseq}(n,t)}{p}}.
\end{aligned}
$$

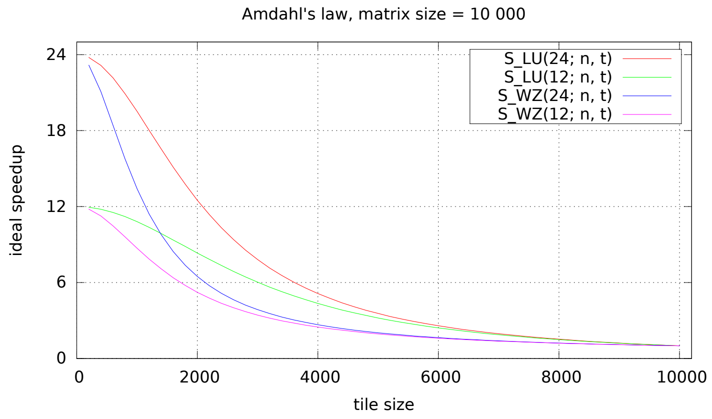

Figure 3. The theoretical maximum speedup for the tiled LU and WZ implementations

Figure 3 shows the theroetical maximum speedup as a function of the size of the block, for fixed $n = 1\,000$ and for selected numbers p of threads ($p \in \{12, 24\}$). We can see from Figure 3 that the speedup should be the best for as small blocks as possible. However, smaller blocks require more communication and synchronization – which is not counted in Amdahl's law. So the size of the block has to be chosen experimentally.

4 IMPLEMENTATIONS

In our work, the matrices are stored as one-dimensional arrays of the tiles and we refer to it as a tiled layout, similarly to [3]. In the tile layout, the matrices are represented as small square tiles of data contiguous in memory so that each core can operate on an individual tile independently.

4.1 OpenMP

In our first implementations, we employ the OpenMP task directive and the BLAS routines for matrices' operations. We call these implementations TLU(q)-OpenMP and TWZ(r)-OpenMP. The well-known OpenMP standard was extended with the task construct introduced in version 3.0 [16] with support for task dependencies by means of the `depend` clause. The clause allows defining lists of data items that are only inputs, only outputs, or both inputs and outputs. The annotated task will be scheduled for execution only when the dependencies expressed by those data items are satisfied with respect to preceding tasks in the same task region. 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 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. Algorithms 3 and 4 present the tiled factorization with the `#pragma omp task` with dependencies (and the color circles representing the content of the particular tasks).

Algorithm 3 Tiled LU factorization – task-based with dependencies

```
Require: A, q
Ensure: L, U
 1: for $k \leftarrow 1, q-1$ do
 2:   #pragma omp task depend(in:$A_{kk}$) depend(out: $L_{kk}$) depend(out: $U_{kk}$)
 3:   LU($\mathbf{A}_{kk}, \mathbf{L}_{kk}, \mathbf{U}_{kk}$)                                  ①
 4:   for $i \leftarrow k+1, q$ do
 5:     #pragma omp task depend(in:$A_{ik}$) depend(in: $U_{kk}$) depend(out: $L_{ik}$)
 6:     DTRSM(nonu, up, q, $\mathbf{U}_{kk}$, $\mathbf{L}_{ik}$, $\mathbf{A}_{ik}$)                 ②
 7:   end for
 8:   for $j \leftarrow k+1, q$ do
 9:     #pragma omp task depend(in:$A_{kj}$) depend(in: $L_{kk}$) depend(out: $U_{kj}$)
10:     DTRSM(u, lo, 1, $\mathbf{L}_{kk}$, $\mathbf{U}_{kj}$, $\mathbf{A}_{kj}$)                    ③
11:   end for
12:   for $i \leftarrow k+1, q$ do
13:     for $j \leftarrow k+1, q$ do
14:       #pragma omp task depend(in:$L_{ik}$) depend(in: $U_{kj}$) depend(inout: $A_{ij}$)
15:       DGEMM($\mathbf{A}_{ij}$, $\mathbf{L}_{ik}$, $\mathbf{U}_{kj}$)                          ④
16:     end for
17:   end for
18: end for
```

