Journal of Computational Mathematics
Vol.xx, No.x, 2025, 1–22.

http://www.global-sci.org/jcm
doi:10.4208/jcm.2505-m2024-0124

HIGH PERFORMANCE PARALLEL BLOCK SUPERNODE DIRECT SOLVER FOR STIFFNESS EQUATION*

Qiang-Gui Jin[1)] and Yao-Hui Ma
National Key Laboratory of Strength and Structural Integrity, Aircraft Strength Research Institute of China, Xi'an 710065, China
Emails: 2753657024@qq.com, 656016575@qq.com

Abstract

The process of direct method to solve large sparse linear equation mainly includes reordering, symbolic factorization, numerical factorization and triangular solving. Traditional symbolic factorization predicts the pattern of $\boldsymbol{L}$ based on single column and single row index. We propose to directly partition supernodes based on characteristics of matrix reordered by METIS, and then perform parallel symbolic factorization based on supernodes and row index fragments. A parallel block supernode numerical factorization strategy is proposed based on the concept of task pool here. In triangular solving stage, unlike traditional algorithms based on DAXPY and DDOT operations, we propose a new parallel triangular solving algorithm based on DGEMM and DTRSM operations. We name the parallel solver as finite element analysis direct solver (FEADS) and compare it with the advanced MKL PARDISO and MUMPS. The stiffness equations of 394770 and 719871 dimensions are solved using the solvers on two different computers. On the first computer, the solving efficiency of FEADS and MKL PARDISO is comparable, while MUMPS is relatively backward. On the second computer, FEADS performs especially well. For solving the case with 394770 dimensions, FEADS leads MKL PARDISO and MUMPS by 21.92% and 42.35%, respectively. For solving the case with 719871 dimensions, FEADS leads 34.75% and 38.38% respectively.

Mathematics subject classification: 65N06, 65B99.
Key words: Cholesky equation solving, Parallel algorithm, Block supernode, Symbolic factorization, Triangular solving.

1. Introduction

The structural finite element analysis forms stiffness equation $\boldsymbol{Ax} = \boldsymbol{B}$. $\boldsymbol{A}$ is the stiffness matrix which is sparse symmetric positive definite (SPD) and diagonally dominant. $\boldsymbol{x}$ is the matrix formed by multiple displacement/rotation component column vectors of finite element nodes to be solved. $\boldsymbol{B}$ is the matrix formed by multiple equivalent node-load column vectors. Iterative method and direct method are two commonly used methods for solving large sparse linear equations [32]. Here we only focus on the direct Cholesky method for SPD stiffness matrix equations [9,27]. At present, the advanced direct solvers such as MUMPS [3], PARDISO [6,31, 32] and SuperLU [22] have been developed based on factorization methods such as multifront method, block supernode method, and Gaussian elimination method. Fully utilizing multi-

* Received June 26, 2024 / Revised version received September 24, 2024 / Accepted May 12, 2025 / Published online October 15, 2025 /
1) Corresponding author

core and high-speed cache resources of computer for solving is an effective way to further improve solving efficiency [6]. The block supernode method constructs dense submatrices and utilizes high-performance multi-threaded and cache-optimized level-3 BLAS [11, 13, 14] and LAPACK [4, 12] subroutines to perform the dense submatrix calculations, which makes full use of multi-core and high-speed cache resources of computers.

The process of direct method usually includes reordering, symbolic factorization, numerical factorization and triangular solving. Reordering can effectively reduce the number of fill-ins generated during the numerical factorization. In order to allocate sufficient memory for numerical factorization and provide reference for scheduling the numerical factoring process, symbolic factorization predicts the positions of fill-ins to obtain necessary information. Numerical factorization formally factors the reordered matrix into a lower triangular matrix $\boldsymbol{L}$, which is the most time-consuming stage in the entire solving process. Triangular solving includes two steps of forward substitution and backward substitution, solutions are obtained after this stage.

For symmetric matrices, symmetric reordering is often used. Common symmetric reordering strategies include minimum degree [16], approximate minimum degree [1, 7, 8], minimum fill-in [28], as well as hybrid algorithms METIS [15, 18–21] and SCOTCH [30]. Guermouche *et al.* [17] found that METIS and SCOTCH yields wider and more balanced elimination trees, which are more suitable for parallel solving. Nowadays, both MUMPS and PARDISO recommend using METIS as reordering algorithm [23, 26]. Traditional symbolic factorization predicts positions of fill-ins based on single column and single non-zero row index, which has a large time complexity [6]. Based on the idea of dimensionality reduction, this paper proposes to directly partition supernodes on the matrix reordered by METIS, and perform symbolic factorization with supernodes and row index fragments. Due to the fact that the numbers of supernodes and row index fragments are much smaller than the numbers of matrix columns and non-zeros, the time complexity of symbolic factorization is greatly reduced.

Numerical factorization is the most time-consuming stage in the entire solving process, and mainstream numerical factorization methods include multi-front method [2], block supernode method [32], Gaussian elimination method [22], etc. This paper only focuses on factoring SPD matrices, using the block supernode Cholesky method for numerical factorization, and applying the task pool model for parallel implementation. For the sparse triangular solving problem (SpTRSV), parallel implementation is not easy because the solutions of the unknowns are often interrelated [5, 24, 25, 29, 33]. To fully utilize the multi-core and multi-threaded resources of computers, this paper proposes a task pool template based on OpenMP, and flexibly applies it to stages of symbolic factorization, numerical factorization and triangular solving. Unlike traditional triangular solving algorithms that operate on single row and column using DAXPY and DDOT operations [6], this paper proposes novel triangular solving algorithms based on DGEMM and DTRSM operations, and combining them with the task pool template for parallel implementation. After test and comparison, the new triangular solving algorithms achieve better efficiency than MKL PARDISO and MUMPS.

This paper is organized as follows. Section 2 presents the principle and parallel algorithm of the proposed approximate symbolic factorization (ASF). Section 3 introduces the principle and parallel algorithm of block supernode Cholesky numerical factorization. Section 4 introduces triangular solving algorithms based on supernode structure, including both forward and backward substitution algorithms. Test results and optimal parallel solving performances of solving real stiffness equations on two different computers are compared among FEADS, MKL PARDISO and MUMPS in Section 5. Section 6 makes brief summary.

2. Parallel Approximate Symbolic Factorization

2.1. Predicting fill-ins' positions and concept of elimination tree

1) Predicting the positions of fill-ins.

As mentioned before, the main job of symbolic factorization is to predict the positions of fill-ins. Here we briefly introduce how to predict the positions of fill-ins in SPD matrices. If there is a set

$$S = \{a_{js} \,|\, a_{js} \neq 0,\ s < j\},$$

whose elements are at row j and on the left side of a_{jj}, and for each element a_{js} of S, there is a set

$$S_S = \{a_{ks} \,|\, a_{ks} \neq 0,\ k > j,\ s < j\},$$

whose elements are at column s and below a_{js}, then set

$$S_{SJ} = \{a_{kj} \,|\, a_{ks} \neq 0,\ a_{kj} = 0,\ k > j,\ s < j\},$$

represents the set of fill-ins in column j and produced by column s. That is, if there is a non-zero a_{ks} of S_s in row k, and a_{kj} is zero, then a_{kj} is the fill-in produced by a_{ks}. In actual situation, we do not consider whether a_{kj} is zero or not for efficiency.

Matrix (2.1) is the lower triangular part of a SPD matrix, X denotes the fill-in of $\boldsymbol{L}$. We now pay attention to the fill-in in column 5. Firstly, $a_{55} = 6.0$, there is a non-zero $a_{54} = 1.0$ on the left side of a_{55} in column $s = 4$, and there is a non-zero $a_{74} = 1.0$ below a_{54} in row 7, and $a_{75} = 0$, then a_{75} is the fill-in of column j=5,

$$\begin{bmatrix} 1 & & & & & & & \\ 0 & 2 & & & & & & \\ 0 & 0 & 3 & & & & & \\ 1 & 0 & 1 & 4 & & & & \\ 0 & 0 & 0 & 0 & 5 & & & \\ 0 & 0 & 0 & 0 & 1 & 6 & & \\ 0 & 0 & 0 & 0 & 0 & 0 & 7 & \\ 0 & 1 & 0 & 1 & 1 & X & 1 & 8 \end{bmatrix}. \tag{2.1}$$

2) Algorithm of elimination tree.

