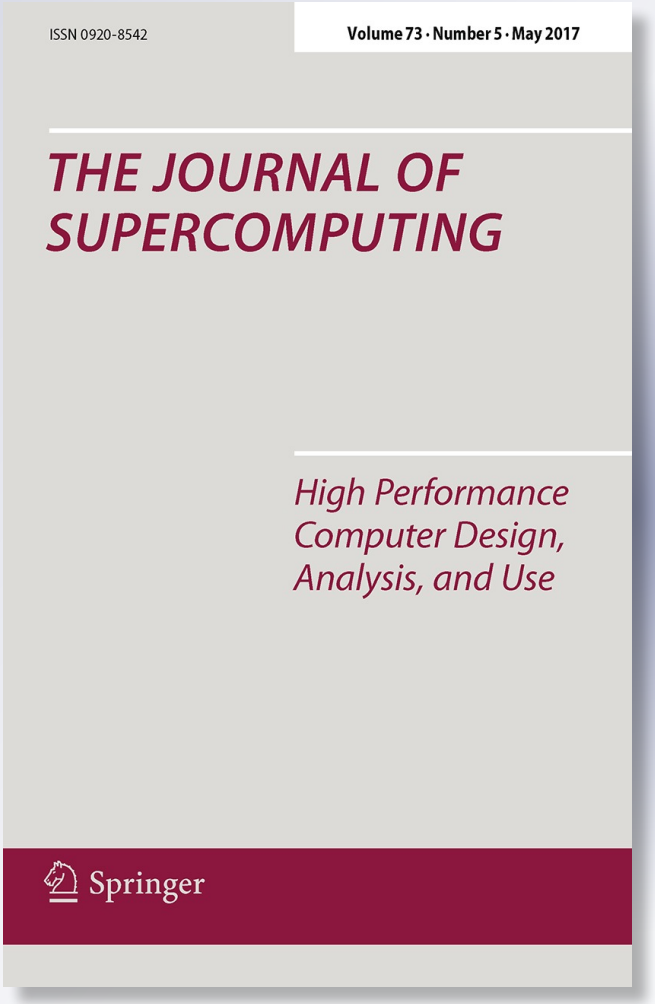
A parallel solving method for block-tridiagonal equations on CPU–GPU heterogeneous computing systems

Wangdong Yang, Kenli Li & Keqin Li

The Journal of Supercomputing
An International Journal of High-Performance Computer Design, Analysis, and Use

ISSN 0920-8542
Volume 73
Number 5

J Supercomput (2017) 73:1760-1781
DOI 10.1007/s11227-016-1881-x



Springer

J Supercomput (2017) 73:1760–1781
DOI 10.1007/s11227-016-1881-x

A parallel solving method for block-tridiagonal equations on CPU–GPU heterogeneous computing systems

Wangdong Yang[1] · **Kenli Li**[1] · **Keqin Li**[1,2]

Published online: 22 September 2016

Abstract Solving block-tridiagonal systems is one of the key issues in numerical simulations of many scientific and engineering problems. Non-zero elements are mainly concentrated in the blocks on the main diagonal for most block-tridiagonal matrices, and the blocks above and below the main diagonal have little non-zero elements. Therefore, we present a solving method which mixes direct and iterative methods. In our method, the submatrices on the main diagonal are solved by the direct methods in the iteration processes. Because the approximate solutions obtained by the direct methods are closer to the exact solutions, the convergence speed of solving the block-tridiagonal system of linear equations can be improved. Some direct methods have good performance in solving small-scale equations, and the sub-equations can be solved in parallel. We present an improved algorithm to solve the sub-equations by thread blocks on GPU, and the intermediate data are stored in shared memory, so as to significantly reduce the latency of memory access. Furthermore, we analyze cloud resources scheduling model and obtain ten block-tridiagonal matrices which are produced by the simulation of the cloud-computing system. The computing performance of solving these block-tridiagonal systems of linear equations can be improved using our method.

✉ Wangdong Yang
yangwangdong@163.com

Kenli Li
lkl@hnu.edu.cn

Keqin Li
lik@newpaltz.edu

1 College of Information Science and Engineering, Hunan University, Changsha 410008, Hunan, China

2 Department of Computer Science, State University of New York, New Paltz, NY 12561, USA

Keywords Block tridiagonal · Linear equations · Sparse matrix-vector multiplication · Solving

1 Introduction

1.1 Motivation

Solving block-tridiagonal systems is one of the key issues in numerical simulations of many scientific and engineering problems. The block-tridiagonal equations are arisen from classical finite-difference discretizations of two-dimensional separable elliptic equations. For instance, the Fourier-transformed partial differential equations (PDEs) have block-tridiagonal characteristics when the underlying equations involve at most second-order derivatives for three-dimensional systems, where two coordinates are angular and one is radial. Particularly, the optimal resource allocation schemes of cloud-computing platform are found by solving the block-tridiagonal equations, which are generated by queuing model.

The frequency of the processor is difficult to significantly increase due to the limitation of the current very large-scale integration (VLSI) technology. In response, multicore processors and/or specialized hardware accelerators have been designed and developed by most hardware manufactures [1]. The new parallel characteristics of new architectures are used and exploited to improve the performance of programs. The appearance of GPUs for general-purpose computing platforms offers powerful parallel processing capabilities at a low cost, and GPUs are widely used in the field of high-performance computing.

There are two methods to solve the block-tridiagonal equations, which are the direct methods and iterative methods. The solutions are obtained by elimination for direct methods, and some zero elements may become non-zero elements in the process of elimination. Furthermore, the error of the solutions will increase with more elimination operations. The iterative methods are suitable for large sparse matrices, such as Jacobi method, Gauss–Seidel method, GMRES, and Conjugate gradient, etc, and have no cumulative error. However, the stability of the iterative methods is poor, and the iterations may be slow convergence for some sparse matrices.

1.2 Our contributions

For block-tridiagonal system of linear equations, we present a solving method which mixes direct and iterative methods. Our method generates a sequence of approximate solutions $\{x^k\}$ in the same way as the conventional iteration methods, where x^k expresses the approximate solutions of the kth iteration. The conventional iteration methods essentially involve a matrix A only in the context of matrix-vector multiplication and do not make full use of tridiagonal characteristics of block-tridiagonal matrices, so as to result in slow convergence. Non-zero elements are mainly concentrated in the blocks on the main diagonal for most block-tridiagonal matrices, and the blocks up and down the main diagonal have little non-zero elements. In our method, the submatrices on the main diagonal are solved by the direct methods in the iteration

processes. Because the approximate solutions obtained by the direct methods are closer to the exact solutions, the convergence speed of solving the block-tridiagonal system of linear equations can be improved. Some direct methods have good performance in solving small-scale equations, and the sub-equations can be solved in parallel. We present an improved algorithm to solve the sub-equations by thread blocks on GPU, and the intermediate data are stored in shared memory, so as to significantly reduce the latency of memory access. Therefore, the computational complexity of the hybrid method is not increased and the convergence speed can be accelerated.

According to our experiments on ten test cases, the performance improvement using our algorithm is very effective and noticeable. The average number of iterations is reduced by 283.15 and 18.34 % using our method compared with CG and BiCGSTAB of PARALUTION library, respectively, and the performance using our method is better than those of the commonly used iterative and direct methods, and the performance of solving the test cases on GPU is improved by 26.98, 11.52, and 9.25 % using our method compared with CG, BiCGSTAB of PARALUTION library, and cuSolverSP of CUDA.

The remainder of the paper is organized as follows. In Sect. 2, we review the related research on solving block-tridiagonal system of linear equations. In Sect. 3, we present an introduction to CUDA. In Sect. 4, we develop the method of solution for solving block-tridiagonal matrices. In Sect. 4.6, we describe parallel implementation of our method on GPU. In Sect. 5, we demonstrate the performance comparison results in our extensive experiments. In Sect. 6, we conclude the paper.