Algorithm 4 Tiled WZ factorization – task-based with dependencies

```
Require: $\mathbf{A}$, $r$
Ensure: $\mathbf{W}$, $\mathbf{Z}$
 1: for $k \leftarrow 1, r/2$ do
 2:     $k_2 \leftarrow r - k + 1$
 3:     #pragma omp task depend(in: $A_{kk}$, $A_{kk_2}$, $A_{k_2k_2}$, $A_{k_2k}$)
        depend(out: $W_{kk}$, $W_{kk_2}$, $W_{k_2k_2}$, $W_{k_2k}$, $Z_{kk}$, $Z_{kk_2}$, $Z_{k_2k_2}$, $Z_{k_2k}$, $D_1$, $E_1$, $D_2$, $E_2$ )
 4:     WZ($\begin{bmatrix} \mathbf{A}_{kk} & \mathbf{A}_{kk_2} \\ \mathbf{A}_{k_2k} & \mathbf{A}_{k_2k_2} \end{bmatrix}$, $\begin{bmatrix} \mathbf{W}_{kk} & \mathbf{W}_{kk_2} \\ \mathbf{W}_{k_2k} & \mathbf{W}_{k_2k_2} \end{bmatrix}$, $\begin{bmatrix} \mathbf{Z}_{kk} & \mathbf{Z}_{kk_2} \\ \mathbf{Z}_{k_2k} & \mathbf{Z}_{k_2k_2} \end{bmatrix}$)   $\textcircled{1}$
 5:     DTRSM(nonu, up, l, $\mathbf{Z}_{kk}$, $\mathbf{D_1}$, $\mathbf{Z}_{kk_2}$)   $\textcircled{1}$
 6:     DGEMM_copy($\mathbf{E_1}$, $\mathbf{Z}_{k_2k}$, $\mathbf{D_1}$, $\mathbf{Z}_{k_2k_2}$)   $\textcircled{1}$
 7:     DTRSM(u, lo, r, $\mathbf{W}_{kk}$, $\mathbf{D_2}$, $\mathbf{W}_{k_2k}$)   $\textcircled{1}$
 8:     DGEMM_copy($\mathbf{E_2}$, $\mathbf{D_2}$,$\mathbf{W}_{kk_2}$,$\mathbf{W}_{k_2k_2}$)   $\textcircled{1}$
 9:     for $i \leftarrow k+1, k_2-1$ do
10:         #pragma omp task depend(in: $A_{ik}$, $A_{ik_2}$, $Z_{kk}$, $Z_{k_2k}$, $D_1$, $E_1$)
                depend(out: $W_{ik}$, $W_{ik_2}$)
11:         DGEMM($\mathbf{A}_{ik_2}$, $\mathbf{A}_{ik}$, $\mathbf{D_1}$)   $\textcircled{2}$
12:         DTRSM(nonu, lo, r, $\mathbf{E_1}$, $\mathbf{W}_{ik_2}$, $\mathbf{A}_{ik_2}$)   $\textcircled{2}$
13:         DGEMM($\mathbf{A}_{ik}$, $\mathbf{W}_{ik_2}$, $\mathbf{Z}_{k_2k}$)   $\textcircled{2}$
14:         DTRSM(nonu, up, r, $\mathbf{Z}_{kk}$, $\mathbf{W}_{ik}$, $\mathbf{A}_{ik}$)   $\textcircled{2}$
15:     end for
16:     for $i \leftarrow k+1, k_2-1$ do
17:         #pragma omp task depend(in: $A_{k_2i}$, $A_{ki}$, $W_{kk_2}$, $W_{kk}$, $D_2$, $E_2$)
                depend(out: $Z_{ki}$, $Z_{k_2i}$)
18:         DGEMM($\mathbf{A}_{k_2i}$, $\mathbf{D_2}$, $\mathbf{A}_{ki}$)   $\textcircled{3}$
19:         DTRSM(u, up, l, $\mathbf{E_2}$, $\mathbf{Z}_{k_2i}$, $\mathbf{A}_{k_2i}$)   $\textcircled{3}$
20:         DGEMM($\mathbf{A}_{ki}$, $\mathbf{W}_{kk_2}$, $\mathbf{Z}_{k_2i}$)   $\textcircled{3}$
21:         DTRSM(u, lo, l, $\mathbf{W}_{kk}$, $\mathbf{Z}_{ki}$, $\mathbf{A}_{ki}$)   $\textcircled{3}$
22:     end for
23:     for $j \leftarrow k+1, k_2-1$ do
24:         for $i \leftarrow k+1, k_2-1$ do
25:             #pragma omp task depend(in: $W_{ik}$, $W_{ik_2}$, $Z_{kj}$, $Z_{k_2j}$) depend(out: $A_{ij}$)
26:             DGEMM($\mathbf{A}_{ij}$, $\mathbf{W}_{ik}$, $\mathbf{Z}_{kj}$)   $\textcircled{4}$
27:             DGEMM($\mathbf{A}_{ij}$, $\mathbf{W}_{ik_2}$, $\mathbf{Z}_{k_2j}$)   $\textcircled{4}$
28:         end for
29:     end for
30: end for
```

It would seem that keeping some matrices in cache between tasks would be profitable. However, forcing it is not possible. Moreover, even if it would be possible, it is not desirable – we use tasks in our implementations and the data are in the cache during one task execution (if the block size is not too big), but locking them between tasks would restrict the task scheduler. The task scheduler itself has to decide, which tasks are to be run on which processors, considering the cache content and the dependencies.

4.2 TBB

Our second set of implementations uses Intel Thread Building Blocks (TBB) and similarly, as in our previous implementations, they call the BLAS routines for matrix operations. We denote these implementations TLU(q)-TBB and TWZ(r)-TBB. The Intel Threading Building Blocks (Intel TBB) [12, 17] is a C++ template library for parallel programming on multicore architectures. This library provides parallel constructs like algorithms, containers, and tasks which the programmer can use to implement an algorithm and run it in parallel.

The TBB task interface requires the declaration of a new class extending the task class and the creation of task object instances. A member function executes the work of the task. However, there are also other tools to run a task-based algorithm – and one of them is the flow graph.

The greatest advantage of this approach is the separation of concerns. We can do the following implementation jobs independently:

- describe the algorithm;
- design and implement small independent computational kernels;
- connect them with the graph to schedule them efficiently.

The use of the flow graph (which can be an arbitrary directed graph, not only a dag) in TBB is different from the OpenMP. Here, the programmer has to build a dependency graph on his own – quoting the dependencies is not enough. For building the graph, there are (among others) following elements (all from the `tbb::flow` namespace):

- the class `graph` – this is the main class which provides the graph implementation;
- the class template `continue_node` – this is an auxiliary class which represents a single node of the graph – and a task at the same time. The main job of the node is storing a functor which describes actions to be performed on this node.
- the class `continue_msg` – it is a helper class used as a signal between consecutive nodes;
- the function template `make_edge` connects the nodes and thus, it determines the sequence of the nodes (and the actions, at the same time) and – which can be more important – also the dependencies between nodes. A `continue_node`

can perform any actions if and only if all its previous nodes (connected with it directly by edges) finished their actions.

