Vectorized Parallel Solver for Tridiagonal Toeplitz Systems of Linear Equations

Beata Dmitruk(✉) and Przemysław Stpiczyński

Institute of Computer Science, Maria Curie–Skłodowska University, ul. Akademicka 9, 20-033 Lublin, Poland
beata.dmitruk@umcs.pl, przem@hektor.umcs.lublin.pl

Abstract. The aim of this paper is to present two versions of a new *divide and conquer* parallel algorithm for solving tridiagonal Toeplitz systems of linear equations. Our new approach is based on a recently developed algorithm for solving linear recurrence systems. We discuss how to reduce the number of necessary synchronizations and show proper data layout that allows to use cache memory and SIMD extensions of modern processors. Numerical experiments show that our new implementations achieve very good seedup on multicore and manycore architectures. Moreover, they are more energy efficient than a simple sequential algorithm.

Keywords: Tridiagonal Toeplitz systems · Parallel algorithms · Vectorization · SIMD extensions · OpenMP · Energy efficiency

1 Introduction

Tridiagonal Toeplitz systems of linear equations play an important role in many theoretical and practical applications. They appear in numerical algorithms for solving boundary value problems for ordinary and partial differential equations [12,14]. For example, a numerical solution to the heat-diffusion equation of the following form

$$\frac{\partial u}{\partial t}(x,t) = \alpha^2 \frac{\partial^2 u}{\partial x^2}(x,t), \text{ for } 0 < x < l \text{ and } 0 < t,$$

with boundary conditions $u(0,t) = u(l,t) = 0$, for $0 < t$ and $u(x,0) = f(x)$, for $0 \leq x \leq l$ can be found using finite difference methods that reduce to the problem of solving tridiagonal Toeplitz systems. Such systems also arise in piecewise cubic interpolation and splines algorithms [2,11,13]. Moreover, such systems are useful when we solve the 2D Poisson equation by the variable separation method and the 3D Poisson equation by a combination of the alternating direction implicit and the variable separation methods [13]. Banded Toeplitz matrices also appear in signal and image processing [1].

There are several methods for solving such systems [3–5,7–9,13–15]. The basic idea comes from Rojo [9]. He proposed a method for solving symmetric

© Springer Nature Switzerland AG 2020
R. Wyrzykowski et al. (Eds.): PPAM 2019, LNCS 12043, pp. 93–103, 2020.
https://doi.org/10.1007/978-3-030-43229-4_9

tridiagonal Toeplitz systems using LU decomposition of a system with almost Toeplitz structure together with Sherman-Morrison's formula [4]. This approach has been modified to obtain new solvers for a possible parallel execution [5,14]. A simple vectorized but non-parallel algorithm was proposed in [3]. A different approach was proposed by McNally et al. [8] who developed a scalable communication-less algorithm. It finds an *approximation* of the exact solution of a system with a given acceptable tolerance level. However, the algorithm does not utilize vectorization explicitly. Terekhov [13] proposed a highly scalable parallel algorithm for solving tridiagonal Toeplitz systems with multiple right hand sides. It should be noticed that well-known numerical libraries optimized for modern multicore architectures like Intel MKL, PLAPACK or NAG do not provide routines for solving tridiagonal Toeplitz systems. These libraries provide solvers for more general systems i.e.non-Toeplitz or dense Toeplitz systems. The case studied here is more detailed, but it allows to formulate more efficient solvers.

In this paper, we present two versions of a new *divide and conquer* parallel vectorized algorithm for finding the exact solution of tridiagonal Toeplitz systems of linear equations. As the starting point, we consider the splitting $T = LR + P$, where L, R are bidiagonal and P has only one non-zero entry [3]. Our new approach for solving bidiagonal Toeplitz systems is based on recently developed algorithms for solving linear recurrence systems [10,12]. We discuss possible OpenMP implementations and show how to reduce the number of necessary synchronizations to improve the performance. Further improvements come from the proper data layout that allows to use cache memory and SIMD extensions of modern processors. Numerical experiments performed on Intel Xeon CPUs and Intel Xeon Phi show that our new implementations achieve good performance on multicore and manycore architectures. Moreover, they are more energy efficient than a simple sequential algorithm.

