PanguLU: A Scalable Regular Two-Dimensional Block-Cyclic Sparse Direct Solver on Distributed Heterogeneous Systems

Xu Fu
SSSLab, China University of Petroleum-Beijing, China
xu.fu@student.cup.edu.cn

Bingbin Zhang
SSSLab, China University of Petroleum-Beijing, China
bingbin.zhang@student.cup.edu.cn

Tengcheng Wang
SSSLab, China University of Petroleum-Beijing, China
tengcheng.wang@student.cup.edu.cn

Wenhao Li
SSSLab, China University of Petroleum-Beijing, China
wenhao.li@student.cup.edu.cn

Yuechen Lu
SSSLab, China University of Petroleum-Beijing, China
yuechenlu@student.cup.edu.cn

Enxin Yi
SSSLab, China University of Petroleum-Beijing, China
enxin.yi@student.cup.edu.cn

Jianqi Zhao
SSSLab, China University of Petroleum-Beijing, China
jianqi.zhao@student.cup.edu.cn

Xiaohan Geng
SSSLab, China University of Petroleum-Beijing, China
xiaohan.geng@student.cup.edu.cn

Fangying Li
SSSLab, China University of Petroleum-Beijing, China
fangying.li@student.cup.edu.cn

Jingwen Zhang
SSSLab, China University of Petroleum-Beijing, China
jingwen.zhang@student.cup.edu.cn

Zhou Jin
SSSLab, China University of Petroleum-Beijing, China
jinzhou@cup.edu.cn

Weifeng Liu
SSSLab, China University of Petroleum-Beijing, China
weifeng.liu@cup.edu.cn

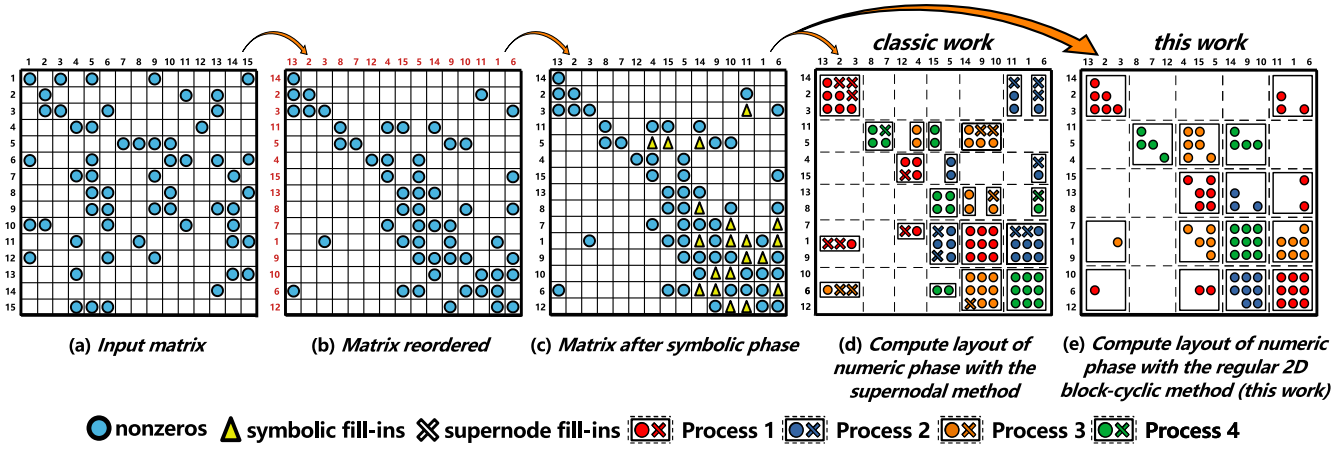


Figure 1: The major procedures of sparse LU factorisation. (a), (b) and (c) illustrate the original matrix, its reordered form, and the matrix after symbolic factorisation (yellow triangles represent the fill-ins), respectively. (d) shows numeric factorisation with a four-process compute layout by using the classic supernodal method, and the crosses mean the extra fill-ins when computing Schur complement (dense GEMM). (e) shows numeric factorisation in PanguLU with a four-process compute layout by using the regular 2D block-cyclic method proposed in this paper. Note that no extra fill-ins are needed, since our work calls sparse kernels instead of dense ones.

