Proceedings of the Federated Conference on
Computer Science and Information Systems pp. 629–639

DOI: 10.15439/2015F354
ACSIS, Vol. 5

Strategies of parallelizing nested loops on the multicore architectures on the example of the WZ factorization for the dense matrices

Beata Bylina, Jarosław Bylina
Marie Curie-Skłodowska University,
Institute of Mathematics,
Pl. M. Curie-Skłodowskiej 5,
20-031 Lublin, Poland
Email: {beata.bylina,jaroslaw.bylina}@umcs.pl

***Abstract*—In the WZ factorization the outermost parallel loop decreases the number of iterations executed at each step and this changes the amount of parallelism in each step. The aim of the paper is to present four strategies of parallelizing nested loops on multicore architectures on the example of the WZ factorization.**

For random dense square matrices with the dominant diagonal we report the execution time, the performance, the speedup of the WZ factorization for these four strategies of parallelizing nested loops and we investigate the accuracy of such solutions.

It is possible to shorten the runtime when utlilizing the appropriate strategies with the use of good scheduling.

Keywords: linear system, WZ factorization, matrix factorization, matrix computations, multicore architecture, parallel nested loops, OpenMP

I. Introduction

MULTICORE computers with shared memory are used to solve the computational science problems. One of more important computational problems is solution of linear systems, the form of which is the following:

$$\mathbf{A}\mathbf{x} = \mathbf{b}, \qquad \text{where} \quad \mathbf{A} \in \mathbb{R}^{n\times n}, \quad \mathbf{b} \in \mathbb{R}^{n}. \tag{1}$$

One of the direct methods of solving a dense linear system (1) is to factorize the matrix $\mathbf{A}$ into some simpler matrices — that is its decomposition into factor matrices of a simpler structure — and then solving simpler linear systems.

Such factorization is hard to compute and that is why it is worth applying different optimization techniques and simultaneously using parellelism of contemporary computers.

The implementation of the factorization contains nested loops. The reasearch of the parallelization of nested loops have been undertaken by different scientistcs.

In the work [6], the authors study five different models for nested parallel loops execution on shared-memory multiprocessors and show a simulation based performance comparison of different techniques using real application. The possibility to take advantage of the parallelism in nested parallel loops with the use of good scheduling and synchronization algorithms is described.

An automatic mechanism to dynamically detect the best way to exploit the parallelism when having nested parallel loops is presented in the study [3]. This mechanism is based on a number of threads, size of the problem, number of iterations in a loop and its was implemented inside IBM XL runtime library. This paper examined (among other) an LU kerner, which decomposes the matrix $\mathbf{A}$ into the matrices: $\mathbf{L}$ (lower triangular matrix) and $\mathbf{U}$ (upper triangular matrix).

An algorithm for finding good distributions of threads to tasks is provided and the implementation of nested parallelism in OpenMP is discussed in the paper [1].

The main focus of [5] was to investigate the possibility of dynamically choosing, at runtime, the loop which best utilises the available threads.

To implement parallel programs on multicore systems with shared-memory, programmers usually use the OpenMP standard [8]. The programming model provides a set of directives to explicitly define parallel regions in applications. The compliator translates these directives. One of its most interesting features in the language is the support for nested parallelism.

This work investigate the issue of the parallelizing nested loops in OpenMP. The OpenMP standard supports loop paralelism. For OpenMP standard, it is done by the utilization of the directive `#pragma omp parallel for`, which provides a shourtcut for specifying a parallel region that contains a single `#pragma omp parallel`.

Parallelism of the nested loops in the WZ factorization is the aim of the work. In the WZ kerner the outermost parallel loop decreases the amount of iterations executed at each step and this changes the number of parallelism in each step. In this paper we investigate the time, the scalability, the speedup and the acuraccy for four different nested loops parallelism strategies for the WZ factorization.

The paper deals with the following issues. In Section II the idea of the WZ factorization [2], [7] and the way the matrix $\mathbf{A}$ is factorized to a product of matrices $\mathbf{W}$ and $\mathbf{Z}$ are described. Such a factorization exists for every nonsingular matrix (with pivoting) which was shown in [2].

Section III provides information about some strategies of parallelizing nested loops and their application to the orginal the WZ factorization. Section IV presents the results of our

978-83-60810-66-8/$25.00

experiments. The time, the speedup, the performance of WZ factorization for different strategies on the two platforms are analysed. The influence of the size of the matrix on the achieved numerical accuracy is studied as well. Section V is a summary of our experiments.

II. WZ Factorization (WZ)

The chapter presents the WZ factorization usage to solve (1). The WZ factorization is described in [2], [4].

Let us assume that the $\mathbf{A}$ is a square nonsingular matrix of an even size (it is somewhat easier to obtain formulas for even sizes than for odd ones). We are to find matrices $\mathbf{W}$ and $\mathbf{Z}$ that fulfill $\mathbf{WZ} = \mathbf{A}$ and the matrices $\mathbf{W}$ and $\mathbf{Z}$ consist of the following rows $\mathbf{w}_i^T$ and $\mathbf{z}_i^T$ respectively:

$$\begin{aligned}
\mathbf{w}_1^T &= (1, \underbrace{0, \ldots, 0}_{n-1}) \\
\mathbf{w}_i^T &= (w_{i1}, \ldots, w_{i,i-1}, 1, \underbrace{0, \ldots, 0}_{n-2i+1}, w_{i,n-i+2}, \ldots, w_{in}) \\
&\qquad \text{for} \quad i = 2, \ldots, \tfrac{n}{2}, \\
\mathbf{w}_i^T &= (w_{i1}, \ldots, w_{i,n-i}, \underbrace{0, \ldots, 0}_{2i-n-1}, 1, w_{i,i+1}, \ldots, w_{in}) \\
&\qquad \text{for} \quad i = \tfrac{n}{2}+1, \ldots, n-1, \\
\mathbf{w}_n^T &= (\underbrace{0, \ldots, 0}_{n-1}, 1) \\
\mathbf{z}_i^T &= (\underbrace{0, \ldots, 0}_{i-1}, z_{ii}, \ldots, z_{i,n-i+1}, 0, \ldots, 0) \\
&\qquad \text{for} \quad i = 1, \ldots, \tfrac{n}{2}, \\
\mathbf{z}_i^T &= (\underbrace{0, \ldots, 0}_{n-i}, z_{i,n-i+1}, \ldots, z_{ii}, 0, \ldots, 0) \\
&\qquad \text{for} \quad i = \tfrac{n}{2}+1, \ldots, n.
\end{aligned} \tag{2}$$

After the factorization we can solve two linear systems:

$$\begin{cases} \mathbf{Wy} = \mathbf{b} \\ \mathbf{Zx} = \mathbf{y} \end{cases} \tag{3}$$

(where $\mathbf{y}$ is an auxiliary intermediate vector) instead of one (1).