2 Related work

Block-tridiagonal matrices have a central diagonal and two adjacent, which are located at a distance m from the center, and m is the size of the block matrix. There has been considerable work in developing solution algorithms for block-tridiagonal matrices. For a block-tridiagonal matrix A, it is possible to obtain an exact inverse (direct solution) with no fill-in using the well-known Thomas [2] serial algorithm, which is easily generalized for block sizes $m = 1$. While this is the fastest algorithm on a serial computer, it is not parallelizable, since each solution step in the algorithm depends on the preceding one. Many authors have considered efficient parallel block solvers for scalar block ($m = 1$) matrices based on cyclic reduction [3]. Cyclic reduction was first described by Heller [4] for block-tridiagonal systems, although an efficient parallel code was not described. The new code BCYCLIC described here fills a software gap in the available codes for solving tridiagonal systems with large ($m = 1$), dense blocks [5]. Reference [6] analyzed the projection of four known parallel tridiagonal system solvers: cyclic reduction [7,8], recursive doubling [9], Bondeli's divide and conquer algorithm [10], and Wang's partition method [11]. Cyclic reduction and recursive doubling focus on a line grain of parallelism, where each processor computes only one equation of the system. Reference [11] developed a partition method with a coarser grain of parallelism. ScaLAPACK [12] provided another method for efficiently solving dense block-tridiagonal systems. Block factorization and solution based on ScaLAPACK are currently implemented in the SIESTA [13] MHD equilibrium code.

This technique scales well with processor count only for very large matrix block sizes. For matrix blocks of interest for small 3D MHD problems ($m = 300$), scalability was found to be limited to about 5–10 processors.

Some works [14–24] presented how hardware and software can work in concert on scalable multiprocessor systems with a number of illustrative matrix-based examples and applications, and provided a historical perspective and relevant context to numerical computation on CPU–GPU heterogeneous computing systems. Some works [25–33] discussed how processor technologies can help scientific computing applications which involve matrix operations. Two-level parallelization [34] was introduced to solve a massive block-tridiagonal matrix system. One-level was used for distributing blocks, whose size is as large as the number of block rows due to the spectral basis, and the other level is used for parallelizing in the block row dimension.

Some researchers have investigated parallel algorithms for solving systems of linear algebraic equations with block-tridiagonal matrices using iteration methods. In [35, 36], the Krylov method is used [37], and special preconditioning procedures, which account for the structure of a block-tridiagonal matrix, are used to increase the rate of convergence. The efficiency of the algorithms described above depends on a great extent on the size of the matrix blocks. As m increases, the local calculations become a greater part of the calculation, and consequently, the efficiency of the method increases. This determines the domain of applicability of presently available parallel algorithms. An important case in practice is the case when the matrix block size is not large, i.e., does not exceed several hundreds. This is because, for such a matrix, one can effectively apply a cost-effective variant of the Gaussian method with overheads of $O(m^3 n)$ [38]. For a large cloud-computing platform, there are tens of thousands of computing resources which have several or dozens of execution states, and state matrix generated by state transition is a block-tridiagonal matrix. Therefore, for this case, it is necessary to develop a parallel algorithm with comparable overheads, which will also enable the effective implementation of thousands of processors.

There are two common methods for solving block-tridiagonal equations: using a block-tridiagonal factorization , or treating the matrix as a band matrix [39]. Some preconditioners are used to improve the performance and robustness of solving block-tridiagonal equations. For a symmetric positive definite block-tridiagonal matrix A, using the Cholesky decomposition once can reduce the operation count [39,40]. Ruggiero and Galligani [41] considered a new form of the arithmetic mean method for solving large block-tridiagonal linear systems and proposed a parallel implementation using the iterative method and the preconditioner on a multi-vector computer. [42] presented an efficient blockwise update scheme for the QR decomposition of block-tridiagonal and block Hessenberg matrices, which come up in generalizations of the Krylov space solvers to block methods for linear systems of equations with multiple right-hand sides, and could improve the performance and stability of block Krylov space solvers. Incomplete LU factorization is a common way to construct preconditioner for sparse linear systems and is known to give excellent results for this class of problems. Some theory for block ILU preconditioner is discussed in [37,43]. This type of approach needs to approximate inverse of pivot blocks.

Work on SpMV [44–46], LU factorization [47], and general hybridized linear algebra routines [48], including tridiagonal solvers on GPUs, has been studied and

optimized in the recent literature [49–52]. For GPUs provided by NVIDIA, NVIDIA provided CUDA (compute unified device architecture) to improve the development efficiency of parallel program [53]. CuSolver library [54] provided by CUDA tools includes some implementation codes of the direct solving algorithms based on the cuBLAS and cuSPARSE libraries [55]. PARALUTION [56] proposed a sparse linear algebra library with focus on exploring fine-grained parallelism, targeting modern processors and accelerators, including multi/many-core CPU and GPU platforms, and provided a portable library for iterative sparse methods on the state-of-the-art hardware. [57] presented the parallel algorithm of the generalized minimal residual iterative method using the Compute Unified Device Architecture programming language and the MPI parallel environment. The GREMLINS code [58] had been developed for solving large sparse linear systems on distributed grids. [59] analyzed the performance of the GREMLINS code obtained with several libraries for solving the linear subsystems and compared with that of the widely used PETSc library [60] that enables one to develop portable parallel applications.

The conventional iteration methods essentially involve a matrix only in the context of matrix-vector multiplication and do not make full use of tridiagonal characteristics of block-tridiagonal matrices, so as to result in slow convergence. Furthermore, the utilization of shared memory has a large effect on the performance of solving on GPU. The whole matrix is stored in the global memory using the conventional GPU-based methods that lead to larger data access latency. However, we present an improved algorithm to solve the sub-equations by thread blocks on GPU, and the intermediate data are stored in shared memory, so as to significantly reduce the latency of memory access to improve solving performance. Although PARALUTION provided some iterative and direct solving algorithms on GPU, no hybrid iterative and direct solving algorithms for block-tridiagonal equations were provided. We present the hybrid solving method which has faster convergence speed and higher parallel efficiency.

3 An introduction of CUDA

GPUs are widely used in parallel computing, because there are more cores and coprocessors to speed the computing in more and more computing systems. CUDA can improve the development efficiency of parallel program. The CUDA heterogeneous programming model is shown in Fig. 1. The CUDA heterogeneous programming model assumes that the CUDA threads execute on a physically separate device that operates as a coprocessor to the host running the main program. This is the case, for example, when the kernels execute on a GPU and the rest of the main program executes on a CPU. The number of threads is decided by the programmer to be executed. A collection of threads (called a block) runs on a multiprocessor at a given time. Multiple blocks can be assigned to a single multiprocessor and their execution is time-shared. For heterogeneous computing systems with CPUs and GPUs, each core of CPU can independently perform its instructions and data, which is called the MIMD model. If there is not dependency, it does not need synchronization between threads on various cores of CPU. For multicore CPUs, each core can be scheduled independently to perform the threads. Therefore, the partitioning methods for multicore CPU do not consider the uniformity of NNZ of rows in the block for SpMV. However, the basic