We should also mention `try_put` (a node method which sends the first `continue_msg` in order to start the computations) and `wait_for_all` (a graph method which waits for all the computations to finish).

Some fragments of the code of TLU(q)-TBB are shown in Listing 1 (it is `LU_Graph` – the main class responsible for building the graph and conducting the computations).

The maps (`nodes_lu`, `nodes_U`, `nodes_L`, `nodes_X`) store (smart) pointers to the nodes and are crucial in building the graph (thanks to them, all the created nodes are easy to reference).

The constructor creates nodes and edges with the use of the functions: `red_node`, `green_node`, `magenta_node` and `blue_node` (names according to colors from Figure 1). Only one (`green_node`) of these functions is shown – the others are similar.

Then, we have some helper functions (`red_action`, `green_action`, `magenta_action` and `blue_action`) which describe the desired computational actions for respective nodes and return computational kernels in the form of lambdas. In them, we use some macros but there are just BLAS routines inside. Again, only one function (`magenta_action` this time) is fully shown here.

Finally, we can see the method `run` which starts the computations and waits for them to finish.

The idea of the TWZ(r)-TBB implementation is the same – although the dependencies are somewhat different (see Figure 2) and the kernels are more complicated (what can be inferred from Algorithm 4).

5 NUMERICAL EXPERIMENTS

We tested the performance of two matrix decompositions, namely the tiled LU factorization and the tiled WZ factorization. We compared four implementations of these matrix decompositions, that is:

- TLU(q)-OpenMP – a parallel implementation of the tiled LU factorization with the use of single-threaded level 3 BLAS routines (`DTRSM` and `DGEMM`) from the MKL library and the OpenMP standard with tasks and the dynamic scheduling;
- TWZ(r)-OpenMP – a parallel implementation of the tiled WZ factorization with the use of single-threaded level 3 BLAS routines (`DTRSM` and `DGEMM`) from the MKL library and the OpenMP standard with tasks and the dynamic scheduling;
- TLU(q)-TBB – a parallel implementation of the tiled LU factorization with the use of single-threaded level 3 BLAS routines (`DTRSM` and `DGEMM`) from the MKL library and a TBB flow graph;
- TWZ(r)-TBB – a parallel implementation of the tiled WZ factorization with the use of single-threaded level 3 BLAS routines (`DTRSM` and `DGEMM`) from the MKL library and a TBB flow graph.

```
using namespace tbb::flow;
class LU_Graph {
private:
    graph g;
    /* other class members */
    std::map<std::vector<int>,
          std::shared_ptr<continue_node<continue_msg>>>
                nodes_lu, nodes_U, nodes_L, nodes_X;
public:
    LU_Graph(int q, /* other parameters */) { /*...*/ }

    void red_node(int k) { /*...*/ }
    void green_node(int k, int i) {
        nodes_U[ {k, i} ] =
            std::make_shared<continue_node<continue_msg>>
            (g,
             green_action(k, i));
        make_edge(*nodes_lu.at({k}),
                  *nodes_U.at({k, i}));
    }
    void magenta_node(int k, int j) { /*...*/ }
    void blue_node(int k, int i, int j) { /*...*/ }

    auto red_action(int k) { /*...*/ }
    auto green_action(int k, int i) { /*...*/ }
    auto magenta_action(int k, int j) {
        return [=](const continue_msg &) {
            TLU_DTRSM_no_copy(TLU_u, TLU_lo, TLU_l,
                              TILE_ADDR_X(L, k, k),
                              TILE_ADDR_X(A, k, j));
        };
    }
    auto blue_action(int k, int i, int j) { /*...*/ }

    void run() {
        nodes_lu.at({0})->try_put(continue_msg());
        g.wait_for_all();
    }
};
```

Listing 1. Fragments of the main class responsible for building the graph and conducting the computations in the TLU(q)-TBB implementation

Table 1 shows details of the specification of the hardware and software used in the numerical experiment. The flags used in compilation and linking were: `-mkl=sequential -fopenmp -O3 -ip -no-prec-div -fp-model fast=2 -std=c++14 -ltbb`. The theoretical peak performance (in Gflops) can be computed from the specification, with the use of the formula:

$$\begin{aligned}\#\text{ of cores} \times \text{clock frequency in GHz} \times \text{flops per cycle} &= 24 \times 2.3 \times 16 \\ &= 883.2\,[\text{Gflops}]\end{aligned}$$

CPU	2 × Intel Xeon E5-2670 v.3 (Haswell)
# of cores	24 (12 per socket)
# of threads	48 (2 per core)
clock	2.30 GHz
level 1 data cache	32 kB per core
level 2 cache	256 kB per core
compiler	Intel ICC 16.0.0
BLAS/LAPACK libraries	MKL 2016.0.109

Table 1. Hardware and software used in the experiments

The input matrices were generated by the authors. They were random, square, dense matrices, with a dominant diagonal of even sizes (1 024, 2 018, . . . 14 336). Various numbers of tiles were tested, namely, each matrix was divided into 16, 32, 64, and 128 tiles for each side (both for the rows and the columns). The matrices are stored (from the beginning) in a tiled format [9] – as shown in Figure 4.

