The 2013 World Congress on
Advances in Structural Engineering and Mechanics (ASEM13)
Jeju, Korea, September 8-12, 2013

An improved Thomas Algorithm for finite element matrix parallel computing

Qingfeng Du[1a)], Zonglin Li[1b)], [*]Hongmei Zhang[2], Xilin Lu[2], Liu Zhang[1]

[1)] *School of Software Engineering, Tongji Univesity,Shanghai, 200092, China*
[2)] *Research Institute of Structural Engineering and Disaster Reduction,*
Tongji University,Shanghai, 200092, China)
zhanghongmei@tongji.edu.cn

ABSTRACT

With the expansion of the scale of linear finite element analyzing data, efficiency of computation is a main technical bottleneck. For solving the bottleneck, parallel computation is playing an increasingly prominent role. At present, the researches on linear finite element parallel computation are mainly concentrated on the pre-processing phase, and goal of these researches is to reduce the communication overhead and improve the homogeneous degree. The researches on computation phase dealing with linear finite element analyzing data are rare. However, most of computation cost is in this phase. In this paper, we study and analyze stiffness matrix decomposition and after comparing the different matrix decomposition algorithms, an improved algorithm for parallel computation based on Thomas algorithm is proposed. Verification by a large amount of data proves that the improved algorithm greatly enhances the parallel performance of linear finite element computation.

1. INTRODUCTION

Nowadays, linear finite element method is widely used in large complex combination structure analysis. With the growth of data scale, linear finite element parallel computation (Ananth 2003) is playing an increasingly important role in engineering field especially in the concrete structure simulation (Liu 2009, Lv 2011). At the moment, the research on linear finite element parallel computation is mainly concentrated on the pre-processing phase, and goal of their researches is to reduce the communication overhead and improve the homogeneous degree (Maurer 2011, Paz 2005). Seldom research is conducted on computation phase dealing with finite element analyzing data. And most of computation cost is in this phase and the computation is mainly in solving large size linear equations. Structural stiffness matrix, which is the coefficient matrix of the equations, is singular, symmetrical and sparse, with non-zero elements spread on a

* Corresponding author, Associate Professor, PhD, E-mail: zhanghongmei@tongji.edu.cn ; du_cloud@tongji.edu.cn
[a]Professor, PhD
[b]Master Student
Note: Copied from the manuscript submitted to “Computers and Concrete, An International Journal” for presentation at ASEM13 Congress

stripe region and is usually a tridiagonal matrix (Turmo 2012). The most mature decomposition algorithm is Gaussian Elimination Algorithm, which is also called *LU* algorithm. For tridiagonal matrix, Thomas proposed chasing algorithm (Thomas algorithm) based on the LU algorithm (Turmo, J. 2012). This algorithm is very effective on solving tridiagonal linear equations, but it is not suitable for parallel computation (Paz. 2005). In order to solve the problem of parallel computation of tridiagonal linear equations, the original Thomas algorithm on single processor wea anlyzed here, and then, an improved algorithm suitable for parallel computation by describing the idea and logic of the algorithm is proposed for complex structural matrix analyze. To verify the efficiency of the improved algorithm, the parallel method named MPI (Message Passing Interface) (Pacheco. 1997) was employed to test on various levels of specificity of data. Test result indicates that the improved algorithm enhances the parallel performance of linear finite element computation significantly.

1. BACKGROUND AND RELEVANT KNOWLEDGE

1.1 The decomposition of the coefficient matrixes of linear equations and Thomas Algorithm

Assuming that $Ax = b$ is a large size linear equations and A is the coefficient matrix. The most mature decomposition algorithm is Gaussian Elimination Algorithm. This decomposition algorithm is divided into two phases. Firstly, some algebraic operations simplify $Ax = b$ into upper triangular equations, so $Ax = b$ can be written as $Ux = y$ (U is a unit upper triangular coefficient matrix). The second phase, backward substitution method is used to solve the equations. An improved method of the method is LUalgorithm, that is the algorithm that decomposes matrix A into upper triangle matrix U and lower triangle matrix L and that is $A = LU$. L is stored in the lower triangle of A and U is stored in the upper triangular of A (since the diagonal elements are not stored, the default value is 1). Gaussian decomposition method is applicable to the general dense matrix. Cholesky decomposition algorithm is widely used to solve positive definite matrix, being seen as a special case of LU algorithm and more suitable to solve the symmetric positive definite matrix. In the algorithm, the matrix is decomposed into a product of a triangular matrix and its transpose, i.e., $A = RR^T$(R is an upper triangular matrix). Obviously, the decomposition algorithm calculated amount is $n^3 /6$, and the calculated amount is only half of Gaussian decomposition method. For Tridiagonal matrix, Thomas proposed chasing algorithm (Thomas algorithm) based on the LU algorithm. The algorithm is very simple and the calculated amount is only $5n - 4$ times of multiplication and division operations. The algorithm is a numerically stable algorithm and is a classical algorithm to solve tridiagonal linear equations too.