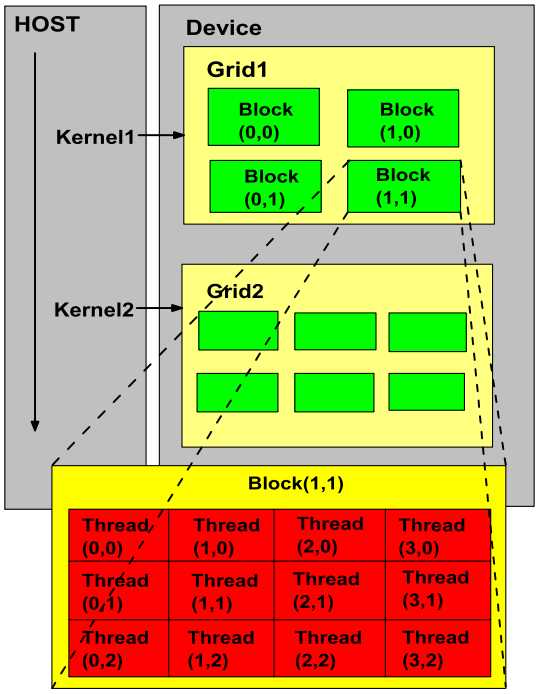


Fig. 1 CUDA heterogeneous programming model

computing unit of a GPU is called streaming multiprocessor (SM). As a component at the bottom of the independent hardware structure, SM can be seen as an SIMD processing unit. Each SM contains some scalar processors (SP) and special function units (SFU) [53]. It needs synchronization between SPs of the same SM on GPU [53]. Therefore, the inequality of the load of different threads will have a larger impact on performance.

4 The solving method for blocked tridiagonal matrix

4.1 The compressed storage format

For many scientific and engineering applications, these sparse matrices may have various sparsity characteristics. Different sparse matrices with different sparse distributions should be stored by the most appropriate compressed storage format.

The 6-by-6 sparse matrix A shown below is used as a running example in this section:

$$A = \begin{pmatrix} 6 & 0 & 2 & 0 & 0 & 0 \\ 1 & 9 & 7 & 0 & 0 & 0 \\ 0 & 5 & 4 & 3 & 0 & 0 \\ 0 & 1 & 0 & 8 & 7 & 3 \\ 0 & 0 & 6 & 2 & 11 & 4 \\ 0 & 0 & 0 & 0 & 6 & 21 \end{pmatrix}.$$

4.1.1 COO storage format

The coordinate (COO) format is a particularly simple storage scheme with a triplet (**row**, **column**, **value**). The arrays **row**, **column**, and **value** store the row indices, column indices, and values of the non-zero elements in a matrix, respectively. For the example sparse matrix A, we have

$$\begin{aligned}
&\mathbf{row} = (0, 0, 1, 1, 1, 2, 2, 2, 3, 3, 3, 3, 4, 4, 4, 4, 5, 5),\\
&\mathbf{column} = (0, 2, 0, 1, 2, 1, 2, 3, 1, 3, 4, 5, 2, 3, 4, 5, 4, 5),\\
&\mathbf{value} = (6, 2, 1, 9, 7, 5, 4, 3, 1, 8, 7, 3, 6, 2, 11, 4, 6, 21).
\end{aligned}$$

4.1.2 CSR storage format

The compressed sparse row (CSR) format is a popular and general-purpose sparse matrix representation scheme. CSR explicitly stores column indices and non-zero values in arrays **Aj** and **Av**. The third array **Ap** represents the starting position of each row in the array **Aj**. For an n-by-m matrix, **Ap** has length $n + 1$ and stores the offset of the ith row in **Ap**$[i]$. The value of the last element is NNZ, which is the number of non-zero elements. For the example sparse matrix A, we have

$$\begin{aligned}
&\mathbf{Ap} = (0, 2, 5, 8, 12, 16, 18),\\
&\mathbf{Aj} = (0, 2, 0, 1, 2, 1, 2, 3, 1, 3, 4, 5, 2, 3, 4, 5, 4, 5),\\
&\mathbf{Av} = (6, 2, 1, 9, 7, 5, 4, 3, 1, 8, 7, 3, 6, 2, 11, 4, 6, 21).
\end{aligned}$$

4.1.3 DIA storage format

For the n-order sparse matrix A with k diagonal line, it can greatly reduce the storage overhead when only the non-zero elements in diagonal are stored. DIA format includes two parts. The first part is an $n \times k$ matrix **D** to store the value on the diagonal. The second part is an array **offset** to store k elements, which are the offsets of a diagonal with respect to the main diagonal. For the sparse matrix A, we have

$$\mathbf{D} = \begin{pmatrix} * & * & 6 & 0 & 2 \\ * & 1 & 9 & 7 & 0 \\ 1 & 5 & 4 & 3 & 0 \\ 6 & 0 & 8 & 7 & 3 \\ 0 & 2 & 11 & 4 & * \\ 0 & 6 & 21 & * & * \end{pmatrix}, \mathbf{Offset} = (-2, -1, 0, 1, 2)\,.$$

DIA is perfect format for the compression and storage of diagonal matrix, but not good for dispersed sparse matrix [44]. The main reason is that the presence of non-zero element in most of diagonals will lead to too much columns in matrix **D**. As the

matrix with non-zero elements in all diagonals, in this extreme case, the data matrix **D** will involves $2n-1$ columns, and it will cost double storage volume than the original matrix.

4.2 The blocked tridiagonal matrix

The block-tridiagonal matrix A is shown as follows:

$$A = \begin{pmatrix} D_1 & U_1 & & & & & & \\ L_2 & D_2 & U_2 & & & & & \\ & L_3 & D_3 & U_3 & & & & \\ & & \ddots & \ddots & \ddots & & & \\ & & & \ddots & \ddots & \ddots & & \\ & & & & L_{m-1} & D_{m-1} & U_{m-1} \\ & & & & & L_m & D_m \end{pmatrix},$$

where $D_1, D_2, \ldots, D_m$, $L_2, L_3, \ldots, L_m$, and $U_1, U_2, \ldots, U_{m-1}$ are $k \times k$ squares, and the other positions in B are zero. Assume that the number of rows in B is n, where n is $m \times k$.

4.3 Partitioning of block-tridiagonal matrix

The block-tridiagonal matrix A is partitioned into $3\times(m-2)$ squares, which are $D_1, D_2, \ldots, D_m$, $L_2, L_3, \ldots, L_m$, and $U_1, U_2, \ldots, U_{m-1}$. $L_2, L_3, \ldots, L_m$, and $U_1, U_2, \ldots, U_{m-1}$ are very sparse in the block-tridiagonal matrices which are arisen from the most scientific computing applications, such as finite difference and resource scheduling, and most non-zero elements are in $D_1, D_2, \ldots, D_m$. Therefore, $L_2, L_3, \ldots, L_m$ and $U_1, U_2, \ldots, U_{m-1}$ are stored as the compressed formats, such as COO or CSR formats. $D_1, D_2, \ldots, D_m$ are stored as the suitable storage formats according to the distribution features of the non-zero elements. They are stored as dense matrices if non-zero elements are more dense, and are stored as DIA format if they are tridiagonal matrices. The block-tridiagonal matrices arisen from resource scheduling are partitioned into $D_1, D_2, \ldots, D_m$, which are tridiagonal matrices.

4.4 Hybrid blocked iterative solving algorithm (HBISA)

The block-tridiagonal equation is given by Eq. (1):

$$Ax = b. \tag{1}$$

Equation (1) can be given by the following:

$$\begin{pmatrix} D_1 & U_1 & & & & & & \\ L_2 & D_2 & U_2 & & & & & \\ & L_3 & D_3 & U_3 & & & & \\ & & \ddots & \ddots & \ddots & & & \\ & & & \ddots & \ddots & \ddots & & \\ & & & & L_{m-1} & D_{m-1} & U_{m-1} & \\ & & & & & L_m & D_m & \end{pmatrix} \times \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ \vdots \\ x_{m-1} \\ x_m \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ b_3 \\ \vdots \\ \vdots \\ b_{m-1} \\ b_m \end{pmatrix},$$

where x_j and b_j for $j = 1, 2, \ldots, m$ are the sub-vectors of x and b. x_j includes the $((j-1) \times k)$th to the $(j \times k)$th elements of x, and b_j includes the $((j-1) \times k)$th to the $(j \times k)$th elements of b.