2 Parallel Algorithm for Tridiagonal Toeplitz Systems

Let us consider a tridiagonal Toeplitz system of linear equations $T\mathbf{x} = \mathbf{b}$ of the following form

$$\begin{bmatrix} d & a & & & \\ 1 & d & a & & \\ & \ddots & \ddots & \ddots & \\ & & 1 & d & a \\ & & & 1 & d \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ \vdots \\ \vdots \\ x_{n-1} \end{bmatrix} = \begin{bmatrix} b_0 \\ b_1 \\ \vdots \\ \vdots \\ b_{n-1} \end{bmatrix}. \tag{1}$$

For the sake of simplicity let us assume that $n = 2^m$, $m \in \mathbb{N}$, and $|d| > 1 + |a|$. Thus, T is not singular and pivoting is not needed to assure numerical stability. To find the solution to (1) we can follow the approach presented in [3] and decompose T as follows

$$\begin{bmatrix} d & a & & & \\ 1 & d & a & & \\ & \ddots & \ddots & \ddots & \\ & & 1 & d & a \\ & & & 1 & d \end{bmatrix} = \underbrace{\begin{bmatrix} r_2 & & & & \\ 1 & r_2 & & & \\ & \ddots & \ddots & & \\ & & 1 & r_2 & \\ & & & 1 & r_2 \end{bmatrix}}_{L} \underbrace{\begin{bmatrix} 1 & r_1 & & & \\ & 1 & r_1 & & \\ & & \ddots & \ddots & \\ & & & 1 & r_1 \\ & & & & 1 \end{bmatrix}}_{R} + \begin{bmatrix} r_1 & 0 & \dots & 0 \\ 0 & 0 & & \vdots \\ \vdots & & \ddots & \vdots \\ 0 & \dots & \dots & 0 \end{bmatrix}, \quad (2)$$

where $r_2 = (d \pm \sqrt{d^2 - 4a})/2$ and $r_1 = d - r_2$. Using this formula we can rewrite the Eq. (1) as follows

$$\begin{bmatrix} x_0 \\ x_1 \\ \vdots \\ x_{n-1} \end{bmatrix} + r_1 x_0 \underbrace{(LR)^{-1} \begin{bmatrix} 1 \\ 0 \\ \vdots \\ 0 \end{bmatrix}}_{\mathbf{u}} = \underbrace{(LR)^{-1} \begin{bmatrix} b_0 \\ b_1 \\ \vdots \\ b_{n-1} \end{bmatrix}}_{\mathbf{v}} \quad (3)$$

or simply $\mathbf{x} + r_1 x_0 \mathbf{u} = \mathbf{v}$. Then the solution to (1) can be found using

$$\begin{cases} x_0 = \frac{v_0}{1 + r_1 u_0} \\ x_i = v_i - r_1 x_0 u_i, \quad i = 1, \dots, n-1. \end{cases} \quad (4)$$

Let $\mathbf{e}_k = (\underbrace{0, \dots, 0}_{k}, 1, 0, \dots, 0)^T \in \mathbb{R}^s$, $k = 0, \dots, s-1$. Note that to apply (4) we only need to have vectors $\mathbf{u}$ and $\mathbf{v}$ that are solutions to systems of linear equations $LR\mathbf{u} = \mathbf{e}_0$ and $LR\mathbf{v} = \mathbf{b}$, respectively. The solution to the system of linear equations $LR\mathbf{y} = \mathbf{f}$ can be found using a simple sequential algorithm based on the following recurrence relations

$$\begin{cases} z_0 = f_0 / r_2 \\ z_i = (f_i - z_{i-1}) / r_2, \quad i = 1, \dots, n-1, \end{cases} \quad (5)$$

and

$$\begin{cases} y_{n-1} = z_{n-1} \\ y_i = z_i - r_1 y_{i+1}, \quad i = n-2, \dots, 0. \end{cases} \quad (6)$$

Thus, to solve (1) using (4), such a simple sequential algorithm based on (5) and (6) should be applied twice for $LR\mathbf{u} = \mathbf{e}_0$ and $LR\mathbf{v} = \mathbf{b}$, respectively. This sequential algorithm requires $9n + O(1)$ flops. It should be pointed out that (5) and (6) contain obvious data dependencies, thus they cannot be parallelized and vectorized automatically by the compiler.