1.2 Structural stiffness matrix

An element K_{ij} of structural stiffness matrix K means: how much force should be exerted on node i when the displacement of node j is one unit value while other nodes are zero. The difference between element stiffness matrix and structural stiffness matrix is that, structure is the collection of unit and every unit affects the structure. As a unit stiffness matrix is symmetrical and singular, the structural stiffness matrix that is integrated by some units is also symmetrical and singular. That is that constraint condition of displacement has to be given in order to remove the singularity of K, so that the displacement of elements can be obtained.

The structural stiffness matrix is the collection of unit stiffness matrixes. Although the total number of elements is more, and the order of structural stiffness is high, most of the elements are zero. So if the number is reasonable, the non-zero elements will spread on a stripe region centered in principal diagonal.

In short, a structural stiffness matrix is singular, symmetrical and sparse, with non-zero elements spread on a stripe region. With reasonable numbering, the matrix is positive definite tridiagonal matrix.

1.3 Existing problems

Finite element analysis is a very important numerical analysis method that has been widely used in the field of engineering and scientific computing. However, large or very large complex structure analysis using finite element analysis method will result in the calculated amount increases exponentially and the usual strategy is to use a supercomputer to calculate. In recent years, the finite element parallel computing researches draw the researchers' attention, and one of the concerns is to improve the finite element parallel computing algorithm to raise the efficiency of large scale complex structural analysis under common distributed parallel computing environment and make it applicable to common users.

The finite element distributed parallel computing can be divided into three stages: pre-processing, computation and post-processing. In the pre-processing stage, finite element model is built, and the unit grid is divided. During the post-processing, we analyze the results to help users extract information and understand the calculated results. The computing cost is mainly in the computation stage. In fact, this majority of computation is to solve large-scale linear equations. The computation process involves linear equations coefficient matrix algorithm, Thomas algorithm and structural stiffness matrix. The key point of our research is the stiffness matrix decomposition.

This paper analyzes the features of large-scale linear equations coefficient matrix and the Thomas algorithm and then we put forward an effective matrix decomposition strategy and an improved Thomas algorithm based on the strategy, which is applicable to large complex structure analysis and suitable for parallel computation.

2. THE IMPROVEMENT OF THOMAS ALGORITHM

2.1 The existing Thomas algorithm on a single process

2.1.1 Algorithm introduction

Firstly, a single process Thomas algorithm is given and assuming a coefficient matrix *A* a positive tridiagonal matrix, that is:

$$A=\begin{bmatrix} b_1 & c_1 & & & \\ a_2 & b_2 & c_2 & & \\ & \ddots & \ddots & \ddots & \\ & & a_{n-1} & b_{n-1} & c_{n-1} \\ & & & a_n & b_n \end{bmatrix} \tag{1}$$

Decompose A according to Crout, that is $A = LU$,

$$L = \begin{bmatrix} \alpha_1 & & & & \\ \gamma_1 & \alpha_2 & & & \\ & \ddots & \ddots & & \\ & & \gamma_{n-1} & \alpha_{n-1} & \\ & & & \gamma_n & \alpha_n \end{bmatrix} \tag{2}$$

$$U = \begin{bmatrix} 1 & \beta_1 & & & \\ & 1 & \beta_2 & & \\ & & \ddots & \ddots & \\ & & & 1 & \beta_{n-1} \\ & & & & 1 \end{bmatrix} \tag{3}$$

a_i , β_i and γ_i are undetermined coefficient. By matrix multiplication, we can get:

$$\left.\begin{array}{l} b_1 = \alpha_1, \quad c_1 = \alpha_1\beta_1 \\ a_i = \gamma_i, \quad b_i = \gamma_i\beta_{i-1} + a_i, \quad i = 2,3,\cdots,n \\ c_i = \alpha_i\beta_i, \quad i = 2,3,\cdots,n-1 \end{array}\right\} \tag{4}$$

$$\left.\begin{array}{ll} \alpha_1 = b_1, & \beta_1 = c_1 / b_1 \\ \alpha_i = b_i - a_i\beta_{i-1}, & i = 2,3,\cdots,n, \\ \beta_i = c_i / \alpha_i, & i = 2,3,\cdots,n-1, \end{array}\right\} \tag{5}$$

Therefore, the existing tridiagonal equations are equivalent to the following two equations $Ly = f$ and $Ux = y$.

2.1.2 The logic of the algorithm

- Step 1: Input data a_i, b_i, c_i, y
- Step 2: Computation

$$\left.\begin{array}{ll} \alpha_1 = b_1, & \beta_1 = c_1 / b_1 \\ \alpha_i = b_i - a_i\beta_{i-1}, & i = 2,3,\cdots,n, \\ \beta_i = c_i / \alpha_i, & i = 2,3,\cdots,n-1, \end{array}\right\} \tag{6}$$

- Step 3: Solving the equations $Ly = f$

$$\begin{array}{l} y_1 = f_1 / b_1, \\ y_i = \left(f_i - a_i y_{i-1}\right) / \alpha_i, \quad i = 2,3,\cdots,n, \end{array} \tag{7}$$

- Step 4: Solving the equations $Ux = y$

$$\begin{array}{l} x_n = y_n, \\ x_i = y_i - \beta_i x_{i+1}, \quad i = n-1, n-2, \cdots, 1. \end{array} \tag{8}$$

- Step5:Output the solution of equations $X = (x_0, x_1, L, x_{m-1})^T$

The process of calculating $\beta_1 \to \beta_2 \to L \to \beta_{n-1}$ and $y_1 \to y_2 \to L \to y_{n-1}$ is the process of forward sweep. The process of $x_n \to x_{n-1} \to L \to x_1$ is the process of backward substitution.