SC '23, November 12–17, 2023, Denver, CO, USA

ACM ISBN 979-8-4007-0109-2/23/11...$15.00
https://doi.org/10.1145/3581784.3607050

ABSTRACT

Sparse direct solvers play a vital role in large-scale high performance computing in science and engineering. Existing distributed sparse direct methods employ multifrontal/supernodal patterns to aggregate columns of nearly identical forms and to exploit dense basic linear algebra subprograms (BLAS) for computation. However, such a data layout may bring more unevenness when the structure of the input matrix is not ideal, and using dense BLAS may waste many floating-point operations on zero fill-ins.

In this paper, we propose a new sparse direct solver called PanguLU. Unlike the multifrontal/supernodal layout, our work relies on simpler regular 2D blocking and stores the blocks in their sparse forms to avoid any extra fill-ins. Based on the sparse patterns of the blocks, a variety of block-wise sparse BLAS methods are developed and selected for higher efficiency on local GPUs. To make PanguLU more scalable, we also adjust the mapping of blocks to processes for overall more balanced workload, and propose a synchronisation-free communication strategy considering the dependencies among different sub-tasks to reduce overall latency overhead.

Experiments on two distributed heterogeneous platforms consisting of 128 NVIDIA A100 GPUs and 128 AMD MI50 GPUs demonstrate that PanguLU achieves up to 11.70x and 17.97x speedups over the latest SuperLU_DIST, and scales up to 47.51x and 74.84x on the 128 A100 and MI50 GPUs over a single GPU, respectively.

CCS CONCEPTS

• **Mathematics of computing** → **Solvers**; **Mathematical software performance**.

KEYWORDS

Sparse LU, Regular 2D Block, Distributed Heterogeneous Systems

ACM Reference Format:

Xu Fu, Bingbin Zhang, Tengcheng Wang, Wenhao Li, Yuechen Lu, Enxin Yi, Jianqi Zhao, Xiaohan Geng, Fangying Li, Jingwen Zhang, Zhou Jin, and Weifeng Liu. 2023. PanguLU: A Scalable Regular Two-Dimensional Block-Cyclic Sparse Direct Solver on Distributed Heterogeneous Systems. In *The International Conference for High Performance Computing, Networking, Storage and Analysis (SC '23), November 12–17, 2023, Denver, CO, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3581784.3607050

1 INTRODUCTION

The purpose of the sparse solver is to calculate the solution vector x for a linear system $Ax = b$ [18, 27, 30], where A is a sparse matrix and b is a dense vector. Direct methods often first use LU factorisation $A = LU$ to obtain two triangular matrices L and U, and complete the solution after two triangular solves. Unlike dense LU factorisation, a typical sparse LU factorisation is divided into three phases to accommodate the sparse matrix: (1) reordering, (2) symbolic factorisation, and (3) numeric factorisation. The purpose of the reordering phase is to reduce fill-in nonzeros and maintain numerical stability. Then, the symbolic factorisation is to determine the structure of the matrices L and U. Finally, the numeric factorisation performs floating-point operations.

The numeric factorisation phase normally contains a large number of floating-point operations [18, 30], thus many direct solvers in recent years have taken the parallelism of the numeric factorisation phase as the main optimisation orientation on single-threaded processors [23], multi-threaded processors [15, 17, 79], heterogeneous processors [49, 52, 61, 81] and distributed memory systems [8, 55, 73]. But there are several problems with these solvers. KLU [23], NICSLU [15], UMFPACK [17] and FLU [79] run on share memory CPUs and do not effectively use the computing power of heterogeneous processors (e.g. GPUs). GLU [49, 52, 61] and SFLU [81] are still only available for a single GPU, which cannot take advantage of the computing power of multiple GPUs in large-scale supercomputers. MUMPS [8], SuperLU_DIST [55] and PARDISO [73] support the computation on both shared and distributed memory systems. In addition, the latest versions of SuperLU_DIST have also been improved for distributed CPU-MIC and CPU-GPU systems [66–69]. These solvers use multifrontal or supernodal methods to aggregate dense columns for using dense BLAS, resulting in good scalability and performance for matrices with many similar column structures. However, it may cause performance problems with irregular matrices on distributed memory systems. Either too few similar columns are aggregated (resulting in insufficient scalability), or too many zero fill-ins are used to gather more similar columns (resulting in lower computational efficiency).