The elimination tree usually records the dependency information between columns or supernodes of a matrix, which can be applied in multiple stages such as symbolic factorization and numerical factorization, and plays an important role in the sparse matrix factorization process [23]. Each column or supernode of the matrix represents a node on the elimination tree, and except the $(N-1)$-th node, all other nodes can have one parent node. An N-dimensional array P can be used to represent the elimination tree structure in the program. If j is the parent of k, then position $*(P+k)$ stores k's parent j, that is, $P[k] = j$. To obtain the elimination tree, it is necessary to first obtain the lower triangular matrix in compressed sparse row (CSR) format, as (2.2),

$$int\ N = 8, \tag{2.2a}$$

$$int\ NNZ = 15, \tag{2.2b}$$

$$int\ RAp[\,] = \{0, 1, 2, 3, 6, 7, 9, 10, 15\}, \tag{2.2c}$$

$$int\ CA[\,] = \{0, 1, 2, 0, 2, 3, 4, 4, 5, 6, 1, 3, 4, 6, 7\}. \tag{2.2d}$$

Algorithm 2.1 is the algorithm of computing elimination tree, which is modified from reference [6] to make it applicable to programming languages numbering from 0. P is the N-dimensional array used to store elimination tree, with its elements initialized to 0, a is an N-dimensional auxiliary array, with elements initialized to -1, i and j are auxiliary variables initialized to 0.

Algorithm 2.1: Algorithm of Computing Elimination Tree.

Input : Lower triangular part of reordered $\boldsymbol{A}$ stored in CSR format.
Output: Array P that contains information of elimination tree.

```
1  P ← 0, a ← −1, i = 0, j = 0.
2  for k = 1, ..., N − 1 do
3      for m = RAp[k], ..., RAp[k + 1] − 1 do
4          i = CA[m].
5          while i ≠ −1 and i < k do
6              j = a[i].
7              a[i] = k.
8              if j = −1 then
9                  P[i] = k.
10             end
11             i = j.
12         end
13     end
14 end
```

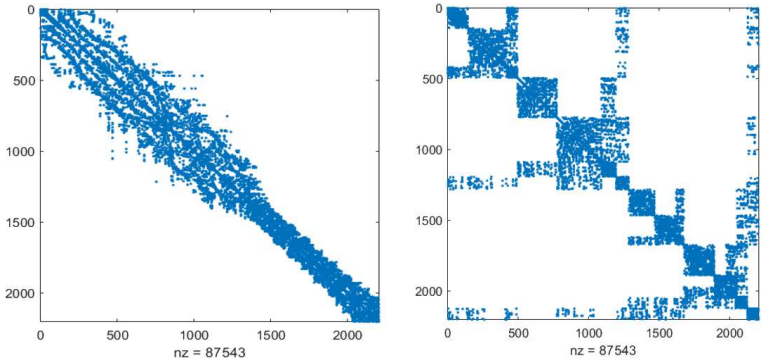
2.2. Approximate symbolic factorization

2.2.1. The process of ASF

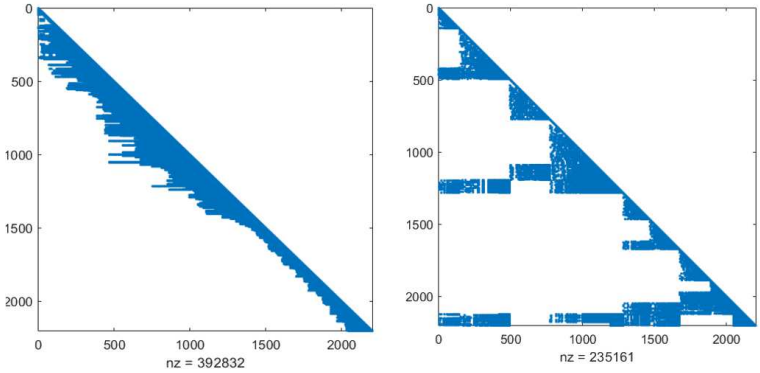
Traditional symbolic factorization uses single column and single row index for $\boldsymbol{L}$'s pattern predicting, the time complexity of the algorithm is $\mathcal{O}(|\boldsymbol{L}|)$. As the dimension of matrix and the number of non-zeros continue to increase, the time required for symbolic factorization will dramatically increase, even exceeding the time required for numerical factorization.

Based on the idea of dimensionality reduction, we propose a new symbolic factorization strategy called approximate symbolic factorization. Traditional symbolic factorization obtains supernode partition after the pattern of $\boldsymbol{L}$ has been gotten, while here we propose to directly partition supernode based on the matrix reordered by METIS. We explore characteristic of matrix reordered by METIS. Patterns in Fig. 2.1(a) belong to matrices before and after METIS reordering and patterns in Fig. 2.1(b) belong to their corresponding $\boldsymbol{L}$. It is evident that the non-zeros after METIS reordering exhibit a regular block-shape, and the corresponding $\boldsymbol{L}$ pattern indicates that they both have similar block regions. Usually, supernodes need to be partitioned based on $\boldsymbol{L}$'s pattern, but here we propose to partition supernodes on the pattern of reorder $\boldsymbol{A}$ as an approximation based on the above feature.

As Fig. 2.2, we mark the boundary rows of block-shape regions with arrows and horizontal lines. How to accurately locate these boundary rows in the program is crucial. Two characteristics of these boundary rows are summarized here. The first characteristic is that the boundary rows have only one nonzero because program stores only the lower and diagonal part of re-



(a) Patterns before and after reordered by METIS



(b) Corresponding patterns of $\boldsymbol{L}$ in Fig. 2.1(a)

Fig. 2.1. Patterns of $\boldsymbol{A}$ and $\boldsymbol{L}$.

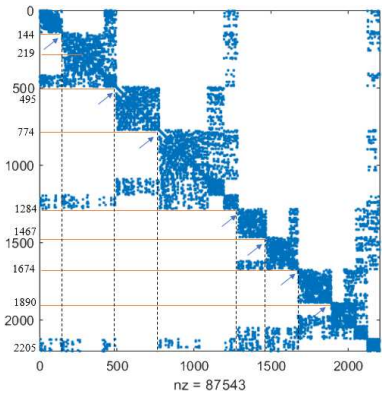


Fig. 2.2. Matrix reordered by METIS.

ordered $\boldsymbol{A}$. This condition is easily met because we use METIS as reordering algorithm which uses the method of multilevel nested dissection. The second characteristic is that the nonzero number of boundary rows' previous row must greater than one. If we denote the nonzero number of the row j as RN_j, then $RN_j - RN_{j-1} < 0$ is bound to hold. We set a threshold TH to filter out more obvious boundary rows. If we set $TH = -15$, then the boundary rows of matrix in Fig. 2.2 obtained by FEADS are $144, 219, 495, 774, 1284, 1467, 1674, 1890$, as marked in the

figure. Then we can use the same columns as these boundary rows to partition supernodes. Supernodes are divided with dashed lines in Fig. 2.2. To control the size of supernodes, we set a threshold STD_SN_SIZE. If the size of a supernode is much greater than STD_SN_SIZE, then partition it into supernodes smaller or a little bit greater than STD_SN_SIZE. If we set $STD\_SN\_SIZE = 100$, then the final partition of supernode can be shown as

$$\begin{aligned} supernode[\,] = \{&0, 144, 219, 319, 495, 595, 774, 874, 974, 1074, 1174,\\ &1284, 1467, 1567, 1674, 1774, 1890, 1990, 2090, 2205\}. \end{aligned}$$