As the formula of Thomas algorithm is very simple, its calculated amount of is $5n-4$ times of multiplication and division. Thomas algorithm is numerically stable algorithm, so it is widely used in serial processing tridiagonal equations.

2.2 Improvement of Thomas algorithm for parallel computing

2.2.1 *The existing storage strategy of large tridiagonal matrix*

Block Storage Strategy: Assume K is an n-order square matrix, and P is the number of node machines. For large finite element analysis, it is $n > p$, under normal circumstances. Assume $n = mp$ and block storage scheme is, the first node machine stores the first m lines, the second node machine for the second m lines, and so on, and the last node machine store the last m lines. If n is not an integer multiple of $\boldsymbol{P}$, the rest of the rows is stored in first node. As the front lines of the matrix first complete the matrix decomposition, when each m lines is decomposed an idle machine is added, so the storage strategy is serious load imbalance.

Single-line Shutter Storage Strategy: Single line shutter storage is that the i line is stored in the $i \bmod p$ node machine. For example, the first node stores line 1, line $p+1$ and line $2p+1$, and so on. When line $(m-1)p+1$ decomposition is completed, the first processor stops operations. This strategy minimizes the load imbalance. However, the communication overhead increased significantly.

Multi-line Shutter Storage Strategy: Multi-line shutter storage strategy takes the advantages of the above two. This algorithm is to decompose the matrix into blocks by rows, each block contains multiple lines and the i block stored into the $i \bmod p$ node machine. When block $(m-1)p+1$ decomposition is completed, the first processor will stop operation. Since each block contains multiple lines, the communication overhead is less than the single line shutter storage.

2.2.2 *The existing Decomposition strategy and mapping technology --- Cholesky decomposition*

The idea of the Cholesky Decomposition:

$$\begin{bmatrix} b_1 & c_1 & & & \\ a_2 & b_2 & c_2 & & \\ & \ddots & \ddots & \ddots & \\ & & a_{n-1} & b_{n-1} & c_{n-1} \\ & & & a_n & b_n \end{bmatrix} \begin{bmatrix} B_1 & C_1 & & & \\ C_1^T & B_2 & C_2 & & \\ & \ddots & \ddots & \ddots & \\ & & C_{q-1}^T & B_{q-1} & C_{q-1} \\ & & & C_q^T & B_q \end{bmatrix}$$

Fig. 1 The left is the matrix before decomposition, the right is the matrix after decmposition

Fig. 1 is the two matrixes before decomposition and after decomposition. C_i is an m-order square matrix, and only the lower left quarter element is not zero. B_i is an m-order positive definite tridiagonal matrix, and it is decomposed Cholesky decomposition method: $B_i = L_i D_i L_i^T$. D_i is a diagonal matrix. L_i is a unit lower triangular matrix. Matrix transformations are as follows.

In Fig. 2, $\overline{C_i} = L_i^{-1} C_i L_{i+1}^{-T} = C_i L_{i+1}^{-T}$ and all its elements, except for those in the last line, are 0. As D_i is a diagonal matrix, the elements of the last line of C_i can be eliminated to zero respectively except for the last element. C_i and D_i are the matrixes after elimination. Thus, the elements of the last lines of C_i, D_i as well as D_i^T form new three-diagonal matrix equations. The small three-diagonal matrix equations can be solved on a single processor, then the solution is sent to other processors. The original problem can be solved.

$$\begin{bmatrix} D_1 & \overline{C_1} & & & \\ \overline{C_1^T} & D_2 & \overline{C_2} & & \\ & \ddots & \ddots & \ddots & \\ & & \overline{C_{p-1}^T} & D_{p-1} & \overline{C_{p-1}} \\ & & & \overline{C_p^T} & D_p \end{bmatrix}$$

Fig. 2 The left is the matrix before decomposition, the right is the matrix after decmposition

Algorithm description:

- Step 1: Solve L_i, D_i Make $B_i = L_i D_i L_i^T$.
- Step 2: Transform $\overline{C_i} = C_i L_{i+1}^{-T}$.
- Step 3: Solve $d_{i,m-1} = d_{i,m-1} - \sum_{j=0}^{m-2} b_{ij}^2 / d_{i+1,j}$, Solve the small three-diagonal matrix equations on a single processor.
- Step 4: Solve the original problem.

This algorithm has good parallelism, but the computational complexity is double that of the serial algorithm. Despite an increase in parallelism, but the computational complexity reduces the efficiency of the algorithm. Therefore, we would like to improve the idea of this algorithm to reduce computational complexity.