In this paper, we propose PanguLU, a novel direct solver for distributed heterogeneous systems. Unlike the multifrontal or supernodal methods, PanguLU uses regular two-dimensional blocks for the layout and intra-block sparse sub-matrices as the basic unit to build a new distributed sparse LU factorisation algorithm. Since the stored matrix blocks are sparse, we take advantage of sparse BLAS for computation to avoid unnecessary fill-ins and optimise the sparse properties to make the computation more efficient.

To exploit large-scale supercomputers with heterogeneous processors [1], we need to address the following three key challenges: (1) how to balance workload on distributed processes for higher scalability, (2) how to take advantage of heterogeneous accelerators by designing appropriate parallel algorithms based on sparse matrix structures, and (3) how to reduce synchronisation cost between processes with irregularly sparse structured dependencies.

Firstly, to make PanguLU more scalable, we devised a static block mapping scheme to balance the load. This approach calculates the corresponding weight of each task, and tasks with high weights are migrated to less loaded processes in time slice order to balance the workload of each process. Secondly, we introduce a dedicated sparse BLAS design and develop 17 sparse kernels with different parallel methods. In addition, we also design a decision tree approach to select a faster sparse kernel based on the structure of the sparse matrix. Finally, we focus on the dependencies between different sub-tasks and design a synchronisation-free scheduling strategy. This strategy uses a synchronisation-free array that allows each process in the distributed system to compute as many executable sparse kernels as possible and ensures the correctness of the sparse LU factorisation. This has the advantage of improving performance by reducing the synchronisation overhead.

We evaluate 16 representative sparse matrices from the SuiteSparse Matrix Collection [22] on two distributed heterogeneous platforms, one with 128 NVIDIA A100 GPUs and the other with 128 AMD MI50 GPUs. Compared to SuperLU_DIST 8.1.2, the geometric mean speedups of PanguLU in numeric factorisation are 2.53x and 2.79x on the NVIDIA and the AMD GPU platforms, with speedups ranging from 1.10x - 11.70x and 1.12x - 17.97x, respectively. We also demonstrate that PanguLU can scale up to 47.51x and 74.84x on the two GPU platforms, respectively. Then we analyse the performance advantage of PanguLU in numeric factorisation with SuperLU_DIST, the kernel time speeds up by an average of 6.54x on a single GPU and the synchronisation time speeds up by an average of 2.20x on 128 GPUs. We also test the optimisations resulting from the sparse kernel selection and the synchronisation-free scheduling strategy in PanguLU, which achieve speedups of 2.3x - 5.4x (3.8x on average) over the baseline version. Finally, we compare

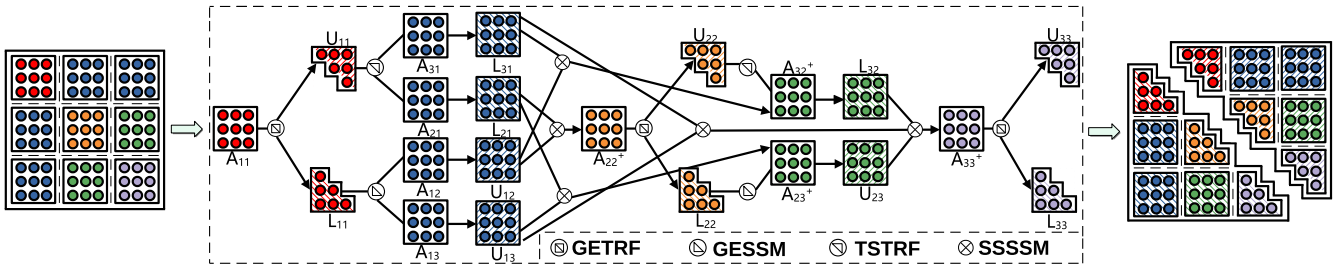


Figure 2: An example of block LU factorisation. The diagonal blocks perform LU factorisation to generate triangular blocks. The upper and lower triangular blocks perform triangular solve to update the other blocks in the same column and row. The updated blocks resulting from the triangular solve perform Schur complement to update the corresponding tail blocks.