Assume that $B = L + D + U$, where

$$L = \begin{pmatrix} L_2 & & & & & \\ & L_3 & & & & \\ & & \ddots & & & \\ & & & \ddots & & \\ & & & & L_{m-1} & \\ & & & & & L_m \end{pmatrix},$$

$$D = \begin{pmatrix} D_1 & & & & & \\ & D_2 & & & & \\ & & D_3 & & & \\ & & & \ddots & & \\ & & & & \ddots & \\ & & & & & D_{m-1} \\ & & & & & & D_m \end{pmatrix},$$

and

$$U = \begin{pmatrix} U_1 & & & & & \\ & U_2 & & & & \\ & & U_3 & & & \\ & & & \ddots & & \\ & & & & \ddots & \\ & & & & & U_{m-1} \\ & & & & & \end{pmatrix}.$$

Therefore, we have

$$(L + D + U)x = b, \tag{2}$$

and

$$Dx = b - Lx - Ux. \tag{3}$$

Letting

$$r = b - Lx - Ux.$$

Then

$$Dx = r. \tag{4}$$

Solving Eq. (4) is much easier than to solve Eq. (1). Define the iterative equation:

$$Dx^{i+1} = b - Lx^i - Ux^i, \quad x^0 = 0. \tag{5}$$

For each block in B, Eq. (5) can be expressed as

$$D_j x_j^{i+1} = \begin{cases} b_j - U_j x_{j+1}^i & j = 1; \\ b_j - L_j x_{j-1}^i - U_j x_{j+1}^i & 1 < j < m; \\ b_j - L_j x_{j-1}^i & j = m. \end{cases} \tag{6}$$

For Algorithm 1, i index represents the ith iteration for solving process, and j index represents the jth block for the block-tridiagonal matrix. r_j is a right-hand vector for $D_j x_j^i = r_j$, which can be stored in an array to reuse in all iterations. r_j are independent vectors for different values of j. There are no data dependency among different r_j. Therefore, lines 5–17 can be processed in parallel.

For $j > 1$, x_{j-1}^{i+1} has been computed in the current iteration, and Eq. (6) can be replaced by Eq. (7):

$$D_j x_j^{i+1} = \begin{cases} b_j - U_j x_{j+1}^i & j = 1; \\ b_j - L_j x_{j-1}^{i+1} - U_j x_{j+1}^i & 1 < j < m; \\ b_j - L_j x_{j-1}^{i+1} & j = m. \end{cases} \tag{7}$$

4.5 The iterative convergence analysis

Lemma 1 *Assume that $Ax = b$, $A = D + L + U$, where A is the nonsingular block-tridiagonal matrix and D, L, U are submatrices which are obtained by Eq. (2). The necessary and sufficient condition of convergence for solving $Ax = b$ using Algorithm 1 is $\rho(D^{-1}(L+U)) < 1$, where $\rho(D^{-1}(L+U))$ is the spectral radius of $D^{-1}(L+U)$.*

Algorithm 1 The hybrid blocked iterative solving algorithm (HBISA) for the block-tridiagonal equation.

Require: The blocks partitioned from A, i.e., $L_2, L_3, \ldots, L_m, D_1, D_2, \ldots, D_m$ and $U_1, U_2, \ldots, U_{m-1}$; The vector, b.
Ensure: The solution, x_t.
$x^0 \leftarrow 0$;
$r \leftarrow b$;
for $i \leftarrow$ 1,2,3,…, until convergence **do**
  //The tridiagonal equation is solved by direct method, such as Gauss elimination or Thomas;
  for $j \leftarrow 1$ to m **do**
    //$D_j x_j^i = r_j$ is solved by direct method, such as Gauss elimination or Thomas;
    Solve $D_j x_j^i = r_j$;
    if $j = 1$ **then**
      $r_j \leftarrow b_j - U_j x_{j+1}^{i-1}$;
    else
      if $j > 1$ and $j < m$ **then**
        $r_j \leftarrow b_j - L_j x_{j-1}^{i-1} - U_j x_{j+1}^{i-1}$;
      else
        $r_j \leftarrow b_j - L_j x_{j-1}^{i-1}$;
      end if
    end if
  end for
  $\beta \leftarrow \|x^i - x^{i-1}\|$;
  if β =0 **then**
    $t \leftarrow i$;
    Stop;
  end if
end for
return x_t.

Proof (Sufficiency): D is the nonsingular matrix, because A is the nonsingular block-tridiagonal matrix. Therefore, Eq. (5) can be transformed into the following equation:

$$x^{i+1} = D^{-1}(b - Lx^i - Ux^i) = -D^{-1}(L + U)x^i + D^{-1}b.$$

Assume that $B = -D^{-1}(L + U)$ and $f = D^{-1}b$, the above iterative equation can be expressed $x^{i+1} = Bx^i + f$. $Ax = b$ has a unique solution, because A is a nonsingular matrix. Assume that the solution is x^* and $x^* = Bx^* + f$. Assume that an error vector is $\epsilon^k = x^k - x^* = B^k\epsilon^0$, $\epsilon^0 = x^0 - x^*$, where B^k expresses the matrix B raised to the power k. $\rho(B) < 1$, because $\rho(-D^{-1}(L + U)) < 1$ and $\rho(B) = \rho(-D^{-1}(L + U))$. $\rho(B)$ is the supremum among the absolute values of the eigenvalues of B, and $\rho(B) < 1$ if and only if $\lim_{k\to\infty} B^k = 0$ [61]. Therefore, $\lim_{k\to\infty} \epsilon^k = 0$ for any x^0, and $\lim_{k\to\infty} x^k = x^*$.

(Necessity): Assume that $\lim_{k\to\infty} x^k = x^*$ for any x^0, where $x^{k+1} = Bx^k + f$. Obviously, x^* is the solution of $x = Bx + f$, and for any x^0, $\epsilon^k = x^k - x^* = B^k\epsilon^0 \to 0\,(k \to \infty)$. Therefore, $\lim_{k\to\infty} B^k = 0$, and $\rho(B) < 1$. Therefore, $\rho(-D^{-1}(L + U)) < 1$, because $\rho(-D^{-1}(L + U)) = \rho(B)$. □

4.6 Parallel implementation

The iterative solving algorithm for the block-tridiagonal equation can be parallelized. We can observe that the codes in lines 5–17 can be executed in parallel, and each i of loops can be processed as a task in parallel. If Algorithm 1 is processed in parallel, Eq. (6) should be used in lines 9, 12, and 14, because x_{j-1}^{i+1} is computed synchronously and cannot be obtained in the same iteration. L_i, D_i, and U_i are assigned to processors as standalone tasks for $i = 1, 2, \ldots, q$, and lines 7–16 in Algorithm 1 are computed in parallel. The residual β of the iterative solution can be piecewise calculated in parallel, and line 18 can be replaced by $\beta_j^i \leftarrow \|x_j^i - x_j^{i-1}\|$ and moved into the loop 5–17. The tasks allocation is shown in Fig. 2.

For CUDA, the threads are allocated and scheduled in accordance with blocks. Furthermore, the number of cores in GPU is far more than that of CPU, and SpMV of lines 12 and 17 in Algorithm 1 is computed in parallel on GPU. Some common parallel algorithms can be used to solve $D_j x_j^i = r_j$, such as CR and PM. Due to the small size of D_j, the performance of direct solving algorithms should be good. SpMV is suited to be computed in parallel on GPU using CUDA, and each row of L_j and U_j is assigned to a thread of the block to execute multiplication with x_j, which is the jth approximate solution (Fig. 3).