To develop a new parallel algorithm for solving $LR\mathbf{y} = \mathbf{f}$ that could utilize vector extensions of modern multiprocessors, we apply the *divide-and-conquer* algorithm for solving first-order linear recurrence systems with constant coefficients [10,12]. Let us assume that $n = rs$ and $r, s > 1$. First, we arrange entries of L into blocks to obtain the following block matrix

$$L = \begin{bmatrix} L_s & & & \\ B & L_s & & \\ & \ddots & \ddots & \\ & & B & L_s \end{bmatrix}, \quad L_s = \begin{bmatrix} r_2 & & & \\ 1 & r_2 & & \\ & \ddots & \ddots & \\ & & 1 & r_2 \end{bmatrix}, \quad B = \begin{bmatrix} 0 & \dots & 0 & 1 \\ \vdots & & 0 & 0 \\ \vdots & \cdot^{\cdot^{\cdot}} & & \vdots \\ 0 & \dots & \dots & 0 \end{bmatrix}. \tag{7}$$

Let $\mathbf{z}_i = (z_{is}, \dots, z_{(i+1)s-1})^T$ and $\mathbf{f}_i = (f_{is}, \dots, f_{(i+1)s-1})^T$. Then the lower bidiagonal system of linear equations $L\mathbf{z} = \mathbf{f}$ can be rewritten in the following recursive form

$$\begin{cases} L_s\mathbf{z}_0 = \mathbf{f}_0 \\ L_s\mathbf{z}_i = \mathbf{f}_i - B\mathbf{z}_{i-1}, \quad i = 1, \dots, r-1. \end{cases} \tag{8}$$

Equation (8) reduces to the following form

$$\begin{cases} \mathbf{z}_0 = L_s^{-1}\mathbf{f}_0 \\ \mathbf{z}_i = L_s^{-1}\mathbf{f}_i - z_{is-1}L_s^{-1}\mathbf{e}_0, \quad i = 1, \dots, r-1. \end{cases} \tag{9}$$

Note that (9) has a lot of potential parallelism. Just after all vectors $L_s^{-1}\mathbf{f}_i$, $i = 0, \dots, r-1$, have been found, we can apply (9) to find $\mathbf{z}_i$, $i = 1, \dots, r-1$, "one-by-one" using the OpenMP "`for simd`" construct. It is clear that to find $\mathbf{z}_i$ we need the last entry of $\mathbf{z}_{i-1}$. Thus, before calculating the next vector, all threads should be synchronized. Alternatively, we can find last entries of all vectors $\mathbf{z}_i$ and then $s-1$ first entries can be found in parallel without the need for the synchronization of threads.

Similarly, in case of the upper bidiagonal system $R\mathbf{y} = \mathbf{z}$, assuming the same as previously, we get

$$R = \begin{bmatrix} R_s & C & & \\ & R_s & \ddots & \\ & & \ddots & C \\ & & & R_s \end{bmatrix}, \quad R_s = \begin{bmatrix} 1 & r_1 & & \\ & 1 & \ddots & \\ & & \ddots & r_1 \\ & & & 1 \end{bmatrix}, \quad C = \begin{bmatrix} 0 & \dots & \dots & 0 \\ \vdots & & \cdot^{\cdot^{\cdot}} & \vdots \\ 0 & 0 & & \vdots \\ r_1 & 0 & \dots & 0 \end{bmatrix}. \tag{10}$$

The solution of the system (i.e. vectors $\mathbf{y}_i$, $i = 0, \dots, r-1$), satisfies

$$\begin{cases} R_s\mathbf{y}_{r-1} = \mathbf{z}_{r-1} \\ R_s\mathbf{y}_i = \mathbf{z}_i - C\mathbf{y}_{i+1}, \quad i = r-2, \dots, 0. \end{cases} \tag{11}$$

Finally, we get

$$\begin{cases} \mathbf{y}_{r-1} = R_s^{-1}\mathbf{z}_{r-1} \\ \mathbf{y}_i = R_s^{-1}\mathbf{z}_i - r_1 y_{(i+1)s}R_s^{-1}\mathbf{e}_{s-1}, \quad i = r-2, \dots, 0. \end{cases} \tag{12}$$

We have a similar situation as previously, but to find $\mathbf{y}_i$, $i = 0, \dots, r-2$, we need to know first entries of all vectors $\mathbf{y}_i$, $i = 1, \dots, r-1$. Then we can find other entries simultaneously. Such a parallel algorithm requires $16n - 3r - 6s + O(1)$ flops.