the preprocessing and symbolic factorisation of PanguLU and SuperLU_DIST, achieving an average speedup of 1.61x and 4.45x and a maximum speedup of 3.16x and 6.80x, respectively.

This work makes the following contributions:

- We propose a regular two-dimensional blocking layout in sparse LU factorisation and adjust the block to process mapping for a more balanced workload.
- We design 17 sparse kernels and a matrix structure-based selection method to improve computational efficiency on heterogeneous processors.
- We develop a synchronisation-free communication strategy on distributed systems to reduce the synchronisation overhead of LU factorisation.
- We achieve significant speedups over SuperLU_DIST in various phases of LU factorisation and prove that PanguLU scales well on heterogeneous distributed systems.

2 BACKGROUND

2.1 Dense LU and its Block Algorithm

LU factorisation is a popular matrix decomposition method that factorises a matrix A into lower and upper triangular matrices L and U. The pseudocode in Algorithm 1 demonstrates the dense LU factorisation of a square matrix A.

Algorithm 1 An algorithm for dense LU factorisation

```
1:  for k = 0 : n − 1 do
2:      u_kk = a_kk
3:      for i = k + 1 : n − 1 do
4:          l_ik = a_ik / u_kk
5:          u_ki = a_ki
6:          for j = k + 1 : n − 1 do
7:              a_ij = a_ij − l_ik × u_kj
8:          end for
9:      end for
10: end for
```

To improve performance, a block LU factorisation algorithm [11] was proposed by exploiting the spatial location of the data. Figure 2 depicts block LU factorisation for a 9×9 matrix. It follows the fixed-size block-wise approach in which each block is processed independently of others. The diagonal blocks are executed sequentially from the upper left corner to the lower right corner of the matrix. Blocks of the same colour can be processed simultaneously to leverage parallelism. Such as, once block A_{11} has completed the LU factorisation, triangular blocks U_{11} and L_{11} can simultaneously perform triangular solve with A_{21}, A_{31}, A_{12} and A_{13}. Then the four blocks can perform the Schur complement at the same time to update A_{22}, A_{23}, A_{32} and A_{33}. Next the diagonal block is computed in a similar way until the last diagonal block is processed, marking the completion of the block LU factorisation. It is clear that block LU factorisation involves numerous computations with complex dependencies, which significantly impact parallel performance.

2.2 Sparse LU and its Multifrontal / Supernodal Algorithms

Sparse LU factorisation is generally considered superior to dense LU factorisation for solving large-scale sparse linear systems. This preference is due to the redundant computational steps caused by zero element computation in dense LU factorisation of sparse matrices. Sparse LU factorisation can be divided into three phases: reordering, symbolic factorisation and numeric factorisation. Figure 1 shows an example of sparse LU factorisation with details.

Certain row and column reordering methods [24, 25] are typically used to minimise nonzero fill-ins before the symbolic factorisation phase, as demonstrated in Figure 1(b). In the symbolic phase illustrated in Figure 1(c), additional nonzero fill-ins are introduced to determine the sparsity pattern of the triangular matrices L and U, and to allocate space for the numeric phase in advance. Finally, in the numeric phase, the necessary floating-point operations are performed to determine the values in L and U. Numeric factorisation often takes a long time due to the complexity and extensive nature of the floating-point operations. Furthermore, uneven data access patterns can arise from the sparsity and uneven distribution of the matrix, which worsens computational time.

