Solving a Kind of BVP for Second-Order ODEs Using Novel Data Formats for Dense Matrices

Conference Paper · November 2008

DOI: 10.1109/IMCSIT.2008.4747254 · Source: IEEE Xplore

CITATIONS
4

READS
42

1 author:

Przemyslaw Stpiczynski
Maria Curie-Sklodowska University in Lublin
70 PUBLICATIONS **230** CITATIONS

SEE PROFILE

Proceedings of the International Multiconference on Computer Science and Information Technology pp. 293–296

ISBN 978-83-60810-14-9
ISSN 1896-7094

Solving a Kind of BVP for Second-Order ODEs Using Novel Data Formats for Dense Matrices

Przemysław Stpiczyński
Department of Computer Science, Maria Curie–Skłodowska University
Pl. M. Curie-Skłodowskiej 1, 20-031 Lublin, Poland
Email: przem@hektor.umcs.lublin.pl

***Abstract*—The aim of this paper is to show that a kind of boundary value problem for second-order ordinary differential equations which reduces to the problem of solving tridiagonal systems of linear equations can be efficiently solved on modern multicore computer architectures. A new method for solving such tridiagonal systems of linear equations, based on recently developed algorithms for solving linear recurrence systems with constant coefficients, can be easily vectorized and parallelized. Further improvements can be achieved when novel data formats for dense matrices are used.**

I. Introduction

LET consider the following boundary value problem which arises in many practical applications [10]:

$$-\frac{d^2u}{dx^2} = f(x) \quad \forall x \in [0,1] \tag{1}$$

where

$$u'(0) = 0, \quad u(1) = 0. \tag{2}$$

Numerical solution to the problem (1)–(2) reduces to the problem of solving tridiagonal systems of linear equations. Simple algorithms based on Gaussian elimination achieve poor performance, since they do not fully utilize the underlying hardware, i.e. memory hierarchies, vector extensions and multiple processors, what is essential in case of modern multicore computer architectures [1]. More sophisticated algorithms like *cyclic reduction*, *recursive doubling* [3] and *Wang's method* [15] lead to a substantial increase in the number of floating-point operations and do not utilize cache memory, what is crucial for achieving reasonable performance of parallel computers with a limited number of processors or modern multicore systems [4], [8], [14]. Following this observation, we have introduced a new fully vectorized version of the Wang's method devoted for solving linear recurrences with constant coefficients [12], [13] using two-dimensional arrays. In this paper we show how to apply this algorithm for solving tridiagonal systems of linear equations which arise after discretization of (1)–(2) and how to improve its performance using novel data formats for dense matrices [6].

II. Simple solution

We want to find an approximation of the solution to the problem (1)–(2) in the following grid points

$$0 = x_1 < x_2 < \ldots < x_{n+1} = 1,$$

where $x_i = (i-1)h$, $h = 1/n$, $i = 1, \ldots, n+1$. Let $f_i = f(x_i)$ and $u_i = u(x_i)$. Using the following approximation

$$u''(x_i) \approx \frac{u_{i+1} - 2u_i + u_{i-1}}{h^2} \tag{3}$$

we get

$$-u_{i-1} + 2u_i - u_{i+1} = h^2 f_i \tag{4}$$

for $i = 2, \ldots, n$. When we apply the boundary condition $u_{n+1} = u(1) = 0$, we get a special case of (4)

$$-u_{n-1} + 2u_n = h^2 f_n. \tag{5}$$

To find an approximation of $u''(0)$ using the boundary condition $u'(0) = 0$, let us observe [5] that

$$u''(0) \approx \frac{u(0-h) - 2u_1 + u_2}{h^2} \tag{6}$$

and

$$u'(0) \approx \frac{u_2 - u(0-h)}{2h}. \tag{7}$$

From (7) and $u'(0) = 0$, we get $u(0-h) \approx u_2$, thus using (1) and (6) we have