In this paper we are interested only in obtaining the matrices $\mathbf{Z}$ and $\mathbf{W}$. The first part of the algorithm consists of setting succesive 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 $(n-1)$st row. Next we update the matrix $\mathbf{A}$.

More formally we can describe the first step of the algorithm the following way.

1) For every $i = 2, \ldots, n-1$ we compute w_{i1} and w_{in} from the system:

$$\begin{cases} a_{11}w_{i1} + a_{n1}w_{in} = -a_{i1} \\ a_{1n}w_{i1} + a_{nn}w_{in} = -a_{in} \end{cases}$$

and we put them in the matrix of the form:

$$\mathbf{W}^{(1)} = \begin{bmatrix} 1 & 0 & \cdots & 0 & 0 \\ w_{21} & 1 & \ddots & \vdots & w_{2n} \\ \vdots & 0 & \ddots & 0 & \vdots \\ w_{n-1,1} & \vdots & \ddots & 1 & w_{n-1,n} \\ 0 & 0 & \cdots & 0 & 1 \end{bmatrix}.$$

2) We compute:

$$\mathbf{A}^{(1)} = \mathbf{W}^{(1)}\mathbf{A}.$$

After the first step we get a matrix of the form:

$$\mathbf{A}^{(1)} = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1,n-1} & a_{1n} \\ 0 & a_{22}^{(1)} & \cdots & a_{2,n-1}^{(1)} & 0 \\ \vdots & \vdots & & \vdots & \vdots \\ 0 & a_{n-1,2}^{(1)} & \cdots & a_{n-1,n-1}^{(1)} & 0 \\ a_{n1} & a_{n2} & \cdots & a_{n,n-1} & a_{nn} \end{bmatrix}, \tag{4}$$

where (for $i, j = 2, \ldots, n-1$):

$$a_{ij}^{(1)} = a_{ij} + w_{i1}a_{1j} + w_{in}a_{nj}. \tag{5}$$

Then, we proceed analogously — but for the inner square matrices — $\mathbf{A}^{(1)}$ of size $(n-2)$ and so on.

So, the whole algorithm is the following.

For $k = 1, 2, \ldots, \frac{n}{2} - 1$:

1) For every $i = k+1, \ldots, n-k$ we compute w_{ik} and $w_{i,n-k+1}$ from the system:

$$\begin{cases} a_{kk}^{(k-1)}w_{ik} + a_{n-k+1,k}^{(k-1)}w_{i,n-k+1} = -a_{ik}^{(k-1)} \\ a_{k,n-k+1}^{(k-1)}w_{ik} + a_{n-k+1,n-k+1}^{(k-1)}w_{i,n-k+1} = -a_{i,n-k+1}^{(k-1)} \end{cases}$$

and we put them in a matrix of the form shown in Figure 1.

2) We compute:

$$\mathbf{A}^{(k)} = \mathbf{W}^{(k)}\mathbf{A}^{(k-1)}.$$

After $(\frac{n}{2} - 1)$ such steps we get the matrix

$$\mathbf{Z} = \mathbf{A}^{(\frac{n}{2}-1)}.$$

Moreover, we know that:

$$\mathbf{W}^{(\frac{n}{2})-1} \cdot \ldots \cdot \mathbf{W}^{(1)} \cdot \mathbf{A} = \mathbf{Z},$$

so we get

$$\mathbf{A} = \{\mathbf{W}^{(1)}\}^{-1} \cdot \ldots \cdot \{\mathbf{W}^{(\frac{n}{2})-1}\}^{-1} \cdot \mathbf{Z} = \mathbf{WZ}.$$

Algorithm 1 shows the WZ algorithm.

The complexity of Algorithm 1 can be expressed by the formule

$$\sum_{k=1}^{\frac{n}{2}-1} \left(3 + \sum_{i=k+1}^{n-k} \left(8 + \sum_{j=k+1}^{n-k} 4 \right) \right) = \frac{4n^3 - 7n - 18}{6}. \tag{6}$$

So, this algorithm requires $O(n^3)$ arthmetic operations.

$$\mathbf{W}^{(k)} = \begin{bmatrix} 1 & & & & & & & \\ & \ddots & & & & & & \\ & & 1 & & & & & \\ & & w_{k+1,k} & \ddots & & w_{k+1,n-k+1} & & \\ & & \vdots & & \ddots & \vdots & & \\ & & w_{n-k,k} & & & w_{n-k,n-k+1} & & \\ & & & & & 1 & & \\ & & & & & & \ddots & \\ & & & & & & & 1 \end{bmatrix}$$

Fig. 1. The matrix $\mathbf{W}^{(k)}$ in kth step.

