BULLETIN OF THE POLISH ACADEMY OF SCIENCES
TECHNICAL SCIENCES, Vol. 66, No. 6, 2018
DOI: 10.24425/bpas.2018.125800

Studying OpenMP thread mapping for parallel linear algebra kernels on multicore system

B. BYLINA* and J. BYLINA

Marie Curie-Skłodowska University, Institute of Mathematics, Pl. M. Curie-Skłodowskiej 5, 20-031 Lublin, Poland

Abstract. Thread mapping is one of the techniques which allow for efficient exploiting of the potential of modern multicore architectures. The aim of this paper is to study the impact of thread mapping on the computing performance, the scalability, and the energy consumption for parallel dense linear algebra kernels on hierarchical shared memory multicore systems. We consider the basic application, namely a matrix-matrix product (GEMM), and two parallel matrix decompositions (LU and WZ). Both factorizations exploit parallel BLAS (basic linear algebra subprograms) operations, among others GEMM. We compare differences between various thread mapping strategies for these applications. Our results show that the choice of thread mapping has the measurable impact on the performance, the scalability, and energy consumption of the GEMM and two matrix factorizations.

Key words: computation performance, OpenMP standard, nonnegative matrix factorization, thread mapping, energy consumption.

1. Introduction

The advance of the shared memory multicore and manycore architectures caused a rapid development of one type of parallelism, namely the thread level parallelism. This kind of parallelism relies on splitting the program into subprograms which can be executed concurrently. Each of such subprograms is performed by one or more software threads.

A software thread is a running sequential part of the program and there are also hardware threads. A hardware thread is an independent physical processing unit – as seen by the operating system. Such a hardware thread can execute one sequential software thread at any particular moment. Operating systems on the shared memory multicore and manycore architectures run numerous software threads and these threads share a complex hierarchical memory. Since the architecture consists of many processing units, these software threads have to be assigned to appropriate processing units (that is, hardware threads). Such an assignment is called thread mapping [8]. Using inadequate thread mapping strategies can produce a bad utilization of the computing power and the memory hierarchy. The adequate ones should be used to efficiently exploit the potential of modern multiprocessors.

The thread mapping can also improve the energy efficiency of parallel applications by reducing the execution time. Energy consumption will be reduced proportionally since the processor is in a high power-consumption state for less time. Energy efficiency is being recently considered as important as raw performance and has become a critical aspect of the development of scalable systems.

Determining the efficiency of the thread mapping depends on the machine and the application. There is not a single thread mapping strategy that suits all the applications. In this work, we are going to try to state rules which guide us to determine efficient thread mapping to improve the performance, the scalability, power and energy consumption of parallel numerical linear algebra applications on shared memory multithreaded machines with hierarchical memory.

Efficient parallel numerical algorithms and their implementations on different contemporary parallel machines are crucial for engineering applications and computational science. One of the important problems of numerical linear algebra is the matrix factorization which is the matrix decomposition into factor matrices of a simpler structure or of some specific properties. The most known factorizations are the LU, QR, and Cholesky factorizations. In this work, we study the LU factorization, and another less known form of the factorization, namely the WZ factorization. We assume that the factorized matrix is dense, non-singular, square. For both factorizations (LU and WZ), we consider block versions which use a standard set of basic linear algebra subprograms (BLAS) [9]. BLAS collects all the vector-matrix operations.

The basic matrix operation, namely the matrix multiplication (GEMM) from BLAS library was analyzed. Furthermore, we considered GEMM which was implemented in Intel MKL library (Math Kernel Library [16]). and discussed a multithreaded version of the block LU factorization which is available in the Intel MKL library too. Another factorization which we considered was the WZ factorization which uses level 3 BLAS routines. The WZ factorization was introduced in [11]. It was a novel method for solving linear systems in parallel, for SIMD (*single instruction multiple data*, [12]) computers. A tiled WZ factorization (here, 'tiled' means a block version [3] with all the blocks being square matrices of the same size) was implemented by the authors with the use of multithreaded

*e-mail: beata.bylina@umcs.pl

Manuscript submitted 2017-11-28, revised 2018-04-21, initially accepted for publication 2018-06-04, published in December 2018.

BLAS operations and the OpenMP standard on multicore architectures.

We studied the OpenMP thread mapping strategies for matrix decompositions on multicore architectures in our work [6]. The results show that the choice of thread affinity has the measurable impact on the executed time of the matrix factorizations. Here, we extend this investigation into analyzing the performance, the scalability and the energy efficiency. We also add power profiles based on Intel's RAPL (Intel Running Average Power Limit) [13] technology that allows measuring power and energy seamlessly by using hardware counter technology available on multicore processors. We not only investigate the matrix decompositions but also add the research GEMM operation, which is a component of the matrix decompositions.

The rest of this paper is organized as follows. In Section 2, we present different thread mapping strategies. Section 3 reviews the matrix decompositions, namely LU and WZ. Section 4 shows the results of numerical experiments carried out on shared memory multicore architectures and evaluates different thread affinities for all GEMM and matrix factorizations. Finally, Section 5 concludes our research and presents the future plans.

2. Thread mapping strategies

Most operating systems have smart algorithms to allocate software threads on multicore architectures. They model the structure of the underlying hardware in order to describe the distance between units. Computer machines have the hierarchical structure in which hardware threads share a common core including functional units and first level cache. Logically, a computer system can be treated like a tree of packages, cores, and hardware threads. Each package contains one or more cores, and each core contains one or more hardware threads. Figure 1 presents the topology of the hardware used in our experiments. In our studies, we have machines consisting of 2 packages each; each package consists of 12 cores and each core has 2 hardware threads.