$$u_1 - u_2 = \frac{1}{2}h^2 f_1. \tag{8}$$

Finally, using (8), (4) and (5) respectively, we get the following system of linear equations [10]:

$$A\mathbf{u} = \mathbf{d} \tag{9}$$

where

$$A = \begin{pmatrix} 1 & -1 & & & \\ -1 & 2 & -1 & & \\ & \ddots & \ddots & \ddots & \\ & & -1 & 2 & -1 \\ & & & -1 & 2 \end{pmatrix} \tag{10}$$

and

$$\mathbf{u} = (u_1, \ldots, u_n)^T, \quad \mathbf{d} = (d_1, \ldots, d_n)^T \tag{11}$$

with $d_1 = \frac{1}{2}h^2 f_1$ and $d_i = h^2 f_i$, $i = 2, \ldots, n$. The solution to the system (9) can be found using Gaussian elimination without pivoting. The matrix A defined by (10) can be factorized as $A = LR$, where L and R are bidiagonal Toeplitz matrices

$$L = \begin{pmatrix} 1 & & & & \\ -1 & 1 & & & \\ & \ddots & \ddots & & \\ & & -1 & 1 & \\ & & & -1 & 1 \end{pmatrix} \qquad (12)$$

and

$$R = \begin{pmatrix} 1 & -1 & & & \\ & 1 & -1 & & \\ & & \ddots & \ddots & \\ & & & 1 & -1 \\ & & & & 1 \end{pmatrix}. \qquad (13)$$

Thus we have to solve two systems of linear equations

$$L\left(y_1, \ldots, y_n\right)^T = \mathbf{d} \qquad (14)$$

and then

$$R\mathbf{u} = \left(y_1, \ldots, y_n\right)^T. \qquad (15)$$

The first stage (forward reduction) of the algorithm for solving (9), namely finding the solution to the system (14), can be done using

$$\begin{cases} y_1 = d_1 \\ y_i = d_i + y_{i-1} \quad \text{for } i = 2, \ldots, n. \end{cases} \qquad (16)$$

Then (second stage, back substitution) we find the solution to the system (15) using

$$\begin{cases} u_n = y_n \\ u_i = y_i + u_{i+1} \quad \text{for } i = n-1, n-2, \ldots, 1. \end{cases} \qquad (17)$$

Note that there is no need to allocate an array for the elements y_i, $i = 1, \ldots, n$. Assuming that initially the array `u` contains the elements of the vector $\mathbf{d}$, namely `u(i)`$= d_i$, the algorithm based on (16) and (17) can be expressed as the following simple loops

```
do i=2,n
  u(i)=u(i)+u(i-1)
end do
do i=n-1,1,-1
  u(i)=u(i)+u(i+1)
end do
```

Unfortunately, this simple algorithm cannot be vectorized or parallelized, so it can utilize only a small fraction of the theoretical peak performance of modern multicore multiprocessors.

III. "Divide and conquer" approach

It is clear that (16) and (17) are the special cases of the following more general problem of solving *linear recurrence systems with constant coefficients*

$$y_k = \begin{cases} 0 & \text{for } k \le 0 \\ d_k + \sum_{j=1}^{m} a_j y_{k-j} & \text{for } 1 \le k \le n, \end{cases} \qquad (18)$$

where simply $m = 1$, $a_1 = 1$ and in case of (17), the unknowns are calculated from n down to 1. In [12] we introduced a new algorithm based on Level 2 and 3 BLAS routines [3], which is a generalization of our earlier fully vectorized algorithm [13]. It can be efficiently implemented on various shared-memory parallel computers [11]. In this section we present a very brief description of the algorithm applied for $m = 1$, $a_1 = 1$ and show how to improve its performance using novel data formats for dense matrices.

