

QKLU: A Two-Dimensional Block-Cyclic Sparse Direct Solver

Renqian Wan¹, Jianqiang Huang^{1,2,3*}, Haodong Bian^{1,2,3}

¹School of Computer Technology and Application, Qinghai University, Xining, 810016, China.

²Intelligent Computing and Application Laboratory of Qinghai Province, Qinghai University, Xining, 810016, China.

³Qinghai Provincial Green Computing Power Engineering Technology Research Center, Xining, 810016, China.

*Corresponding author(s). E-mail(s): hjqxaly@163.com;
Contributing authors: 1343323213@qq.com; hpc_bhd@163.com;

Abstract

Finite element analysis solves sparse linear systems via sparse LU decomposition. Supernodal and multifrontal techniques accelerate this process by grouping structurally identical or similar columns (or rows) into dense blocks to leverage highly optimized GEMM routines. However, their efficiency critically depends on the matrix structure – performance drops sharply if repeating patterns are scarce. Although the PanguLU algorithm improves on this by introducing a two-dimensional block-cyclic method, there is still room for improvement in its CPU performance. In our two-dimensional block-cyclic algorithm, we manually unrolled loops for sparse blocks, yielding a 23% speedup over the compiler’s automatic unrolling. We also introduced a dense-row detection algorithm since accessing dense rows is faster, this delivered an 11% improvement in sparse-block performance. In addition, we implemented a dense-block check: because dense operators far outperform sparse ones, applying dense operators to dense blocks further accelerates computation. Furthermore, PanguLU employs a static scheduling algorithm for load balancing, which is ill-suited for single-machine (shared-memory) systems. We adapted it to leverage OpenMP’s dynamic multithreading scheduling, utilizing the depend clause to manage dependencies between blocks. This approach dynamically dispatches tasks based on the actual runtime of blocks, offering significantly greater flexibility. Experimental results show that, compared to PanguLU, our method achieves average speedups of $1.67\times$ on the Kunpeng chips and $1.71\times$ on the intel chips. Our implementation is available at <https://github.com/mainfun/QKLU>.

Keywords: LU decomposition, LU, linear solver, direct method, sparse linear system.

1 Introduction

Sparse LU decomposition is a crucial numerical technique used for solving large systems of linear equations, particularly when the coefficient matrix is sparse. In many practical applications, such as finite element analysis, fluid dynamics,

and optimization problems, the matrices involved are often large but contain a significant number of zero elements. This sparsity presents challenges in computational efficiency. The primary goal of the sparse linear solver is to determine the solution vector \mathbf{x} for the linear equation $\mathbf{Ax} = \mathbf{b}$ (Duff, 1977), where \mathbf{A} represents a sparse matrix and \mathbf{b}

is a dense vector. Direct methods typically start by applying LU factorization $\mathbf{A} = \mathbf{L}\mathbf{U}$, which produces two triangular matrices, \mathbf{L} and \mathbf{U} . The solution process is then completed through two triangular solves. Unlike dense LU factorization, sparse LU factorization is structured into three distinct phases to effectively handle the sparse matrix: (1) reordering (Davis et al., 2004; Amestoy et al., 2004), (2) symbolic factorization (Grigori et al., 2007; Gaihare et al., 2021), and (3) numeric factorization. The reordering phase analyzes the sparse matrix to determine an optimal ordering that minimizes fill-in (new non-zero entries created during factorization). Algorithms such as Minimum Degree Ordering and Nested Dissection are commonly used, with libraries like SuiteSparse and Metis providing the necessary tools. Following this, the symbolic factorization phase establishes the structure of the matrices \mathbf{L} and \mathbf{U} . The final phase, numeric factorization, decomposes the sparse matrix \mathbf{A} into a lower triangular matrix \mathbf{L} and an upper triangular matrix \mathbf{U} such that $\mathbf{A} = \mathbf{L}\mathbf{U}$. The multifrontal (Davis, 2004) and supernodal (Eswar et al., 1993; Demmel et al., 1999a,b) method are popular algorithms for this step, with implementations available in libraries like UMFPACK (Davis, 2004), MUMPS (Amestoy et al., 2000), Pardiso (Schenk and Gärtner, 2002, 2004; Schenk et al., 2001) and SuperLU. Additionally, there are many direct solvers available on GPUs, such as GLU (He et al., 2015; Lee et al., 2018; Peng and Tan, 2020), SFLU (Zhao et al., 2021).

The predominant approaches to LU decomposition, namely the supernodal method and the multifrontal method (Puglisi, 2000; Duff, 1986; Amestoy et al., 2015; Ashcraft and Grimes, 1989), aggregates identical columns based on the elimination tree (Liu, 1990) into supernodes and utilizes Level 3 BLAS functions for their updates (Demmel et al., 1999b). Given that the efficiency of General Matrix Multiply (GEMM) computations exceeds that of other sparse operations, these methods can improve performance to some extent. However, the limited number of super columns restricts the opportunities for executing GEMM acceleration.

To solve this problem, PanguLU (Fu et al., 2023) introduce a novel sparse linear solver that departs from the supernodal method. Rather than

searching for rows or columns with identical structures, PanguLU performs LU decomposition using a 2D block cyclic algorithm. It partitions the matrix into regular two-dimensional submatrix blocks and calls SpGEMM to calculate them. However, there is still room for improvement in its CPU performance.

Our solver specifically addresses the performance bottleneck associated with 2D block-cyclic algorithm on CPU chips. It classifies matrix blocks by their density into dense blocks and sparse blocks. For sparse blocks, it applies manual loop unrolling and a dense-row detection routine: whenever a row within a sparse block is found to be dense, processing it as a dense row reduces indirect-addressing overhead. For dense blocks, it leverages the higher performance of GEMM to accelerate computation. In addition, PanguLU employs a static scheduling algorithm for load balancing, which is ill-suited for single-machine (shared-memory) systems. We adapted it to leverage OpenMP’s dynamic multithreading scheduling, utilizing the `depend` clause to manage dependencies between blocks. This approach dynamically dispatches tasks based on the actual runtime of blocks, offering significantly greater flexibility. Experimental results show that, compared to PanguLU, our method achieves average speedups of $1.67\times$ on the Kunpeng chips and $1.71\times$ on the intel chips.

This work makes the following contributions:

1. Optimize the computation of sparse submatrices.
2. Dense-block check to apply fast dense operators on dense submatrices.
3. Dynamic OpenMP scheduling with `depend` for better load balancing.