2.2.3 *The improvement of storage strategy (Improved Multiline Wrapped Interleaved Row Storage IMWIRS)*

Large tridiagonal matrix belongs to large sparse matrixes, all elements, except for those near the diagonal, are zero. Storing those zero elements into the memory is a waste of memory resources. A new algorithm is to transform the matrix, and only store nonzero elements. This storage strategy is an improvement of the multi-line roller shutter storage method, which is specifically applicable to tridiagonal matrix.

Assuming a linear equations $Ax = y$, A is a tridiagonal matrix of n orders. The original storage method is to store A into a $n * n$ two-dimensional array. The improved method only needs 3*n memory space, which is only 3/n of the original one. With the increase of n, the advantage becomes more obvious.

$$A=\begin{bmatrix} b_1 & c_1 & & & \\ a_2 & b_2 & c_2 & & \\ & \ddots & \ddots & \ddots & \\ & & a_{n-1} & b_{n-1} & c_{n-1} \\ & & & a_n & b_n \end{bmatrix}$$

$$A'=\begin{bmatrix} 0 & b_1 & c_1 \\ a_2 & b_2 & c_2 \\ \vdots & \vdots & \vdots \\ a_{n-1} & b_{n-1} & c_{n-1} \\ a_n & b_n & 0 \end{bmatrix}$$

Fig. 3 A is the original storage method, which needs $n \times n$ array , and A’ is the transformed one with only $n \times 3$ memory space

In fig. 3, in order to be clear, the corresponding elements have the same label. In actual storage, we assume that A and A' respectively represent arrays before transformation and after transformation. a_{ii} is the element of A, and a'_{ii} is of A'. $a'_{il} = a_{ii}$, $i = 0,1,2,3,L,n$, $a'_{i0} = a_{ii-1}$,$i = 0,1,2,3,L,n$, $a'_{i2} = a_{ii+1}$ $i = 0,1,2,3,L,n$, $a'_{00} = 0$ a'_{n-12}.

2.2.4Decomposition strategy and mapping technology based on improved storage strategy

- **Step1**, decomposing a large tridiagonal matrix into blocks; assuming that the matrix is an n-order matrix, and decomposing it into m-order small square matrixes. Assuming q = n / m and there are p processors, according to multi-line shutter store strategy, the first processor stores the first m rows, the second processor stores the second m rows and so on, the i-th processor stores $i \bmod p$ m rows; there are $q \times m$ rows.

$$\begin{bmatrix} b_1 & c_1 & & & \\ a_2 & b_2 & c_2 & & \\ & \ddots & \ddots & \ddots & \\ & & a_{n-1} & b_{n-1} & c_{n-1} \\ & & & a_n & b_n \end{bmatrix} \begin{bmatrix} B_1 & C_1 & & & \\ A_2 & B_2 & C_2 & & \\ & \ddots & \ddots & \ddots & \\ & & A_{q-1} & B_{q-1} & C_{q-1} \\ & & & A_q & B_q \end{bmatrix}$$

Fig. 4 the left one is the original tridiagonal matrix, the right one is the matrix after decomposition

In Fig. 4, B_i is a tridiagonal matrix; C_i is a square matrix with only the lower left corner element is non-zero; A_i is a square matrix with only the top right corner element is non-zero. A_i, B_i and C_i are all m-order square matrixes.

- **Step2**, storing A_i , B_i and C_i into the i-th node according to the storage method introduced above.

In Fig. 5, $a_{(i-1)m+1}$ is the none-zero element of A_i; C_{im} is the none-zero element of C_i; the remaining elements constitute tridiagonal matrix B_i.

$$\begin{bmatrix} a_{(i-1)m+1} & b_{(i-1)m+1} & c_{(i-1)m+1} & & & \\ & a_{(i-1)m+2} & b_{(i-1)m+2} & c_{(i-1)m+2} & & \\ & \ddots & \ddots & \ddots & & \\ & & a_{im-1} & b_{im-1} & c_{im-1} & \\ & & & a_{im} & b_{im} & c_{im} \end{bmatrix} [A_i \quad B_i \quad C_i]$$

Fig. 5 The elements and blocks of the i-th node

$$\begin{bmatrix} a_{(i-1)m+1} & b_{(i-1)m+1} & c_{(i-1)m+1} \\ a_{(i-1)m+2} & b_{(i-1)m+2} & c_{(i-1)m+2} \\ \vdots & \vdots & \vdots \\ a_{im-1} & b_{im-1} & c_{im-1} \\ a_{im} & b_{im} & c_{im} \end{bmatrix}$$

Fig. 6 Showing how data is stored into the i-th node

- **Step3**, for each node, solving small tridiagonal matrix B_i and only one non-zero element matrixes A_i, C_i using Thomas algorithm. The first node don't have to deal with A_i, and the last node don't have to solve C_i.

2.2.5 *Description of Improved Algorithm*

According to the analysis above, we give the concrete description of the algorithm. The algorithm only stores the elements near the diagonal therefore can save memory overhead. In terms of decomposition strategy and mapping technology, we decompose large-scale three-diagonal matrix into small square matrixes; each processor needs to solve one small tridiagonal matrix and two square matrixes that contain only one element. Then each processor will send the results to the master processor to calculate the results of the original problem. The algorithm is described below.