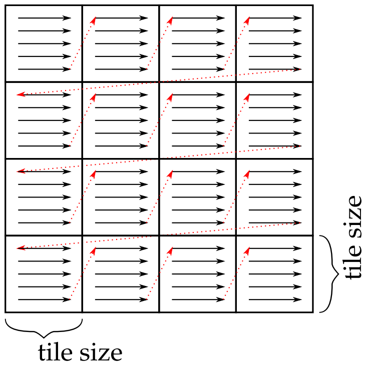

Figure 4. Memory layout of the test matrices. Arrows show data sequence in memory (black: within a tile; red: between tiles).

The performance times were measured with the use of a standard function, namely `(omp_get_wtime())`. The measured performance time does not include the time needed for the matrix generation and for storing it in the aforementioned tiled format. However, it was quite short ($O(n^2)$), relative to the time of the factorizations ($O(n^3)$).

We set the number of OpenMP threads using the `omp_set_num_threads` function and the number of TBB threads with the use of `tbb::task_scheduler_init`. All the experiments reported below were performed with the use of the double-precision arithmetic.

5.1 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 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

$$\frac{2n^3}{3 \cdot T \cdot 10^9},$$

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

Figure 5 presents the performance (in Gflops) of the TWZ(r)-OpenMP, TWZ(r)-TBB, TLU(q)-OpenMP and TLU(q)-TBB for the number of threads 24 for four different number of tiles (16, 32, 64, 128) as a function of the matrix size. We tested matrices of the sizes being multiples of 1 024, thus we were limited to the numbers dividing 1 024, that is, powers of 2. Thus, we chose above-mentioned numbers. However, some other tests (conducted on the matrix of the size 15 120 which has many more divisors) showed that the best results are obtained for q between 32 and 64 (LU) and r between 64 and 128 (WZ).

We can observe that the number of tiles has a great impact on the performance. For a wrongly chosen number of tiles (especially in the WZ factorization), the performance can drop drastically (e.g., even to about 100 Gflops for WZ with $r = 16$ in both frameworks). For the LU factorization and $q = 128$, the performance is also poor. Having analyzed all the experiments for TLU-OpenMP, we can see that the values $q = 16$ and $q = 128$ can be dismissed. On the other hand, for smaller matrices (up to the size 8 192), $q = 32$ is the best and for bigger ones (12 288 and more), we should choose $q = 64$. Between 8 192 and 12 288, the choice is ambiguous – $q = 32$ or $q = 64$ is better, but they are similar. For TLU-TBB, we can ignore $q = 128$ and $q = 64$ (never being the best choices). For small matrices (up to 6 144), the better is smaller of the remaining ones (that is, $q = 16$) and for bigger matrices (7 168 and more), the better is $q = 32$. For the tiled LU factorization, both frameworks are very close – usually, OpenMP prevails, but the differences are very minute. However, we can see that in TLU, OpenMP needs q to be twice as big as for TBB.

After the analysis for TWZ-OpenMP, we can see that this implementation behaves the best for $r = 64$ (up to the size 10 240) and for $r = 128$ (for the matrix size 11 264 and more). The parameter $r = 16$ is never the best and for very small matrices, $r = 32$ gives good results. The TWZ-TBB implementation gives the best results for $r = 32$ (but only for the size 4 096 and less) and for $r = 64$ (from 5 120). The other values ($r = 16$ and $r = 128$) do not perform well. Again, the best results for both tiled WZ implementations are very close and we cannot assess which is the best. Moreover, in TWZ, we can also see that OpenMP needs r to be twice as big as for TBB.

Both the algorithms perform better in TBB if the parameter (q and r) is two times smaller than in OpenMP. In other words, the TBB versions work better for bigger portions of the data. Precisely: four times bigger – because the optimal linear size of the tile is twice bigger in TBB than in OpenMP; so the amount of data is four times bigger in TBB. That leaves an open question: why is this so?

To sum up, the performance depends strongly on the size of the matrix (what is quite obvious) and on the number of tiles (that is q or r) – thus, indirectly on the sizes of a single tile. The framework itself (OpenMP or TBB) has only a slight impact.

Implementation	Time [s]	Performance [Gflops]	% of Peak Performance
MKL LU	3.15	623.67	70.61 %
TLU(64)-OpenMP	2.98	658.28	74.53 %
TLU(32)-TBB	3.24	606.66	68.69 %
TWZ(128)-OpenMP	3.85	509.62	57.70 %
TWZ(64)-TBB	3.68	533.40	60.39 %

Table 2. The comparison of the best times and performances for four presented implementations for 24 threads and the matrix size of 14 336

In Table 2, we can see the best times and performances (with its values of r or q) and peak performance percentages, chosen experimentally for a matrix of size 14 336 and 24 threads – for each considered implementation and for a vendor MKL LU factorization. For the largest matrix size (14 336), the TLU algorithm achieves the best performance in the OpenMP implementation, although for the TWZ algorithm, the TBB implementation is better. Our implementations gave comparable results to the results of a vendor implementation (namely, the LU factorization without pivoting from the MKL library, that is `dgetrfnpi`). The same tests were also conducted for similar sizes (like 14 208, 14 464 and others; to exclude problems with cache associativity) and the general performance is very similar. However, not always the TLU(64)-OpenMP implementation was the best.

5.2 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.