2 Background

2.1 Basic LU And Its Block Decomposition Algorithm

LU decomposition is a mathematical method that breaks a matrix into the product of two matrices: a lower triangular matrix (\mathbf{L}) and an upper triangular matrix (\mathbf{U}). It is used to solve linear equations, invert matrices, and compute determinants efficiently. The pseudocode presented in Algorithm 1

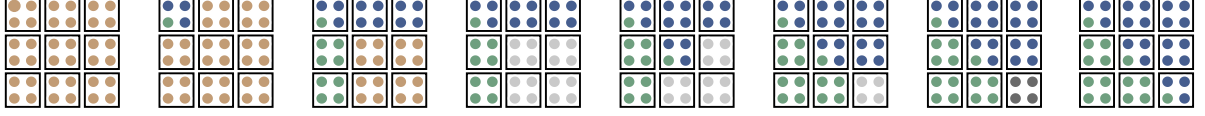


Fig. 1 This figure utilizes a 9×9 matrix divided into 3×3 blocks as an example, illustrating the elements of L and U calculated at each step. The original matrix A is represented in orange, U is shown in green, L is depicted in blue, and two kinds of gray represent the results of the first and second schur complement calculations respectively

Algorithm 1 LU Decomposition

```

1: Input: Matrix  $A$  of size  $N \times N$ 
2: for  $i = 0$  to  $N - 1$  do
3:    $pivot \leftarrow A(i, i)$ 
4:   for  $j = i + 1$  to  $N - 1$  do
5:      $l \leftarrow A(j, i) \leftarrow A(j, i)/pivot$ 
6:     for  $k = i + 1$  to  $N - 1$  do
7:        $A(j, k) \leftarrow A(j, k) - l \cdot A(i, k)$ 
8:     end for
9:   end for
10: end for

```

illustrates the dense LU factorization process for a square matrix A .

To enhance performance, a block LU factorization algorithm (Barker et al., 2001) was introduced to exploit data locality. This paragraph explains the LU decomposition of the two-dimensional block loop algorithm. Figure 1 illustrates the LU decomposition of a 9×9 matrix divided into 3×3 blocks. This approach employs a fixed-size block-wise methodology, wherein each block is processed independently of the others. The diagonal blocks are executed sequentially, progressing from the upper left corner to the lower right corner of the matrix. For instance, once block A_{11} has completed the LU factorization, the triangular blocks U_{11} and L_{11} can simultaneously perform triangular solves with A_{21} , A_{31} , A_{12} and A_{13} . Subsequently, these four blocks can concurrently compute the Schur complement to update A_{22} , A_{23} , A_{32} , and A_{33} . This process continues in a similar manner for the diagonal blocks until the final diagonal block is computed, indicating the completion of the block LU factorization.

2.2 Excellent Sparse LU Algorithms

Sparse LU factorization demonstrates significant advantages over dense LU factorization when solving large-scale sparse linear systems. The superiority stems from dense LU factorization’s inherent

computational redundancy caused by unnecessary operations on zero elements in sparse matrices. As shown in Figure 2, the sparse LU process follows a three-stage framework: (1) matrix reordering, (2) symbolic factorization, and (3) numerical factorization.

In reordering phase, the MC64 algorithm is employed to transform the matrix into a diagonally dominant form for improved numerical stability, followed by a graph-partitioning algorithm to minimize fill-ins. Symbolic analysis is then performed to determine the nonzero fill-in structure of the LU factors. The final numerical factorization phase computes actual numerical values for L and U through floating-point operations. This phase faces two key challenges: (1) the sheer computational intensity from complex dependency patterns, and (2) irregular memory access caused by the sparse matrix’s non-uniform structure. These characteristics frequently lead to suboptimal cache utilization and extended computation times, particularly on modern architectures with deep memory hierarchies.

PanguLU (Fu et al., 2023), an excellent sparse direct solver on distributed heterogeneous systems, performs LU decomposition using a regular two-dimensional block-cyclic algorithm. The algorithm is shown in the figure 1. This fundamentally differs from the traditional approach, addressing three critical limitations in SuperLU_DIST (Sao et al., 2019, 2014): non-uniform block dimensions, redundant zero-fill elements, and excessive synchronisation overhead (Fu et al., 2023).

2.3 Motivation

PanguLU, a state-of-the-art sparse direct solver, employs the regular two-dimensional block-cyclic algorithm. However, our analysis indicates that it exhibits suboptimal performance on CPU chips. We conducted tests on PanguLU’s floating-point operations per second (FLOPS) using the apple

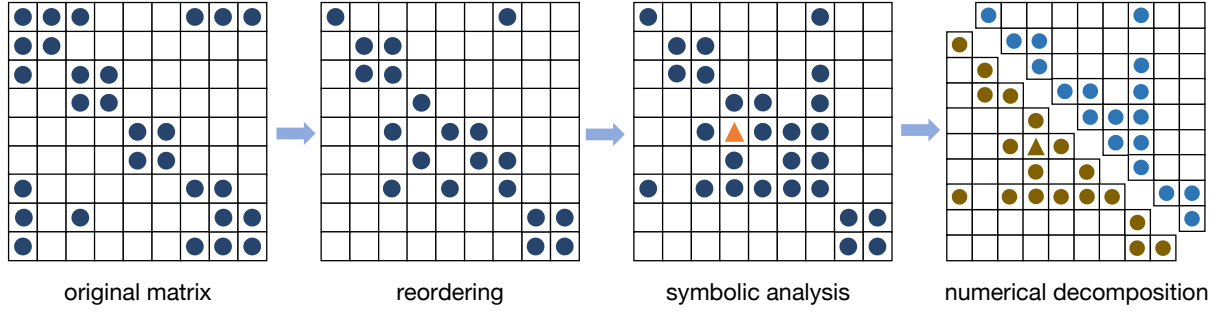


Fig. 2 The major procedures of sparse LU factorisation. The circles represent the non-zero elements of the matrix. The triangles represent the fill-ins.