Assuming the network topology of parallel computers is master-slave architecture and the linear equations are described as $Ax = b$.

- **Step1**, Inputting the order n of the large stiffness matrix into master computer. Assuming program decomposes the large matrix by line into q blocks (that is q slave nodes), then according to multi-line shutter storage strategy, master computer calculates how many lines each block contains, that is $m = n/q$.

- **Step2**, Inputting the elements of the large matrix $a_{ij}(i,j = 0,1,2,L,n-1)$ and vector b. While inputting the elements, master computer assigns from the first line to the $\mathrm{m'th}$ line and vector b to the first node, the line $\mathrm{m}+1$ to the line $2\mathrm{m}$ and vector b to the second node and so on, the remaining lines are assigned to the first node. It's worth noting that when assigning elements, master node only processes the principal diagonal elements.
- **Step3**, For each slave node, allocating an $m \times 3$ array and the elements assigned to it are stored in the array by IMWIRS storage strategy**.**
- **Step4**, For each slave node, decomposing the elements of the array into three m-order square matrix A_i, B_i and C_i by logic; the first element of the array is regarded as the lower right element of A_i, the last element is regarded as the top left element of C_i and the remaining elements is regarded as B_i's; the mapping method refers to figure 4.
- **Step5**, For each slave node, B_i is calculated according to single process Thomas algorithm.
- **Step6**, For each node, calculating $A_i(i = 2,3,L,p)$ and $C_i(i = 2,3,L,p-1)$, which contain only one element (The first element does not need to calculate A_i, and the last node (p) do not have to calculate C_i).
- **Step7**, For each node, returning the computation results to master computer, the master computer calculates the final result, and outputs the result vector $X_i = (x_{m(i-1)+1}, x_{m(i-1)+2}, L, x_{im})^T$.

2.2.6*Algorithm implementation using pseudo-code*

```
 master
 procedure distribution()
 Input n:integer;// order of the coefficient matrix
 q:integer;// number of the blocks(sometimes equals to the number of slave
computers)
 resultVector:Array; // result vector of Ax = b
 Begin
   m=n/q;// the number of lines in each block
   P current=0;//mark the current processed block
   while(p current<q){  //have not distributed all blocks
   for (j = 0; j < m; ++j)
     for(i=0; i < n; ++i)
     {
      If(|i-j|<=1)only the tridiagnal element be sent to slave computers
      {
        input a[i,j];// a[i,j] is an element of the coefficient matrix
        distribute a[i,j]to node computer P current%p;//p is the total number
   of  slave  computer
       }
      }
            Distribute resultArray to node Pcurrent%P;
            Pcurrent++;
  }
end
Slave p
procedure store()
Input a[i,j]//the coefficient elements which the master computer distribute to
it
new resultVector // result vector of Ax=b
Begin
   new coefficientArray // m*3 Array to store the elements which the master
        computer distribute to it.
```

```
29.        for(i=1;i<=m;++i)
30.        {
31.          coefficientArray[I,0] = a[i,i-1];
32.        }
33.        for(i=0;i<=m;i++i)
34.        {
35.          coefficientArray[i,1]=a[i,j];
36.        }
37.        for(i=0;i<=m-1;++i)
38.        {
39.          coeefficientArray[i,2]=a[i,i+1];
40.        }
41.        coefficientArray[0,0]=0;//the first element of the array
42.        coefficientArray[m-1,2]=0;//the last element of the array
43.        new resultArray:Array//n*1 Array to store result
44.        Copy resultVector to resultArray;
45.      End
46.      Slave computer p
47.      procedure calculate()//Thomas algorithm
48.      Begin
49.        $\alpha_0$ =coefficientArray[0,1]
50.        $\beta_0$ =coefficientArray[0,2]/ coefficientArray [0,1]
51.        $\alpha_i$= coefficientArray[i,1]- coefficientArray[i,0]$/\beta_{i-1} i = 1,2L,m-1$
52.        $\beta_i$= coefficientArray[i,2]/$\alpha_i$ $i = 1,2L,m-2$
53.        $y_0 = resultArray[0]/coefficientArray[0,1]$
54.        $y_{i=}$(resultArray[1]-$\alpha_i y_{i-1}$)/$\alpha_i$ $i = 1,2,L,m-1$
55.        $x_{m-1} = y_{m-1}$
56.        $x_i = y_i - \beta_i x_{i+1}$ $i = m-2, m-3, L, 0$
57.        send $X = (x_0, x_1, L, x_{m-1})$ to master
58.      End
59.      Master
60.      print $X = (x_0, x_1, Lx_{m\_1}, x_{m+0}, x_{m+1}, Lx_{2m-1} Lx_{pm+0}, x_{pm+1}, Lx_{n-1})^T$
```

2.2.7 Time and Space Complexity Analysis of Improved Algorithm

The original algorithm uses Gaussian elimination method and its time complexity is O(n) and space complexity is O(n^2).