Algorithm 1 Outline of the WZ factorization algorithm (WZ)

Require: A
Ensure: W, Z
1: **for** $k = 1$ to $n/2 - 1$ **do**
2: $k2 \leftarrow n - k + 1$
3: $det \leftarrow a_{kk} * a_{k2k2} - a_{k2k} * a_{kk2}$
4: **for** $i = k + 1$ to $k2 - 1$ **do**
5: $w_{ik} \leftarrow (a_{k2k2} * a_{ik} - a_{k2k} * a_{ik2})/det$
6: $w_{ik2} \leftarrow (a_{kk} * a_{ik2} - a_{kk2} * a_{ik})/det$
7: **for** $j = k + 1$ to $k2 - 1$ **do**
8: $a_{ij} \leftarrow a_{ij} - w_{ik} * a_{kj} - w_{ik2} * a_{k2j}$
9: **end for**
10: **end for**
11: **end for**
12: $Z \leftarrow A$

III. Nested loops parallelism strategies

An application with nested loops can be performed parallely in different ways depending on compilators, hardware and run-time system support available. Nested loops require from of a programmer taking a decision concerning details of parallelism.

In this work we deal with the following parallelization strategies for nested loops:

1) *outer*
2) *inner*
3) *nested*
4) *split*

While all variables used in a parallel region are by default `shared`, in each strategy we declare explicitly all variables as `private` or `shared` for all directives respectively. Using the `private` clause, we specify that each thread has its own copy of variables.

To ensure load balancing for all threads we use the schedule clause, which specifies how the iterations of the loop are assigned to the threads. In the clause `schedule` of the directive `#pragma omp parallel for` we set the value `static`, because the computational cost of the tasks is known.

A. Outer

Outer — the simplest parallelization strategy of nested loops is parallel execution of the most outer loop. All inner loops are executed in a sequence. This approach gives good results if the number of iterations in a loop is big and the iteration's granularity is coarse enough, which happens exactly in case of the WZ factorization. Algorithm 2 presents outer strategy for WZ factorization. The outermost k-loop cannot be parallelized, however, we can parallelize the i-loop. In this simple parallelization strategy the loop is divided equally between threads, so every thread performs the same amount of work, which ansure regular distribution of work beetwen threads.

B. Inner

Another strategy of paralelizing nested loops involves executing the inner loops in parallel on all processors, but the outer loop is executed in a sequence. Clearly, in case of WZ factorization blocking barrier is used at the end of each parallel

Algorithm 2 Outline of the WZ factorization algorithm (WZ) — outer strategy

Require: A
Ensure: W, Z
1: **for** $k = 1$ to $n/2 - 1$ **do**
2: $k2 \leftarrow n - k + 1$
3: $det \leftarrow a_{kk} * a_{k2k2} - a_{k2k} * a_{kk2}$
4: **#pragma omp parallel for private(i) shared(k, k2, w, a, det,j)**
5: **for** $i = k + 1$ to $k2 - 1$ **do**
6: $w_{ik} \leftarrow (a_{k2k2} * a_{ik} - a_{k2k} * a_{ik2})/det$
7: $w_{ik2} \leftarrow (a_{kk} * a_{ik2} - a_{kk2} * a_{ik})/det$
8: **for** $j = k + 1$ to $k2 - 1$ **do**
9: $a_{ij} \leftarrow a_{ij} - w_{ik} * a_{kj} - w_{ik2} * a_{k2j}$
10: **end for**
11: **end for**
12: **end for**
13: $Z \leftarrow A$

loop, which prevents incorect results. Parallelizing the inner loop will potentially provide smaller pieces of work so they can be distributed evenly between the available threads but it has more overhead due to work distribution and synchronization beetwen threads. This overhead may be high if the loop granularity is too fine. Algorithm 3 presents inner strategy for WZ factorization, in which the j-loop are parallelized.

C. Nested

The third strategy of execution of nested loops paralelization is exploiting the paralelism on each level — nested parallelism. Standard OpenMP (from 2.5 version) makes it possible to nest parallel loops, however, it must be switched on by means of the environment variable `OMP_NESTED` or the function `omp_set_nested`. Each task needs at least one thread to its own disposal. Algorithm 4 presents the nested strategy.

This algorithm shows how a 2-level parallelism can be implemented in OpenMP based on the directives. Nesting parallel loops is a way to use more threads in a computation. This can easily create a large number of threads as their number is the product of the number of threads forked at each level of nested loops.

D. Split

The final strategy concernes division of i-loops into two separate loops. Such a split facilites presentation of kth step in the form of a dag (directed acyclic graph), which shows the order of the task execution. The dag represents computational solutions in which the nodes represent tasks to be executed and edges represent precedence among the tasks. In the figure 2 a dag for kth step and shows, which part of the matrix is processed in a particular task is presented. By Task 1 we understand determining of valiables $k2$ and det (lines 2 and 3 in Algorithm 1. Task 2 is the computation of kth and $k2$nd column of the matrix $\mathbf{W}$ (lines 4, 5 and 6 in Algorithm 1). Task 3 is the computation of values in the matrix $\mathbf{A}$ (lines 4, 7 and 8 in Algorithm 1).

Algorithm 5 shows the *split* strategy for WZ factorization.

The first loop is parallelized. The second loop is nested loop and we use outer version to parallelize this loop.

IV. Numerical Experiments

In this section we tested the time, the performance, the speedup and the absolute accuracy of the WZ factorization. Our intention was to investigate diffrent nested loops parallelization strategies for the WZ factorization on multicore architecuters. We examined five versions algorithms of the WZ factorization:

- *sequential* (Algorithm 1),
- *outer* (Algorithm 2),
- *inner* (Algorithm 3),
- *nested* (Algorithm 4),
- *split* (Algorithm 5).

Here we used experiments, based on information collected at runtime, to decide whether a loop should execute clause `static` or `dynamic` and we chose `static`.

The input matrices are generated (by the authors). They are random, dense, square matrices with a dominant diagonal of even sizes (1000, 2000,..., 9000)

We used two hardware platforms for testing: E5-2660 and X5650. Their details specifications are presented in Table I.

The algorithms *sequential*, *outer*, *inner*, *nested* and *split* were implemented with the use of the C language with the use of the double precision. Our codes were compiled by INTEL C Compiler (icc) with optimization flag `-O3`. Additionally, all algorithms were linked with the OpenMP library.

A. The Time

All the processing times are reported in seconds. The time is measured with an OpenMP function `open_get_wtime()`. They were tested in the double precision.