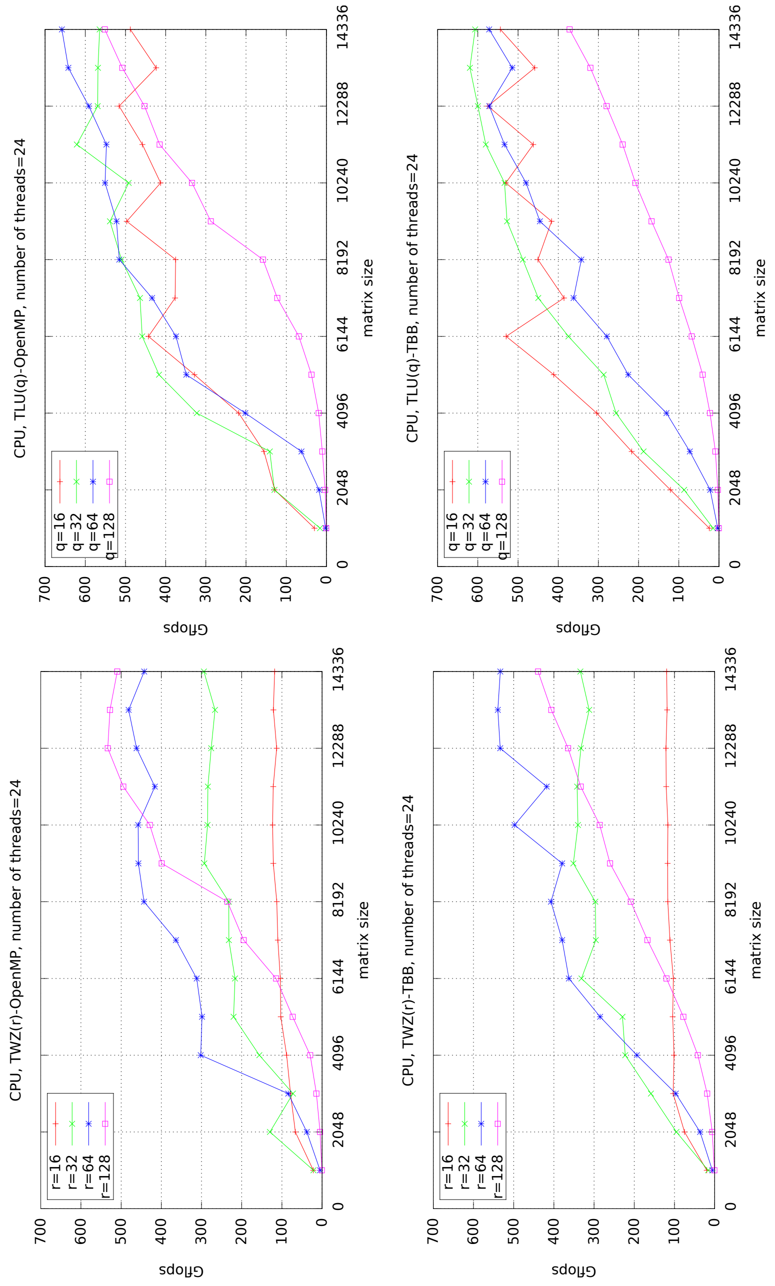

Figure 5. The performance in Gflops of the parallel tiled LU and WZ factorization algorithms for the number of threads 24 for four different number of tiles (16, 32, 64, 128) as a function of the matrix size

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

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

Figure 6 shows the experimental speedup (relative to the same algorithm run with the use of one thread – the times are shown in Table 3) as a function of the number of threads (1–27 threads) for different values of r and q for a matrix of the size 14 336.

Considering the best choices of q and r, the OpenMP implementations give a significantly better speedup (for 24 threads, it is more than 20). The best what the TBB implementations gain is speedup of 20. However, Table 3 shows that the TBB implementations have better performance for one thread, and thus they achieve poorer results in terms of relative speedup (Section 5.1 shows similar performance for OpenMP and TBB implementations of the same algorithm).

Implementation	Time [s]		
	1 Thread	12 Threads	24 Threads
TLU(64)-OpenMP	68.93	5.84	2.98
TLU(32)-TBB	62.99	5.48	3.23
TWZ(128)-OpenMP	75.05	6.98	3.85
TWZ(64)-TBB	70.02	6.25	3.65

Table 3. The performance time for selected numbers of threads and the matrix size of 14 336

We can see that the OpenMP implementations scale better – up to 24 threads. For more threads, the hyperthreading turns on and it does not improve the performance – aggravating the results sometimes. In the case of TBB, the scalability collapses somewhat earlier. For both frameworks, we should choose the maximum number of physical cores as the number of threads, that is, 24 in our environment.

The speedup is sensitive to the values r and q. However, both implementations are scalable (up to the number of physical cores, that is 24) for well-chosen q and r.

5.3 Scalability

Figure 7 shows the weak scalability of the algorithms. The tests here are run for various numbers of processors, however, the amount of the work is chosen to be proportional to the number of employed cores. To achieve a nice weak scalability [10], we expect the plots (of the execution time versus the number of processors employed) to be horizontal. We can see that both frameworks (that is OpenMP and TBB) and both methods (LU and WZ) achieve similar, very good, weak scalability.