After the partitioning of supernode, it needs to get fragments information of each supernode to further reduce the complexity of symbolic factorization. Taking matrix (2.1) as an example, whose supernode partition is $supernode[\,] = \{0, 4, 6, 8\}, SN = 3$, then its row index information can be kept in the form (2.3) where array FN stores the fragments number of each supernode, array $Start$ and End store the start and end information of each fragment. Based on the fragments, we mark block supernodes with boxes in Fig. 2.3. In order to efficiently use level-3 BLAS and LAPACK subroutines for dense matrix operation, we fill traditional trapezoidal supernode into block supernode with zeros,

$$\begin{aligned} &FN[\,] = \{2, 2, 1\},\\ &Start[0][\,] = \{0, 7\}, \quad Start[1][\,] = \{4, 7\}, \quad Start[2][\,] = \{6\},\\ &End[0][\,] = \{4, 8\}, \quad End[1][\,] = \{6, 8\}, \quad End[2][\,] = \{8\}. \end{aligned} \tag{2.3}$$

Afterwards, based on the obtained fragments information, it can construct the adjacency matrix of supernodes, whose size is $SN * SN$. For each supernode J, if one of its fragments has common row indices as the diagonal matrix of its subsequent supernode K, then the two supernodes have a connection relationship. Or it can be said that K is J's ancestor supernode. Then J's all ancestor supernodes in reordered $\boldsymbol{A}$ can be represented as set

$$\begin{aligned} S_J = \{K \,|\, &(Start[J][i] \geq supernode[K] \text{ and } Start[J][i] < supernode[K+1]) \text{ or}\\ &(Start[J][i] < supernode[K] \text{ and } End[J][i] > supernode[K]),\\ &i \in [0, FN[J]), K \in [J+1, SN)\}. \end{aligned} \tag{2.4}$$

According to this, the supernode adjacency matrix (only the lower triangular part) of matrix (2.1) is (2.5). All diagonal positions are filled with 1, and non-diagonal positions are filled

$$\begin{bmatrix} 1 & 0 & 0 & 0 & & & & \\ 0 & 2 & 0 & 0 & & & & \\ 0 & 0 & 3 & 0 & & & & \\ 1 & 0 & 1 & 4 & & & & \\ 0 & 0 & 0 & 0 & 5 & 0 & & \\ 0 & 0 & 0 & 0 & 1 & 6 & & \\ 0 & 0 & 0 & 0 & 0 & 0 & 7 & 0 \\ 0 & 1 & 0 & 1 & 1 & X & 1 & 8 \end{bmatrix}$$

Fig. 2.3. Block supernodes of matrix (2.1).

with 1 to indicate a connection relationship, 0 indicates no connection relationship. At this point, we have successfully transformed the symbolic factorization problem from 8-th order to 3-th order. Subsequently, by applying Algorithm 2.1, the elimination tree of supernodes can be quickly obtained to guide the parallel symbolic factorization process. The elimination tree of supernodes represented by a tree diagram is shown in Fig. 2.4,

$$\begin{bmatrix} 1 & & \\ 0 & 1 & \\ 1 & 1 & 1 \end{bmatrix}. \tag{2.5}$$

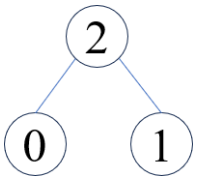


Fig. 2.4. Elimination tree of matrix (2.1)'s supernodes.

2.2.2. The parallel algorithm of ASF

Task pool is a commonly used parallel strategy that achieves parallel computing. A thread can add new tasks into task pool once it finishes a task and any idle thread can try to receive an executable task without waiting, which can maximize parallelism. The parallel algorithm of ASF is achieved by a template of task pool based on OpenMP, as Algorithm 2.2. In the first line of Algorithm 2.2, two task pools are created to store two different kinds of tasks. *task_add* is used to store the type of tasks that child supernodes add their contribution fragments to their parent supernodes, while *task_merge* stores the type of tasks that merge fragments of supernodes. For example, in matrix (2.1), supernode 2 is the parent supernode of 0 and 1, then both 0 and 1 need to add their contribution fragments to 2. Then, after receiving the contribution fragments of 0 and 1, the supernode 2 needs to merge its fragments to ensure that there are no duplicate fragments. If I represents a fragment of J, then the contribution fragments set of J to its parent node F can be represented as set

$$S_{JF} = \{I \,|\, \exists\, k \in I,\ k \geq supernode[F]\}. \tag{2.6}$$

Afterwards, it adds all independent child supernodes to *task_add* (line 3), and then enters the OpenMP parallel area. It is prior for each thread to receive merge tasks first in each loop. When receiving add tasks, it is important to note that multiple child supernodes cannot add fragments to the same parent node simultaneously. Therefore, an additional auxiliary array needs to be used to restrict the receiving of add tasks. For example, a SN-dimensional array J_lock can be used for state tagging. If $J\_lock[J] = 1$, it indicates that one thread is adding fragments to J, and the current thread cannot receive the add task whose parent is J. Tasks are performed according to its type next (line 16). When determining whether all tasks have been completed, it can be checked whether all root supernodes have completed the merge tasks. Usually, only $SN-1$ is the root supernode, but there may be multiple root supernodes in special cases.

Algorithm 2.2: Algorithm of Parallel Approximate Symbolic Factorization.

Input : Matrix reordered by METIS and supernode partition information.
Output: Approximate pattern of $\boldsymbol{L}$.

```
1  task_merge ← 0, task_add ← 0.
2  task_merge_N = 0, task_add_N = 0, TASK_FINISHED = 0.
3  Add all independent supernodes into task_add and initialize task_add_N.
4  #pragma omp parallel.
5  {
6  while TASK_FINISHED = 0 do
7  |  current_task = −1, task_type = −1.
8  |  #pragma omp critical.
9  |  {
10 |  if task_merge_N > 0 then
11 |  |  task_merge_N = task_merge_N − 1, task_type = 0.
12 |  |  current_task = task_merge[task_merge_N].
13 |  end
14 |  if task_type = −1 and task_add_N > 0 then
15 |  |  Get an executable task from task_add, then update task_add and task_add_N.
16 |  |  task_type = 1.
17 |  end
18 |  }
19 |  Perform current_task according to task_type.
20 |  #pragma omp critical.
21 |  {
22 |  Add new tasks into task pools.
23 |  }
24 |  if There are no more tasks needed to be performed then
25 |  |  TASK_FINISHED = 1.
26 |  end
27 end
28 }
```

3. Parallel Numerical Factorization

3.1. Block supernode Cholesky numerical factorization

Numerical factorization is the most time-consuming stage in the entire equation solving process. This paper adopts the block supernode Cholesky factorization method for SPD matrices. Based on the approximate $\boldsymbol{L}$ pattern obtained by ASF, the original reordered matrix is factored into $\boldsymbol{L}\boldsymbol{L}^\top$. Daydé and Duff [10] detailed various methods of dense matrix block factorization in their article, such as JIK-SDOT, JKI-GAXPY, KJI-AXPY, etc. For large sparse matrices, the concept of block partition was extended to the concept of supernode. Based on the non-zero's characteristics in $\boldsymbol{L}$, multiple columns with similar non-zeros are flexibly regarded as a whole and called supernode. Combining the concept of supernode, this paper extends the JIK_SDOT method to the Cholesky factorization of large SPD matrices, known as block

supernode Cholesky factorization (see Fig. 3.1).