In Figures 3 and 4 we have compared the average running time of the four parallel WZ decomposition algorithms and the sequential version on two platforms.

Algorithm 3 Outline of the WZ factorization algorithm (WZ) — inner strategy

Require: A
Ensure: W, Z
1: **for** $k = 1$ to $n/2 - 1$ **do**
2: $k2 \leftarrow n - k + 1$
3: $det \leftarrow a_{kk} * a_{k2k2} - a_{k2k} * a_{kk2}$
4: **for** $i = k + 1$ to $k2 - 1$ **do**
5: $w_{ik} \leftarrow (a_{k2k2} * a_{ik} - a_{k2k} * a_{ik2})/det$
6: $w_{ik2} \leftarrow (a_{kk} * a_{ik2} - a_{kk2} * a_{ik})/det$
7: **#pragma omp parallel for private(j) shared(k, k2, w, a, det,i)**
8: **for** $j = k + 1$ to $k2 - 1$ **do**
9: $a_{ij} \leftarrow a_{ij} - w_{ik} * a_{kj} - w_{ik2} * a_{k2j}$
10: **end for**
11: **end for**
12: **end for**
13: $Z \leftarrow A$

Algorithm 4 Outline of the WZ factorization algorithm (WZ) — nested strategy

Require: A
Ensure: W, Z
1: **for** $k = 1$ to $n/2 - 1$ **do**
2: $k2 \leftarrow n - k + 1$
3: $det \leftarrow a_{kk} * a_{k2k2} - a_{k2k} * a_{kk2}$
4: **#pragma omp parallel for private(i) shared(k, k2, w, a, det)**
5: **for** $i = k + 1$ to $k2 - 1$ **do**
6: $w_{ik} \leftarrow (a_{k2k2} * a_{ik} - a_{k2k} * a_{ik2})/det$
7: $w_{ik2} \leftarrow (a_{kk} * a_{ik2} - a_{kk2} * a_{ik})/det$
8: **#pragma omp parallel for private(j) shared(k, k2, w, a, det)**
9: **for** $j = k + 1$ to $k2 - 1$ **do**
10: $a_{ij} \leftarrow a_{ij} - w_{ik} * a_{kj} - w_{ik2} * a_{k2j}$
11: **end for**
12: **end for**
13: **end for**
14: $Z \leftarrow A$

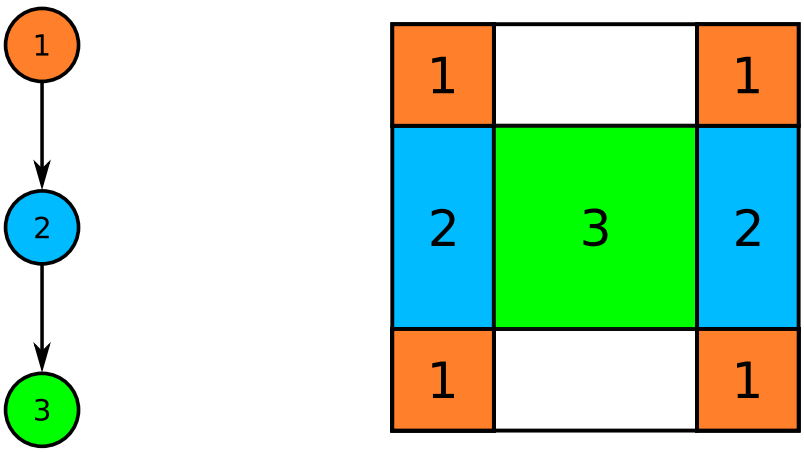

Fig. 2. The dag of the tasks (left). The sequence of calculations in the matrix in the WZ factorization in every step (right).

Algorithm 5 Outline of the WZ factorization algorithm (WZ) — split strategy

Require: A
Ensure: W, Z
1: **for** $k = 1$ to $n/2 - 1$ **do**
2: $k2 \leftarrow n - k + 1$
3: $det \leftarrow a_{kk} * a_{k2k2} - a_{k2k} * a_{kk2}$
4: **#pragma omp parallel for private(i) shared(k, k2, w, a, det)**
5: **for** $i = k + 1$ to $k2 - 1$ **do**
6: $w_{ik} \leftarrow (a_{k2k2} * a_{ik} - a_{k2k} * a_{ik2})/det$
7: $w_{ik2} \leftarrow (a_{kk} * a_{ik2} - a_{kk2} * a_{ik})/det$
8: **end for**
9: **#pragma omp parallel for private(i) shared(k, k2, w, a, det)**
10: **for** $i = k + 1$ to $k2 - 1$ **do**
11: **for** $j = k + 1$ to $k2 - 1$ **do**
12: $a_{ij} \leftarrow a_{ij} - w_{ik} * a_{kj} - w_{ik2} * a_{k2j}$
13: **end for**
14: **end for**
15: **end for**
16: $Z \leftarrow A$

TABLE I
SOFTWARE AND HARDWARE PROPERTIES OF E5-2660 AND X5650 SYSTEMS

	E5-2660 System	X5650 System
CPU	2x Intel Xeon E5-2660 (20M Cache, 2.20 GHz, 8 cores with HT)	2x Intel Xeon X5650 (12M Cache, 2.66 GHz, 6 cores with HT)
CPU memory	48GB DDR3	48GB DDR3
Operating system	CentOS 5.5 (Linux 2.6.18-164.el5)	Debian (GNU/Linux 7.0)
Libraries	OpenMP, Intel Composer XE 2013	OpenMP, Intel Composer XE 2013
Compilers	Intel	Intel

Figure 3 shows the dependence of the time on the number of threads for the matrix of the size 9000 on two platforms (X5650 on the right side, E5-2660 on the left side).

Figure 4 shows the dependence of the time on the matrix size for 12 threads for X5650 system (the right side) and 16 threads for E5-2660 system (the left side).