Figure 8 shows the strong scalability of the algorithms. This time, the tests were run for various numbers of processors, but the amount of the work was always the same (the size of the matrix was 14 336). The plot (of the execution time versus the number of processors employed) was done in log-log scales [10]. In such a plot,

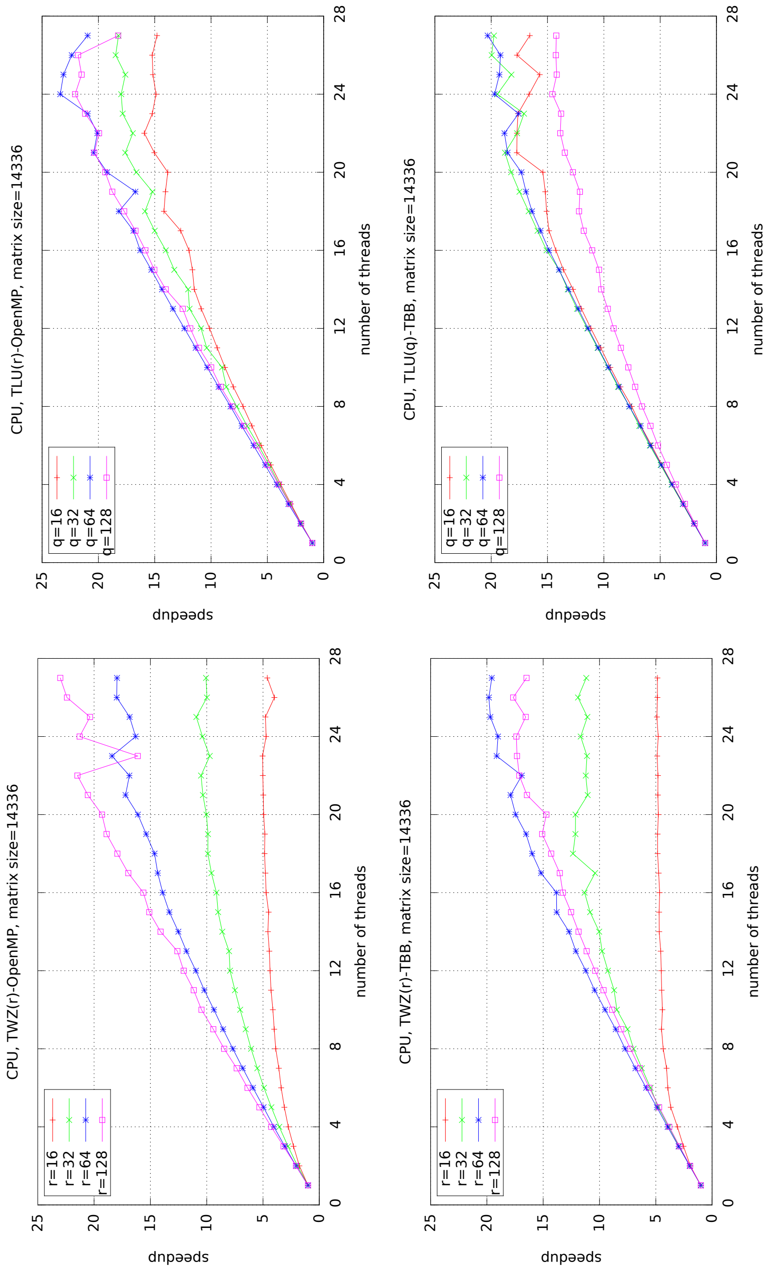

Figure 6. The speedup of the parallel tiled LU and WZ factorization algorithms as a function of the number of threads for different values of r and q for matrix size equals 14 336

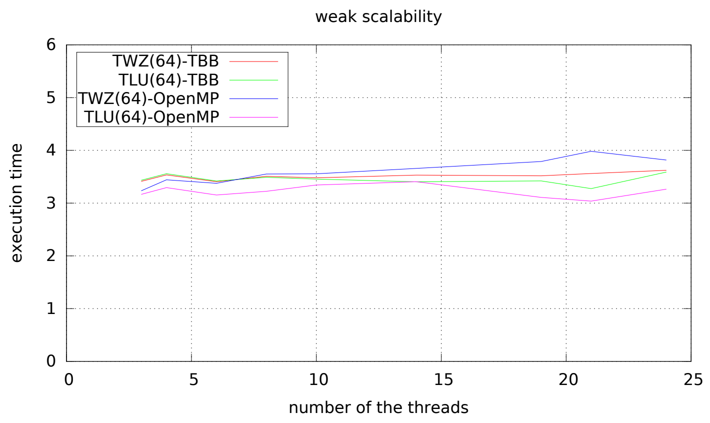

Figure 7. Weak scalability of the tested algorithms for the selected parameters

a good strong scalability should give a straight line with the slope −1. We can see that the scalability is not bad for all cases. However, close to the maximal number of processors, something spoils (what can be also seen in Figure 6). It can be explained by an automatic constraint on the energy used by the processor.

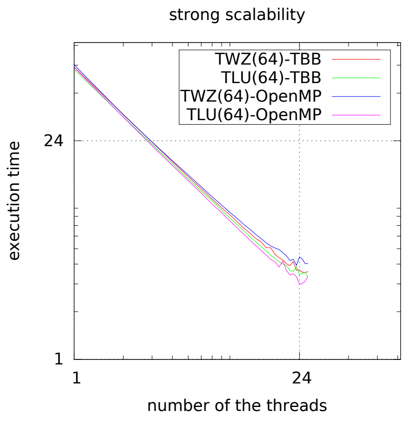

Figure 8. Strong scalability of the tested algorithms for the selected parameters

6 CONCLUSION

In this work, we reported numerical experiments aimed to compare two parallel task frameworks, namely OpenMP and TBB for two matrix factorizations. We focused on two matrix factorizations which use BLAS functions from MKL library and are computationally intensive. We implemented these matrix factorizations using tasks with dependencies in OpenMP and TBB. We chose these frameworks because they differ significantly in their approach to defining dags and dependencies. Moreover, OpenMP is a language extension, but TBB is an ordinary library.