In order to avoid the operating system moving threads around, it is possible to bind a thread. The goal of the thread mapping is to bind software threads to the hardware threads in such a way that memory accesses to data shared between software threads are optimized and all the cores are equally loaded. Thread mapping aims to improve the performance and energy efficiency. Thus, it is important to choose an appropriate thread mapping strategy. Some thread mappings can be applied directly through options of the runtime environment without modifying the parallel application. There are various utilities to control the thread mapping on a modern Linux system. The OpenMP 4.0 or 4.5 [17] specification provides the vendor-neutral `OMP_PLACES` and `OMP_PROC_BIND` environment variables which specify how the software threads in a program are bound to hardware threads. These two environment variables are often used in conjunction with each other. `OMP_PLACES` is used to specify the places in the multicore architecture to which the threads are mapped (for example cores, threads). Cores denote that the software thread can migrate between cores. Threads denote that the software thread cannot migrate between cores. `OMP_PROC_BIND` is used to specify the mapping policy which determines how the threads are bound to places (for example `spread`, `close`). Spread mapping denotes that the OpenMP runtime distributes the threads as evenly within the places in the system, as possible. Close mapping denotes that the OpenMP runtime packs the threads in the same place, as closely to one another.

We consider several examples of binding software threads to hardware threads. Let the number of the software threads be equal to 24. Let Ti denote ith hardware thread (where $i = 0, \ldots, 23$), STi denote ith software thread.

Example 1. For
`OMP_PLACES = threads OMP_PROC_BIND = close`
we have the following binding:

- ST0 is bound to T0,
- ST1 is bound to T24,
- ST2 is bound to T1,
- ST3 is bound to T25,

  ⋮
- ST22 is bound to T11,
- ST23 is bound to T35.

It means that each core has exactly two software threads, P1 package is idle, the hyperthreading is used, and there is no thread migration.

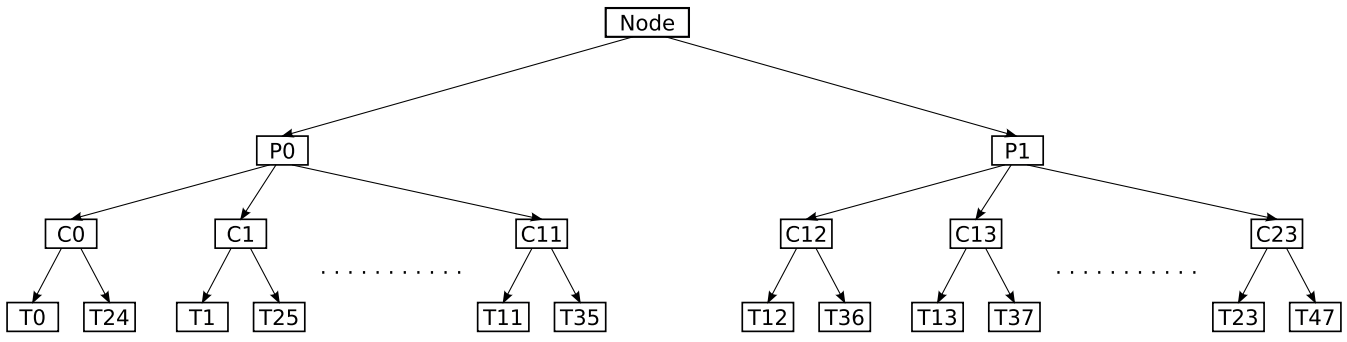

Fig. 1. The topology of the hardware used in the experiments (P – package, C – core, T – thread)

Example 2. For
`OMP_PLACES = threads OMP_PROC_BIND = spread`
we have the following binding:

- ST0 is bound to T0,
- ST1 is bound to T12,
- ST2 is bound to T1,
- ST3 is bound to T13,
- ⋮
- ST22 is bound to T11
- ST23 is bound to T23.

It means that both packages are loaded evenly. Each core has exactly one software thread, the hyperthreading was not used, and there is no thread migration.

Some vendors recommend setting the thread mapping on the OpenMP threads to associate them with a particular processing units. The Intel OpenMP runtime library has the ability to bind OpenMP threads to physical processing units. The `KMP_AFFINITY` environment variable in the Intel compilers allows adjusting software threads to hardware threads. This environment variable has got the following form:

`KMP_AFFINITY=[verbose,]granularity=`*level*, *type*

We can set the *level* of the granularity to `core` or `thread`. Software threads can migrate between processing units in accordance with the operating system preferences. Averting the migration inhibits transferring temporary data between cores and caches. Forcing the static mapping of the threads of a particular program can give a performance growth. If the granularity is set to `core`, the threads can migrate – although only within one core. To forbid threads to migrate at all, the granularity should be set to `thread`.

The optional `verbose` modifier in the `KMP_AFFINITY` variable leads to printing the mapping information. This information regards the machine topology e.g. the number of packages, the number of cores in each package, the number of hardware threads for each core and the actual software threads pinned to hardware threads (for the `thread` granularity) or sets of the hardware and software threads (for the `core` granularity). This modifier `verbose` allows checking the thread mapping in our examples.

The `KMP_AFFINITY` environment variable for CPU can have the following main values for *type*: `compact`, `scatter`, and `none`. Threads remain unbound for the `none` value. However, in this case, the operating system assigns threads according to its own algorithm. If `KMP_AFFINITY` is set to `compact`, it means that all threads are put close together. The new thread is first allocated to one core until the core reaches its maximum load. For the `scatter` value, all threads are put far apart. `Scatter` spreads threads evenly across the system. The new thread is firstly allocated to the package that has the lightest load. `Scatter` is the opposite of `compact`.

We consider several examples of binding software threads to hardware threads. Let the number of the software threads be equal 24. Let T*i* denote *i*th hardware thread (where $i = 0, \ldots, 23$) and ST*i* denote *i*th software thread.

Example 3. For
`KMP_AFFINITY=granularity=thread, compact`
we have the following binding:

- ST0 is bound to T0,
- ST1 is bound to T24,
- ST2 is bound to T1,
- ST3 is bound to T25,
- ⋮
- ST22 is bound to T11,
- ST23 is bound to T35.