A. Simple 2D-array algorithm

The main idea of the algorithm is to rewrite (18) as a block-bidiagonal system of linear equations [12], [13]. Without loss of generality, let us assume that there exist two positive integers r and s such that $rs \le n$ and $s > 1$. The method can be used for finding $y_1, \ldots, y_{rs}$. To find $y_{rs+1}, \ldots, y_n$, we use (16) directly. Then for $j = 1, \ldots, r$, we define vectors

$$\mathbf{d}_j = \left(d_{(j-1)s+1}, \ldots, d_{js}\right), \quad \mathbf{y}_j = \left(y_{(j-1)s+1}, \ldots, y_{js}\right) \in \mathbb{R}^s$$

and find all $\mathbf{y}_j$ using the following formula

$$\begin{cases} \mathbf{y}_1 = L_s^{-1}\mathbf{d}_1 \\ \mathbf{y}_j = L_s^{-1}\mathbf{d}_j + y_{(j-1)s}\mathbf{e} \quad \text{for } j = 2, \ldots, r \end{cases} \qquad (19)$$

where $\mathbf{e} = (1, \ldots, 1)^T$ and the matrix $L_s \in \mathbb{R}^{s \times s}$ is of the same form as L given by (12). However, it is better to find the matrix

$$Y = (\mathbf{y}_1, \ldots, \mathbf{y}_r) \in \mathbb{R}^{s \times r}$$

instead of individual vectors $\mathbf{y}_j$, $j = 1, \ldots, r$. Indeed, for $k = 2, \ldots, s$ we perform

$$Y_{k,1:r} \leftarrow Y_{k,1:r} + Y_{k-1,1:r}, \qquad (20)$$

and this operation (simplified `AXPY` [3]) can be vectorized and parallelized. Then we use (19) to find the last entry of each vector $\mathbf{y}_j$, $j = 2, \ldots, r$, and finally we use (19) to find $s-1$ first entries of these vectors (note that this operation can also be vectorized and parallelized). This approach has one disadvantage: the number of floating-point operations required by the algorithm is twice as many as for the simple algorithm based on (16) (see [12]). Analogously, we can find the solution to (15) using

$$\begin{cases} \mathbf{u}_n = R_s^{-1}\mathbf{y}_r \\ \mathbf{u}_j = R_s^{-1}\mathbf{y}_j + u_{js+1}\mathbf{e} \quad \text{for } j = r-1, \ldots, 1, \end{cases} \qquad (21)$$

where

$$\mathbf{u}_j = \left(u_{(j-1)s+1}, \ldots, u_{js}\right) \in \mathbb{R}^s$$

and $R_s \in \mathbb{R}^{s \times s}$ is of the same form as R given by (13). Note that there is no need to allocate an array for the matrix Y. We can allocate a two-dimensional array `u` as the storage for the matrix

$$U = \begin{pmatrix} u_{11} & \ldots & u_{1r} \\ \vdots & & \vdots \\ u_{s1} & \ldots & u_{sr} \end{pmatrix} \in \mathbb{R}^{s \times r}. \qquad (22)$$

and assign `u(i,j)`$=d_{(i-1)s+j}$, $i = 1, \ldots, s$, $j = 1, \ldots, r$. The The leading dimension of the array (`LDA` for short) can be `LDA` $= s$, however the performance of the algorithm can be improved by the use of *leading dimension padding* [9], what

```
        1 10 19 28 37 46 55 64 73 82
        2 11 20 29 38 47 56 65 74 83
        3 12 21 30 39 48 57 66 75 84
        4 13 22 31 40 49 58 67 76 85
    A = 5 14 23 32 41 50 59 68 77 86
        6 15 24 33 42 51 60 69 78 87
        7 15 25 34 43 52 61 70 79 88
        8 17 26 35 44 53 62 71 80 89
        *  *  *  *  *  *  *  *  *  *
```