The TBB library seems to be more flexible – as it is just a library seamlessly fitting into the C++ language and its other libraries. Arbitrary C++ types can be used with TBB, whereas OpenMP have troubles with some more complicated types (like classes, templates, lambdas) and they cannot be used directly. High-level abstraction (provided by these types) does not port well into OpenMP. Conversely, in TBB they are treated quite transparently, because TBB is not an overlay onto the language – as OpenMP is. As a library, TBB synergizes with C++ standard library as well as with external libraries. We can also easily use templates and lambdas which facilitate the creation of flexible and reusable code. On the other hand, in OpenMP, pragmas do not accept many C++ constructs (sometimes not even macros) and we must employ some tricks to achieve our goals.

The TBB library also offers more tools to better control the execution. With the use of OpenMP, we are not building the graph – this is done by the compiler and run-time system. We can only give the dependencies and trust that the graph will be correct. However, sometimes (especially when the graph is explicit – as in our case) it is easier to build the graph than to write complex (and sometimes artificial) dependencies. Moreover, the dependency graph built by a programmer in TBB can be an arbitrary graph (contrary to OpenMP, where graphs are not arbitrary and they must be given implicitly, by dependencies – as we mentioned above). Each graph node represents a task and its edges describe arbitrary dependencies between them. The task scheduling is a very important part of TBB. It automatically allocates tasks to workers (threads) to maintain the best load balancing. But the main advantage of the TBB is that it is completely compatible with the C++ language and can be freely used with other libraries – which is priceless in advanced applications.

On the other hand, OpenMP is very popular and quite simple to use. It is also a portable and (de facto) standard approach. However, OpenMP has some limitations. It causes problems when dependencies are more complicated, does not allow using some C/C++ constructs (even some simple expressions or data members) in pragmas and clauses, forcing a programmer to use unnatural notations (as illegible casts). Specifically to dependencies, if array sections appear in them, they must be either the same or disjoint. It is also a C-based standard so it does not treat well some C++ elements (like references). Thus, OpenMP is a common and quite efficient tool, although TBB is more programmer-friendly and sometimes TBB's features and flexibility make TBB the only option.

There was not a clear performance relation among the considered frameworks, and the differences between them were small in most cases. In fact, the average performance difference between the slowest and the fastest implementation in our tests was about 19 %. However, the OpenMP implementations relative speedup is clearly better than that for TBB.

There is also a programmer's experience issue. For simpler projects, OpenMP seems easier – especially when we have a working sequential implementation. On the other hand, for more complicated problems, notably ones needing some code reuse (as, for example, refinement techniques [2, 4] where the same algorithm is used with different precision types), TBB is better for an experienced developer, although, TBB is also more demanding. However, the merits of TBB (its flexibility, generality, code readability) prevails over the OpenMP (its limitations and error proneness).

Our corollaries can be generalized to a wide class of algorithms. Namely, all the linear algebra algorithms – as, for example, various factorizations (Cholesky, QR, etc.), matrix-matrix multiplication (GEMM) and iterative methods (Jacobi, Gauss-Seidel, GMRES) – which can be designed as a tiled version (that is, with the use of square blocks and the special storing format mentioned in Section 5) can be also implemented with tasks (using both OpenMP and TBB) similarly.

Further work is needed to determine other ways in which OpenMP and TBB frameworks could potentially be improved and whether additional information could be provided to enable better performance. Also, there is a plenty other computing areas where we can use the task approach – as, for example, in sparse computations, machine learning, etc.

REFERENCES

[1] Agullo, E.—Demmel, J.—Dongarra, J.—Hadri, B.—Kurzak, J.—Langou, J.—Ltaief, H.—Luszczek, P.—Tomov, S.: Numerical Linear Algebra on Emerging Architectures: The PLASMA and MAGMA Projects. Journal of Physics: Conference Series, Vol. 180, 2009, No. 1, Art. No. 012037, doi: 10.1088/1742-6596/180/1/012037.

[2] Baboulin, M.—Buttari, A.—Dongarra, J.—Kurzak, J.—Langou, J.—Langou, J.—Luszczek, P.—Tomov, S.: Accelerating Scientific Computations with Mixed Precision Algorithms. Computer Physics Communications, Vol. 180, 2009, No. 12, pp. 2526–2533, doi: 10.1016/j.cpc.2008.11.005.

[3] Buttari, A.—Langou, J.—Kurzak, J.—Dongarra, J.: A Class of Parallel Tiled Linear Algebra Algorithms for Multicore Architectures. Parallel Computing, Vol. 35, 2009, No. 1, pp. 38–53, doi: 10.1016/j.parco.2008.10.002.

[4] Bylina, B.—Bylina, J.: Mixed Precision Iterative Refinement Techniques for the WZ Factorization. Proceedings of the 2013 Federated Conference on Computer Science and Information Systems, 2013, pp. 425–431.

[5] Bylina, B.—Bylina, J.: OpenMP Thread Affinity for Matrix Factorization on Multicore Systems. Proceedings of the 2017 Federated Conference on Computer Science and Information Systems (FedCSIS), Annals of Computer Science and Information Systems, Vol. 11, 2017, pp. 489–492, doi: 10.15439/2017F231.

[6] Chandra, R.—Dagum, L.—Kohr, D.—Maydan, D.—McDonald, J.—Menon, R.: Parallel Programming in OpenMP. Morgan Kaufmann Publishers, San Francisco, 2001, doi: 10.1016/b978-155860671-5/50003-7.

[7] Dongarra, J. J.—Du Croz, J.—Hammarling, S.—Duff, I. S.: A Set of Level-3 Basic Linear Algebra Subprograms. ACM Transactions on Mathematical Software, Vol. 16, 1990, pp. 1–17, doi: 10.1145/77626.79170.