Using obtained results we conclude that:

- For a growing number of threads E5-2660 architecture outperforms X5650, due to the fact that the latter one is its older. We were expecting this result.
- The time is the shortest for 12 threads on the X5650 system and for 16 threads on the E5-2660 system. For bigger number of threads the time is the same as for 12 threads on the X5650 system nad for 16 threads on E5-2660, which proves weakness of the hyperthrading technology.
- If the size matrix is increased, then the runtime is increased too and it becomes more profitable to use a big number of the threads.
- *split* and *outer* algorithms achieve very similar execution time, which is the shortest compared with other algorithms.
- The worse execution time was achieved by the *nested* algorithm and for E5-2660 it is even worse than sequential algorithm.

B. The Performance

Figures 5 and 6 compare the performance (in Gflops) results obtained for those five algorithms (*sequential*, *outer*, *inner*, *nested*, *split*) — in the double precision on two platforms. The performance is based on the number of floating-point operations in the WZ factorization (6).

Figure 5 shows dependence of the performance on the number of threads (maximum number of the threads is 24) for the matrix of the size 9000 for two platforms (X5650 — the right, E5-2660 — the left).

Figure 6 shows dependence of the performance on matrix size for 12 threads for X5650 system (the right side) and 16 threads for E5-2660 (the left side).

We can see the best performance (about 5.5 Gflop/s) achieved by *split* algorithm for the matrix of the size 9000 for 16 threads on E5-2660 system, and worst (less than 1 Gflop/s) is for *nested* version and *sequential* algorithm for all matrix sizes. On X5650 system we obtain worse performance for all tested algorithms than on E5-2660 System. The performance is very low for all algorithms on X5650 system and almost the same for inner, outer and *split* algorithms on X5650 system.

C. The Speedup

Figures 7 and 8 present the speedup results obtained for four algorithms implementations — in the double precision on two platforms. Figure 7 shows dependence of the speedup on the number of threads (maximum number of the threads is

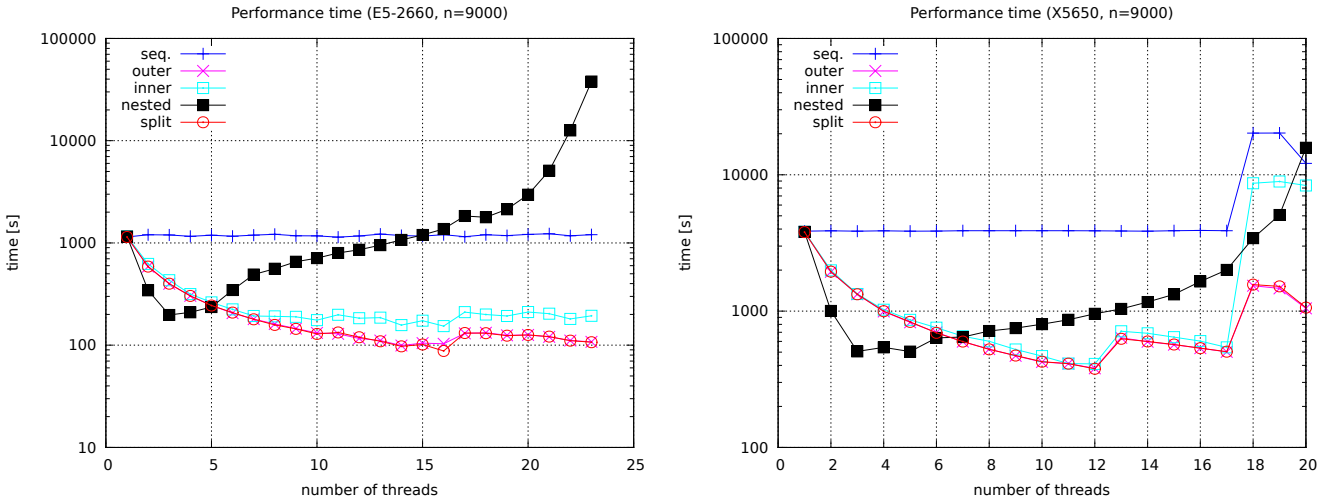

Fig. 3. The average running time of the WZ matrix decomposition as a function of the number of threads — for the five algorithms using the double precision on two platforms (E5-2660 on the left side and X5650 on the right side) for the matrix of the size 9000 (logarithmic y-axis).

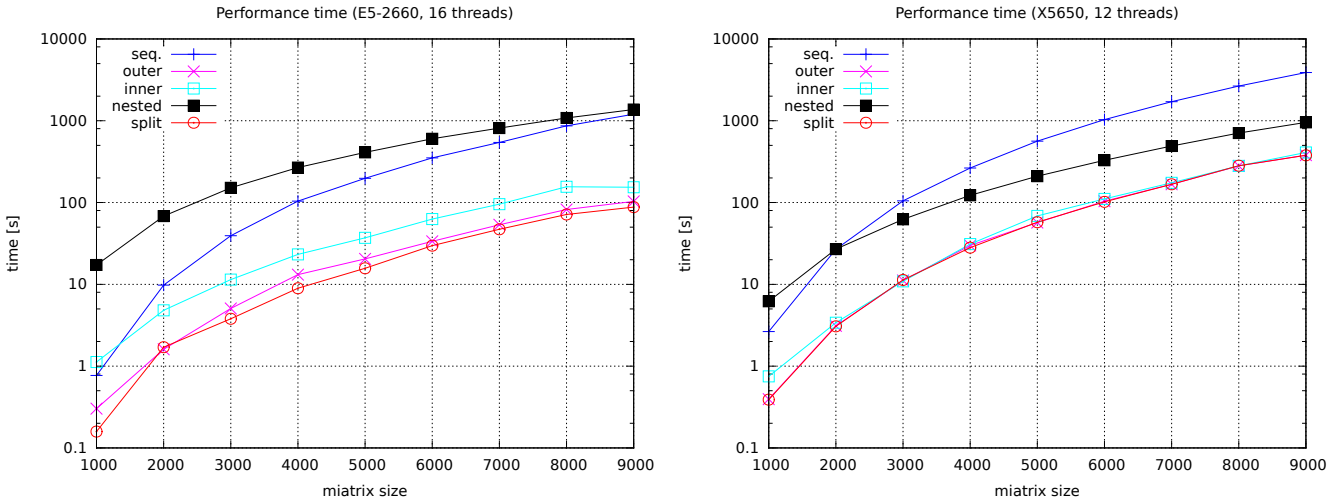

Fig. 4. The average running time of the WZ matrix decomposition as a function of the matrix size — for 16 threads on E5-2660 system (the left side) and for 12 threads on X5650 system (the right side) (logarithmic y-axis).