L_j, D_j, and U_j of each block can be loaded into the shared memory on GPU to reduce access latency, because these data sets can be used in the all iterations. The approximate solutions x_j generated by the iterations are used by multiple SpMV, and should be stored in the shared memory. The residual β of the iterative solution can be calculated by two steps, which are inside a block and between blocks. The parallel reduction algorithm can be used to get the residual β, and the parallel reduction algorithm of the residual calculation using CUDA is shown in Algorithm 2.

In Algorithm 2, the maximum residual of a block is calculated from lines 5–12, and then, it is stored in the first element in the residual sub-array, which corresponds to the block. The maximum residual of the whole residual array is obtained from the maximum residuals of the blocks, and it is stored in the first element of the residual array.

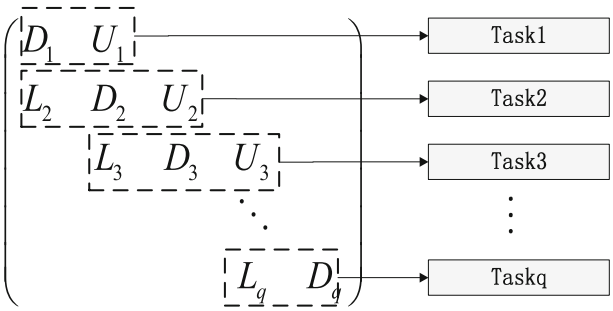


Fig. 2 Tasks allocation for solving the block-tridiagonal equation using Algorithm 1 in parallel

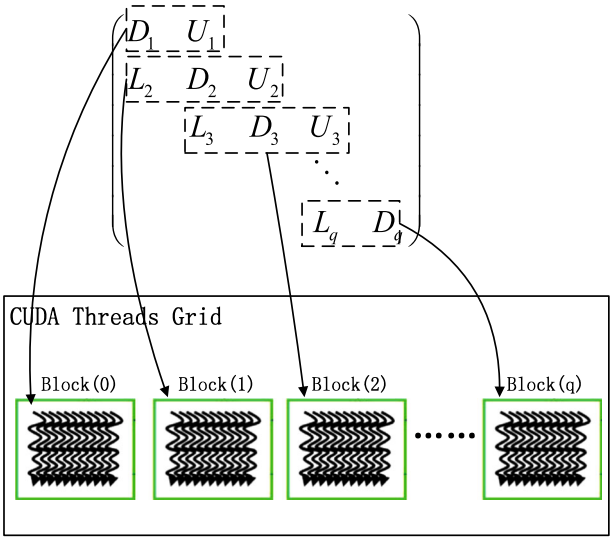


Fig. 3 Parallel computing tasks for Algorithm 1 on GPU

5 A case study: cloud resources scheduling

More and more computing resources are deployed in the cloud-computing platform, and massive amounts of computing tasks are processed in the cloud-computing platform. The computing resources should be allocated reasonably, and the computing tasks should be processed in time. Some computing resources will be idle if too much computing resources are turned on. Therefore, some resources should be turned off when the number of computing tasks is declined; in contrast, more computing resources should be turned on when the number of computing tasks is increased. The number of computing resources that are turned on should be consistent with that of the arrived tasks. However, some computing resources may not be fully used, because the arrival time of tasks is random.

5.1 $M/M/n + k$ dynamic queuing model

Assume that the arrival of tasks meets Poisson distribution and the processing time of the tasks meets exponential distribution. $M/M/n + k$ dynamic queuing model can describe the process of cloud resources scheduling if computing resources are the same. A new computing resource will be turned on if the number of tasks processed on the computing resource reaches an upper limit value. In contrast, an used computing resource will be turned off if the average number of tasks processed on the computing resources is below a lower limit value. The tasks migrated to new computing resources will be delayed, because turning on and off computing resources must be time-consuming. The state transition diagram of task allocation and resource scheduling using $M/M/n + k$ dynamic queuing model is a birth and death process,

Algorithm 2 The parallel reduction algorithm of the residual calculation using CUDA.

Require: The residual array, i.e., $R_0, R_2, \ldots, R_{q\times k-1}$; The number and size of the block, k and q.
Ensure: The maximum of residual, r.

```
tid ← threadIdx.x;
bid ← blockIdx.x;
start_idx ← blockIdx.x * blockDim.x;
i_loop_num ← ⌈lg_2^q⌉;
for i ← 0 to i_loop_num do
    intervals ← 2^i;
    width ← ⌈q / 2^(i+1)⌉;
    if tid > width then
        pos ← tid × 2^(i+1);
        if R_pos < R_(pos+intervals) then
            R_pos ← R_(pos+intervals);
        end if
    end if
end for
i_loop_num ← ⌈lg_2^k⌉;
for i ← 0 to i_loop_num do
    intervals ← 2^i;
    width ← ⌈k / 2^(i+1)⌉;
    if bid > 2^(i−1) then
        pos ← bid × 2^(i+1);
        if R_(bid*q) < R_((bid+intervals)×q) then
            R_(bid×q) ← R_((bid+intervals)*q);
        end if
    end if
end for
r ← R_0
return r.
```

and instantaneous states of flow are showed in Fig. 4, where the ellipses express the states of the resources and the numbers in ellipses express the numbers of pending tasks assigned to the resources. The first line of Fig. 4 expresses the states of the cloud-computing system with one resource and the second line expresses the system with two resources, and so on. λ, μ, η, and ν express request arrival rate, processor's service rate, processor's opening rate, and processor's closing rate. The number i in ellipses expresses that there are i pending tasks in the cloud-computing system.

Assume that there are n computing resources in a cloud-computing platform. When more than k-computing tasks assigned into a computing resource are pended, a new computing resource will be turned on to process the new tasks. Furthermore, the computing resources without computing task should be turned off.

For a cloud-computing system, how many computing resources need to be deployed? The utilization efficiency of resources should be computed. The probability that the resources are used must be calculated for the poisson distribution of the arrival of tasks. Define that $S_{i,j}$ is the state, which has i pending tasks in j computing resources. The transition probability between the system states can be calculated by

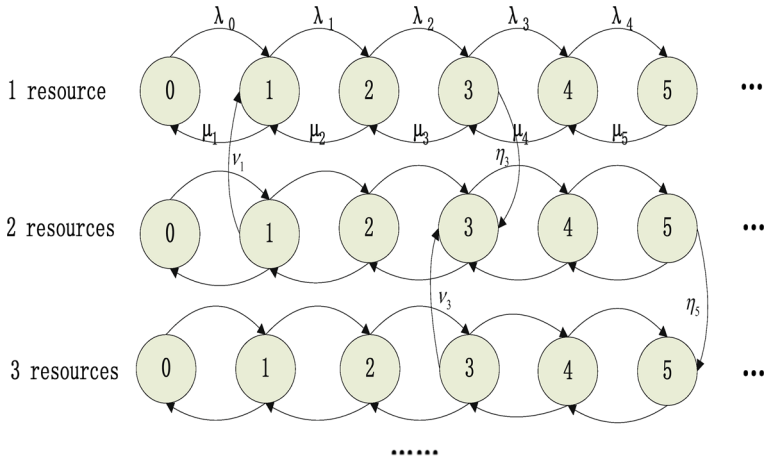


Fig. 4 State transition diagram of task allocation and resource scheduling using $M/M/n+k$ dynamic queuing model

$$\begin{cases} \lambda_{i,j} = \begin{cases} \lambda, & i \le j \times k \\ 0, & i > j \times k \end{cases} \\ \mu_{i,j} = \begin{cases} \lfloor \frac{i}{k} \rfloor \times \mu, & j \le i \le j \times k \\ k \times \mu, & i > j \times k \\ \mu, & i < j \end{cases} \\ \eta_{i,j} = \begin{cases} 0, & i \le j \times k \\ \lceil \frac{i-j\times k}{k} \rceil \times \eta, & i > j \times k \end{cases} \\ \nu_{i,j} = \begin{cases} 0, & i \ge j \times k \\ \lfloor \frac{j\times k-i}{k} \rfloor \times \nu, & i < j \times k \end{cases} \end{cases} \tag{8}$$