Here, based on the above pseudo-code logic of improved algorithm, we analyze the time and space complexity of improved algorithm. Observing pseudo-code of line 10 to line 21 (matrix partition), we can see that it is double layer “for loop”**.** We know that when there are several loops, the time complexity of an algorithm is decided by the frequency f(n) of the innermost statement in the maximum loop nesting. In the pseudo-code, the maximum frequency of statements is line 15 and line 16 with n times inner loop and m times outer loop. According to line 7, we know that m=n/q (q is the number of computers), that is m and n is linear relationship and therefore the time complexity of this part of pseudo-code is T(n)=O(mn)=O(n^2). From line 29 to line 40 (matrix assignment), there are three single layer “for loop”, the scale of each loop is m, m and m+1, and corresponding time complexity are O(m), O(m) and O(m+1), that is their complexity are all O(n). In short, the time complexity should be the maximum times of statements executed within the whole code; therefore the final time complexity of improved algorithm is $T(n)=O(n^2+3n)=O(n^2)$.

As to space complexity, the main space cost is matrix element storage. According to algorithm description, there are q slave node computers and each node computer needs to create an array of m×3; therefore the total space needed is q×m×3=n×3, that is space complexity is $S(n)=O(3n)=O(n)$.

We can see that the time complexity of improved algorithm does not decrease compared with the original algorithm (owing to matrix partition increasing the complexity). However, the advantage of improved algorithm are mainly embodied in the high performance of parallel computing and great alleviation of space cost, especially for large size stiffness matrix, such as matrix data file is larger than 500M, that is matrix order is more than 10^8.

Through the verification below, we can see that the improved algorithm could enhance the computing speed of large size stiffness matrix equations greatly.

3. IMPROVED ALGORITHM VERIFICATION

Here, we use the improved algorithm to solve tridiagonal linear equations $AX \quad b$ to verify the efficiency of the algorithm. We employ a parallel method named MPI. MPI (a standard, a model of message passing interface, with a variety of implementations such as MPICH) is a tool to connect multiple hosts through network for parallel computing. We can also utilize it for multicore or multi-CPU parallel computing on one single machine but the efficiency is poor. It can coordinate several hosts together for parallel computing, and therefore it has good scalability in parallel computing. However, communication among processes could also lead to the problems of large memory overhead, low parallel efficiency as well as complexity in programing. In addition, we also consider stimulating parallel computing by employing OpenMP. OpenMP is designed for parallel computing on single host with multiple CPUs or multiple cores. In other words, OpenMP is more suitable for parallel computing on single machine with shared memory and since threads for parallel computing could share memory, it is of high efficiency and low memory overhead. Yet OpenMP is only available for parallel computing on single host rather than cluster. In order to verify the high efficiency of the improved algorithm and use as much resource as possible during the verification, we choose MPI, that is to apply multi hosts cooperating together for parallel computing.

For the tridiagonal linear equations $AX \quad Y$, we assign different orders of coefficient matrix by data scale, and these orders of coefficient matrix are $1 * 10^7$,$5 * 10^7$,$1 * 10^8$, $5 * 10^8$ and $1 * 10^9$. For each order, we verify it by 1 processor, 2 processors, 4 processors, 8 processors and 16 processors respectively. When the number of processor is 1, it means the original serial algorithm. In addition, T**(unit is second)** represents computation time. S represents speedup ratio and E represents parallel efficiency. Results of verification are shown from Table 1 to Table 5. In these tables, $S = T_1/T_m$, when m = 1, T_1 is the original serial computing time, and when m>1, T_m is the parallel computing time using different numbers of processors. The parallel efficiency E is equal to $T_m / m * T_m$.

Table 1 verification result of a $1*10^7$ order matrix

m(processor amount)	T_m(unit is second(s))	S($S=T_1/T_m$)	E($E=T_1/m*T_m$)
1	32.1	1	1
2	18.0	1.78	0.89
4	13.3	2.41	0.60
8	11.0	2.92	0.37
16	9.40	3.4	0.21

Table 2 verification result of a $5*10^7$ order matrix

m(processor amount)	T_m(unit is second(s))	S($S=T_1/T_m$)	E($E=T_1/m*T_m$)
1	130	1	1
2	59.4	2.19	1.10
4	45.1	2.88	0.72
8	40.9	3.18	0.40
16	33.6	3.87	0.24

Table 3 verification result of a $1*10^8$ order matrix

m(processor amount)	T_m(unit is second(s))	S($S=T_1/T_m$)	E($E=T_1/m*T_m$)
1	271	1	1
2	114	2.37	1.19
4	87.1	3.11	0.78
8	77.4	3.50	0.44
16	66.2	4.11	0.26

Table 4 verification result of a $5*10^8$ order matrix

m(processor amount)	T_m(unit is second(s))	S($S=T_1/T_m$)	E($E=T_1/m*T_m$)
1	1409	1	1
2	582	2.42	1.21
4	436	3.23	0.81
8	376	3.75	0.47
16	327	4.31	0.27

Table 5 verification result of a $1*10^9$ order matrix

m(processor amount)	T_m(unit is second(s))	S($S=T_1/T_m$)	E($E=T_1/m*T_m$)
1	2018	1	1
2	804	2.51	1.26
4	606	3.33	0.83
8	516	3.91	0.49
16	458	4.40	0.28

Based on the verification results shown in Table 1 to Table 5, we can see when the order is same, with the increasing of the number of processors, the computing time decreases significantly; but, with the number of processors increasing, the decreasing extent of computing time is slowing down. Verification manifests that with the increasing of order of the stiffness matrix, the computing efficiency of improved algorithm improves greatly.