As shown in Fig. 3.1, the numerical factorization process of a supernode J consists of two main steps. The first step is external modification, where J needs to separately perform DGEMM operation with all its descendant supernodes. Taking the modification of S_2 to J as an example, it performs DGEMM operation as (3.1), where $\boldsymbol{B} = (\boldsymbol{A1})^\top$ due to symmetry. This operation is performed using the high-performance multi-threaded and cache-optimized level-3 BLAS subroutine `dgemm()`,

$$\begin{bmatrix} \boldsymbol{C1} \\ \boldsymbol{C2} \\ \boldsymbol{C3} \\ \boldsymbol{C4} \end{bmatrix} = \begin{bmatrix} \boldsymbol{C1} \\ \boldsymbol{C2} \\ \boldsymbol{C3} \\ \boldsymbol{C4} \end{bmatrix} - \begin{bmatrix} \boldsymbol{A1} \\ \boldsymbol{A2} \\ \boldsymbol{A3} \\ \boldsymbol{A4} \end{bmatrix} \begin{bmatrix} \boldsymbol{B} \end{bmatrix}. \tag{3.1}$$

After all the descendant supernodes of J have completed their external modification to J, J can start independent internal factorization, as shown in Fig. 3.2. Firstly, it performs Cholesky factorization on the diagonal square matrix of J, as Eq. (3.2) which is accomplished by the LAPACK subroutine `dpotrf()`. Since the matrix targeted here is symmetric positive definite, which is numerically stable, the pivoting operation is not needed. Then, it updates the fragments below the diagonal based on the $\boldsymbol{L}_1$ matrix as shown in Eq. (3.3), which is completed by the BLAS subroutine `dtrsm()`,

$$\boldsymbol{C1} = \boldsymbol{L}_1 \boldsymbol{L}_1^\top, \tag{3.2}$$

$$\begin{bmatrix} \boldsymbol{C2} \\ \boldsymbol{C3} \\ \boldsymbol{C4} \\ \boldsymbol{C5} \\ \boldsymbol{C6} \end{bmatrix} = \begin{bmatrix} \boldsymbol{C2} \\ \boldsymbol{C3} \\ \boldsymbol{C4} \\ \boldsymbol{C5} \\ \boldsymbol{C6} \end{bmatrix} \left[\left(\boldsymbol{L}_1^\top \right)^{-1} \right]. \tag{3.3}$$

There are S_1, S_2, J, and K four supernodes in Fig. 3.1. Since the non-zero fragments of S_1 and S_2 have common row indices with J's diagonal square matrix, $P[S_1] = J$ and $P[S_2] = J$. Similarly, $P[J] = K$, and the corresponding elimination tree is shown in Fig. 3.3. To correctly

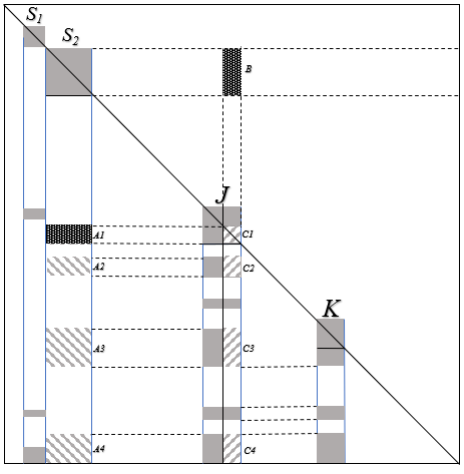


Fig. 3.1. Block supernode Cholesky factorization.

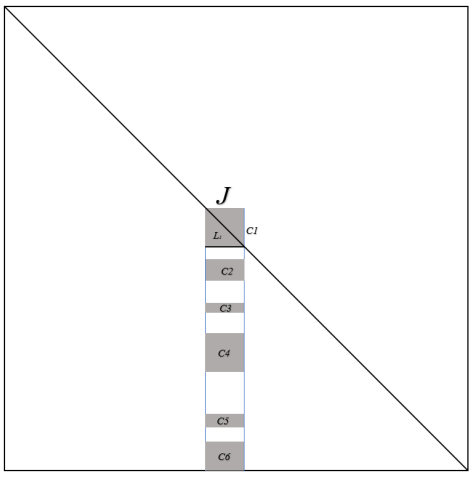


Fig. 3.2. Independent internal factorization.

complete the factorization process, the factorization tasks of these four supernodes must be carried out in a certain order.

Firstly, S_1 and S_2 do not have any descendant supernodes, their internal factorization tasks can be performed simultaneously. When S_1 and S_2 are factored, three new external modification tasks are immediately generated, they are S_1 to J, S_2 to J and S_2 to K respectively . Among the three tasks, the tasks have the same ancestor cannot be executed simultaneously. That is, tasks S_1 to J and S_2 to J cannot be executed at the same time. The reason is that they will both change J's data, which may cause data race and result in incorrect result. However, modification tasks for different ancestors can be executed simultaneously, such as S_2 to J and S_2 to K. This is the basis of numerical factorization to achieve parallelism.

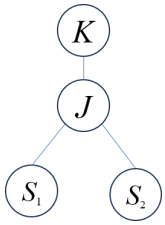


Fig. 3.3. Elimination tree of supernodes.

Based on the elimination tree, a more detailed supernode relationship storing structure is proposed here, as shown in (3.4). The two-dimensional array FJ stores the information of each supernode's all ancestors that constructed from the elimination tree in Fig. 3.3, while FJ_N stores the number of ancestors. Conversely, a two-dimensional array SJ is also used to store the information of each supernode's all descendants. It should be pointed out that although all the ancestor information of any supernode S can be directly obtained from the elimination tree, not all of these ancestors need S's external modification. For example, although K is an ancestor of S_1 on the elimination tree, actually S_1 does not have contribution to K. In Fig. 3.1, since S_1's fragments do not have the same row indices as K's diagonal matrix, S_1 to K is an invalid modification task. Therefore, it is necessary to verify the ancestors of each supernode. The invalid ancestor can be removed directly, as (3.5) and (3.6),

$$\begin{aligned} &FJ[S_1][\,] = \{J, K\}, \quad &&FJ\_N[S_1] = 2, \\ &FJ[S_2][\,] = \{J, K\}, \quad &&FJ\_N[S_2] = 2, \\ &FJ[J][\,] = \{K\}, \quad &&FJ\_N[J] = 1, \\ &FJ[K][\,] = \{\,\}, \quad &&FJ\_N[K] = 0, \end{aligned} \tag{3.4}$$

$$\begin{aligned} &FJ[S_1][\,] = \{J\}, \quad &&FJ\_N[S_1] = 1, \\ &FJ[S_2][\,] = \{J, K\}, \quad &&FJ\_N[S_2] = 2, \\ &FJ[J][\,] = \{K\}, \quad &&FJ\_N[J] = 1, \\ &FJ[K][\,] = \{\,\}, \quad &&FJ\_N[K] = 0, \end{aligned} \tag{3.5}$$

$$\begin{aligned} &SJ[S_1][\,] = \{\,\}, \quad &&SJ\_N[S_1] = 0, \\ &SJ[S_2][\,] = \{\,\}, \quad &&SJ\_N[S_2] = 0, \\ &SJ[J][\,] = \{S_1, S_2\}, \quad &&SJ\_N[J] = 2, \\ &SJ[K][\,] = \{S_2, J\}, \quad &&SJ\_N[K] = 2. \end{aligned} \tag{3.6}$$

3.2. Parallel algorithm of numerical factorization

Algorithm 3.1 is the parallel algorithm of numerical factorization based on task pool. $task\_in$ is the task pool of internal factorization, $task\_S$ and $task\_J$ store the tasks of external modification. We initialize the arrays and variables first and then put all supernodes without descendants into $task\_in$ for parallel internal factorization tasks. Simply, if $SJ\_N[J] = 0$, it means that J has no descendants. In parallel area, it initializes the current task and task type to -1 at the

Algorithm 3.1: Parallel Algorithm of Numerical Factorization.