3 Implementation and Results of Experiments

Following (9), (12) and (4) we can formulate two OpenMP versions of our algorithm for solving (1). They are presented in Fig. 1. The overall structure of both versions is the same. We assume that the value of s is a power of two, thus each column is properly aligned in memory (lines 14, 32). Moreover, each column occupies a contiguous block in memory. We have two kinds of loops. The first one (lines 12–19) does not utilize vector extensions, but can be executed in parallel. Note that the inner loop (lines 17–18) retrieves successive elements of columns, thus necessary entries can be found in cache memory. Lines 35–37 contain another kind of loop. It is a parallel loop that utilize vector extensions (using the OpenMP "`for simd`" construct). The difference between the versions is relatively small. In case of the first version, there is an implicit barrier after the inner loop 35–37, because we need the last entry of the previous column in the next iteration of the outer loop (lines 30–39). Alternatively, we find all last entries in a sequential loop (lines 26–28) and then the inner loop (lines 35–37) can be launched with the "`nowait`" clause. However, the explicit barrier must be issued after the outer loop (line 39) to ensure that all necessary computations have been completed.

All experiments have been carried out on a server with two Intel Xeon E5-2670 v3 processors (CPU) (totally 24 cores, 2.3 GHz, 256-bit AVX2), 128 GB RAM and a server with Intel Xeon Phi Coprocessor 7120P (KNC, 61 cores with multithreading, 1.238 GHz, 16 GB RAM, 512-bit vector extensions) which is an example of Intel MIC architecture, running under Linux with Intel Parallel Studio ver. 2017. Experiments on Xeon Phi have been carried out using its native mode. We have tested `Sequential` implementation based on (5), (6), (4), optimized automatically by the compiler, and two versions of our parallel algorithm (`Version_1` and `Version_2`). On both architectures (CPU and MIC), in most cases, the best performance of the parallel algorithm is obtained for one thread per core. However, on MIC for larger problem sizes, `Version_1` achieves better performance for two threads per core.

Examples of the results are presented in Figs. 2, 3 and Tables 1, 2. Figures 2, 3 show the execution time for various $n \in \{2^{26} \text{ or } 2^{27}, \ldots, 2^{29} \text{ or } 2^{30}\}$ (depending on the architecture) and r.

It should be observed that the right choice of r and $s = n/r$ is very important for achieving good performance. When a bigger value of r is used, the performance is better only to a certain value of r. A bigger r implies smaller s. Thus we need to find a trade-off between getting the benefits form separating one part of loop and making this loop too long. Both parallel implementations achieve the best performance when the value of r is a small multiple of the number of available cores.

```
1  void method(int n,int r,double a,double d,double *b,double *u){
2      const double r2=(d>0)?((d+sqrt(d*d-4*a))/2):((d-sqrt(d*d-4*a))/2);
3      const double r1=d-r2, tmp=-1/r2;
4      u[0]=1./r2;
5      int s=n/r;
6      double *es=_mm_malloc(s*sizeof(double), 64); es[s-1]=1;
7
8      #pragma omp parallel  // parallel region starts here
9      {
10      double tmp=u[0];
11      #pragma omp for nowait schedule(static)
12      for(int j=0;j<r;j++){
13        double *col;
14        __assume_aligned(col,64); // each column is properly aligned
15        col=&b[j*s];
16        col[0]/=r2;
17        for(int i=1;i<s;i++)
18          col[i]=(col[i]-col[i-1])/r2;
19      }
20      #pragma omp single
21      for(int i=1;i<s;i++)
22        u[i]=u[i-1]*tmp;
23      // implicit barrier
```

```
24  //version 1
25
26
27
28
29
30  for(int j=1;j<r;j++){
31    double *col;
32    __assume_aligned(col,64);
33    col=&b[j*s];
34    double last=b[j*s-1];
35    #pragma omp for simd \
36            schedule(static)
37    for(int i=0;i<s;i++)
38      col[i]-=last*u[i];
39    // implicit barrier
40  }
41  ...the rest of the implementation
```

```
24  //version 2
25
26    #pragma omp single
27    for(int j=1;j<r;j++)
28      b[(j+1)*s-1]-=b[j*s-1]*u[s-1];
29
30    for(int j=1;j<r;j++){
31      double *col;
32      __assume_aligned(col,64);
33      col=&b[j*s];
34      double last=b[j*s-1];
35      #pragma omp for simd nowait \
36              schedule(static)
37      for(int i=0;i<s-1;i++)
38        col[i]-=last*u[i];
39    }
40  #pragma omp barrier
41  ...the rest of the implementation
```

```
41      #pragma omp single
42      b[0]/=(1+r1*u[0]);
43
44      tmp=r1*b[0];
45      #pragma omp for simd schedule(static)
46      for(int i=1;i<n;i++)
47        b[i]-=tmp*u[i];
48
49   } // end of parallel region
50    _mm_free(es);
51  }
```