As the order of the matrix and the number of processors are changing, the figures from figure 6 to figure8 show the relationship among order of matrix, number of processor m, computing time T, speedup ratio S and parallel efficiency E.

In Fig. 7, when the number of processors increases from one to two, the algorithm transforms from serial computing to parallel computing, and the computing time drops sharply, especially for large-scale matrix. When the scale of the matrix is larger, the less time it uses for parallel computing. On the other hand, with the increasing number of processors, the computing-time is becoming closer. Because when the number of processors is large enough, such as m=16, the processing performance is high enough to deal with different orders of matrix in a short time.

In Fig. 8, with the increasing number of processors, the speedup S of different scale matrix has a trend of linear increasing, especially when the number of processors increases from 1 to 2, speedup enhances greatly; then when m is more than 2, speedup ratio increases gently. In addition, we can see that when the number of processors is the same, bigger-scale matrix has higher speedup than smaller-scale matrix. It is because when communication cost is the same, the advantage of processing performance is more obvious when the scale of matrix is large.

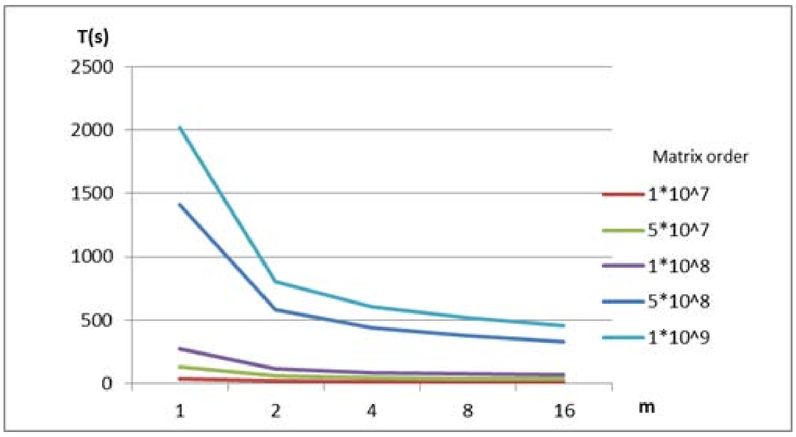

Fig. 7 The trends of T(s) with the increasing of m

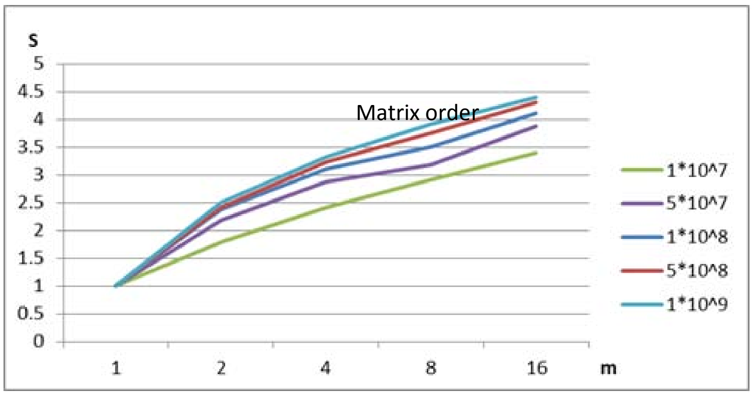

Fig. 8 The trends of S with the increasing of m

In Fig. 9, with the increasing number of processors, the parallel efficiency E of different scale matrix has a trend of linear decreasing. It is because the more processors are, the greater the communication cost is, which leads to the decline of parallel efficiency. In addition, we can see that when the number of processors is the same, bigger-scale matrix has higher parallel efficiency than smaller-scale matrix. The reason for this is that the advantage of parallel performance counteracts the communication cost when the scale is great enough. It also reflect the efficiency of the improved algorithm.