Fig. 1. Standard column major storage of a 8×10 array with `LDA=9`.

```
         1  5  9 13 | 33 37 41 45 | 65 69  *  *
         2  6 10 14 | 34 38 42 46 | 66 70  *  *
         3  7 11 15 | 35 39 43 47 | 67 71  *  *
         4  8 12 16 | 36 40 44 48 | 68 72  *  *
    A = ---------------------------------------
        17 21 25 29 | 49 53 57 61 | 73 77  *  *
        18 22 26 30 | 50 54 58 62 | 74 78  *  *
        19 23 27 31 | 51 55 59 63 | 75 79  *  *
         *  *  *  * |  *  *  *  * |  *  *  *  *
```

Fig. 2. Square blocked full column major order of a 7×10 matrix with 4×4 blocks.

means that we insert unused array elements between columns (Fortran) or rows (C/C++) of a multidimensional array by increasing the leading dimension of the array (Figure 1). Usually, it is sufficient to set $\texttt{LDA} = M$, where $M \geq s$ and M is *not* a multiple of a power of two.

The algorithm for solving (9) can be easily parallelized. Each processor can be responsible for computing a block of columns of the matrix U. The first stage (forward reduction) proceeds as follows. During the first (parallel) step, for $k = 2, \ldots, s$, each processor applies the operation (20) restricted to its own columns. Then the last entries of all vectors $\mathbf{y}_j$ are calculated sequantially. Finaly, each processor uses (19) to find $s - 1$ first entries of its columns. Similarly we can parallelize the second stage of the algoritm (back substitution).

The value of the parameter r should be at least $r = vp$, where p is the number of processors and v is the length of vector registers in a vector processor (if applicable). However, the best performance is achieved if $r = O(\sqrt{n})$ and $s = O(\sqrt{n})$ [12]. It should be noticed that sometimes the performance of the algorithm can be unsatisfactory. When the standard column major storage is used, successive elements of vectors $Z_{k,*}$ and $Z_{k-1,*}$ in (20) do not occupy contiguous memory locations, thus may not belong to the same *cache line* and *cache misses* may occur [9].

B. Using novel data formats

Suppose that we have a $m \times n$ matrix A. The matrix can be partitioned as follows

$$A = \begin{pmatrix} A_{11} & \ldots & A_{1n_g} \\ \vdots & & \vdots \\ A_{m_g1} & \ldots & A_{m_gn_g} \end{pmatrix}. \qquad (23)$$

Each block A_{ij} contains a submatrix of A and it is stored as a square $n_b \times n_b$ block which occupies a contiguous block of memory (Figure 2). The value of the blocksize n_b should be chosen to map nicely into L1 cache [6], [7]. Note that when $n_g n_b \neq n$ or $m_g n_b \neq m$, then $A_{1n_g}, \ldots, A_{m_gn_g}$ or $A_{m_g1}, \ldots, A_{m_gn_g}$ are not square blocks and some memory will be wasted ('`*`' in Figure 2). As mentioned in [6] the novel data format can be represented by using a four dimensional array where `A(i,j,k,l)` refers to the `(i,j)` element within the `(k,l)` block.

Let us observe that the algorithm introduced in the previous subsection can be implemented using a matrix representation based on the *square blocked full data format* presented above. The matrix U can be partitioned as (23). Then the algorithm will operate on small n_b-vectors within $(n_b \times n_b)$-blocks and the number of cache misses will be substantially reduced.

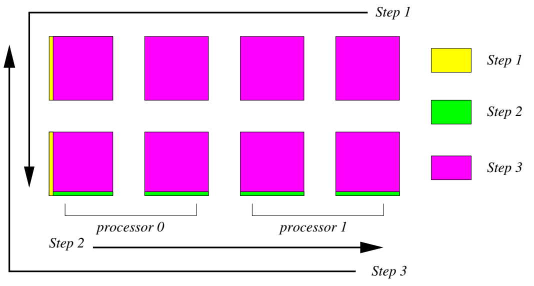

Fig. 3. The first stage using square blocked data format.

The first and the third step of the first stage can be done in parallel, while the second is sequential. To reduce the number of cache misses, blocks should be computed in the appropriate order. During the first step, columns of blocks should be computed from right to left and 'top-down' within each block column. Then during the second step we update elements in the lower row of blocks and finally (the third step) we update blocks from right to left and 'bottom-up' within each column of blocks (see Figure 3).