It means that the P1 package is idle. Each core has exactly two software threads, the hyperthreading was used, and there is lack of the thread migration. This is like Example 1.

Example 4. For
`KMP_AFFINITY=granularity=thread, scatter`
we have the following binding:

- ST0 is bound to T0,
- ST1 is bound to T12,
- ST2 is bound to T1,
- ST3 is bound to T13,
- ⋮
- ST22 is bound to T23,
- ST23 is bound to T11.

It means that both packages are loaded evenly. Each core has exactly one software thread, the hyperthreading was not used, and there is lack of the thread migration. This is like Example 2.

3. Matrix decomposition

The decomposition of a matrix or the factorization of a matrix is used to solve an $n \times n$ system of linear equations [1, 11], to find the inverse of the matrix, to compute the determinant of the matrix or as the preconditioning for iterative methods [4, 5]. The LU decomposition factorizes a matrix into two matrices, namely a lower triangular matrix **L** and an upper triangular matrix **U**. For improving computing performance, a block version of the LU decomposition is applied in high performance computing. The block LU decomposition is a matrix decomposition of a block matrix into a lower block triangular matrix **L** and an upper block triangular matrix **U**. The block version of the LU decomposition was implemented in LAPACK (Linear Algebra PACkage) [1] That implementation is based on BLAS. The parallelism of that block version of the LU factorization arises from the use of a multithreaded BLAS. The MKL library provides exactly such an implementation of BLAS and such a parallel version of the block LU decomposition.

In the latter [2] the concept of tiled algorithms is reminded and a class of parallel tiled linear algebra algorithms (LU, QR and Cholesky factorization) for multicore architectures is presented. In our work, we investigate the tiled WZ decomposition. The WZ factorization is described in [7, 11, 15]. Let's assume that **A** is a square nonsingular matrix of the size $n \times n$ (we consider only even n, for simplicity's sake). We are to find matrices

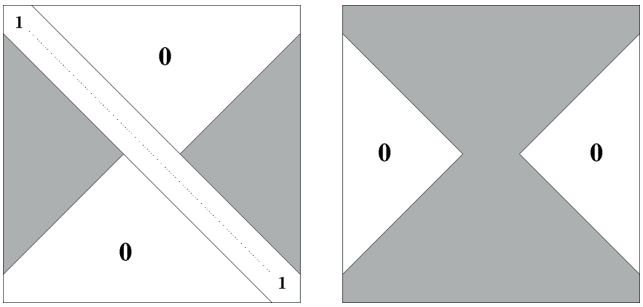

Fig. 2. The form of the result matrices in the WZ factorization

W and **Z** that fulfill **WZ** = **A** where the matrices **W** and **Z** have the structure shown in Fig. 2 – here, gray fields are non-zeros. The main diagonal of the matrix **W** consists only of ones. The second diagonal consists of zeros.

These 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 **Z** has non-zeros where the matrix **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 **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 $n-1$st row. Next, we update the inner submatrix **A** of the size $(n-2)\times(n-2)$ and for $k = 2, \ldots, n/2$ we zero elements in the kth and $(n-k+1)$st columns – from the $(k+1)$st row to the $(n-k)$th row.

In order to achieve high efficiency of the WZ factorization algorithm on a shared memory multicore system, we use a tiled algorithm. The tiled WZ factorization algorithm performs the majority of its floating-point operations (flop) using the level 3 BLAS operations, which use the memory hierarchy. That hierarchy is utilized very efficiently, and thus, the modern BLAS implementations achieve almost the peak performance of the processor. Let's assume that the **A** is a square nonsingular matrix of an even size n and it is partitioned into $r\times r$ (r is also even) parts (r of each side – rows and columns). The tiled WZ algorithm consists of repeating four stages $r/2$ times.

Stage 1 consists in the WZ factorization of a matrix built from four corner blocks of the input matrix. Stage 2 computes $2s$ (where $s = n/r$) columns of the matrix **W** – s right columns and s left columns. Stage 3 computes $2s$ rows of the matrix **Z** – s bottom rows and s top rows. Stage 4 updates the inner submatrix **A** – that is, **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.

- `WZ(B,W,Z)`. This subroutine performs a sequential WZ factorization for matrix **B**.
- `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 **X** and **B** are $s\times s$ matrices, **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 **A**, **B**, and **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 **A**, **B**, **C**, and **D** are $s\times s$ matrices.

Algorithm 1 presents the tiled WZ factorization algorithm expressed by the above-mentioned operations (`WZ`, `DTRSM`, `DGEMM`, `DGEMM_copy`) for a nonsingular matrix **A** partitioned into $r\times r$ blocks. The matrices **W** and **Z** are the results of this algorithm.

Algorithm 1 Tiled WZ factorization algorithm for even r based on four elementary operations

Require: **A**, r
Ensure: **W**, **Z**
for $k \leftarrow 1, r/2-1$ **do**
  $k_2 \leftarrow r-k+1$
  The WZ for the corner blocks of **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`(**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 kth and k_2nd columns of **W** *— STAGE 2*
  `DTRSM(nonu, up, l,` $\mathbf{Z}_{kk}$`,` **D**`,` $\mathbf{Z}_{kk_2}$`)`
  `DGEMM_copy(E,` $\mathbf{Z}_{k_2k}$`,` **D**`,` $\mathbf{Z}_{k_2k_2}$`)`
  for $i \leftarrow k+1, k_2-1$ **do**
    `DGEMM(`$\mathbf{A}_{ik_2}$`,` $\mathbf{A}_{ik}$`,` **D**`);`
    `DTRSM(nonu, lo, r, 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 kth and k_2nd rows of **Z** *— STAGE 3*
  `DTRSM(u, lo, r,` $\mathbf{W}_{kk}$`,` **D**`,` $\mathbf{W}_{k_2k}$`)`
  `DGEMM_copy(E,` **D**`,`$\mathbf{W}_{kk_2}$`,`$\mathbf{W}_{k_2k_2}$`)`
  for $i \leftarrow k+1, k_2-1$ **do**
    `DGEMM(`$\mathbf{A}_{k_2i}$`,` **D**`,` $\mathbf{A}_{ki}$`);`
    `DTRSM(u, up, l, 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 **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