If there are too many pending tasks in a computing node, another computing nodes should be turned on to process the new tasks. The probability of the state $S_{i,j}$ will be very little when $i > k \times (j+1)$ for j computing resources, and the probability of the state $S_{i,j}$ will be very little when $i < k \times (j-1)$ for j computing resources, because the computing resources without computing task should be turned off, so the states can be ignored. The transition probability matrix P is a block-tridiagonal matrix composed of Eq. (8), whose size is $n \times n$ and the size of blocks is k. The $k \times k$ submatrices along the main diagonal are tridiagonal matrices, and another submatrices outside the main diagonal are sparse matrices. The computing resources should not be turned on and off frequently, because it takes time to turn off and on. Therefore, a certain number of computing resources should be turn on in advance, but how many computing resources should be turn on in advance? Define that π_i is the probability of the state with i pending tasks in cloud system. The probabilities of π_i for $i = 1, 2, \ldots, n$ are obtained by solving the following Markov stationary equation:

$$\pi = \pi P. \tag{9}$$

Assume that the state with maximum probability is π_M, which has M pending tasks, M/k computing resources should be turn on in advance to reduce waiting time of pending tasks.

5.2 Hybrid storage format for block-tridiagonal matrix

D_i for $i = 1, 2, \ldots, n$ are tridiagonal matrices in the block-tridiagonal matrix obtained by $M/M/n+k$ dynamic queuing model for cloud resources scheduling modeling, and U_i and L_i for $i = 1, 2, \ldots, n-1$ are sparse matrices.

The 9-by-9 block-tridiagonal matrix B shown below is an example obtained by $M/M/n+k$ dynamic queuing model:

$$\begin{pmatrix}
1 & -4 & 0 & \vdots & -2 & 0 & 0 & \vdots & 0 & 0 & 0 \\
-3 & 7 & -4 & \vdots & 0 & -2 & 0 & \vdots & 0 & 0 & 0 \\
0 & -3 & 5 & \vdots & -4 & 0 & 0 & \vdots & 0 & 0 & 0 \\
\cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\
-1 & 0 & 0 & \vdots & 5 & -4 & 0 & \vdots & -2 & 0 & 0 \\
0 & 0 & -3 & \vdots & -3 & 9 & -4 & \vdots & 0 & -1 & 0 \\
0 & -1 & 0 & \vdots & 0 & -2 & 9 & \vdots & 0 & -2 & 0 \\
\cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\
0 & 0 & 0 & \vdots & 0 & -1 & 0 & \vdots & 5 & -3 & 0 \\
0 & 0 & 0 & \vdots & 0 & 0 & -2 & \vdots & -4 & 9 & -4 \\
0 & 0 & 0 & \vdots & 0 & 0 & -1 & \vdots & 0 & -3 & 12
\end{pmatrix}.$$

Matrix B has seven diagonal blocks which have three rows and three columns. D_1, D_2, and D_3 are tridiagonal matrices, which can be stored as DIA format. Furthermore, **offset** array can be omitted, because the offsets of three diagonals are fixed $(-1, 0, 1)$. L_2, L_3, U_1, and U_2 are sparse matrices, which have few non-zero elements in each row. Therefore, L_2, L_3, U_1, and U_2 should be stored as COO format. The number of storage units of D_1, D_2, and D_3 is 21 and that of L_2, L_3, U_1, and U_2 is 36. Few storage units are occupied using COO format for L_2, L_3, U_1, and U_2, because there are few non-zero elements and irregular distributions of non-zero elements in L_2, L_3, U_1, and U_2.

5.3 Experimental evaluation

5.3.1 Experiment settings

The following test environment has been used for all benchmarks. The test computer is equipped with two AMD Opteron 6376 CPUs running at 2.30 GHz and an NVIDIA Tesla K20c GPUs. Each CPU has 16 cores. The GPU has 2496 CUDA processor cores, working at 0.705GHz clock, and possessing 4 GB global memory with 320 bits

Table 1 Parameters of the test computer

Parameters	Description	Values
S_i	The size of integer	4 Byte
S_s	The size of single	4 Byte
S_d	The size of double	8 Byte
C	The number of stream processor	2496
f_s	The clock speed of SP	0.705 GHz
f_a	The clock speed of the global memory	2.6 GHz
AW	The bus width of the global memory	320 bits
TW	The bandwidth of PCIe	8 GiB/s

Table 2 General information of the block-tridiagonal matrices used in the evaluation

No.	Sparse matrix	n	k	NNZ
1	Cloud16–100	1600	16	7231
2	Cloud16–500	8000	16	35,431
3	Cloud16–800	12,000	16	56,331
4	Cloud16–1000	16,000	16	70,931
5	Cloud16–2000	32,000	16	135,931
6	Cloud32–100	3200	32	14,196
7	Cloud32–500	16,000	32	66,196
8	Cloud32–1000	32,000	32	136,696
9	Cloud64–100	6400	64	25,215
10	Cloud64–500	32,000	64	127,915

bandwidth at 2.6 GHz clock, and the CUDA compute capacity is 3.5. As for the software, the test machine runs the 64 bit Windows 7 and NVIDIA CUDA toolkit 7.0. The hardware parameters of the testing computer are shown in Table 1.

All benchmarks are chosen from the simulation of the cloud-computing system. Ten kinds of cloud-computing systems were simulated to get ten block-tridiagonal matrices. Further characteristics of these matrices are given in Table 2, where n, k, and NNZ are the number of matrices, the size of block, and the number of non-zero elements, respectively, in matrices.

5.3.2 The test using HBISA on GPU

The tridiagonal submatrices along the main diagonal of the coefficient matrix are solved by our parallel CR algorithm for GPU. SuiteSparse library provides a sparse direct solver SuiteSparseQR for GPU [62,63]. CUDA tools also provide cuSolver library, which are a high-level package based on the cuBLAS and cuSPARSE libraries [54]. The cuSolverSP of cuSolver library provides a new set of sparse routines based on a sparse QR factorization. However, the performance of SuiteSparseQR is worse

Table 3 Number of iterations for solving the cases using iterative algorithms

Sparse matrix	HBISA	CG	BiCGSTAB
Cloud16–100	120	154	175
Cloud16–500	128	956	150
Cloud16–800	127	971	164
Cloud16–1000	125	959	157
Cloud16–2000	131	836	159
Cloud32–100	162	163	165
Cloud32–500	186	195	217
Cloud32–1000	181	No	203
Cloud64–100	309	333	356
Cloud64–500	371	334	365

than that of cuSolverSP. For QR factorization, non-zero elements will increase sharply resulting in a decline in performance because of irregular distributions of non-zero elements. The cases were solved using the iterative solvers of PARALUTION library for parallel and serial modes on CPUs and GPU [56], and the solving process is not convergent using the GMRES method of PARALUTION for the test cases. Therefore, the iterative solvers of PARALUTION library using the CG and BiCGSTAB methods are used to test the cases.