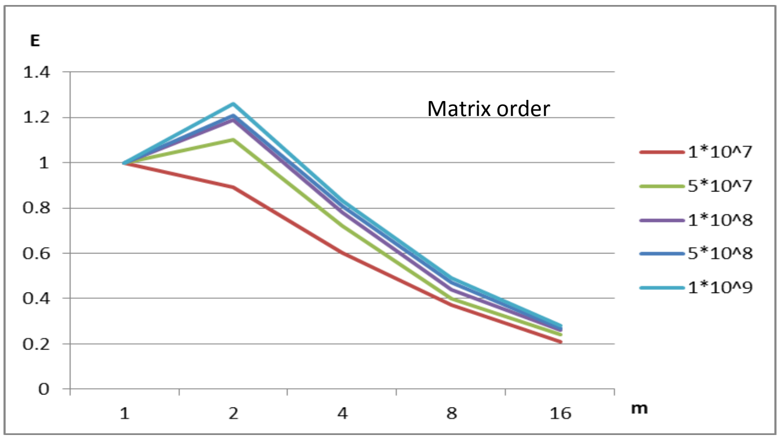

Fig. 9 The trends of E with the increasing of m

4. **CONCLUSIONS**

As you can see, in this paper, we analyze the original Thomas algorithm on single processor first, and we discuss the existing storage strategy of large tridiagonal matrix, existing decomposition strategy and mapping technology. And then we propose our improved Thomas algorithm based on our analysis. At last, we present the pseudo-code according to the idea of the improved algorithm and verify the efficiency of the improved algorithm by employing the parallel method MPI on various levels of specificity of data. The results of verification show the improved algorithm has good performance for linear finite element parallel computing, and it embodies in four aspects:

- Saving storage space. The space complexity of the improved algorithm is S(n)=O(3n)=O(n) and the space complexity of the original algorithm is S(n)=O(n^2). That is to say the space cost of improved algorithm is only 3/n times of original algorithm (Thomas Algorithm).
- Lower interaction overhead and computation complexity. Interaction overhead is small when the number of processors is in specific scope, and the computation complexity is far less than cholesky algorithm.
- Parallel efficiency decreases with too many processors. With the increasing of the number of processors, the speedup ratio increases. But, when the number is over 16, the increasing of speedup ratio lowers down and parallel efficiency starts to decline because of the overhead of communication increasing.
- The efficiency of parallel computation increases with the increasing size of matrix. The performance of the improved algorithm is very high, especially for large size matrix because the communication overhead among different computer nodes could be overlooked when the size of matrix is large enough.

In future, we will do more experiments using structure stiffness matrix in actual scenes to verify or further improve our algorithm and we will also apply our research results in engineering field especially in the concrete structure simulation.

ACKNOWLEDGMENTS

The authors gratefully acknowledge the financial support provided by Kwang-Hua Fund for College of Civil Engineering in Tongji University, the National Natural Science Foundation of China（Grant No: 41171303，51008226, 90915008 and 51178426），Hong Kong, Macao and Taiwan Science & Technology Cooperation Program of China (2012DFH70130), the Fundamental Research Funds of the Central Universities (2011QNA4016), Zhejiang Provincial Natural Science Foundation of China (LR13E080001). The authors also thanks the hard work provided by Xuefei Zhou and Xiaowei Zhou in this research program.

REFERENCES

Ananth Grama, Anshul Gupta, George Karypis, et al. (2003), “Introduction to Parallel Computing”, Beijing, China, June.

C.Xavier, S.S (2004), “IyengarIntroduction to Parallel Algorithms”, Beijing, China, June.

Efendiev, Y., Hou, T. Y. (2009), “Multiscale finite element methods”, *Applied and Computational Mathematics*, 217, 50.

Ferziger, J. H., Perić, M. (1996), “Computational methods for fluid dynamics (Vol. 3)”. Berlin: Springer.

Hendrickson, B., Kolda, T. G. (2000), “Graph partitioning models for parallel computing”, *Parallel Computing*, 26(12), 1519-1534.

Kim, H.S., Wu, S.Z., Chang, L.W. (2001), “A Scalable Tridiagonal Solver for GPUs[C] 2011 International Conference on Parallel Processing (ICPP), , 2011(9), 444-453.

Law, K. H. (1986), “A parallel finite element solution method. *Computers & Structures*”, 23(6), 845-858.

Liu, W.J., Wang R.Q. (2009), “Parallel Computing Based Finite Element Analysis”. 2009 1st International Conference on Information Science and Engineering (ICISE 2009), 2009, 295-298

Lv H, Di R.H., Gong H., Li C.X. (2011), “A MPI/OpenMP hybrid parallel nonlinear equation solver used in finite element analysis”. 2011 Sixth China Grid Annual Conference (China Grid), 2011, 215-219

Maurer, D., Wieners, C. (2011), “A parallel block LU decomposition method for distributed finite element matrices”. *Parallel Computing*, 2011, **37(12)**, 742-58

Paz, C. N. M., Alves, J. L. D., Ebecken, N. F. F. (2005), “Assessment of computational performance for a vector parallel implementation: 3D probabilistic model discrete cracking in concrete”. *Computers & Concrete*, 2(5), 345-366.

Pacheco, P. S. (1997), “Parallel programming with MPI”, Morgan Kaufmann Pub.

Sun, W.Y., Du, Q.K., Chen, J.R. (2007). “Calculation Method”, Beijing, China, May.

Takizawa, K., Tezduyar, T. E. (2012), “Computational methods for parachute fluid–structure interactions”, *Archives of Computational Methods in Engineering,* **19**(1), 125-169.

Turmo, J., Ramos, G., & Aparicio, A. C. (2012). “Towards a model of dry shear keyed joints: modelling of panel tests”, *Computers & Concrete*, **10**(5), 469-487.

Wang, X.B., Zhong, Z.H. (2004), “Generalized Thomas algorithm for solving cyclic tridiagonal equations”, *Journal of Computer Mechanics,* **104**(2),73-76.