4. Numerical experiments

4.1. Environment. In this section, we compare the different thread mapping strategies on a shared hierarchical memory multicore architecture. We present the performance, the speedup, power and energy consumption. These evaluations were conducted by the execution of parallel versions of the following applications:

- a multithreaded implementation of GEMM routines from the MKL library which computes a matrix multiplication. This operation is denoted by GEMM.
- a multithreaded implementation of the `dgetrf` routine from the MKL library, which computes the complete LU factorization of a general matrix with pivoting. In our case, the matrices are square of the size $n \times n$. In the implementation of the `dgetrf` routine the panel factorization (factorization of a block of columns) is used, as well as the level 3 BLAS routines (`DTRSM` and `DGEMM`). This LU factorization is denoted by LU.
- a parallel tiled WZ factorization with the use of level 3 BLAS routines (`DTRSM` and `DGEMM`) and the OpenMP standard (denoted by TWZ(r)-OpenMP). OpenMP is used to parallelize loops (lines: 9, 18 and 25 in Algorithm 1) with `dynamic` scheduler.

Table1 shows details of the specification of the hardware and software used in the numerical experiment.

Table 1
Hardware and software used in the experiments

CPU	**Intel ®Xeon E5–2670 v.3 (Haswell)**
# cores	2 sockets
# threads	48 threads
Clock speed	2.30 GHz
Level 1 instruction cache	32kB per core
Level 1 data cache	32kB 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, LAPACK	MKL 2016.0.109

All applications were compiled with `icc` using the following options: `-xHost`, `-mkl`, `-openmp`, `-O3`. Here, the `-xHost` option generates instructions for the highest instruction set and the processor available on the compilation host machine. The `-mkl` and `-openmp` options link the program with two libraries (MKL and OpenMP). The last one, `-O3`, orders the compiler to optimize the code automatically with the use of vectorization and parallelization (among others). The MKL library already provides vectorized implementations of BLAS routines.

All floating point calculations were performed with the double precision. The input matrices were generated by the authors. They were random (all off-diagonal elements were generated with the use of a uniform distribution) matrices with a dominant diagonal (diagonal elements were increased above the sum of all the other elements in the same row – to ensure the existence of the factorization without pivoting). The matrices' sizes are chosen as powers of two (and their multiples, namely: 128, 256, 512, $1024 \times \{1, \ldots, 14\}$). Because using the power of two can be a bad idea for cache layout we tested additionally codes for other matrix sizes (namely as a multiple of 1000 $\{1000, 2000, \ldots, 15\,000\}$) only for GEMM and LU factorization from MKL library. The number of tiles equals $r = 128$ was tested for WZ factorization (a different number of tiles was tested in [6]).

The times were measured with the use of a standard function, namely `omp_get_wtime` from OpenMP Standard. We set the number of the OpenMP threads using the `OMP_NUM_THREADS` environment variable. The MKL uses OpenMP and thus, this variable enables a parallel execution with the given number of threads.

Another environment variables used in the tests are `OMP_PLACES` and `OMP_PROC_BIND`, which are set to one of the three values:

- `OMP_PLACES = threads`, `OMP_PROC_BIND = close` denoted by `close` (see Example 1)
- `OMP_PLACES = threads`, `OMP_PROC_BIND = spread` denoted by `spread` (see Example 2).

Another environment variable used in the tests is `KMP_AFFINITY` delivery Intel, which is set to one of the three settings:

\begin{itemize}

- `KMP_AFFINITY = granularity = thread`, `compact` denoted by `compact` (see Example 3)
- `KMP_AFFINITY = granularity = thread`, `scatter` denoted denoted by `scatter` (see Example 4)
- `KMP_AFFINITY = none` denoted by `none`.

To better control assigning software threads to hardware threads we chose the granularity as `thread`. Such a setting enables treating each hardware thread separately. Thus, each software thread is mapped to one hardware thread. If we used the `core` granularity, one core would be treated as a unit containing two software threads (because it consists of two hardware threads through hyperthreading) and can cause thread migration.

The MKL procedures are available in the form of precompiled DLLs. However, they use the OpenMP standard internally, and thus, the user can set the above-mentioned environment variables and these variables do influence the MKL behavior (they are not compile-time settings but the run-time ones). The MKL is built to allow such a tuning and the affinity mapping can be adjusted by a user during launching the application.

We begin by introducing our energy consumption measurement methodology along with the metrics used to analyze the results on the multicore system.

4.2. Performance. In our experiments, we use the number of floating-point operations per second (flops) as a metric. The number of floating point operations for GEMM equals n^3. Both the LU factorization and the WZ factorization have the number of floating point operations equal $\frac{2}{3}n^3 + O(n^2)$, so this numer approximately equals $\frac{2}{3}n^3$.