Table 1 The information about the tested matrices and the GFLOPs of PanguLU.* represents the error $< 10^{-8}$

Matrix	dim	nnz	kind	GFLOPs
Chebyshev4	68,121	5,377,761	Structural Problem	9.70
bcsstk35	30,237	20,619	Structural Problem	3.27
ASIC_680k	682,862	2,638,997	Circuit Simulation Problem	17.88
bcircuit	68,902	375,558	Circuit Simulation Problem	0.35
onetone1	36,057	335,552	Frequency Domain Circuit Simulation Problem	6.13
twotone	120,750	1,206,265	Frequency Domain Circuit Simulation Problem	10.412
epb3	84,617	463,625	Thermal Problem	2.01
thermomech_dK	204,316	2,846,228	Thermal Problem	4.51
ACTIVSg70K	69,999	238,627	Power Network Problem	0.27
hvdc2	189,860	1,347,273	Power Network Problem	0.54
viscoplastic2	32,769	381,326	Materials Problem	*
xenon1	4,860	118,112	Materials Problem	7.85
BenElechi1	245,874	13,150,496	2D/3D Problem	*
heart3	2,339	680,341	2D/3D Problem	7.98
Zd_Jac6_db	22,835	663,643	Chemical Process Simulation Problem	7.46
Zd_Jac6	22,835	1,711,557	Chemical Process Simulation Problem	11.15
lung2	109,460	492,564	Computational Fluid Dynamics Problem	0.09
water_tank	60,740	2,035,281	Computational Fluid Dynamics Problem	*
psmigr_3	3,140	543,162	Economic Problem	15.02
g7jac200	59,310	837,936	Economic Problem	*

M2 chip, with results presented in Table 1. The table data indicates that its computation speed falls significantly short of the theoretical limit of the apple M2 chip.

The matrices evaluated in our experiments are publicly available from the SuiteSparse Matrix Collection (Davis and Hu, 2011). Our test suite comprises 20 representative matrices spanning 10 application domains(structural problem, circuit simulation problem, 2D/3D problem, etc.), with problem scales ranging from 10^4 to 10^8 non-zero

entries (nnz). For each of the ten application domains, we include two matrices: one large-scale instance and one medium-scale instance, creating a balanced benchmark set that covers both computational intensity and algorithm adaptability.

3 QKLU

3.1 Overview

QKLU is a sparse linear solver based on LU decomposition that achieves good performance

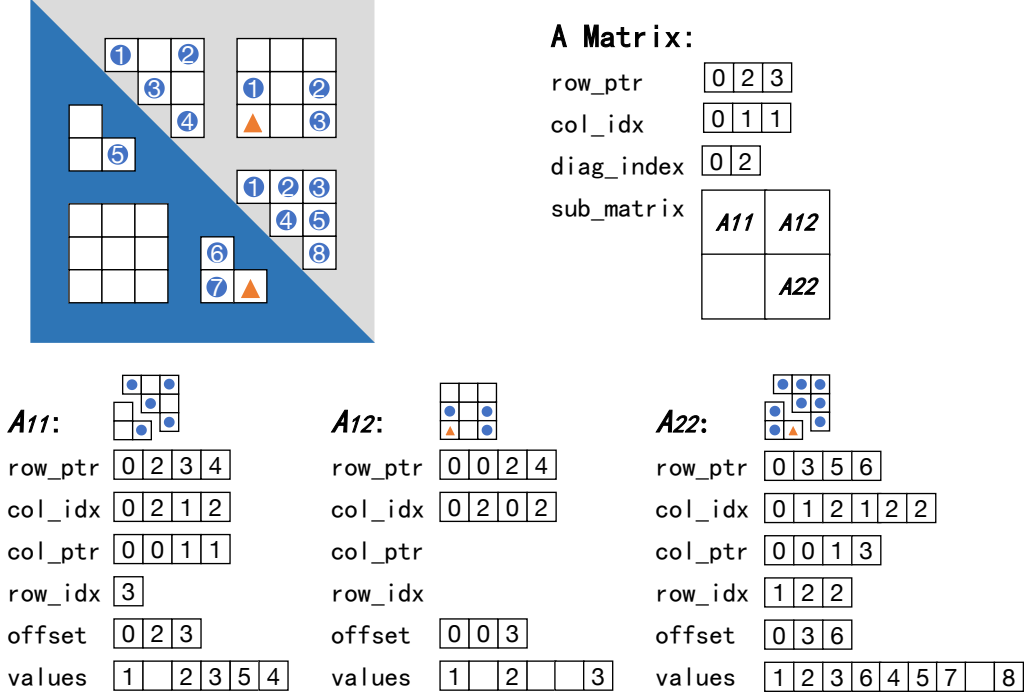


Fig. 3 Block matrix data format diagram. Circles represent the non-zero elements of the original matrix A, and triangles represent fill-in elements.

across multiple platforms. It consists of the following phases to solve the problem.

First, the MC64 algorithm (Duff and Koster, 2001, 1999) is applied to preprocess the matrix into a diagonally dominant matrix, improving numerical stability. Let the original matrix be A . After this step, it is transformed into $P_1 D_r A D_c$, where P is a permutation matrix, and D_r and D_c are row scaling and column scaling matrices, respectively.

Second, the reduction fill-ins reordering algorithm (Amestoy et al., 2004) is used to main sparsity. After this step, the matrix becomes $P_2 P_1 D_r A D_c P_2^T$, denoted as A' , and then the matrix is decomposed into L' and U' .

Finally, solve the system $Ax = b$, which involves the following steps: Solve $D_r^{-1} P_1^T P_2^T L' U' P_2 D_c^{-1} x = b$, or equivalently solve $L' U' P_2 D_c^{-1} x = P_2 P_1 D_r b$.

Let $b' = P_2 P_1 D_r b$ and $y = U' P_2 D_c^{-1} x$. This results in solving the following:

$$L' y = b',$$

followed by solving:

$$U' x' = y,$$

where $x' = P_2 D_c^{-1} x$. Finally, recover the solution as:

$$x = D_c P_2 x'.$$

The numerical factorization of A' was a key focus of our optimization efforts. QKLU partitions the matrix into 2D submatrix blocks, with each block's storage format dynamically selected based on its sparsity and position. Matrix blocks with a density greater than a threshold utilize a dense structure (two-dimensional array), while those below the threshold use a sparse structure. The experiment shows that setting the threshold to 40% yields the best performance. The storage for sparse blocks located in the upper triangular position is an QK_CSR structure, while those in the lower triangular position use an QK_CSC structure. QKLU have four main computational functions: general triangular factorization (GETRF), sparse lower triangular solve (GESSM), sparse upper triangular solve (TSTRF), and Schur complement. This is similar to PanguLU (Fu et al.,

2023); however, the computation for the Schur complement requires determining whether to execute SpGEMM or GEMM based on whether the computation block is dense or sparse. During parallel computation, all four functions are executed using OpenMP tasks, with their dependencies managed using OpenMP depend clause.