23) for the matrix of the size 9000 for two platforms (X5650 — the right, E5-2660 — the left).

Figure 8 shows dependence of the performance on the matrix size for 12 threads for X5650 system (the right side) and 16 threads for E5-2660 (the left side).

Note that:

- All algorithms scale well with the size of a matrix; moreover,the bigger the matrix, the better the speedup.
- The speedup increases steadily until 12 threads on E5-2660 System and 16 threads on X5650 system, before it starts to level off.
- *Split* algorithm has the better speedup, even value up to 14 for 16 threads on E5-2660 system.
- On the X5650 system *split* and outer algorithm have similary seedup, but on E5-2660 System *split* algorithm has higher speedup than *split* algorithm.

D. Numerical Accuracy

The purpose of this section is not to accomplish a full study of the numerical stability and accuracy of the WZ factorization, but justify experimentally that our implementation of the WZ algorithm can be used in practice.

As a measure of accuracy we took the following expression (where $||\mathbf{M}||$ is the Frobenius norm of the matrix $\mathbf{M}$) based on the absolute error:

$$||\mathbf{A} - \mathbf{WZ}||.$$

Table II illustrates the accuracy (given as the norms $||\mathbf{A} - \mathbf{WZ}||$) of the WZ factorization. The norms on both platforms (E5-2660 and X5650) are the same for appropriate matrix

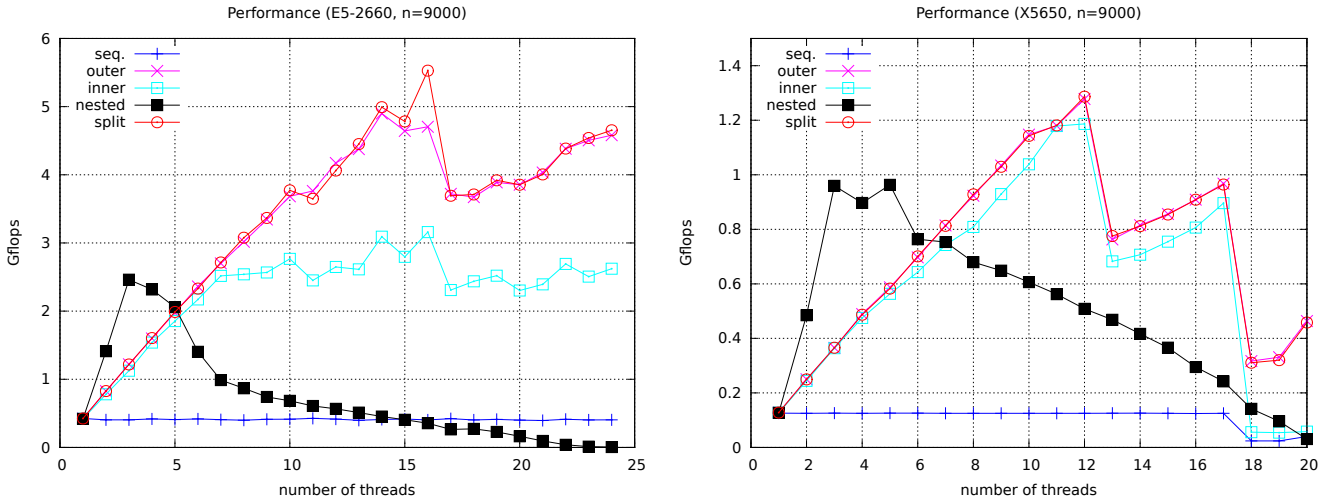

Fig. 5. The performance results for the WZ factorization — using the double precision on two platforms (E5-2660 — the left side; X5650 — the right side) for the five algorithms as the function of number of threads.

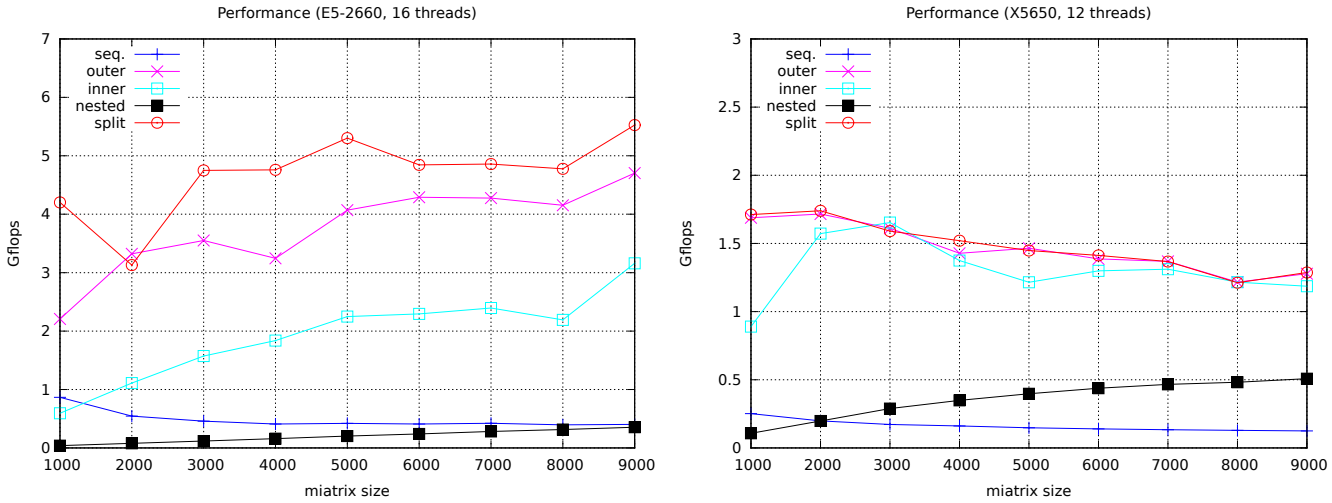

Fig. 6. The performance results for the WZ factorization — using the double precision for 16 threads on E5-2660 system (the left side) and for 12 threads on X5650 system (the right side) for the five algorithms as a function of the matrix size.

sizes. Values of the norm do not depend on the number of the threads and do not depend on a choice of algorithms (for all algorithms the norm depends only on the matrix size).