Input : Lower triangular part of reordered $\boldsymbol{A}$ stored in CSC format and $\boldsymbol{L}$'s approximate pattern obtained by ASF.
Output: The factored lower triangular matrix $\boldsymbol{L}$.

```
1  task_in ← 0, task_S ← 0, task_J ← 0.
2  task_in_N = 0, task_out_N = 0, TASK_FINISHED = 0.
3  Add all supernodes with 0 descendant into task_in and initialize task_in_N.
4  #pragma omp parallel.
5  {
6  while TASK_FINISHED = 0 do
7      S = −1, J = −1, task_type = −1.
8      #pragma omp critical.
9      {
10     if task_in_N > 0 then
11         task_in_N = task_in_N − 1.
12         S = task_in[task_in_N].
13         task_type = 0.
14     end
15     if task_type = −1 and task_out_N > 0 then
16         Get an executable modification task and update task_S, task_J and task_out_N.
17     end
18     }
19     if task_type=0 then
20         Perform internal factorization task S.
21     end
22     if task_type=1 then
23         Perform external modification task S to J.
24     end
25     #pragma omp critical.
26     {
27     Add new tasks into task pools.
28     }
29     if There are no more tasks needed to be performed then
30         TASK_FINISHED = 1.
31     end
32 end
33 }
```

beginning of each loop. When receiving tasks, it is prior to receive internal factorization tasks first, which are currently placed in variable S. Only when there are no internal factorization tasks, then external modification tasks can be obtained. Generally, the earlier a task is added, the prior it should be obtained. It should be noted again that multiple external modification tasks with the same ancestor cannot be performed simultaneously. Therefore, an auxiliary array should be used to mark the ancestor who is being operated, and this task should be skipped when receiving tasks. The operations of altering the contents of auxiliary array should be performed within the critical region.

Once a valid task is received, the task will be executed according to its type (lines 16-21). Every time a supernode S completes its internal factorization task, the external modification tasks that S to all its ancestors can be immediately added to task pool. When the times that J participates in external modification tasks as ancestor are equal to the number of J's descendants, J should be put into *task_in*. It can break from while loop only when the completion amount of internal factorization is equal to the number of supernode.

4. Parallel Triangular Solving

4.1. Parallel supernode forward substitution

4.1.1. The introduction of supernode forward substitution

The triangular solving includes two steps of forward and backward substitution, which are performed based on the lower and upper triangular matrices obtained from numerical factorization. For Cholesky factorization, due to symmetry, only the $\boldsymbol{L}$ lower triangular matrix is obtained, and both forward and backward substitution are performed on it. Firstly, the right-hand side matrix $\boldsymbol{B}$ is necessary to perform the same row permutation as $\boldsymbol{A}$. If b_{ij} is located in the row i and column j of $\boldsymbol{B}$, then after permutation, it is located in the row $iperm[i]$ and column j. $iperm$ is the inverse permutation sequence obtained by METIS.

The traditional forward substitution algorithm updates the right-hand side column by column with a low efficiency based on DAXPY operation. To fully utilize the characteristics of supernodes, a novel forward substitution method based on dense matrix operations is proposed here. Its core operations are completed by the multi-threaded high-performance `dtrsm()` and `dgemm()` subroutines. As shown in Fig. 4.1, a supernode J and the right-hand side matrix $\boldsymbol{B}$ are presented, where $\boldsymbol{B}$ is stored in a dense form with column major. When it is time to operate on supernode J to update $\boldsymbol{B}$, the first step is to perform a dense triangular solving, as Eq. (4.1), where $\boldsymbol{C1}$ is the lower triangular part of supernode J's diagonal square matrix. After that, a DGEMM operation is performed, as Eq. (4.2),

$$\boldsymbol{B1} = (\boldsymbol{C1}^{-1})(\boldsymbol{B1}), \tag{4.1}$$

$$\begin{bmatrix} \boldsymbol{B2} \\ \boldsymbol{B3} \\ \boldsymbol{B4} \\ \boldsymbol{B5} \\ \boldsymbol{B6} \end{bmatrix} = \begin{bmatrix} \boldsymbol{B2} \\ \boldsymbol{B3} \\ \boldsymbol{B4} \\ \boldsymbol{B5} \\ \boldsymbol{B6} \end{bmatrix} - \begin{bmatrix} \boldsymbol{C2} \\ \boldsymbol{C3} \\ \boldsymbol{C4} \\ \boldsymbol{C5} \\ \boldsymbol{C6} \end{bmatrix} \begin{bmatrix} \boldsymbol{B1} \end{bmatrix}. \tag{4.2}$$

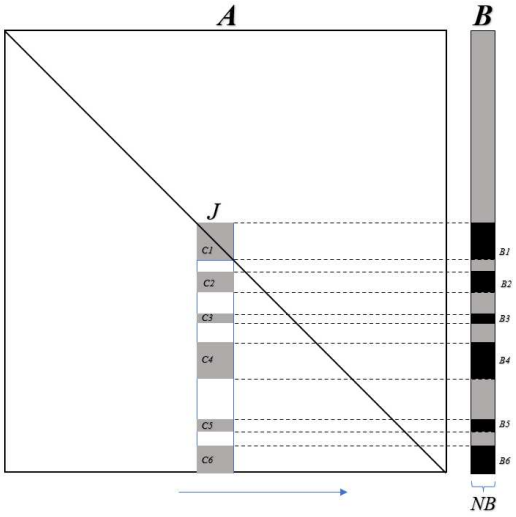


Fig. 4.1. Supernode forward substitution.

4.1.2. The parallel algorithm of supernode forward substitution

The parallel algorithm of supernode forward substitution is Algorithm 4.1. In the initial stage, any supernode without descendants can start forward substitution at the same time. When executing a forward substitution corresponding to a supernode, there is no possibility of data race during dense triangular solving. However, data race may occur during DGEMM operation. At this point, each thread needs a temporary array $\boldsymbol{TB}$ and its initial data is initialized to 0, then Eq. (4.2) is changed to Eq. (4.3). After the calculation is completed, OpenMP `atomic` operation can be used to Eq. (4.4) to avoid data race,

$$\boldsymbol{TB} = \boldsymbol{TB} - \begin{bmatrix} \boldsymbol{C2} \\ \boldsymbol{C3} \\ \boldsymbol{C4} \\ \boldsymbol{C5} \\ \boldsymbol{C6} \end{bmatrix} \begin{bmatrix} \boldsymbol{B1} \end{bmatrix}, \tag{4.3}$$

$$\begin{bmatrix} \boldsymbol{B2} \\ \boldsymbol{B3} \\ \boldsymbol{B4} \\ \boldsymbol{B5} \\ \boldsymbol{B6} \end{bmatrix} = \begin{bmatrix} \boldsymbol{B2} \\ \boldsymbol{B3} \\ \boldsymbol{B4} \\ \boldsymbol{B5} \\ \boldsymbol{B6} \end{bmatrix} + \boldsymbol{TB}. \tag{4.4}$$

In addition to paying attention to the problem of data race, a supernode J can be added into task pool only if all of its descendants have finished their update. Besides, due to the sparsity of the matrix $\boldsymbol{B}$, it can be determined whether $\boldsymbol{B1}$ is a 0 matrix or not before performing Eqs. (4.3) and (4.4). This optimization often has a significant acceleration effect.

Algorithm 4.1: Parallel Algorithm of Supernode Forward Substitution.