[8] Evans, D. J.—Hatzopoulos, M.: A Parallel Linear System Solver. International Journal of Computer Mathematics, Vol. 7, 1979, No. 3, pp. 227–238, doi: 10.1080/00207167908803174.

[9] Gustavson, F. G.: High-Performance Linear Algebra Algorithms Using New Generalized Data Structures for Matrices. IBM Journal of Research and Development, Vol. 47, 2003, No. 1, pp. 31–55, doi: 10.1147/rd.471.0031.

[10] Heath, M. T.: A Tale of Two Laws. The International Journal of High Performance Computing Applications, Vol. 29, 2015, No. 3, pp. 320–330, doi: 10.1177/1094342015572031.

[11] Igual, F. D.—Chan, E.—Quintana-Ortí, E. S.—Quintana-Ortí, G.—van de Geijn, R. A.—Van Zee, F. G.: The FLAME Approach: From Dense Linear Algebra Algorithms to High-Performance Multi-Accelerator Implementations. Journal of Parallel and Distributed Computing, Vol. 72, 2012, No. 9, pp. 1134–1143, doi: 10.1016/j.jpdc.2011.10.014.

[12] Kukanov, A.—Voss, M. J.: The Foundations for Scalable Multi-Core Software in Intel Threading Building Blocks. Intel Technology Journal, Vol. 11, 2007, No. 4, pp. 309–322, doi: 10.1535/itj.1104.05.

[13] Kurzak, J.—Luszczek, P.—YarKhan, A.—Faverge, M.—Langou, J.—Bouwmeester, H.—Dongarra, J.: Multithreading in the PLASMA Library. In: Rajasekaran, S., Fiondella, L., Ahmed, M., Ammar, R. A. (Eds.): Multicore Computing: Algorithms, Architectures, and Applications. Chapter 5. Chapman and Hall/CRC, 2013, p. 119–142, doi: 10.1201/b16293-11.

[14] Marowka, A.: TBBench: A Micro-Benchmark Suite for Intel Threading Building Blocks. Journal of Information Processing Systems, Vol. 8, 2012, No. 2, pp. 331–346, doi: 10.3745/jips.2012.8.2.331.

[15] Marqués, M.—Quintana-Ortí, G.—Quintana-Ortí, E. S.—van de Geijn, R. A.: Using Desktop Computers to Solve Large-Scale Dense Linear Algebra Problems. The Journal of Supercomputing, Vol. 58, 2011, No. 2, pp. 145–150, doi: 10.1007/s11227-010-0394-2.

[16] Olivier, S. L.—Prins, J. F.: Comparison of OpenMP 3.0 and Other Task Parallel Frameworks on Unbalanced Task Graphs. International Journal of Parallel Programming, Vol. 38, 2010, No. 5-6, pp. 341–36, doi: 10.1007/s10766-010-0140-7.

[17] Pheatt, C.: Intel Threading Building Blocks. Journal of Computing Sciences in Colleges, Vol. 23, 2008, No. 4, pp. 298–298.

[18] Quintana-Ortí, G.—Quintana-Ortí, E. S.—van de Geijn, R. A.—Van Zee, F. G.—Chan, E.: Programming Matrix Algorithms-by-Blocks for Thread-Level Parallelism. ACM Transactions on Mathematical Software, Vol. 36, 2009, No. 3, pp. 14:1–14:26, doi: 10.1145/1527286.1527288.

[19] Rao, S. C. S.: Existence and Uniqueness of WZ Factorization. Parallel Computing, Vol. 23, 1997, No. 8, pp. 1129–1139, doi: 10.1016/s0167-8191(97)00042-2.

[20] Stpiczyński, P.: Language-Based Vectorization and Parallelisation Using Intrinsics, OpenMP, TBB and Cilk Plus. Journal of Supercomputing, Vol. 74, 2018, pp. 1461–1472, doi: 10.1007/s11227-017-2231-3.

[21] Tang, P.: Measuring the Overhead of Intel C++ Concurrent Collections over Threading Building Blocks for Gauss–Jordan Elimination. Concurrency and Computation: Practice and Experience, Vol. 24, 2012, pp. 2282–2301, doi: 10.1002/cpe.2811.

[22] Virouleau, P.—Brunet, P.—Broquedis, F.—Furmento, N.—Thibault, S.—Aumage, O.—Gautier, T.: Evaluation of OpenMP Dependent Tasks with the KASTORS Benchmark Suite. In: DeRose, L., de Supinski, B. R., Olivier, S. L., Chapman, B. M., Müller, M. S. (Eds.): Using and Improving OpenMP for Devices, Tasks, and More (IWOMP 2014). Springer, Heidelberg, Lecture Notes in Computer Science, Vol. 8766, 2014, pp. 16–29, doi: 10.1007/978-3-319-11454-5_2.

[23] Yalamov, P.—Evans, D. J.: The WZ Matrix Factorization Method. Parallel Computing, Vol. 21, 1995, No. 7, pp. 1111–1120, doi: 10.1016/0167-8191(94)00088-r.

[24] `https://www.cilkplus.org/`.

[25] `https://www.khronos.org/opencl/`.

Jarosław Bylina is a mathematics graduate (1998) with his Ph.D. in computer science (Distributed methods to solve Markovian models of computer networks, done at The Silesian University of Technology in Gliwice, Poland in 2006. He has been working in the Institute of Mathemtaics of Marie Curie-Skłodowska University in Lublin (Poland) since 1998, now as Assistant Professor. He is interested in numerical methods for Markov chains, modelling of teleinformatic systems and parallel and distributed computing and processing.