V. Conclusion

In this paper we examined several practical aspect of nested parallel loop execution. We used four different strategies for executing nested parallel loops on the examples of the WZ factorization. All proposed approaches usually accelerate sequential computations, except the *nested* algorithm.

Nested algorithm for a small number of threads proved to be the fastest, but for a big numer of threads its execution took longer time even than for a *sequential* algorithm. We may explain that creating any parallel region will cause the overhead. Overhead from nesting of parallel regions may cause overheads greater than necessary if, for example, an outer region could simply employ more threads in a computation. The appliaction lost the time on scheduling threads. OpenMP allows the specification of nested parallel loops, but for WZ factorization does not acquire satisfactory results. OpenMP uses nesting poorly.

The available number of threads exploited both outer and *split* algorithms best. *Split* approach achievs the best speedup. The speedup of 14 was achived for 16 threads on the E5-2660 system. We find this result very satisfactory.

The implementation had no impact on the accuracy of the factorization — the accuracy depended only on the size of the matrix what is quite self-evident.

The implementation of the *split* algorithm presented in this paper achieves high performance results, which has a direct

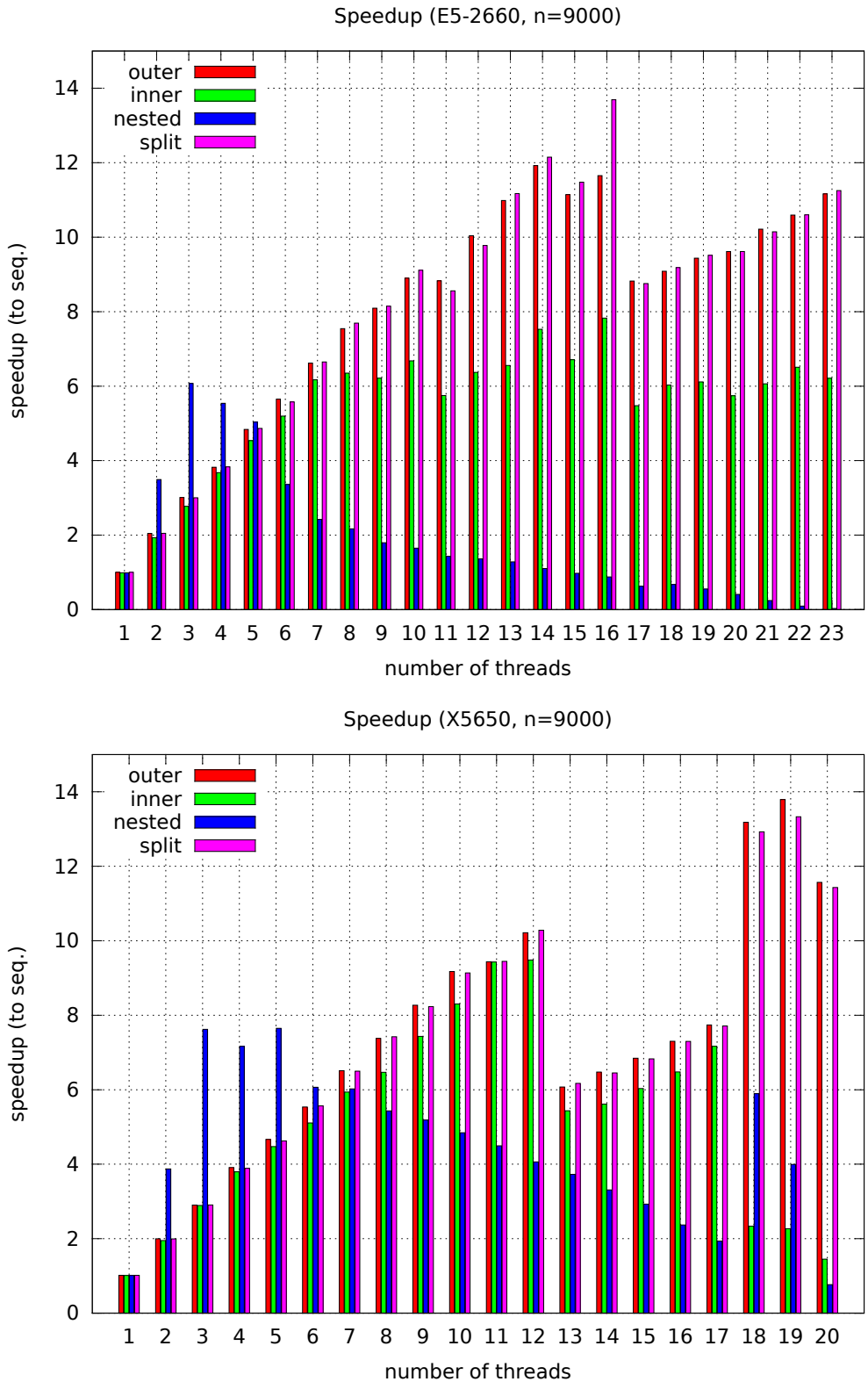

Fig. 7. The speedup results for the WZ factorization — using the double precision on two platforms (E5-2660 — top; X5650 — bottom) for the four algorithms as the function of number of threads.