Input : The factored lower triangular matrix $\boldsymbol{L}$ and matrix $\boldsymbol{B}$.
Output: Modified matrix $\boldsymbol{B}$ after forward substitution.

```
task_pool ← 0, task_N = 0, TASK_FINISHED = 0.
Add all supernodes with 0 descendant into task_pool and initialize task_N.
#pragma omp parallel.
{
while TASK_FINISHED = 0 do
    J = −1.
    #pragma omp critical.
    {
    if task_N > 0 then
        task_N = task_N − 1.
        J = task_pool[task_N].
    end
    }
    if J! = −1 then
        Perform J's forward substitution.
    end
    F = P[J].
    if F's all descendants have finished their update tasks then
        Add F into task_pool and update task_N in critical section.
    end
    if There are no more tasks needed to be performed then
        TASK_FINISHED = 1.
    end
end
}
```

4.2. Parallel backward substitution based on supernode

4.2.1. The introduction of supernode backward substitution

Backward substitution updates $\boldsymbol{B}$ from column $N-1$ to column 0. The traditional backward substitution algorithm operates on single column and is based on the DDOT operation. Here we introduce the improved algorithm based on supernode structure. When it is time for supernode J to update matrix $\boldsymbol{B}$, a DGEMM operation as Eq. (4.5) is performed firstly. Then, a dense triangular solving as Eq. (4.6) follows,

$$\boldsymbol{B1} = \boldsymbol{B1} - \begin{bmatrix} \boldsymbol{C2} \\ \boldsymbol{C3} \\ \boldsymbol{C4} \\ \boldsymbol{C5} \\ \boldsymbol{C6} \end{bmatrix}^{\top} \begin{bmatrix} \boldsymbol{B2} \\ \boldsymbol{B3} \\ \boldsymbol{B4} \\ \boldsymbol{B5} \\ \boldsymbol{B6} \end{bmatrix}, \tag{4.5}$$

$$\boldsymbol{B1} = \left((\boldsymbol{C1})^{\top}\right)^{-1}(\boldsymbol{B1}). \tag{4.6}$$

4.2.2. The algorithm of supernode backward substitution

The parallel supernode backward substitution algorithm is shown as Algorithm 4.2. In the initial stage, all supernodes without ancestors should be added to the task pool. Due to the fact that each update only changes the content of $\boldsymbol{B1}$, there will be no data race issue in both DGEMM and DTRSM operations. Only when all ancestors of a supernode S have completed their updates, can S be added to the task pool (lines 16-18). It can be judged based on whether the number of S's ancestors that has finished their update is equal to the number of S's ancestors.

Algorithm 4.2: Parallel Algorithm of Supernode Backward Substitution.

Input : The factored lower triangular matrix $\boldsymbol{L}$ and modified matrix $\boldsymbol{B}$ after forward substitution.
Output: Modified matrix $\boldsymbol{B}$ after backward substitution.

```
1  task_pool ← 0, task_N = 0, TASK_FINISHED = 0.
2  Add all supernodes with 0 ancestor into task_pool and initialize task_N.
3  #pragma omp parallel.
4  {
5  while TASK_FINISHED = 0 do
6  |  J = −1.
7  |  #pragma omp critical.
8  |  {
9  |  if task_N > 0 then
10 |  |  task_N = task_N − 1.
11 |  |  J = task_pool[task_N].
12 |  end
13 |  }
14 |  if J! = −1 then
15 |  |  Perform J's backward substitution.
16 |  end
17 |  for k = 0, . . . , SJ_N[J] − 1 do
18 |  |  S = SJ[J][k].
19 |  |  if S's all ancestors have finished their update tasks then
20 |  |  |  Add S into task_pool and update task_N in critical section.
21 |  |  end
22 |  end
23 |  if There are no more tasks needed to be performed then
24 |  |  TASK_FINISHED = 1.
25 |  end
26 end
27 }
```

5. Test and Comparison

5.1. The effect of supernode size on factorization efficiency

The size of ASF's supernodes can be controlled by variable STD_SN_SIZE as mentioned in Section 2. Here we explore the effect of supernode size on the efficiency of ASF and numerical factorization (NF). We construct a 294636 dimensional stiffness equation with 6574641 nonzeros and FEADS solves it with 4 OpenMP threads. Table 5.1 shows the effect of supernode size on efficiency of factoring 3D_29W. As the increase of supernode size, the supernode number decreases and the downward trend slows down. The trend of ASF time is consistent with the trend of supernode number. The number of NNZ increases nearly linearly as the increase of supernode size and more NNZ means there are more logical zero elements in $\boldsymbol{L}$. The total factorization time decreases largely first and increases slowly as STD_SN_SIZE trends to be very large, as Fig. 5.1. A minimum total time is obtained when STD_SN_SIZE is nearly 250 and we think this is the appropriate size for supernode.

The size of supernode indirectly affects various deeper factors, such as NNZ of $\boldsymbol{L}$, the efficiency of level-3 BLAS subroutines, and the cache utilization rate, etc. The combined effect of all these factors determines the actual efficiency of factorization. The level-3 BLAS subroutines usually have saturation threshold and the appropriate supernode size should be below but close to the saturation region [32]. When STD_SN_SIZE is too small, too many small supernodes

Table 5.1: Effect of supernode size on efficiency of factoring 3D_29W.

STD_SN_SIZE	SN	NNZ of $\boldsymbol{L}$	ASF/s	NF/s	Total/s
50	5634	377163927	2.910374	35.522215	38.43259
100	2689	390949729	1.312708	20.983140	22.29585
150	1727	405048984	0.738118	17.846209	18.58433
200	1239	416607989	0.432265	16.897974	17.33024
250	967	427990107	0.308880	16.743526	17.05241
300	821	441017334	0.273872	16.964127	17.23800
400	690	465595734	0.248627	18.482175	18.73080
1000	555	534380934	0.170784	22.933425	23.10421
1500	538	552517434	0.176485	26.056129	26.23261

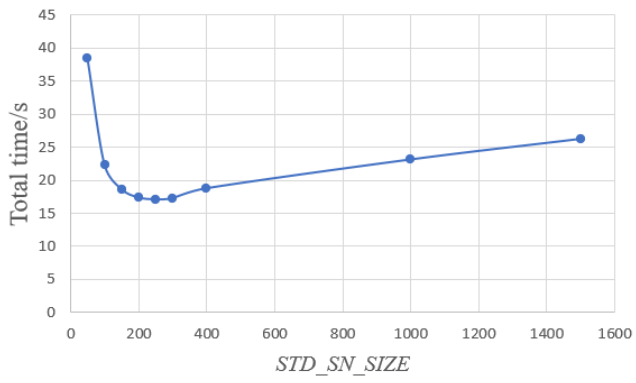


Fig. 5.1. Total time as the growth of STD_SN_SIZE.

cannot maximize performance of level-3 BLAS subroutines, which ultimately leads to extremely low factorization efficiency. And when STD_SN_SIZE trends to be very large, too many NNZ of $\boldsymbol{L}$ leads to more calculation of factorization, which also reduces factorization efficiency.

5.2. Logical zeros in $\boldsymbol{L}$ produced by ASF

The maximum memory requirement during the total solving process is keeping data of $\boldsymbol{L}$'s nonzeros. For large-scale problems, this item typically accounts for over 95% of the total memory requirements. ASF generates approximate pattern of $\boldsymbol{L}$, hence it is easy to think that there is possibility that the NNZ produced by ASF is much greater than the minimum NNZ by instinct, which may cause significant waste of memory.

Five stiffness matrices with different scales are constructed to explore the trend of logical zeros in $\boldsymbol{L}$ produced by ASF. We set STD_SN_SIZE as 250, and record NNZ of each $\boldsymbol{L}$ produced by ASF, as Table 5.2. The trend of logical zero proportion as the increase of matrix dimension is as Fig. 5.2. It is obvious that the proportion trend converges to around 20% as the increase of matrix dimension, which indicates feasibility of ASF for matrices of various scales.

5.3. The impact of OpenMP threads

To explore the impact of OpenMP threads on solver's performance, the 394770 dimensional stiffness equation is used. The number of OpenMP threads is continuously increased for FEADS, MKL PARDISO and MUMPS. The final timing results are shown in Table 5.3. The computer CPU is Intel (R) Core (TM) i5-8250U, with 4 cores and 8 threads, and available physical memory is 8 GB.

Table 5.2: Basic information of matrices.