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

$$\frac{n^3}{T \cdot 10^9}, \text{ for GEMM}$$

$$\frac{2n^3}{3 \cdot T \cdot 10^9}, \text{ for both factorization matrix}$$

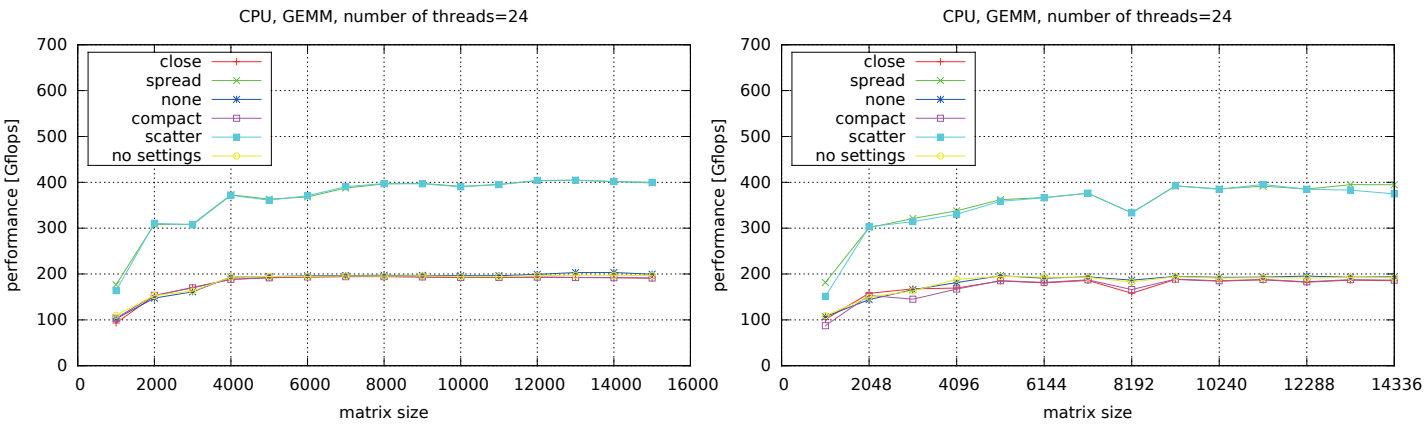

Fig. 3. The performance of GEMM from the MKL library

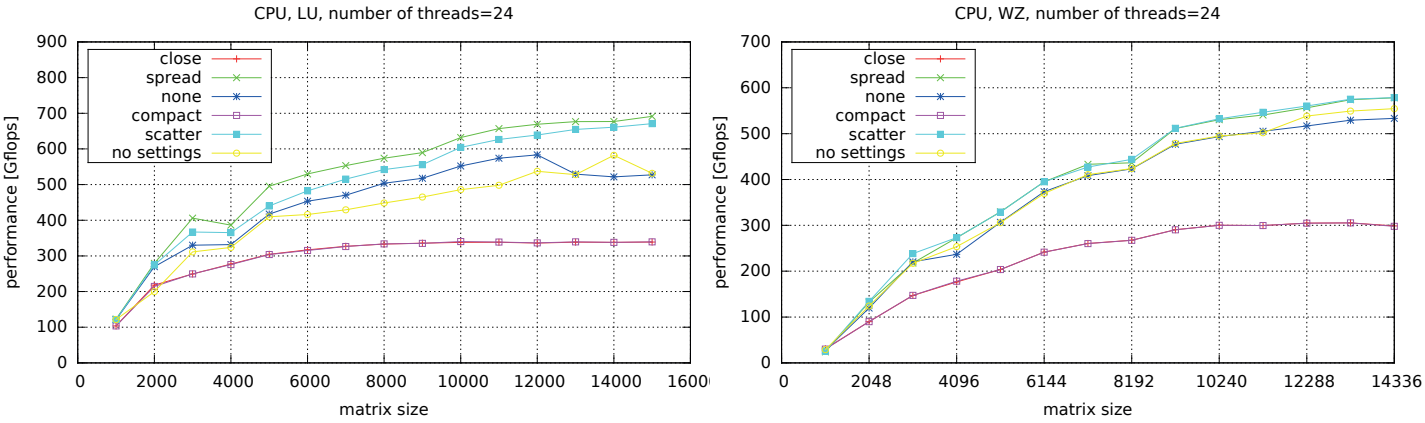

Fig. 4. The performance of LU from the MKL library

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

Figures 3, 4 and 5 present the performance (in Gflops) of the GEMM, LU, and WZ for the fixed number of threads as a function of the matrix size depending on the thread mapping.

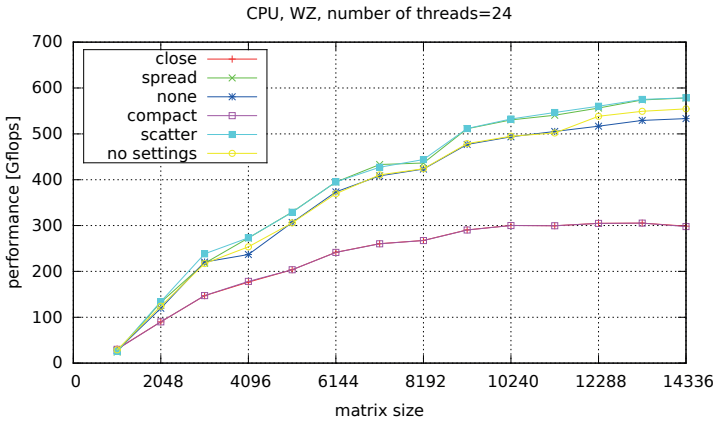

Fig. 5. The performance of authors' WZ implementation

The thread mapping had an important impact on the performance of all the tested applications. All three applications are the most efficient for `scatter` or `spread`, and the least efficient for `compact` and `close` – and they do not depend on the matrix size. Such results were connected with the fact the multicore machine was evenly loaded for `scatter` or `spread` and one package was idle for `compact` and `close`. The same results were obtained for the matrix size being powers of two (or the multiple of powers of two) as and for the multiple of 1000. The decrease in the performance was observed only for 4096 and 8192 matrix size and it seemed to be connected with the cache size. In the latter part of the article, we will only investigate the matrix size being the multiple of the power of two.