IV. Results of experiments

Let us consider the algorithms described in the previous sections:

SEQ	sequential algorithm based on (16) and (17),
DC	divide and conquer algorithm based on (19) and (21) using two dimensional arrays,
ND	divide and conquer algorithm based on (19) and (21) using the square blocked full data format.

The algorithms have been implemented in Fortran 95 with OpenMP directives [2] and all vector operations have been implemented using array section assignments. The experiments have been carried out on a dual processor Quad-Core Xeon (2.33 GHz, 12 MB L2 cache, 4 GB RAM running under Linux) workstation using the g95 Fortran Compiler.

We have measured the execution time (in seconds), the performance (in megaflops) of the algorithms for various values of n, r, s and blocksizes n_b and the speedup relative to Algorithm SEQ for optimal n_b, r, s and various numbers of processors to find out how we can improve the performance of the simple scalar code (Algorithm SEQ) by the use of advanced computer hardware. Exemplary results are presented in Figures 4, 5 and Table I. The results of experiments can be summarized as follows.

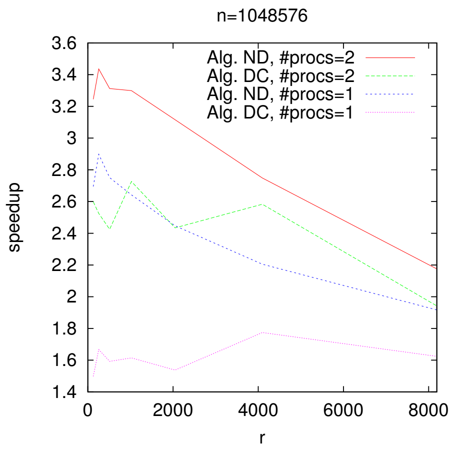

Fig. 4. Speedup of the algorithms DC and ND relative to Algorithm SEQ for various r.

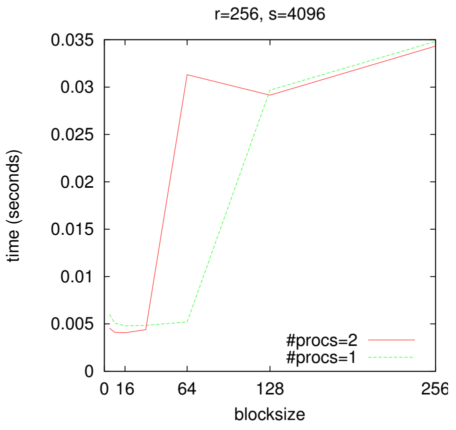

Fig. 5. Execution time of Algorithm ND for various blocksizes n_b.

TABLE I
EXECUTION TIME, SPEEDUP AND PERFORMANCE OF THE ALGORITHMS SEQ, DC, ND FOR $n = 1048576$

	Time		Speedup		Mflops	
alg.	#p=1	#p=2	#p=1	#p=2	#p=1	#p=2
SEQ	1.40E-2	1.40E-2	–	–	225	225
DC	7.96E-3	5.49E-3	1.75	2.55	659	954
ND	4.80E-3	4.06E-3	2.89	3.43	1093	1286

Both introduced algorithms (DC and ND) run faster than Algorithm SEQ. The performance of Algorithm ND depends on the blocksize n_b, so it should be chosen carefully. The optimal blocksize is $n_b = 16$ (Figure 5). When the blocksize n_b is too large, the performance of the algorithm decreases dramatically because blocks do not fit into L1 cache and cache misses occur. For the optimal blocksize, the algorithm achieves reasonable speedup relative to Algorithm SEQ. Moreover, in case of the simple algorithm, fast vector instructions cannot be used without manual optimization of the simple scalar code. Although all optimization switches have been turned on, the compiler has produced rather slow output for the scalar code. The values of the parameters r and s should be chosen to minimize the number of flops required by the algorithms, however $r = s$ is rather a good choice in case of Algorithm DC and $r = \sqrt{n}/4$ in case of Algorithm ND. Finally, comparing the performance of the new algorithm which utilizes the novel data format with the performance of the divide and conquer algorithm which uses standard Fortran two-dimensional arrays with the column major storage order, we can observe that Algorithm ND is faster than Algorithm DC.