Fig. 1. Two OpenMP versions of the parallel algorithm (abbreviated)

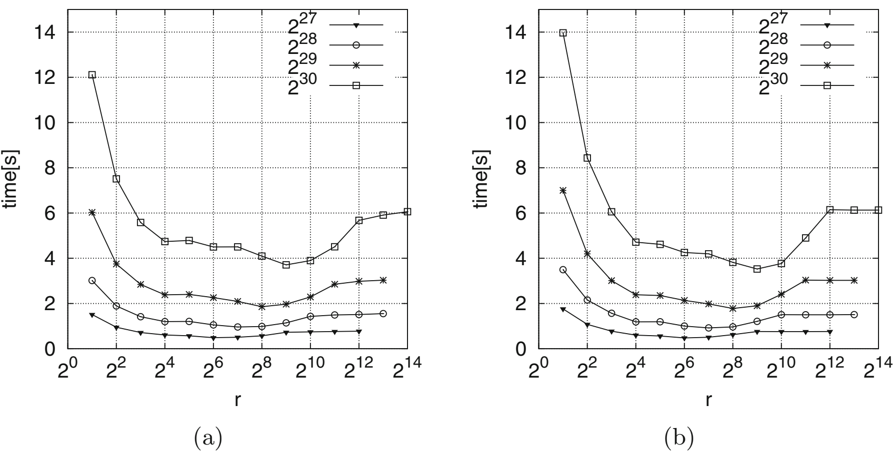

Fig. 2. Execution time for various n and r on CPU: `Version_1` (a), `Version_2` (b)

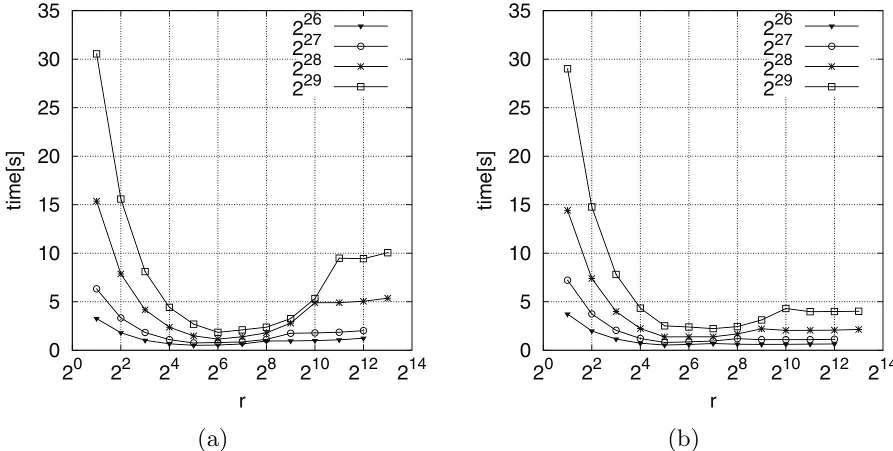

Fig. 3. Execution time for various n and r on MIC: `Version_1` (a), `Version_2` (b)

Tables 1, 2 show the execution time and speedup obtained for optimal values of r. On CPU, `Version_1` achieves better performance for smaller values of n, but on MIC the situation is reversed: `Version_1` achieves better performance for bigger values of n. Better speedup (up to 30) and efficiency can be observed on MIC, where the use of vector extensions is crucial for achieving good performance.