4.3. Speedup. Figure 6 presents the speedup of the GEMM, LU and WZ for the fixed matrix size as the function of the number of threads depending on the thread mapping in relation to the application version executed on one core. These figures show the scalability of applications when the number of threads is increased on the multicore system. Our algorithm scales well

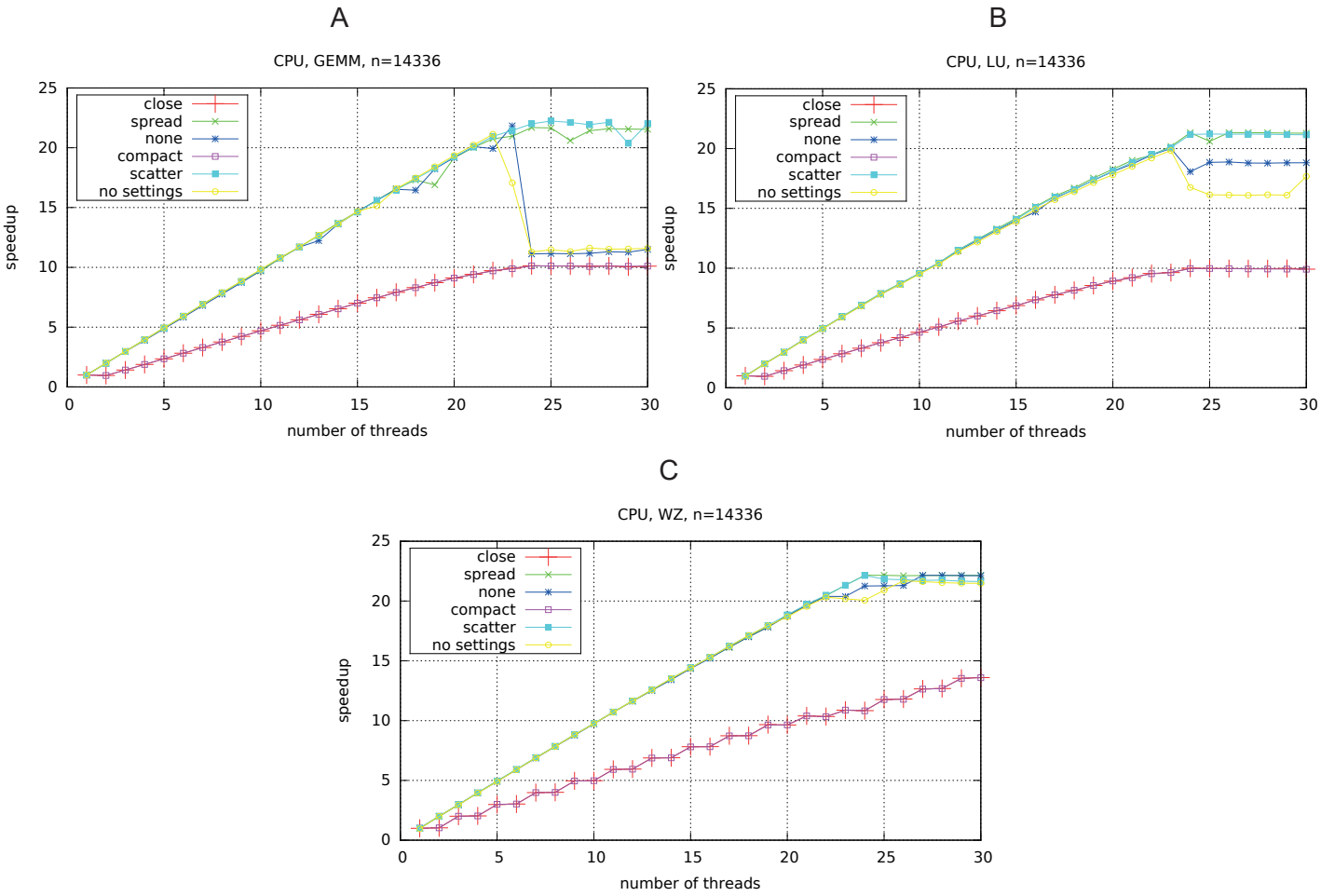

Fig. 6. The speedup of GEMM (A) and LU (B) from the MKL library, and of the authors' WZ implementation (C)

up to 24 threads. To achieve the best speedup, it is the advisable to use all the physical cores (here, 24 threads), without hyperthreading (the MKL trims the number of the threads down to the number of physical cores, because the hyperthreading gives almost no improvement when the cache is well utilized). The hyperthreading also impairs the results for `none` in some cases and it does not improve the results.

However, we can see a speedup breakdown in all these Figures. It is caused by 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 seem that the breakdown should be somewhat earlier – but we do not know the real frequencies used (apart from the fact they are between a producer-specified minimum and maximum).

All three implementations scale best with regard to the threads and to the matrix size when the environment variables are set to `scatter` or `spread`.

4.4. Power and energy. In this section, we focus on a detailed study of power and energy characteristics. We measured the power and the energy consumption of three numerical algorithms with different settings of the thread mapping. The Intel RAPL counter monitor was considered as the measurement tool with an adjustable sampling rate that we set to 100 ms, similarly to the work [10]. This sampling rate was sufficient to show important transitions. Using the RAPL, values can be read from the Intel processor's registers and the power consumption of the CPU and DRAM can be estimated in a very accurate way. We measured the power and energy for two CPUs (denoted by Package 0 and Package 1) and two DRAMs (denoted by DRAM 0 and DRAM 1) because our system was dual-socket, thus allowing us to see which of the subsystems was more heavily loaded.

In this section, we show the results for a selected matrix of the size 14336. The results are very similar for other sizes of matrices. For these experiments, we use the optimal number of threads and the thread count is equal to the number of physical cores, namely 24.