The multifrontal [36] method is proposed based on the right-looking sparse LU factorisation. This method reorganises sparse matrices into a sequence of partial factorisation of smaller dense matrices. The elimination tree [25, 57, 72] is used to identify the rows and columns of the matrix involved in each factorisation step. It is usually constructed according to the following rule: Consider each diagonal element as a node, and a node k is a child of a node j in the elimination tree if the factorisation of the diagonal element j follows the diagonal element k. Unlike the multifrontal method, the supernodal [24] method merges a number of similar row structure columns to form supernodes. As shown in Figure 1(d), the layout of the supernodal method includes crosses representing the additional zero fill-ins introduced during the supernode formation process. In addition, the supernodal method of identifying columns with similar row structures can be challenging due to the identification of the unevenness in matrix structures.

3 MOTIVATION

3.1 Uneven Block Sizes

Many studies have proposed optimisation algorithms for sparse LU factorisation, such as the supernodal method, which groups columns with the same nonzero pattern together and compute them as dense blocks. Figure 3 illustrates the significantly different supernode sizes of two typical matrices, the number of rows of supernode for G3_circuit and audikw_1 are concentrated in [4, 64) and [32, 512), respectively, and the number of columns is concentrated in [1, 32) and [2, 32), respectively. It can be seen that the matrix blocks generated by the supernodes can be very irregular, which affects the computational and storage efficiency, and the irregular structure makes it difficult to optimise the performance at the kernel level.

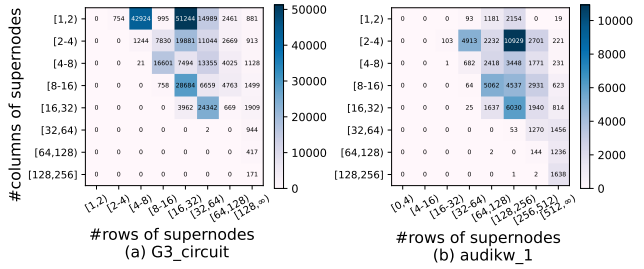


Figure 3: Uneven-size distribution of supernode blocks. The x-axis and y-axis are the row and column sizes of the supernodes, and the value of the heatmap colour represents the number of supernode blocks within a certain interval.

3.2 Redundant Zero Fill-ins

Multifrontal and supernodal algorithms divide the matrix into uneven dense blocks to be computed with level-3 BLAS routines, which may cause two problems. Firstly, as shown in Figure 1(d), redundant zero fill-ins can occur when forming dense blocks that adds extra floating-point operations. Secondly, the local sparsity of the matrix cannot be exploited during GEMM calculations on dense blocks, leading to possible performance degradation. In Figure 4, the density of block matrices involved in GEMM operations demonstrate great differences, ASIC_680k is concentrated at [0,10)% while audikw_1 is concentrated at [90,100)%, CoupCons3D has an overall even distribution but a significant portion is less than 50%. As a result, there is a chance that sparse BLAS is faster compared to dense BLAS.

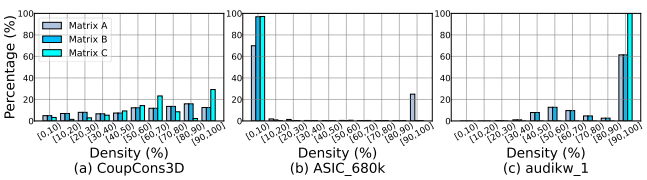


Figure 4: Density distribution of matrices involved in GEMM ($C = AB$) in SuperLU_DIST, where the x-axis represents the density, or the proportion of nonzeros in the matrix, and the y-axis shows the ratio of the number of matrices for each density range to the total number of matrices.

3.3 High Synchronisation Costs

SuperLU_DIST uses the level-set method to generate an elimination tree with tree nodes as the minimum scheduling unit. The tree nodes consist of multiple dense BLAS operations and there are dependencies between each level that need to be synchronised on completion, resulting in additional synchronisation overhead. In Figure 5, we show the synchronisation cost ratio of SuperLU_DIST (on 64 NVIDIA A100 GPUs, from 1 to 64 process) on six matrices from different applications. As shown, the cost of synchronisation gradually increases as the number of processes increases, and at 64 processes, it can account for up to 60% of the total computation time. Therefore, for some matrices with a high synchronisation time ratio, we can consider reducing the synchronisation cost to explore the optimisation space.