3.2 QK_CSR and QK_CSC

Algorithm 2 Naive Gauss elimination

```

1: Input: Matrix  $A$  of size  $N \times N$ 
2: for  $i = 0$  to  $N - 1$  do
3:    $pivot \leftarrow A(i, i)$ 
4:   for  $j = i + 1$  to  $N - 1$    where  $A(j, i) \neq 0$ 
     do
5:      $l \leftarrow A(j, i)/pivot$ 
6:      $A(j, i) \leftarrow l$ 
7:     for  $k = i + 1$  to  $N - 1$    where
        $A(i, k) \neq 0$  do
8:        $A(j, k) \leftarrow A(j, k) - l \times A(i, k)$ 
9:     end for
10:  end for
11: end for

```

Algorithm 2 presents a naive Gaussian elimination implementation, whose performance can be further optimized. To avoid checking whether elements are zero, the nonzero entries of the upper triangular part can be indexed using the Compressed Sparse Row (CSR) format, while those of the lower triangular part use the Compressed Sparse Column (CSC) format. However, relying solely on CSC/CSR makes efficient reading and writing of $A(j, k)$ difficult. Accessing matrix entries by row and column indices is most conveniently supported by a two-dimensional array layout. Thus, one can intuitively use CSC/CSR to represent the locations of nonzero entries, and use a two-dimensional array to store their values. However, when the data is sparse, storing all those zeros in a two-dimensional array leads to significant memory waste. We therefore eliminate the unnecessary zeros from the two-dimensional array. As depicted in Figure 4, empty rows, leading zeros, and trailing zeros in the matrix are first eliminated, and the remaining values are flattened into a one-dimensional array. Concurrently, the offset between the reduced value array and

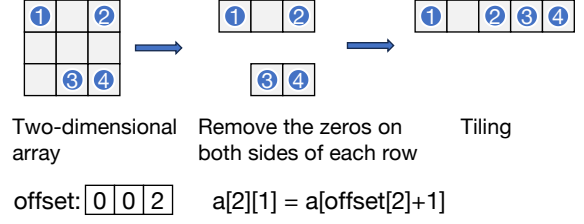


Fig. 4 Sparse matrix format adjustment.

the original value array is recorded. For instance, in a dense matrix, accessing the element a_{21} is typically performed using $a[2 \cdot N + 1]$. In the reduced structure, this operation is achieved via $a[\text{offset}[2]+1]$, where $\text{offset}[2]$ replaces the original $2 \cdot N$. The key advantage of this approach lies in its ability to facilitate rapid row- and column-based element access, akin to a two-dimensional array, while simultaneously preserving the sparsity of the matrix. As shown in Figure 3, the row_ptr and col_idx of block A11 represent the positions of the upper triangular non-zero elements (including fill-ins), while the col_ptr and row_idx indicate the positions of the lower triangular non-zero elements (excluding the diagonal but including fill-ins). The values array contains some extra redundant zeros compared to the traditional CSR/CSC format (these zeros are not used in the computation). In this paper, we refer to the adjusted CSR and CSC formats as QK_CSR and QK_CSC, respectively.

3.3 2D block decomposition

We employ a blocked computation strategy:

1. Sparse blocks are stored in the QK_CSR/QK_CSC format
2. Dense blocks use the conventional two-dimensional array layout
3. Computational routines are selected dynamically based on each block's sparsity rate

The block format is shown in Figure 3. Within the blocked scheme, the SSSSM operation Fu et al. (2023) is the primary performance bottleneck. The SSSSM operation of sparse blocks as shown in Algorithm 3. This implementation requires three indirect addressing operations per multiply-subtract instruction, resulting in excessive computational overhead. We introduce two optimizations:

Table 2 Table of performance optimization effect.

Matrix	Naive(ms)	Loop unrolling(ms)	Identifying dense rows(ms)	Speedup
Chebyshev4	21,528	9,531	6,432	3.34
bcsstk35	630	185	118	5.33
ASIC_680k	804	449	393	2.04
bcircuit	19	15	15	1.26
onetone1	674	221	115	5.86
twotone	4,108	1,288	853	4.81
epb3	409	144	94	4.35
thermomech_dK	6,273	1,783	993	9.72
hvdc2	1,598	803	540	2.95
ACTIVSg70K	21	14	14	1.5
viscoplastic2	676	216	118	5.72
xenon1	3,142	2,987	1,002	3.14
BenElechi1	30,844	7,884	4,730	6.52
heart3	243	75	42	5.78
Zd_Jac6_db	428	128	69	3.34
Zd_Jac6	5,264	1,924	1,278	4.11
lung2	3	3	3	1.00
water_tank	70,498	19,192	9,941	7.09
psmigr_3	1,577	545	373	4.22
g7jac200	32,975	9,166	4,778	6.90

Algorithm 3 SSSSM function**Require:** Block matrices m_1, m_2, m_3 **Ensure:** Result matrix $m_3 = m_1 \times m_2$

```

1:  $b \leftarrow m_2.\text{values}$ 
2:  $c \leftarrow m_3.\text{values}$ 
3:  $\text{offset}_2 \leftarrow m_2.\text{offset}$ 
4:  $\text{offset}_3 \leftarrow m_3.\text{offset}$ 
5: for  $i \leftarrow 0$  to  $\text{BLOCK\_SIDE} - 1$  do
6:   for  $p \leftarrow m_1.\text{col\_pointers}[i]$  to
      $m_1.\text{col\_pointers}[i + 1] - 1$  do
7:      $j \leftarrow m_1.\text{row\_indices}[p]$ 
8:      $l \leftarrow A(j, i)$   $\triangleright$  Get block  $(j, i)$  from  $m_1$ 
9:      $b\_ptr \leftarrow b + \text{offset}_2[i]$ 
10:     $c\_ptr \leftarrow c + \text{offset}_3[j]$ 
11:    for  $k \leftarrow m_2.\text{row\_pointers}[i]$  to
         $m_2.\text{row\_pointers}[i + 1] - 1$  do
12:       $b\_col \leftarrow m_2.\text{col\_indices}[k]$ 
13:       $c\_ptr[b\_col] \leftarrow c\_ptr[b\_col] - l \times$ 
         $b\_ptr[b\_col]$ 
14:    end for
15:  end for
16: end for

```

1. Manually unroll the second-level for loop to reduce the number of indirect addressings;

2. Incorporate a row-level sparsity check: for rows whose density exceeds a given threshold, switch to a dedicated dense-row computation path to further cut down on indirections. Since the dense-row computation code is naive and simple, its implementation is omitted.

Performance gains are quantified in Table 2. The organization of blocked computation is illustrated in Figure 1. During parallel execution, each block’s compute function is implemented as an OpenMP task, with inter-task dependencies automatically managed via the depend clause.