4.4.1. Power. Figure 7 shows three graphs of the power consumption of GEMM for matrix size of 14336 with different settings of the thread mapping (we obtained the similar graphs for LU and WZ). On each graph, we can see three levels of the

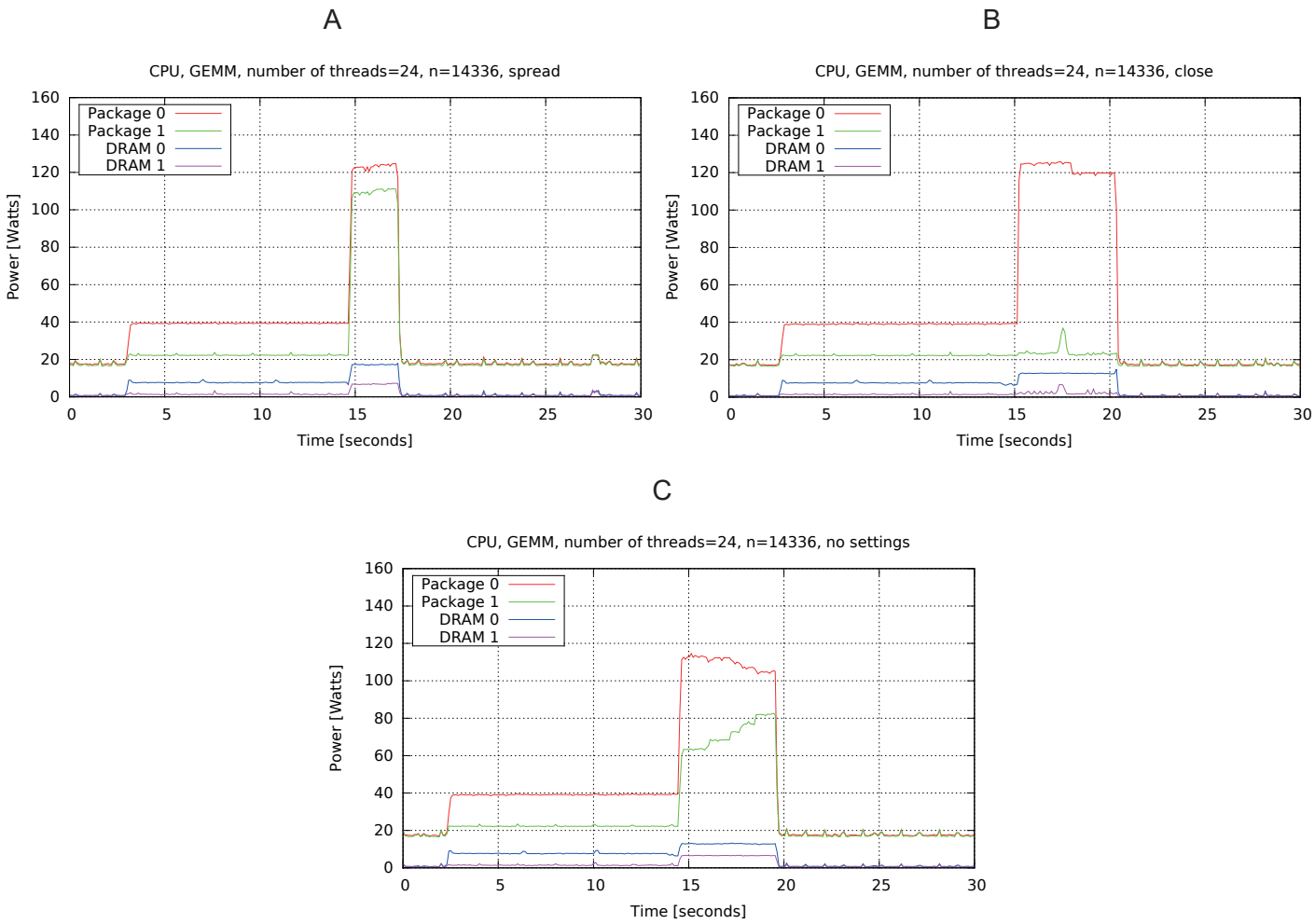

Fig. 7. The power profiling of GEMM from the MKL library with RAPL (A: `spread`; B: `close`; C: *no settings*)

power usage. The first one is idle or low power and is equal to about 20 W for both packages and close to 0 W for both DRAMs. The second level is the memory allocation and is equal to about 40 W for Package 0 and without change for Package 1 (still 20 W) and about 5 W for DRAM 0 and without change for DRAM 1 (0 W). The third one is the proper computation and the power depends on the thread mapping (in Section \4.4.3 we study the energy consumption for this level).

For `spread` (we obtained the similar graphs for `scatter`) we see that Package 0 consumes over 120 W, Package 1 about 110 W, DRAM 0 consumes about 20 W and DRAM 1 about 10 W. For `close` (we obtained the similar graphs for `compact`) we see that Package 0 consumes over 120 W, Package 1 about 20 W. For *no settings* (we obtained the similar graphs for `none`), we can see that the results are not so balanced – sometimes one package uses more energy, sometimes the other one. The same goes for memory. For `spread`, both the sockets were loaded with computations – although one of the sockets had to manage the work division. For `close`, only one socket was used, thus, the running time extended. For *no settings*, we can see that the thread migration takes place because the loads of both sockets are different at various moments.

4.4.2. Power efficiency. We investigated the whole system which implies adding idle, or unused, components to any metric. We assumed a metric of the number of floating-point operations per second per Watt (flop/s/W) (as [14]). Table 2 compares the effects of the thread mapping on the power efficiency for three applications. We obtained better power efficiency for the thread mapping from the vendor-neutral runtime system OpenMP settings. The best power efficiency is achieved for `spread` and `scatter`. In these cases, we observe load balanced system. The worst power efficiency is achieved for `none`, `close` or `compact`. The machine is badly load balanced in the case of `close` and `compact`.

Table 2
Power efficiency [Gflop/s/W] for different thread mapping for GEMM, LU and WZ factorization

	`close`	`spread`	`none`	`compact`	`scatter`	*no settings*
GEMM	3.58	**4.53**	2.89	3.53	**4.53**	2.90
LU	2.09	**2.78**	1.79	1.99	2.69	1.86
WZ	1.88	**2.41**	**2.41**	1.88	2.38	2.38

The best power efficiency we obtained for GEMM. The worst power efficiency we obtained for WZ factorization.