Table 1. Execution time [s] and speedup for optimal values of r (CPU)

	`Sequential`	`Version_1`			`Version_2`		
n	time	r	time	speedup	r	time	speedup
2^{20}	0.0160	2^4	0.0095	1.69	2^5	0.0090	1.79
2^{21}	0.0336	2^4	0.0141	2.37	2^6	0.0119	2.83
2^{22}	0.0704	2^6	0.0233	3.02	2^6	0.0219	3.22
2^{23}	0.1393	2^4	0.0409	3.41	2^8	0.0435	3.21
2^{24}	0.2786	2^4	0.0746	3.74	2^4	0.0783	3.56
2^{25}	0.5572	2^4	0.1410	3.95	2^4	0.1433	3.89
2^{26}	1.1143	2^6	0.2609	4.27	2^6	0.2621	4.25
2^{27}	2.2263	2^6	0.4873	4.57	2^6	0.4755	4.68
2^{28}	4.4509	2^7	0.9606	4.63	2^7	0.9183	4.85
2^{29}	8.9077	2^8	1.8586	4.79	2^8	1.7777	5.01
2^{30}	17.8155	2^9	3.7072	4.81	2^9	3.5225	5.06

Table 2. Execution time [s] and speedup for optimal values of r (MIC)

	`Sequential`	`Version_1`			`Version_2`		
n	time	r	time	speedup	r	time	speedup
2^{20}	0.1095	2^6	0.1503	0.73	2^6	0.1391	0.79
2^{21}	0.2177	2^6	0.1711	1.27	2^7	0.1524	1.43
2^{22}	0.4328	2^5	0.1981	2.19	2^7	0.1705	2.54
2^{23}	0.8582	2^4	0.2458	3.49	2^8	0.2008	4.27
2^{24}	1.7095	2^5	0.2974	5.75	2^8	0.2633	6.49
2^{25}	3.4111	2^5	0.3649	9.35	2^9	0.3671	9.29
2^{26}	6.8127	2^5	0.5095	13.37	2^5	0.5357	12.72
2^{27}	13.6193	2^5	0.7563	18.01	2^5	0.8088	16.84
2^{28}	27.2880	2^6	1.1568	23.59	2^5	1.3780	19.80
2^{29}	54.5765	2^6	1.8531	29.45	2^7	2.2310	24.46

4 The Energy Efficiency

Figure 4 and Table 3 present the exemplary results of our experiments concerning the energy efficiency of our implementations. Data for this plot have been collected on the server with two Intel Xeon E5-2670 v3 processors using Intel's Running Average Power Limit (RAPL) [6]. This interface enables to measure the power consumption for CPUs and DRAMs. Figure 4 shows how the power consumption changes during the execution of the program, which comprises calls to `Sequential`, `Version_1` and `Version_2`, respectively.

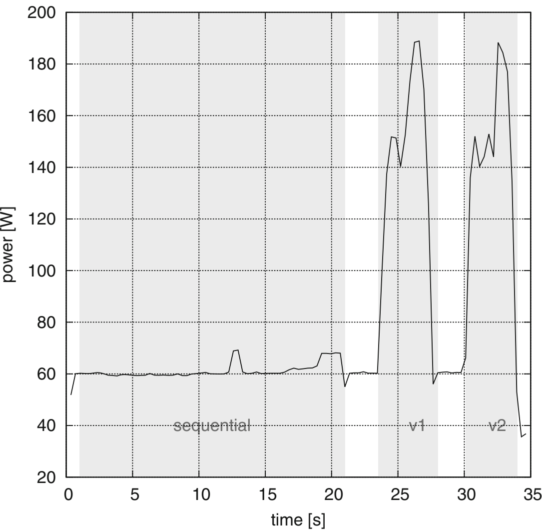

Fig. 4. Total power consumption required by all considered implementations.

We also present the total power consumption of CPUs and DRAMs for $n = 2^{30}$ and the optimal value of r (in case of the parallel implementations). We start with the sequential method (it takes 18.2 s), next we perform `Version_1` (4.2 s) and `Version_2` (3.8 s). It is clear that current power draw during the execution of `Version_1` and `Version_2` is much higher, but it only lasts for a short time.

The power consumption [J] for various problem sizes is presented in Table 3. We can observe that both parallel versions need about 50% of the energy required by `Sequential`.

Table 3. Total power consumption [J] on CPU required by all considered implementations

n	2^{20}	2^{21}	2^{22}	2^{23}	2^{24}	2^{25}	2^{26}	2^{27}	2^{28}	2^{29}	2^{30}
`Sequential`	1.06	2.10	4.10	8.57	17.57	35.88	70.25	139.68	277.47	544.99	1092.91
`Version_1`	1.25	1.98	3.17	5.44	10.26	20.80	40.17	75.03	147.28	291.43	583.29
`Version_2`	0.97	1.95	3.11	6.03	11.02	20.78	39.17	71.32	142.93	281.67	543.25

5 Conclusions and Future Work

We have presented the new vectorized parallel algorithm for solving tridiagonal Toeplitz systems of linear equations. Numerical experiments have shown that it achieves good performance and speedup on multicore and especially manycore

architectures. Moreover, it is more energy efficient than the simple sequential algorithm optimized automatically by the compiler. We plan to show that our approach can be easily implemented on GPUs using OpenACC.