4 EXPERIMENTAL RESULTS

4.1 Experimental Setup

We performed our experiments on three platforms: a server equipped with a HUAWEI Kunpeng 920 (7270Z) processor—128 cores and 2 TB of RAM; a server featuring an Intel Xeon Silver 4216 CPU—32 cores and 250 GB of RAM; and a workstation powered by an Apple M2 chip with 8 cores and 16 GB of RAM.

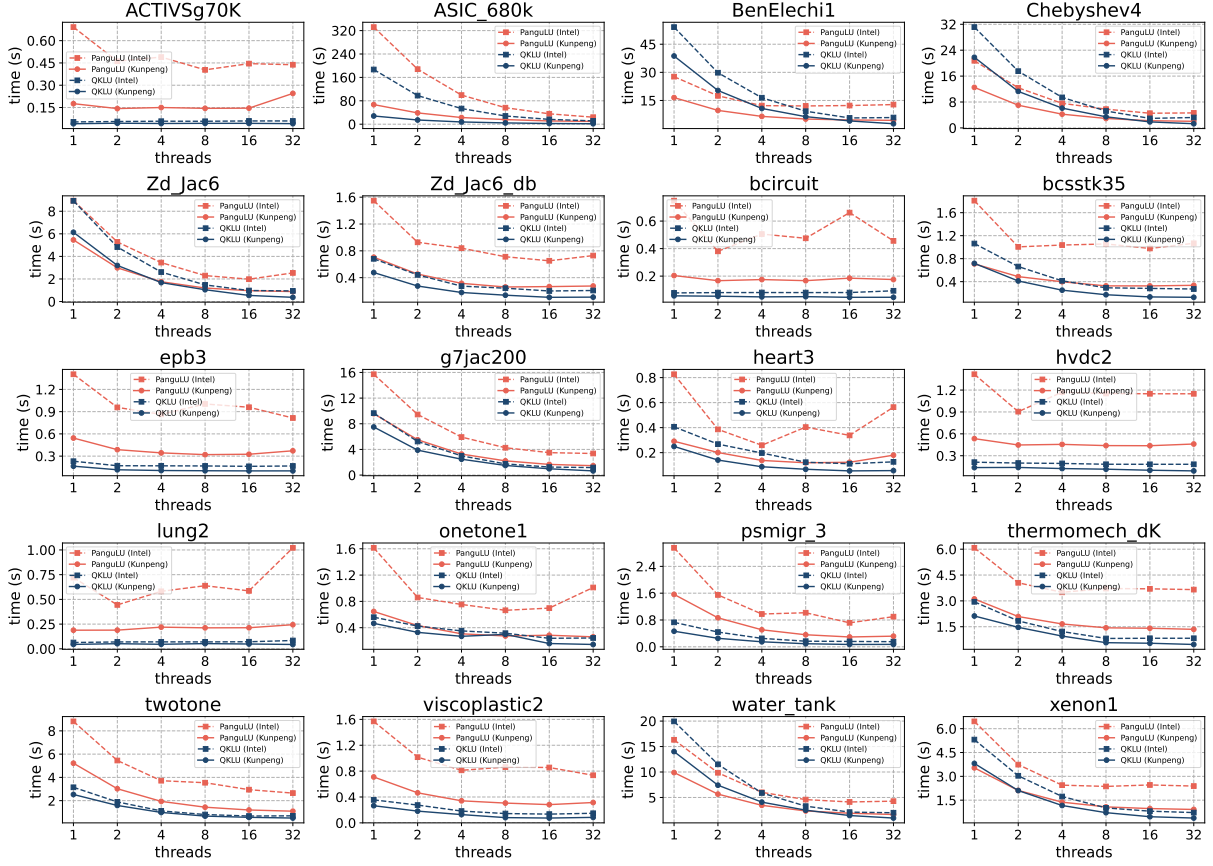


Fig. 5 Performance comparison between QKLU and PanguLU on HUAWEI Kunpeng 920 7270Z chip and intel chip platforms. The comparison time is computed as the total time minus both the MC64 time and the graph-partitioning time. The red lines and blue lines indicate the execution time of PanguLU and QKLU, respectively, while the solid lines and dotted lines indicate the execution time on the Kunpeng and intel platforms, respectively.

For MAC platforms, we compile PanguLU 4.0.0 and QKLU with clang 19.1.4, and OpenBLAS 0.3.28. PanguLU 4.0.0 is not the most recent release; we chose this version because the newer PanguLU versions cannot be built on macOS. For other platforms, we compile PanguLU 5.0.0 and QKLU with gcc 14.1, and OpenBLAS 0.3.28. Regarding the data set, we use 20 representative sparse matrices from the SuiteSparse Matrix Collection (Davis and Hu, 2011), covering various domains such as structural problems, circuit simulation problems, and computational fluid dynamics. The problem scales range from 10^4 to 10^8 non-zero entries (nnz). Details of the matrices are shown in Table 1. For MAC platforms, comparison of total execution time data between QKLU in single-thread mode and PanguLU in parallel mode is shown in Table 4. Where

* in the table indicates incorrect results. For other platforms, parallel computation time comparisons are shown in Fig. 5. When PanguLU and QKLU are run with more than 32 tasks in parallel, their performance does not improve due to data-dependency issues, so data for task counts above 32 have been omitted from the figure.

The above presents a performance comparison with PanguLU, which adopts the same algorithm, the regular two-dimensional block-cyclic algorithm. In addition, to provide a more comprehensive performance evaluation, we also include comparisons with two other algorithms, the Supernodal and Multifrontal methods. We compared the performance on Intel platform against two other mainstream solvers—SuperLU_DIST 9.1.0 and MUMPS 5.8.1—using 32-core parallel runs. All solvers were compiled with gcc 14.1 and linked

Table 3 Total Time Comparison on Intel Platform (Parallelism = 32)