Name	Dimension	NNZ of $\boldsymbol{A}$	Accurate NNZ of $\boldsymbol{L}$	NNZ of $\boldsymbol{L}$
3D_3W	30342	613618	9920285	19125474
3D_14W	144264	3181533	91810308	142794164
3D_29W	294636	6574641	291535293	427990107
3D_39W	394770	8822715	403476936	558936006
3D_71W	719871	16152141	944898786	1237953840

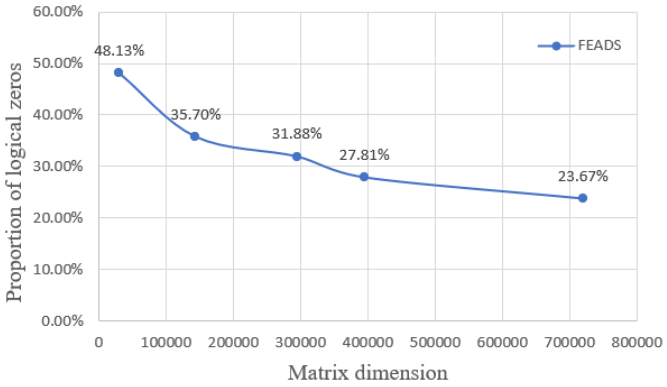


Fig. 5.2. Logical zero proportion of FEADS.

Table 5.3: The impact of OpenMP threads.

OT	1	2	3	4	5	6	7	8
PARDISO/s	60.152143	45.842895	45.684765	37.336508	-	-	-	-
FEADS/s	68.629308	48.679464	44.533408	39.848192	39.216628	37.962909	37.911802	39.926214
MUMPS/s	73.521681	57.920461	54.698671	53.911495	53.899999	55.88733	56.863012	58.181173

Table 5.4: The impact of OpenMP threads on FEADS's solving stages.

OT	Input/s	Reorder/s	Perm/s	SF/s	NF/s	TS/s	Other/s	Total/s
1	6.357382	1.998104	0.285237	1.599802	57.776788	0.576791	0.035204	68.629308
2	6.031456	1.911884	0.286091	1.288729	38.802608	0.325058	0.033638	48.679464
3	6.053243	1.902922	0.284711	1.157926	34.549765	0.545013	0.039828	44.533408
4	6.040819	1.903081	0.282596	1.094084	30.053294	0.440691	0.033627	39.848192
5	6.034115	1.917723	0.285761	1.123825	29.502116	0.31733	0.035758	39.216628
6	6.046970	1.907614	0.281014	1.097730	28.246871	0.348561	0.034149	37.962909
7	6.056986	1.918699	0.279768	1.085133	28.111950	0.425163	0.034103	37.911802
8	6.177256	1.909398	0.280108	1.098898	29.942594	0.483011	0.034949	39.926214

PARDISO limits its maximum number of OpenMP threads to not exceed the computer's number of physical cores, so its maximum OpenMP thread number is 4. On this computer, the optimal number of OpenMP threads for PARDISO is 4, FEADS is around 6, and MUMPS is also around 4. It can be seen that the optimal solving performance is not achieved when the number of OpenMP threads is set to the maximum thread number. The reason is that hyperthreading overhead outweighs its benefits. On this solving platform, the solving performance of PARDISO and FEADS is not significantly different, with an optimal solving time of around 37 seconds, while MUMPS is slower with an optimal solving time of around 54 seconds.

FEADS uses parallel algorithms in the stages of approximate symbolic factorization, numerical factorization and triangular solving (TS). Table 5.4 provides a detailed list of FEADS's solving time for each stage under different OpenMP threads (OT) number. It can be seen that as the number of OpenMP threads increases, there is a significant improvement in the efficiency of both symbolic factorization and numerical factorization. However, we can not see significant acceleration effect in the triangular solving stage on this solving case. This may be the reason that the solving scale is relatively small. And it can be seen that numerical factorization has the most significant parallel acceleration effect in the entire solving process. Using the model of task pool, our solver fully utilizes the multi-threaded resource of the computer and achieves excellent parallel performance.

5.4. Comparison of the optimal solving performance

To compare the optimal solving performance of the three solvers, stiffness equations are solved on two computers with different hardwares, the solving time of each solving stage was recorded in detail. The hardware parameters of the two computers are shown in Table 5.5.

Tables 5.6 and 5.7 show the optimal solving time to solve 394770 dimentional stiffness equation for MKL PARDISO, FEADS, and MUMPS on two test computers, respectively. For this case, the MKL PARDISO produces 40415 supernodes, FEADS produces 1530 supernodes

Table 5.5: Parameters of test computers.

Numbering	1	2
CPU	Intel(R) Core(TM) i5-8250U	AMD Ryzen 7 7840H with Radeon 780M Graphics
CPU number	4	8
Maximum thread	8	16
Maximum memory/GB	8	32
Frequency/GHz	1.8	3.8
L1 cache/KB	256	512
L2 cache/MB	1.0	8.0
L3 cache/MB	6.0	16.0

Table 5.6: The optimal time to solve 394770 dimentional stiffness equation on Computer 1.

Stage	Step	PARDISO (4 THREADS)	FEADS (6 THREADS)	MUMPS (4 THREADS)
File input	Input/s	6.055577	6.04697	6.621816
Analysis	Reorder/s	1.840393	1.907614	4.128700
	SF/s	1.017117	1.097730	
	Other/s	0.554760	0.281014	
NF	NF/s	26.617478	28.246871	35.123100
Solving	TS/s	1.034659	0.348561	3.4066
Other/s		0.216524	0.034149	4.6312795
Total/s		37.336508	37.962909	53.911495

Table 5.7: The optimal time to solve 394770 dimentional stiffness equation on Computer 2.

Stage	Step	PARDISO (8 THREADS)	FEADS (12 THREADS)	MUMPS (8 THREADS)
File input	Input/s	2.654813	2.695898	3.203672
Analysis	Reorder/s	1.076132	1.211387	2.360400
	SF/s	0.479456	0.466742	
	Other/s	0.31095	0.25241	
NF	NF/s	11.686333	8.256473	14.824800
Solving	TS/s	0.463209	0.134385	2.190200
Other/s		0.205308	0.019411	1.045818
Total/s		16.876201	13.036706	23.624890

and MUMPS produces 9132 supernodes. It is obvious that the supernode number of FEADS is much smaller than the other two solvers. The average supernode size for the three solvers are $9.8, 258.0$ and 43.2 respectively. According to the table, the solving performance of MKL PARDISO and FEADS is basically equivalent on computer 1. PARDISO is slightly faster than

Table 5.8: The optimal time to solve 719871 dimentional stiffness equation on Computer 2.

Stage	Step	PARDISO (8 THREADS)	FEADS (12 THREADS)	MUMPS (8 THREADS)
File input	Input/s	4.937067	4.920709	5.680402
Analysis	Reorder/s	2.065474	2.322934	4.876800
	SF/s	1.097088	1.189592	
	Other/s	0.593856	0.428325	
NF	NF/s	46.709760	27.719550	44.635700
Solving	TS/s	1.109359	0.290667	4.641700
Other/s		0.287476	0.035467	2.018214
Total/s		56.80008	36.907244	61.852815

FEADS in the numerical factorization stage, but FEADS is slightly faster than PARDISO in the symbolic factorization and triangular solving stage. This fully shows that the parallel ASF method and the supernode triangular solving algorithm proposed in this paper are progressive. However, MUMPS lags behind the other two solvers in all stages. On the second computer, FEADS performs especially well. FEADS leads MKL PARDISO and MUMPS by 21.92% and 42.35%, respectively. Our solver mainly leads in the numerical factorization and triangular solving stages at this point compared with the other two solvers.

Another larger stiffness equation with 719871 dimensions is built for the three solvers to solve on Computer 2, the timing result is as Table 5.8. For this case, the MKL PARDISO produces 98105 supernodes, FEADS produces 2783 supernodes and MUMPS produces 16457 supernodes. It is obvious that FEADS leads more than MKL PARDISO for this solving case, especially in the numerical factorization stage. FEADS takes only 27.7 seconds to finish parallel numerical factorization, while MKL PARDISO takes 46.7 seconds, almost twice as much as FEADS. For the total solving time, FEADS leads 34.75% than MKL PARDISO and 38.38% than MUMPS.