4.4.3. Energy. When considering energy consumption, we considered the whole system which implies adding the energy consumption for all components namely Package 0, Package 1, DRAM 0 and DRAM 1. In Fig. 8 we see energy consumption of the computation level only for six tested settings of the thread mapping in three applications. It can be clearly seen that using the `none` or *no settings* for GEMM and LU factorization and `close` and `compact` for WZ factorization consumes more energy than `spread` and `scatter` for GEMM and LU factorization and than `spread`, `scatter`, `none` and *no settings* for WZ factorization which may be expected. `None` and *no settings* behave differently for GEMM and LU factorization than for WZ factorization.

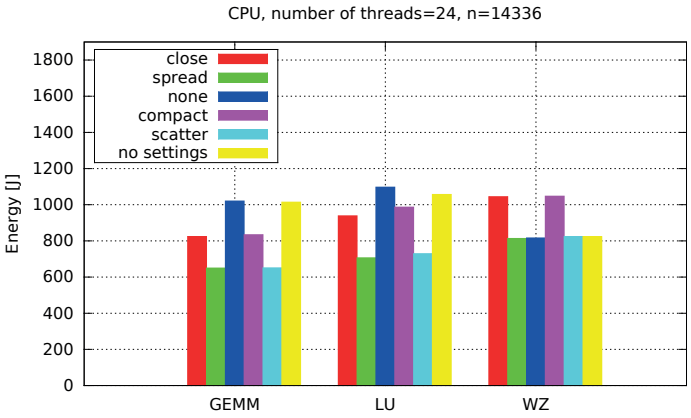

Fig. 8. The energy consumption with RAPL for different thread mapping for GEMM, LU and WZ factorization

`Spread` consumed about 22% less energy than `close`. `Scatter` consumed about 23% less energy than `compact`.

The largest energy consumption is for LU factorization with `none`. The smallest energy consumption is for GEMM with `spread` and `scatter`.

5. Conclusion

In this article, we analyzed the OpenMP thread mapping mechanism for numerical dense linear algebra on a shared memory multicore architecture. We focused on the decompositions of square dense nonsingular matrices into two factors. Such a problem is of computationally intensive nature. To reduce computing time significantly and to use the contemporary computers architectures, the authors considered a partition of the matrix **A** into blocks or tiles and the use of BLAS operations among others GEMM.

The paper highlights the significant impact of thread mapping on the performance, the speedup and power and energy consumption of the matrix decompositions which use BLAS operations in their implementations. Our results showed that there is one thread mapping strategy adapted for block-based numerical dense linear algebra on shared memory multicore architectures. For the matrix decomposition, the environment variable should be set to `scatter` or `spread` because in this way we efficiently exploit the potential of modern shared memory multicore machines and energy saving. With this setting, threads are put far from each other (as on different packages) which provides a better usage of hardware resources (uniform load) and reduces the execution time and the energy consumption.

In future works, the authors plan to research the impact of the thread mapping on numerical dense linear algebra on Intel Xeon Phi and to compare it with the results obtained in this work. The authors are going to use the vendor-neutral BLAS library and evaluate the results for other compilers on multicore and manycore architectures.

References

[1] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen: LAPACK Users' Guide, Society for Industrial and Applied Mathematics, Philadelphia, PA, Third Edition, 1999.

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

[3] B. Bylina, "The Block WZ factorization", Journal of Computational and Applied Mathematics 331, 119–132 (2018).

[4] B. Bylina and J. Bylina, "Incomplete WZ factorization as an alternative method of preconditioning for solving Markov chains", PPAM, volume 4967 of *Lecture Notes in Computer Science*, 99–107 (2007).

[5] B. Bylina and J. Bylina, "Influence of preconditioning and blocking on accuracy in solving Markovian models", *Applied Mathematics and Computer Science*, 19 (2), 207–217 (2009).

[6] B. Bylina and J. Bylina "OpenMP thread affinity for matrix factorization on multicore systems", *Proceedings of the 2017 Federated Conference on Computer Science and Information Systems, volume 11 of Annals of Computer Science and Information Systems*, 489–492 (2017).

[7] S. Chandra Sekhara Rao, "Existence and uniqueness of WZ factorization", *Parallel Computing*, 23 (8), 1129–1139 (1997).

[8] M. Diener, E. H. M. Cruz, M. A. Z. Alves, P. O. A.Navaux, and I. Koren "Affinity-based thread and data mapping in shared memory systems", *ACM Comput. Surv.*, 49 (4), 64:1–64:38 (Dec. 2016).

[9] J. Dongarra, J. DuCroz, I. S. Duff, and S. Hammarling, "A set of level-3 Basic Linear Algebra Subprograms", *ACM Trans. Math. Software*, 16, 1–28 (1990).

[10] J. Dongarra, H. Ltaief, P. Luszczek, and V. M.Weaver, "Energy footprint of advanced dense numerical linear algebra using tile algorithms on multicore architectures", *2012 Second International Conference on Cloud and Green Computing*, 274–281 (Nov. 2012).

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

[12] M. J. Flynn. "Some computer organizations and their effectiveness", *IEEE Trans. Comput.*, 21 (9), 948–960 (Sep. 1972).

[13] E. Rotem, A. Naveh, A. Ananthakrishnan, E. Weissmann, and D. Rajwan, "Power-management architecture of the intel microarchitecture code-named sandy bridge", *IEEE Micro*, 32 (2), 20–27 (Mar. 2012).
[14] M. Weiland and N. Johnson, "Benchmarking for power consumption monitoring", *Computer Science – Research and Development*, 30 (2), 155–163 (May 2015).
[15] P. Yalamov and D. J. Evans, "The WZ matrix factorisation method", *Parallel Computing*, 21 (7), 1111–1120 (1995).
[16] Intel Math Kernel Library, 2014. http://software.intel.com/en-us/articles/intel-mkl/
[17] OpenMP Architecture Review Board: OpenMP application program interface version 4.5, May 2015.