Matrix	QKLU	MUMPS	speedup ratio	SuperLU_DIST	speedup ratio
Chebyshev4	5,213 ms	5,385ms	1.03	5,857ms	1.12
bcsstk35	290 ms	291ms	1.00	860ms	2.96
ASIC_680k	16,171ms	109,301ms	6.75	140,308ms	8.67
bcircuit	533ms	642ms	1.20	1,856ms	3.48
onetone1	613ms	*	*	1,571ms	2.56
twotone	2,194ms	*	*	9,254ms	4.21
epb3	767ms	856ms	1.11	2,321ms	3.02
thermomech_dK	1,908ms	2,044ms	1.07	6,328ms	3.31
hvd2	232ms	1,466ms	6.31	5,135ms	22.13
ACTIVSg70K	959ms	633ms	0.66	2,166ms	2.25
viscoplastic2	638ms	699ms	1.09	4,060ms	6.36
xenon1	1,072ms	1,136ms	1.05	1,899ms	1.77
BenElechi1	5,889ms	2,632ms	0.44	7,333ms	1.24
heart3	126ms	128ms	1.01	209ms	1.65
Zd_Jac6_db	618ms	831ms	1.34	2,797ms	4.5
Zd_Jac6	2,108ms	*	*	8,064ms	3.82
lung2	404ms	621ms	1.53	1,760ms	4.35
water_tank	1,797ms	2,933ms	1.63	3,856ms	2.14
psmigr_3	173ms	1,192ms	6.89	1,756ms	10.15
g7jac200	2,568ms	*	*	6,283ms	2.44

Table 4 Table comparing the total execution time of QKLU in single-thread mode with the total execution time of PanguLU 4.0.0 in parallel mode.

Matrix	QKLU	PanguLU	Speedup
Chebyshev4	6,432 ms	9,619ms	1.49
bcsstk35	371ms	520ms	1.40
ASIC_680k	1,470ms	55,026ms	37.43
bcircuit	87ms	346ms	3.97
onetone1	2,71ms	667ms	2.46
twotone	1,411ms	4,358ms	3.08
epb3	300ms	621ms	2.07
thermomech_dK	2,126ms	2,536ms	1.19
hvd2	540ms	671ms	1.24
ACTIVSg70K	88ms	236ms	2.68
viscoplastic2	327ms	*	*
xenon1	1,002ms	2,132ms	2.13
BenElechi1	9,534ms	*	*
heart3	111ms	166ms	1.49
Zd_Jac6_db	540ms	867ms	1.60
Zd_Jac6	3,033ms	4,614ms	1.52
lung2	63ms	282ms	4.47
water_tank	13,183ms	*	*
psmigr_3	656ms	897ms	1.36
g7jac200	7,051ms	*	*

against OpenBLAS 0.3.28. Table 3 presents the experimental results. In the table, * indicates a program crash.

4.2 Block Size And Density Threshold

The LU factorization process exhibits maximum computational intensity during its numerical phase, where Schur complement operations dominate the time complexity. Our decomposition methodology employs uniform two-dimensional block partitioning. For matrix blocks with high density (dense blocks), we implement GEMM-based Schur complement updates, whereas sparse blocks utilize SpGEMM-optimized operations. The critical parameters of block size and density threshold require empirical determination through systematic experimentation. On the M2 chip, through comprehensive benchmarking across four representative matrices (visualized in Figure 6’s performance heatmap), our analysis reveals optimal performance occurs when:

- Block density falls within 20%-40%
- Block size approximates 80

Notably, cooler color gradients in the heatmap correspond to superior computational throughput. Considering memory footprint constraints,

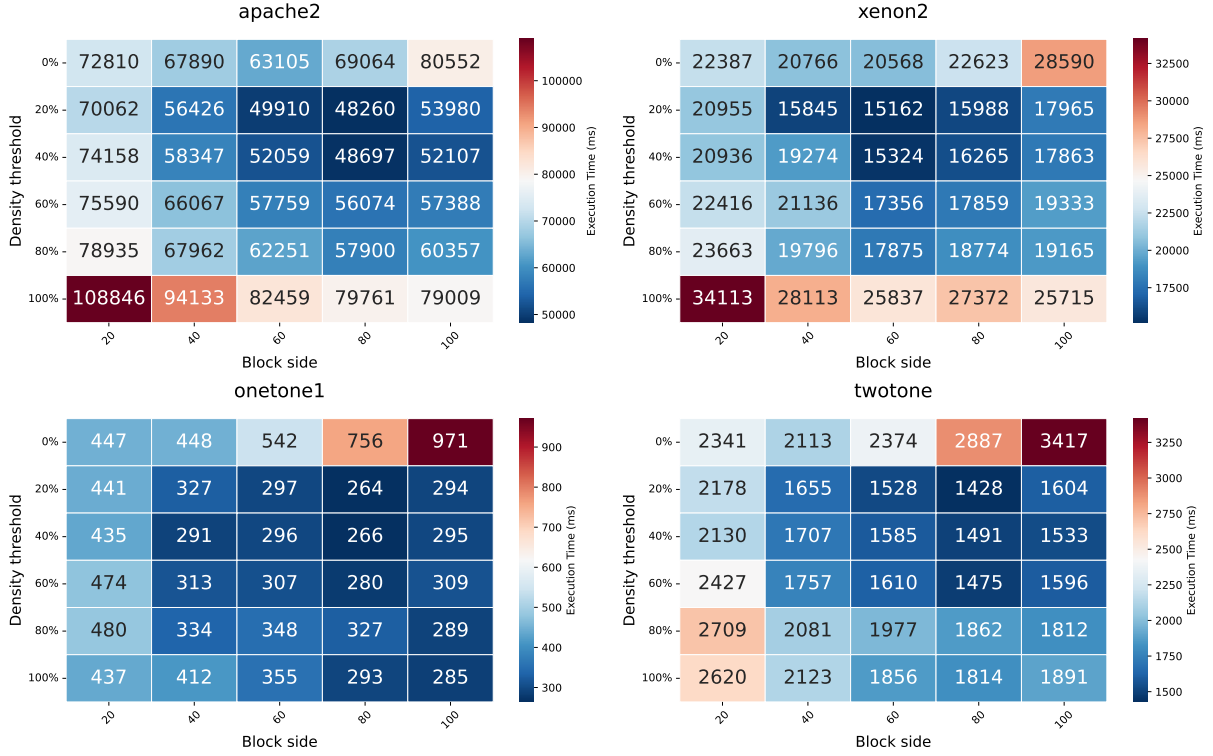


Fig. 6 Heatmap of the relationship between density and block size with performance.

we ultimately configure the density threshold at 40% with fixed block dimensions of 80×80 .

5 RELATED WORK

Significant advancements have been made over the past few decades to enhance the performance of sparse direct solvers, including efforts to maintain the numerical stability of LU factorization through pivoting (Duff et al., 2018; Duff and Pralet, 2005; Li and Demmel, 1998), reduce computational complexity in sparse LU factorization via reordering (Kumar et al., 1994), accelerate symbolic factorization (Gaihre et al., 2021), improve the parallelism of numeric factorization, and achieve higher performance in sparse triangular solves (Liu et al., 2016, 2023). Nevertheless, there remains potential for further enhancement of parallelism in sparse direct solvers.