V. Conclusion

We have shown that the performance of the simple algorithm for solving a kind of boundary value problem for second-order ordinary differential equations which reduces to the problem of solving tridiagonal systems of linear equations can be highly improved by the use of the divide and conquer vectorized algorithm which operates on the square blocked full data format for dense matrices. The algorithm can also be parallelized, thus it should be suitable for novel multicore architectures.

References

[1] A. Buttari, J. Dongarra, J. Kurzak, J. Langou, P. Luszczek, and S. Tomov, "The impact of multicore on math software," *Lecture Notes in Computer Science*, vol. 4699, pp. 1–10, 2007.
[2] R. Chandra, L. Dagum, D. Kohr, D. Maydan, J. McDonald, and R. Menon, *Parallel Programming in OpenMP*. San Francisco: Morgan Kaufmann Publishers, 2001.
[3] J. Dongarra, I. Duff, D. Sorensen, and H. Van der Vorst, *Numerical Linear Algebra for High Performance Computers*. Philadelphia: SIAM, 1998.
[4] E. Elmroth, F. Gustavson, I. Jonsson, and B. Kågström, "Recursive blocked algorithms and hybrid data structures for dense matrix library software," *SIAM Rev.*, vol. 46, pp. 3–45, 2004.
[5] G. Golub and J. M. Ortega, *Scientific Computing: An Introduction with Parallel Computing*. Academic Press, 1993.
[6] F. G. Gustavson, "New generalized data structures for matrices lead to a variety of high performance algorithms," *Lect. Notes Comput. Sci.*, vol. 2328, pp. 418–436, 2002.
[7] ——, "High-performance linear algebra algorithms using new generalized data structures for matrices," *IBM J. Res. Dev.*, vol. 47, pp. 31–56, 2003.
[8] ——, "The relevance of new data structure approaches for dense linear algebra in the new multi-core / many core environments," *Lecture Notes in Computer Science*, vol. 4967, pp. 618–621, 2008.
[9] M. Kowarschik and C. Weiss, "An overview of cache optimization techniques and cache-aware numerical algorithms," *Lecture Notes in Computer Science*, vol. 2625, pp. 213–232, 2003.
[10] L. R. Scott, T. Clark, and B. Bagheri, *Scientific Parallel Computing*. Princeton University Press, 2005.
[11] P. Stpiczyński, "Numerical evaluation of linear recurrences on various parallel computers," in *Proceedings of Aplimat 2004, 3rd International Conference, Bratislava, Slovakia, February 4–6, 2004*, M. Kovacova, Ed. Technical Univerity of Bratislava, 2004, pp. 889–894.
[12] ——, "Solving linear recurrence systems using level 2 and 3 BLAS routines," *Lecture Notes in Computer Science*, vol. 3019, pp. 1059–1066, 2004.
[13] P. Stpiczyński and M. Paprzycki, "Fully vectorized solver for linear recurrence systems with constant coefficients," in *Proceedings of VECPAR 2000 – 4th International Meeting on Vector and Parallel Processing, Porto, June 2000*. Facultade de Engerharia do Universidade do Porto, 2000, pp. 541–551.
[14] P. Stpiczyński, "Evaluating linear recursive filters using novel data formats for dense matrices," *Lecture Notes in Computer Science*, vol. 4967, pp. 688–697, 2008.
[15] H. Wang, "A parallel method for tridiagonal equations." *ACM Trans. Math. Softw.*, vol. 7, pp. 170–183, 1981.