The convergence speed refers to the number of iterations of solving the equation using iterative algorithms. The less number of iterations, the faster convergence speed. The convergence tolerance is set to 0.0000001 for the iterative algorithms in the test. The cases are solved by the tested algorithms using double precision. The number of iterations for solving the cases using iterative algorithms HBISA, CG, and BiCGSTAB is shown in Table 3, where *No* expresses that the solving process is not convergent using the iterative algorithm for the test case. It is observed from Table 3 that Cloud32–1000 cannot be solved by CG. HBISA and BiCGSTAB have better robustness than CG, and the number of iterations using CG is far more than that of HBISA and BiCGSTAB for Cloud16–500, Cloud16–800, Cloud16–1000, and Cloud16–2000. The convergence speeds of HBISA are faster than those of CG and BiCGSTAB exception Cloud64–500. For the test cases, the average number of iterations is reduced by 283.15 and 18.34 % using HBISA compared with CG and BiCGSTAB of PARALUTION library.

The performance of solving the test cases on K20c GPU is shown in Table 4, where *No* represents that the case cannot be solved using the algorithm. For Cloud16–500, Cloud16–800, Cloud16–1000, and Cloud16–2000, the performance of CG is significantly worse than those of HBISA and BiCGSTAB, because the number of iterations using CG is far more than that of HBISA and BiCGSTAB for these cases. The performance of HBISA is worse than those of CG, BiCGSTAB, and cuSolverSP, because the convergence speed of HBISA is slow for Cloud64–500.

The parallel efficiency refers to the speedup, which is a metric used to express relative performance improvement. To more clearly describe the performance improvement, the performance improvement percentages using HBISA on GPU compared with CG, BiCGSTAB of PARALUTION library, and cuSolverSP of CUDA in Fig. 5 are calculate by $(T_i - T_H)/T_H \times 100\,(i = 1, 2, 3)$, where T_1, T_2, and T_3 are the perfor-

Table 4 Performance of solving the cases (unit: second)

Sparse matrix	HBISA	CG	BiCGSTAB	cuSolverSP
Cloud16–100	0.196	0.267	0.297	0.263
Cloud16–500	0.354	0.733	0.401	0.428
Cloud16–800	0.525	0.754	0.622	0.598
Cloud16–1000	0.698	0.913	0.731	0.719
Cloud16–2000	0.960	1.186	0.962	0.965
Cloud32–100	0.284	0.286	0.288	0.297
Cloud32–500	0.460	0.462	0.469	0.489
Cloud32–1000	0.509	No	0.595	0.566
Cloud64–100	0.324	0.336	0.350	0.334
Cloud64–500	0.703	0.682	0.696	0.667

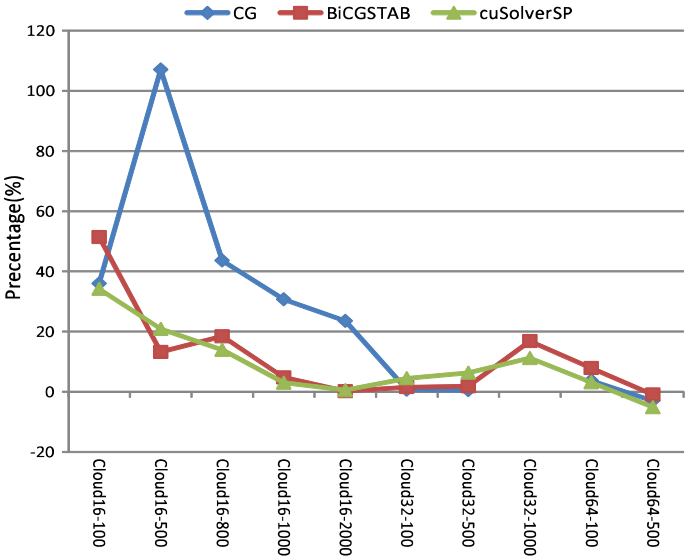


Fig. 5 Performance improvement of solving the cases using HBISA on K20c GPU

mance of CG, BiCGSTAB of PARALUTION library, and cuSolverSP, respectively, and T_H is the performance of HBISA. It is observed from Fig. 5 that the average performance of solving the test cases on GPU is improved by 26.98, 11.52, and 9.25 % using HBISA compared with CG, BiCGSTAB of PARALUTION library, and cuSolverSP of CUDA.

6 Conclusion

In this paper, a parallel hybrid solving algorithm is proposed for block-tridiagonal systems of linear equations, and the performance is better than that of the other iterative

algorithms and direct algorithms on GPU because of faster convergence speed and higher parallel efficiency. Furthermore, we analyse cloud resources scheduling model and obtain ten block-tridiagonal matrices produced by the simulation of the cloud-computing system. The computing performance of solving these block-tridiagonal systems of linear equations can be improved by using our method. Other sparse linear systems arising from numerical simulation may be quasi-block-diagonals equations, and how to solve them quickly will be our next step of investigation.

Acknowledgements The authors deeply appreciate the anonymous reviewers for their comments on the manuscript. The research was partially funded by the Key Program of National Natural Science Foundation of China (Grant Nos. 61133005 and 61432005), the National Natural Science Foundation of China (Grant Nos. 61370095, 61472124, and 61572175), and the Science and technology project of Hunan Province (Grant No. 2015SK20062).

References

1. Geer D (2005) Chip makers turn to multicore processors. Computer 38(5):11–13
2. Thomas LH (1949) Elliptic problems in linear difference equations over a network. Watson Sci. Comput. Lab. Rept. Columbia University, New York
3. Stone HS (1975) Parallel tridiagonal equation solvers. ACM Trans Math Softw 1:289–307
4. Heller D (1976) Some aspects of the cyclic reduction algorithm for block tridiagonal linear systems. SIAM J Numer Anal 13(4):484–496
5. Hirshman SP, Perumalla KS, Lynch VE, Sanchez R (2010) Bcyclic: a parallel block tridiagonal matrix cyclic solver. J Comput Phys 229(18):6392–6404
6. Lamas-Rodrıguez J, Heras D, Bóo M, Argüello F (2011) Tridiagonal system solvers internal report. Department of Electronics and Computer Science Internal Report, University of Santiago de Compostela, Spain
7. Buzbee BL, Golub GH, Nielson CW (1970) On direct methods for solving poisson's equations. SIAM J Numer Anal 7:627–656
8. Hockney RWA (1965) fast direct solution of Poisson's equation using fourier analysis. J ACM 12:95–113
9. Stone HS (1973) An efficient parallel algorithm for the solution of a tridiagonal linear system of equations. J ACM 20:27–38
10. Bondeli S (1990) Divide and conquer: a parallel algorithm for the solution of a tridiagonal linear system of equations. In: Joint International Conference on Vector and Parallel Processing, CONPAR 90, vol. IV. Springer, Berlin, pp 419–434
11. Wang HH (1981) A parallel method for tridiagonal equations. ACM Trans Math Softw 7:170–183
12. Lorenzo PAR, Müller A, Murakami Y, Wylie BJN (1996) High performance fortran interfacing to scalapack. In: Proceedings of the Third International Workshop on Applied Parallel Computing, Industrial Computation and Optimization, pp 457–466
13. Sanchez R, Hirshman S, Lynch V (2010) Siesta: an scalable island equilibrium solver for toroidal applications. American Physical Society, Providence
14. Arabnia HR, Oliver MA (1986) Fast operations on raster images with SIMD machine architectures. Comput Graph Forum 5(3):179–188
15. Arabnia HR, Oliver MA (1987) A transputer network for the arbitrary rotation of digitised images. Comput J 30(30):425–432
16. Arabnia HR, Oliver MA (1987) Arbitrary rotation of raster images with simd machine architectures. Comput Graphics Forum 6(1):3–11
17. Arabnia HR, Oliver MA (1989) A transputer network for fast operations on digitised images. Comput Graphics Forum 8(8):3–11
18. Arabnia HR (1990) A parallel algorithm for the arbitrary rotation of digitized images using process-and-data-decomposition approach. J Parallel Distrib Comput 10(2):188–192
19. Arabnia HR (1995) Distributed stereo-correlation algorithm. In: Proceedings of the International Conference on Computer Communications and Networks, pp 707–711