The key challenge faced by sparse direct solvers is maintaining the sparse properties of the LU factorization while fully leveraging the scalability of modern supercomputers. To address this, Duff

and Reid (1983) introduced multifrontal methods, while Demmel et al. (1999b) and Li (2005); Li and Demmel (2003) developed the supernodal method. These approaches necessitate transforming input matrices into a relatively regular pattern by grouping similar columns, combining them into dense sub-matrices, and utilizing dense BLAS for computations. A considerable amount of work (Duff and Scott, 2004; Eswar et al., 1991; Chenhan et al., 2011; Xia et al., 2010) has been optimized based on these two methods, as they not only preserve sparse properties but also significantly enhance scalability when handling regular matrices. Additionally, Gupta (2000) achieved improved load balancing through task stealing with a task parallelism engine, while Amestoy et al. (2001) implemented dynamic task scheduling based on a multifrontal algorithm to balance loads in distributed systems. However, both multifrontal and supernodal methods are heavily dependent on matrix structure, which can result in suboptimal performance for irregular matrices. PanguLU proposes a new idea for distributed solver design,

using a simpler regular 2D blocking approach to exploit the sparse properties of matrices and balancing load by mapping computational tasks to less busy processes.

But there is still room for improvement in its CPU performance. In our two-dimensional block-cyclic algorithm, we manually unrolled loops for sparse blocks and introduced a dense-row detection algorithm since accessing dense rows is faster. In addition, we implemented a dense-block check: because dense operators far outperform sparse ones, applying dense operators to dense blocks further accelerates computation. Furthermore, PanguLU employs a static scheduling algorithm for load balancing, which is ill-suited for single-machine (shared-memory) systems. We adapted it to leverage OpenMP’s dynamic multi-threading scheduling, utilizing the depend clause to manage dependencies between blocks. This approach dynamically dispatches tasks based on the actual runtime of blocks, offering significantly greater flexibility.

6 CONCLUSION

In this paper, we propose QKLU, a scalable regular two-dimensional block-cyclic sparse direct solver. It performs well on both ARM architecture and intel architecture, achieving a speedup of 1.19 to 37 times compared to the PanguLU on most matrices. For matrices with more than 10^8 non-zero elements in $L + U - E$, the parallel speedup is satisfactory. However, for small matrices with fewer than 10^6 non-zero elements, no speedup has been observed in numerical computations. Additionally, we utilize a hybrid structure of dense matrices and adjusted CSC/CSR formats (QK_CSR/QK_CSC), which retains some zeros compared to standard CSC/CSR. The amount of redundant zero storage does not exceed three times that of the original.

Acknowledgements. This paper is partially supported by the Science and Technology Project of Qinghai Province(No.2023-QY-208), National Natural Science Foundation of China (No.62462052).

Conflict of Interest

On behalf of all authors, the corresponding author states that there is no conflict of interest.

References

- Duff, I.S.: A survey of sparse matrix research. *Proceedings of the IEEE* **65**(4), 500–535 (1977)
- Davis, T.A., Gilbert, J.R., Larimore, S.I., Ng, E.G.: Algorithm 836: Colamd, a column approximate minimum degree ordering algorithm. *ACM Transactions on Mathematical Software (TOMS)* **30**(3), 377–380 (2004)
- Amestoy, P.R., Davis, T.A., Duff, I.S.: Algorithm 837: Amd, an approximate minimum degree ordering algorithm. *ACM Transactions on Mathematical Software (TOMS)* **30**(3), 381–388 (2004)
- Grigori, L., Demmel, J.W., Li, X.S.: Parallel symbolic factorization for sparse lu with static pivoting. *SIAM Journal on Scientific Computing* **29**(3), 1289–1314 (2007)
- Gaihare, A., Li, X.S., Liu, H.: Gsofa: Scalable sparse symbolic lu factorization on gpus. *IEEE Transactions on Parallel and Distributed Systems* **33**(4), 1015–1026 (2021)
- Davis, T.A.: Algorithm 832: Umfpack v4. 3—an unsymmetric-pattern multifrontal method. *ACM Transactions on Mathematical Software (TOMS)* **30**(2), 196–199 (2004)
- Eswar, K., Sadayappan, P., Huang, C.-H., Visvanathan, V.: Supernodal sparse cholesky factorization on distributed-memory multiprocessors. In: *1993 International Conference on Parallel Processing-ICPP’93*, vol. 3, pp. 18–22 (1993). IEEE
- Demmel, J.W., Gilbert, J.R., Li, X.S.: An asynchronous parallel supernodal algorithm for sparse gaussian elimination. *SIAM Journal on Matrix Analysis and Applications* **20**(4), 915–952 (1999)
- Demmel, J.W., Eisenstat, S.C., Gilbert, J.R., Li, X.S., Liu, J.W.: A supernodal approach to sparse partial pivoting. *SIAM Journal on Matrix Analysis and Applications* **20**(3), 720–755 (1999)
- Amestoy, P.R., Duff, I.S., L’Excellent, J.-Y., Koster, J.: Mumps: a general purpose distributed memory sparse solver. In: *International Workshop on Applied Parallel Computing*, pp.