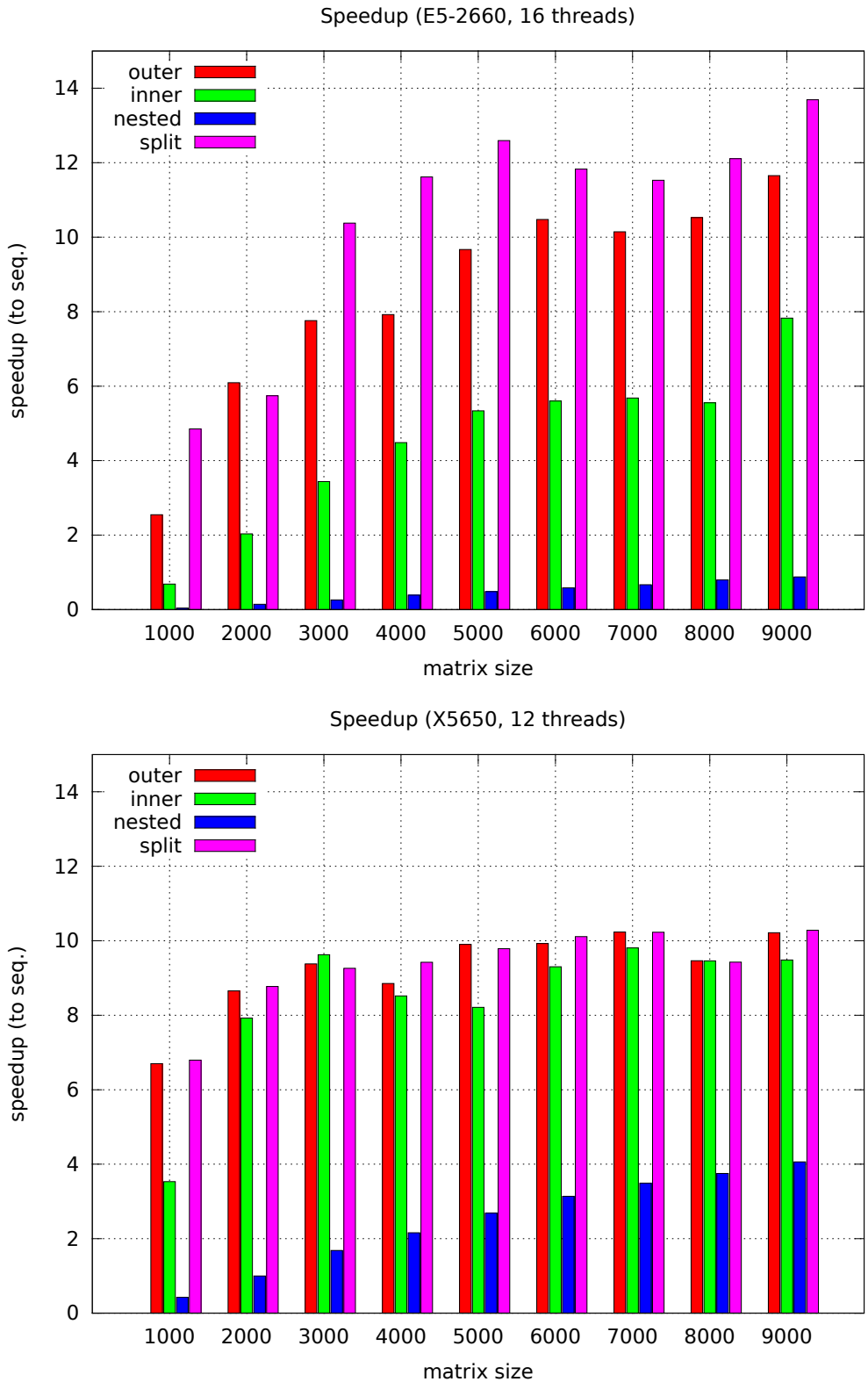

Fig. 8. The speedup results for the WZ factorization — using the double precision for 16 threads on E5-2660 System (top) and 12 threads on X5650 System (bottom) for the four algorithms as a function of the matrix size.

TABLE II
THE NORMS FOR THE WZ FACTORIZATIONS IN DOUBLE PRECISION ON E5-2660 SYSTEM AND X5650 SYSTEM FOR ALL THE ALGORITHMS IN DOUBLE PRECISION

matrix size	$\|\|\mathbf{A}-\mathbf{WZ}\|\|$
1000	$2.89 \cdot 10^{-22}$
2000	$1.18 \cdot 10^{-21}$
3000	$2.97 \cdot 10^{-21}$
4000	$5.59 \cdot 10^{-21}$
5000	$9.32 \cdot 10^{-21}$
6000	$1.40 \cdot 10^{-20}$
7000	$2.06 \cdot 10^{-20}$
8000	$2.83 \cdot 10^{-20}$
9000	$3.78 \cdot 10^{-20}$

impact on the solution of linear systems.

This paper is another example of the succesful use of OpenMP for solving scientific appliactions.

REFERENCES

[1] R. Blikberg, T. Sørevik: "Load balancing and OpenMP implementation of nested parallelism", *Parallel Computing* 31, Elsevier, 2005, pp. 984–998.

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

[3] A. Duran, R. Silvera, J. Corbalan, J. Labarta: "Runtime adjustment of parallel nested loops", *Proceedings of the 5th international conference on OpenMP Applications and Tools: shared Memory Parallel Programming with OpenMP*, Houston, 2004, pp. 137–147.

[4] D. J. Evans, M. Hatzopoulos: "The parallel solution of linear system", *Int. J. Comp. Math.* **7** (1979), pp. 227–238.

[5] A. Jackson, O. Agathokleous: "Dynamic Loop Parallelisation", arXiv: 1205.2367v1, 10 May 2012.

[6] A. Sadun, W. W. Hwu: "Executing nested parallel loops on shared-memeory multiprocessors", *Proceedings of the 21st Annual International Conference on Parallel Processing*, 1992.

[7] P. Yalamov, D. J. Evans: "The WZ matrix factorization method", *Parallel Computing* 21, 1995, pp. 1111–1120.

[8] OpenMP, http://openmp.org/wp/, April 2015.