20. Bhandarkar SM, Arabnia HR, Smith JW (2011) A reconfigurable architecture for image processing and computer vision. Int J Pattern Recognit Artif Intell 9(2):201–229
21. Bhandarkar SM, Arabnia HR (1995) The hough transform on a reconfigurable multi-ring network. J Parallel Distrib Comput 24(1):107–114
22. Bhandarkar SM, Arabnia HR (1995) The refine multiprocessor theoretical properties and algorithms. Parallel Comput 21(11):1783–1805
23. Arabnia HR, Bhandarkar SM (1996) Parallel stereocorrelation on a reconfigurable multi-ring network. J Supercomput 10(3):243–269
24. Wani MA, Arabnia HR (2003) Parallel edge-region-based segmentation algorithm targeted at reconfigurable multiring network. J Supercomput 25(1):43–62
25. Thapliyal H, Srinivas MB, Arabnia HR (2005) A need of quantum computing: Reversible logic synthesis of parallel binary adder-subtractor. In: International Conference on Embedded Systems and Applications. ESA, Las Vegas
26. Thapliyal H, Arabnia H, Vinod AP (2006) Combined integer and floating point multiplication architecture (CIFM) for fpgas and its reversible logic implementation. Comput Sci 2:438–442
27. Gopineedi PD, Thapliyal H, Srinivas MB, Arabnia HR (2006) Novel and efficient 4: 2 and 5: 2 compressors with minimum number of transistors designed for low-power operations. In: International Conference on Embedded Systems Applications, Las Vegas, pp 160–168
28. Thapliyal H, Arabnia HR (2006) Reversible programmable logic array (RPLA) using fredkin and feynman gates for industrial electronics and applications. Computer Science
29. Thapliyal H, Arabnia HR, Bajpai R, Sharma KK (2007) Combined integer and variable precision (CIVP) floating point multiplication architecture for fpgas. Comput Sci
30. Thapliyal H, Arabnia HR, Srinivas MB (2009) Efficient reversible logic design of BCD subtractors. Springer, Berlin
31. Balasubramanian P, Edwards DA, Arabnia HR (2011) Robust asynchronous carry lookahead adders. In: International Conference on Computer Design, pp 321–324
32. Balasubramanian P, Arabnia HR, Arisaka R (2012) Rb_dsop: a rule based disjoint sum of products synthesis method. In: International Conference on Computer Design
33. Thapliyal H, Jayashree HV, Nagamani AN, Arabnia HR (2013) Progress in reversible processor design: a novel methodology for reversible carry look-ahead adder
34. Lee J, Wright JC (2014) A block-tridiagonal solver with two-level parallelization for finite element-spectral codes. Comput Phys Commun 185(10):2598–2608
35. Ruggiero V, Galligani E (1992) A parallel algorithm for solving block tridiagonal linear systems. Comput Math Appl 24(4):15–21
36. Li HB, Huang TZ, Zhang Y, Liu XP, Li H (2009) On some new approximate factorization methods for block tridiagonal matrices suitable for vector and parallel processors. Math Comput Simul 79(7):2135–2147
37. Henk A, Vorst VD (2003) Iterative krylov methods for large linear systems, vol 13. Cambridge University Press, Cambridge xiv+221
38. Samarskii A A, Nikolaev E S (1989) Numerical methods for grid equations. Birkhäuser, Basel
39. Varah JM (1972) On the solution of block-tridiagonal systems arising from certain finite-difference equations. Math Comput 26(120):859–868
40. Terekhov AV (2011) A fast parallel algorithm for solving block-tridiagonal systems of linear equations including the domain decomposition method. Parallel Comput 39(s 6–7):475–484
41. Ruggiero V, Galligani E (1992) A parallel algorithm for solving block tridiagonal linear systems. Comput Math Appl 24(4):15–21
42. Gutknecht MH, Schmelzer T (2007) Updating the qr decomposition of block tridiagonal and block hessenberg matrices. Appl Numer Math 58(2008):871–883
43. Koulaei MH, Toutounian F (2007) On computing of block ilu preconditioner for block tridiagonal systems. J Comput Appl Math 202(2):248–257
44. Yang W, Li K, Liu Y, Shi L, Wang C (2014) Optimization of quasi diagonal matrix-vector multiplication on gpu. Int J High Perform Comput Appl 28(2):181–193
45. Li K, Yang W, Li K (2015) Performance analysis and optimization for SPMV on GPU using probabilistic modeling. IEEE Trans Parallel Distrib Syst 26:196–205. doi:10.1109/TPDS.2014.2308221
46. Yang W, Li K, Mo Z, Li K (2015) Performance optimization using partitioned SPMV on GPUs and multicore cpus. IEEE Trans Comput 64(9):2623–2636
47. DAzevedo E, Hill J C (2012) Parallel lu factorization on GPU cluster. Proc Comp Sci 9(11):67–75

48. Tomov S (2012) A hybridization methodology for high-performance linear algebra software for GPUs, Chap 34. Elsevier, Amsterdam
49. Agullo E, Demmel J, Dongarra J, Hadri B, Kurzak J, Langou J, Ltaief H, Luszczek P, Tomov S (2009) Numerical linear algebra on emerging architectures: the plasma and magma projects. J Phys Conf Seri, p 012037
50. Davidson A, Zhang Y, Owens JD (2011) An auto-tuned method for solving large tridiagonal systems on the gpu. In: Proceedings of the 2011 IEEE International Parallel Distributed Processing Symposium, pp 956–965
51. Göddeke D, Strzodka R (2011) Cyclic reduction tridiagonal solvers on GPUs applied to mixed-precision multigrid. IEEE Trans Parallel Distrib Syst 22(1):22–32
52. László E, Giles M, Appleyard J (2016) Manycore algorithms for batch scalar and block tridiagonal solvers. ACM Trans Math Softw 42(4):31:1–31:36
53. NVIDIA (213) NVIDIA CUDA C programming guide, Tech. Rep
54. NVIDIA (2015) Cusolver library, Tech. Rep
55. NVIDIA (2015) Cusparse library, Tech. Rep
56. PARALUTION Labs UG & Co. KG (2015) Paralution—user manual, Tech. Rep., Gaggenau
57. Ziane Khodja L, Couturier R, Giersch A, Bahi J (2014) Parallel sparse linear solver with gmres method using minimization techniques of communications for gpu clusters. J Supercomput 69(1):200–224. doi:10.1007/s11227-014-1143-8
58. Couturier R, Denis C, Jzquel F (2008) Gremlins: a large sparse linear solver for grid environment. Parallel Comput 34(6C8):380–391. Parallel Matrix Algorithms and Applications. http://www.sciencedirect.com/science/article/pii/S0167819107001354
59. Jezequel F, Couturier R, Denis C (2012) Solving large sparse linear systems in a grid environment: the gremlins code versus the petsc library. J Supercomput 59(3):1517–1532. doi:10.1007/s11227-011-0563-y
60. Smith B (2001) PETSC: portable, extensible toolkit for scientific computation. Encyclopedia of Parallel Computing, pp 1530–1539
61. Householder AS (1964) The theory of matrices in numerical analysis. Dover, New York
62. Davis T A (2011) Algorithm 915, suitesparseqr: multifrontal multithreaded rank-revealing sparse qr factorization. ACM Trans Math Softw (TOMS) 38(1):8
63. Davis TA, Yeralan SN, Ranka S (2015) Algorithm 9xx: sparse qr factorization on the GPU. ACM Trans Math Softw 1:1–28. doi:10.1145/0000000.0000000