- 121–130 (2000). Springer
- Schenk, O., Gärtner, K.: Two-level dynamic scheduling in pardiso: Improved scalability on shared memory multiprocessing systems. *Parallel Computing* **28**(2), 187–197 (2002)
- Schenk, O., Gärtner, K.: Solving unsymmetric sparse systems of linear equations with pardiso. *Future Generation Computer Systems* **20**(3), 475–487 (2004)
- Schenk, O., Gärtner, K., Fichtner, W., Stricker, A.: Pardiso: a high-performance serial and parallel sparse linear solver in semiconductor device simulation. *Future Generation Computer Systems* **18**(1), 69–78 (2001)
- He, K., Tan, S.X.-D., Wang, H., Shi, G.: Gpu-accelerated parallel sparse lu factorization method for fast circuit analysis. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* **24**(3), 1140–1150 (2015)
- Lee, W.-K., Achar, R., Nakhla, M.S.: Dynamic gpu parallel sparse lu factorization for fast circuit simulation. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* **26**(11), 2518–2529 (2018)
- Peng, S., Tan, S.X.-D.: Glu3. 0: Fast gpu-based parallel sparse lu factorization for circuit simulation. *IEEE Design & Test* **37**(3), 78–90 (2020)
- Zhao, J., Wen, Y., Luo, Y., Jin, Z., Liu, W., Zhou, Z.: Sflu: Synchronization-free sparse lu factorization for fast circuit simulation on gpus. In: 2021 58th ACM/IEEE Design Automation Conference (DAC), pp. 37–42 (2021). IEEE
- Puglisi, C.: An unsymmetrized multifrontal lu factorization (2000)
- Duff, I.S.: Parallel implementation of multifrontal schemes. *Parallel computing* **3**(3), 193–204 (1986)
- Amestoy, P., Ashcraft, C., Boiteau, O., Buttari, A., l’Excellent, J.-Y., Weisbecker, C.: Improving multifrontal methods by means of block low-rank representations. *SIAM Journal on Scientific Computing* **37**(3), 1451–1474 (2015)
- Ashcraft, C., Grimes, R.: The influence of relaxed supernode partitions on the multifrontal method. *ACM Transactions on Mathematical Software (TOMS)* **15**(4), 291–309 (1989)
- Liu, J.W.: The role of elimination trees in sparse factorization. *SIAM journal on matrix analysis and applications* **11**(1), 134–172 (1990)
- Fu, X., Zhang, B., Wang, T., Li, W., Lu, Y., Yi, E., Zhao, J., Geng, X., Li, F., Zhang, J., *et al.*: Pangulu: A scalable regular two-dimensional block-cyclic sparse direct solver on distributed heterogeneous systems. In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–14 (2023)
- Barker, V.A., Blackford, L.S., Dongarra, J., Croz, J.D., Hammarling, S., Marinova, M., Waśniewski, J., Yalamov, P.: *LAPACK95 Users’ Guide*. SIAM, Philadelphia (2001)
- Sao, P., Li, X.S., Vuduc, R.: A communication-avoiding 3d algorithm for sparse lu factorization on heterogeneous systems. *Journal of Parallel and Distributed Computing* **131**, 218–234 (2019)
- Sao, P., Vuduc, R., Li, X.S.: A distributed cpu-gpu sparse direct solver. In: *Euro-Par 2014 Parallel Processing: 20th International Conference, Porto, Portugal, August 25–29, 2014. Proceedings 20*, pp. 487–498 (2014). Springer
- Davis, T.A., Hu, Y.: The university of florida sparse matrix collection. *ACM Transactions on Mathematical Software (TOMS)* **38**(1), 1–25 (2011)
- Duff, I.S., Koster, J.: On algorithms for permuting large entries to the diagonal of a sparse matrix. *SIAM Journal on Matrix Analysis and Applications* **22**(4), 973–996 (2001)
- Duff, I.S., Koster, J.: The design and use of algorithms for permuting large entries to the diagonal of sparse matrices. *SIAM Journal on Matrix Analysis and Applications* **20**(4), 889–901 (1999)
- Duff, I., Hogg, J., Lopez, F.: A new sparse symmetric indefinite solver using a posteriori threshold pivoting. *NLAFET Working Note* (2018)
- Duff, I.S., Pralet, S.: Strategies for scaling and pivoting for sparse symmetric indefinite problems. *SIAM Journal on Matrix Analysis and Applications* **27**(2), 313–340 (2005)
- Li, X.S., Demmel, J.W.: Making sparse gaussian elimination scalable by static pivoting. In: *SC’98: Proceedings of the 1998 ACM/IEEE Conference on Supercomputing*, pp. 34–34 (1998). IEEE
- Kumar, B., Sadayappan, P., Huang, C.-H.: On sparse matrix reordering for parallel factorization. In: *Proceedings of the 8th International Conference on Supercomputing*, pp. 431–438

- (1994)
- Liu, W., Li, A., Hogg, J., Duff, I.S., Vinter, B.: A synchronization-free algorithm for parallel sparse triangular solves. In: Euro-Par 2016: Parallel Processing: 22nd International Conference on Parallel and Distributed Computing, Grenoble, France, August 24-26, 2016, Proceedings 22, pp. 617–630 (2016). Springer
- Liu, Y., Ding, N., Sao, P., Williams, S., Li, X.S.: Unified communication optimization strategies for sparse triangular solver on cpu and gpu clusters. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, pp. 1–15 (2023)
- Duff, I.S., Reid, J.K.: The multifrontal solution of indefinite sparse symmetric linear. ACM Transactions on Mathematical Software (TOMS) **9**(3), 302–325 (1983)
- Li, X.S.: An overview of superlu: Algorithms, implementation, and user interface. ACM Transactions on Mathematical Software (TOMS) **31**(3), 302–325 (2005)
- Li, X.S., Demmel, J.W.: Superlu_dist: A scalable distributed-memory sparse direct solver for unsymmetric linear systems. ACM Transactions on Mathematical Software (TOMS) **29**(2), 110–140 (2003)
- Duff, I.S., Scott, J.A.: A parallel direct solver for large sparse highly unsymmetric linear systems. ACM Transactions on Mathematical Software (TOMS) **30**(2), 95–117 (2004)
- Eswar, K., Sadayappan, P., Visvanathan, V.: Multifrontal factorization of sparse matrices on shared-memory multiprocessors. In: ICPP (3) (1991)
- Chenhan, D.Y., Wang, W., Pierce, D.: A cpu-gpu hybrid approach for the unsymmetric multifrontal method. Parallel Computing **37**(12), 759–770 (2011)
- Xia, J., Chandrasekaran, S., Gu, M., Li, X.S.: Superfast multifrontal method for large structured linear systems of equations. SIAM Journal on Matrix Analysis and Applications **31**(3), 1382–1411 (2010)
- Gupta, A.: Wsmp: Watson sparse matrix package (part-i: direct solution of symmetric sparse systems). IBM TJ Watson Research Center, Yorktown Heights, NY, Tech. Rep. RC **21886** (2000)
- Amestoy, P.R., Duff, I.S., L’Excellent, J.-Y.,

Koster, J.: A fully asynchronous multifrontal solver using distributed dynamic scheduling. SIAM Journal on Matrix Analysis and Applications **23**(1), 15–41 (2001)



Renqian Wan is a Master’s student in the School of Computer Technology and Applications, Qinghai University, China. His research interests include high-performance computing and direct solvers for sparse linear systems.



Jianqiang Huang is a Professor in the School of Computer Technology and Applications, Qinghai University. His research interests include graph computing, heterogeneous computing (CPUs/GPUs), and parallel and distributed systems.



Haodong Bian is a Lecturer in the School of Computer Technology and Applications at Qinghai University. He received his Master’s degree from Qinghai University. His research interests include high-performance computing and graph computing systems.