Acknowledgements. The use of computer resources installed at Maria Curie-Skłodowska University in Lublin is kindly acknowledged.

References

1. Belhaj, S., Dridi, M.: A fast algorithm of two-level banded Toeplitz systems of linear equations with application to image restoration. New Trends Math. Sci. **2**, 277–283 (2017). https://doi.org/10.20852/ntmsci.2017.178
2. Chung, K.L., Yan, W.M.: Parallel B-spline surface fitting on mesh-connected computers. J. Parallel Distrib. Comput. **35**, 205–210 (1996). https://doi.org/10.1006/jpdc.1996.0082
3. Chung, K.L., Yan, W.M.: Vectorized algorithms for solving special tridiagonal systems. Comput. Math. Appl. **32**, 1–14 (1996). https://doi.org/10.1016/S0898-1221(96)00203-9
4. Du, L., Sogabe, T., Zhang, S.L.: A fast algorithm for solving tridiagonal quasi-Toeplitz linear systems. Appl. Math. Lett. **75**, 74–81 (2018). https://doi.org/10.1016/j.aml.2017.06.016
5. Garey, L., Shaw, R.: A parallel method for linear equations with tridiagonal Toeplitz coefficient matrices. Comput. Math. Appl. **42**(1), 1–11 (2001). https://doi.org/10.1016/S0898-1221(01)00125-0
6. Khan, K.N., Hirki, M., Niemi, T., Nurminen, J.K., Ou, Z.: RAPL in action: experiences in using RAPL for power measurements. ACM Trans. Model. Perform. Eval. Comput. Syst. **3**(2), 9:1–9:26 (2018). https://doi.org/10.1145/3177754
7. McNally, J.M., Garey, L.E., Shaw, R.E.: A split-correct parallel algorithm for solving tridiagonal symmetric Toeplitz systems. Int. J. Comput. Math. **75**(3), 303–313 (2000). https://doi.org/10.1080/00207160008804986
8. McNally, J.M., Garey, L., Shaw, R.: A communication-less parallel algorithm for tridiagonal Toeplitz systems. J. Comput. Appl. Math. **212**, 260–271 (2008). https://doi.org/10.1016/j.cam.2006.12.001
9. Rojo, O.: A new method for solving symmetric circulant tridiagonal systems of linear equations. Comput. Math. Appl. **20**, 61–67 (1990). https://doi.org/10.1016/0898-1221(90)90165-G
10. Stpiczyński, P.: Solving linear recurrence systems using level 2 and 3 BLAS routines. In: Wyrzykowski, R., Dongarra, J., Paprzycki, M., Waśniewski, J. (eds.) PPAM 2003. LNCS, vol. 3019, pp. 1059–1066. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-24669-5_137
11. Stpiczyński, P., Potiopa, J.: Piecewise cubic interpolation on distributed memory parallel computers and clusters of workstations. In: Fifth International Conference on Parallel Computing in Electrical Engineering (PARELEC 2006), Bialystok, Poland, 13–17 September 2006, pp. 284–289. IEEE Computer Society (2006). https://doi.org/10.1109/PARELEC.2006.68
12. Stpiczyński, P., Potiopa, J.: Solving a kind of boundary-value problem for ordinary differential equations using Fermi—the next generation CUDA computing architecture. J. Comput. Appl. Math. **236**, 384–393 (2011). https://doi.org/10.1016/j.cam.2011.07.028

13. Terekhov, A.V.: A highly scalable parallel algorithm for solving Toeplitz tridiagonal systems of linear equations. J. Parallel Distrib. Comput. **87**, 102–108 (2016). https://doi.org/10.1016/j.jpdc.2015.10.004
14. Vidal, A.M., Alonso, P.: Solving systems of symmetric Toeplitz tridiagonal equations: Rojo's algorithm revisited. Appl. Math. Comput. **219**, 1874–1889 (2012). https://doi.org/10.1016/j.amc.2012.08.030
15. Wang, H.H.: A parallel method for tridiagonal equations. ACM Trans. Math. Softw. **7**(2), 170–183 (1981). https://doi.org/10.1145/355945.355947