6. Summary

We use METIS as reordering algorithm to reorder the SPD stiffness matrix in this paper, and directly partition supernodes based on the reordered matrix. Approximate symbolic factorization is carried out based on supernodes and row index fragments, which has a much lower time complexity due to much less operating dimensions. To fully utilize the multi-core and multi-threaded resources of computers, task pool is applied to the stages of ASF, numerical factorization, and triangular solving. In the numerical factorization stage, the block supernode Cholesky factorization method is used to construct dense submatrices, and then high-performance multi-threaded and cache-optimized level-3 BLAS and LAPACK subroutines are used for calculating. In the triangular solving stage, new algorithms upgrade traditional algorithms based on DAXPY and DDOT operations to algorithms based on DGEMM and DTRSM operations. The new algorithms perform better than the algorithms of MKL PARDISO and MUMPS according to the test results.

We compare our FEADS with MKL PARDISO and MUMPS. The stiffness equations of 394770 and 719871 dimensions are solved using these three solvers on two different computers. On the first computer, the solving efficiency of FEADS and MKL PARDISO is comparable, while MUMPS is relatively backward. On the second computer, FEADS performs especially

well. For solving the case with 394770 dimensions, FEADS leads MKL PARDISO and MUMPS by 21.92% and 42.35%, respectively. For solving the case with 719871 dimensions, FEADS leads 34.75% and 38.38%, respectively.

References

[1] P.R. Amestoy, T.A. Davis, and I.S. Duff, An approximate minimum degree ordering algorithm, *SIAM J. Matrix Anal. Appl.*, **1996**:17 (1996), 886–905.

[2] P.R. Amestoy, I.S. Duff, J.Y. L'Excellent, Multifrontal parallel distributed symmetric and unsymmetric solvers, *Comput. Methods Appl. Mech. Engrg.*, **184**:2-4 (2000), 501–520.

[3] P.R. Amestoy, I.S. Duff, J.-Y. L'Excellent, and J. Koster, A fully asynchronous multifrontal solver using distributed dynamic scheduling dynamic scheduling, *SIAM J. Matrix Anal. Appl.*, **23**:1 (2001), 15–41.

[4] E. Anderson et al., *LAPACK Users' Guide*, SIAM, 1999.

[5] E. Anderson and Y. Saad, Solving sparse triangular linear system on parallel computers, *Int. J. High Speed Comput.*, **1**:1 (1989), 73–95.

[6] M. Bollhöfer, O. Schenk, R. Janalik, S. Hamm, and K. Gullapalli, State-of-the-art sparse direct solvers, in: *Parallel Algorithms in Computational Science and Engineering. Modeling and Simulation in Science, Engineering and Technology*, Birkhäuser, (2020), 3–33.

[7] T.A. Davis, J.R. Gilbert, S.I. Larimore, and E.G. Ng, A column approximate minimum degree ordering algorithm, *ACM Trans. Math. Software*, **30**:3 (2004), 353–376.

[8] T.A. Davis, A column pre-ordering strategy for the unsymmetric-pattern multifrontal method, *ACM Trans. Math. Software*, **30**:2 (2004), 165–195.

[9] T.A. Davis, Algorithm 849: A concise sparse Cholesky factorization package, *ACM Trans. Math. Software*, **31**:4 (2005), 587–591.

[10] M.J. Daydé and I.S. Duff, Level 3 BLAS in LU factorization on the CRAY-2, ETA-10P, and IBM 3090-200/VF, *Int. J. High Perform. Comput. Appl.*, **3** (1989), 40–70.

[11] J. Dongarra, Preface: Basic linear algebra subprograms technical (BLAST) forum standard, *Int. J. High Perform. Comput. Appl.*, **16**:1 (2002), 15–15.

[12] J. Dongarra and J. Demmel, LAPACK: A portable linear algebra library for high-performance computers, *Supercomput.*, **8** (1991), 33–38.

[13] J. Dongarra, J. Du Croz, S. Hammarling, and I. Duff, A set of level 3 basic linear algebra subprograms, *ACM Trans. Math. Software*, **16**:1 (1990), 1–17.

[14] I.S. Duff, M.A. Heroux, and R. Pozo, An overview of the sparse basic linear algebra subprograms: The new standard from the BLAS technical forum, *ACM Trans. Math. Software*, **28**:2 (2002), 239–267.

[15] A. George and J.W.H. Liu, *Computer Solution of Large Sparse Positive Definite Systems*, Prentice-Hall, 1981.

[16] A. George and J.W.H. Liu, The evolution of the minimum degree ordering algorithm, *SIAM Rev.*, **31** (1989), 1–19.

[17] A. Guermouche, J. L'Excellent, and G. Utard, Impact of reordering on the memory of a multifrontal solver, *Parallel Comput.*, **29** (2003), 1191–1218.

[18] G. Karypis and V. Kumar, *METIS: A Software Package for Partitioning Unstructured Graphs, Partitioning Meshes, and Computing Fill-Reducing Orderings of Sparse Matrices*, Technical Report 97-061, University of Minnesota, 1997.

[19] G. Karypis and V. Kumar, Multilevel k-way partitioning scheme for irregular graphs, *J. Parallel Distrib. Comput.*, **48**:1 (1998), 96–129.

[20] G. Karypis and V. Kumar, A fast and high quality multilevel scheme for partitioning irregular graphs, *SIAM J. Sci. Comput.*, **20**:1 (1999), 359–392.

[21] D. LaSalle and G. Karypis, Multi-threaded graph partitioning, in: *Proceedings of the 2013 IEEE 27th International Symposium on Parallel and Distributed Processing*, IEEE, (2013), 225–236.

[22] X.S. Li, J.W. Demmel, and J.R. Gilbert, *SuperLU Users' Guide*, 2018. `https://link.gitcode.com/i/f66c6423af7c90a1db5e15ba1adbfecd?uuid_tt_dd=10_37039561030-1756432671823-909358&isLogin=1&from_id=142778029`

[23] J.W.H. Liu, The role of elimination trees in sparse factorization, *SIAM J. Matrix Anal. Appl.*, **11**:1 (1990), 134–172.

[24] W. Liu et al., A synchronization-free algorithm for parallel sparse triangular solves, in: *Parallel Processing. Euro-Par 2016. Lecture Notes in Computer Science*, Vol. 9833, Springer, (2016), 617–630.

[25] Y. Liu, M. Jacquelin, P. Ghysels, and X.S. Li, Highly scalable distributed-memory sparse triangular solution algorithms, in: *2018 Proceedings of the Seventh SIAM Workshop on Combinatorial Scientific Computing*, SIAM, (2018), 87–96.

[26] T. Mumps, *Multifrontal Massively Parallel Solver (MUMPS 5.6.2) Users' guide*, 2023. `http://mumps-solver.org/index.php?page=home`

[27] E. Ng and B. Peyton, A supernodal Cholesky factorization algorithm for shared-memory multiprocessors, *SIAM J. Sci. Comput.*, **14** (1993), 761–769.

[28] E. Ng and P. Raghavan, Performance of greedy ordering heuristics for sparse Cholesky factorization, *SIAM J. Matrix Anal. Appl.*, **20** (1999), 902–914.

[29] J. Park, M. Smelyanskiy, N. Sundaram, and P. Dubey, Sparsifying synchronization for high-performance shared-memory sparse triangular solver, in: *Supercomputing. ISC 2014. Lecture Notes in Computer Science*, Vol. 8488, Springer, (2014), 124–140.

[30] F. Pellegrini, *SCOTCH 3.4 User's Guide*, Technical Report RR 1264-01, Université Bordeaux, 2001.

[31] O. Schenk, *Scalable Parallel Sparse LU Factorization Methods on Shared Memory Multiprocessors*, ETH Zürich, 2000.

[32] O. Schenk, K. Gartner, and W. Fichtner, Efficient sparse LU factorization with left-right looking strategy on shared memory multiprocessors, *BIT*, **40**:1 (2000), 158–176.

[33] E. Totoni, M.T. Heath, and L.V. Kale, Structure-adaptive parallel solution of sparse triangular linear systems, *Parallel Comput.*, **40**:9 (2014), 454–470.