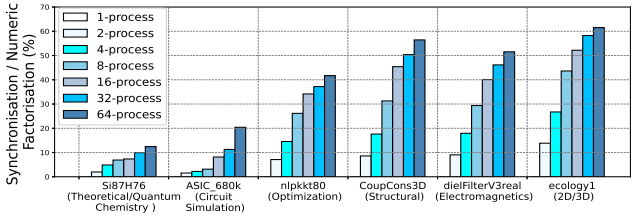


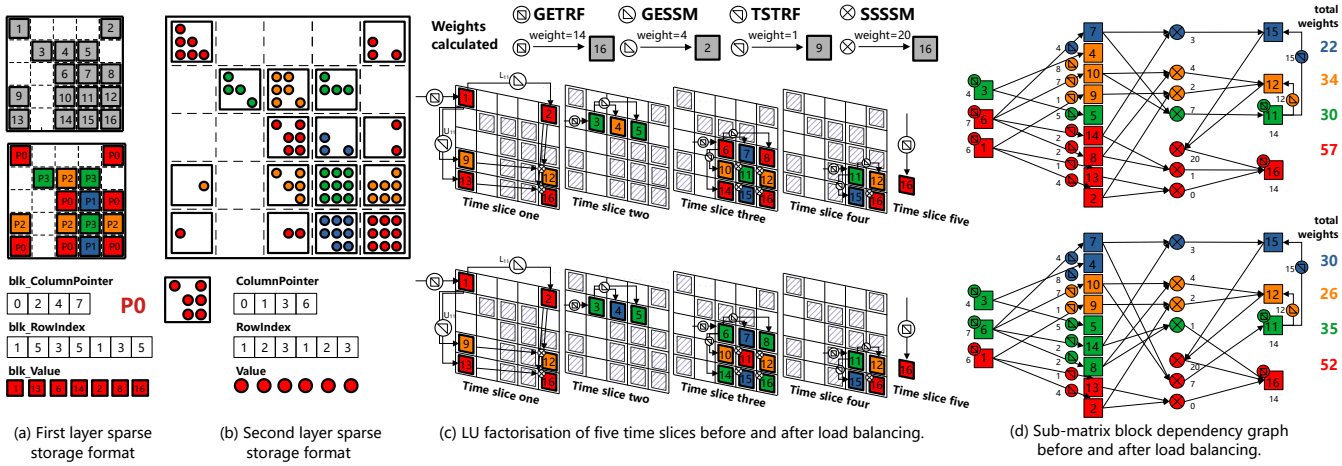
Figure 5: Ratio of synchronisation time to numeric factorisation time in multiple processes. The x-axis is multiple matrices with 1, 2, 4, 8, 16, 32 and 64 processes, and the y-axis is the ratio.

4 PANGULU

4.1 Overview

In this paper, we propose a new sparse direct solver on distributed heterogeneous systems called PanguLU, which includes five steps: reordering, symbolic factorisation, preprocessing, numeric factorisation and triangular solve. (1) In the reordering, PanguLU uses the MC64 [32, 33] algorithm to maintain the numerical stability, and METIS [50] to reduce the nonzero fill-ins during the symbolic factorisation. (2) The symbolic factorisation in PanguLU uses symmetric pruning [41] to reduce the computational complexity and improve performance over unsymmetric pruning [40]. (3) The preprocessing divides the matrix into sub-matrix blocks and sends them to each process. The block size is calculated from the matrix order and the density of the matrix after symbolic factorisation to balance the computation and communication. Then each process constructs its own two-layer sparse structure. (4) The numeric factorisation contains a large number of floating-point operations to determine the numerical values of L and U. (5) Finally, PanguLU uses triangular solve to solve $Ly = b$ and $Ux = y$ for the final solution x, where x, y and b are vectors and b is known.

To enable PanguLU to better exploit the computational power of heterogeneous distributed systems, we propose a new sparse LU numeric factorisation algorithm. It consists of three major components to eliminate the drawbacks of the existing work: (1) A regular 2D sparse block structure, (2) Adaptive sparse BLAS kernels, and (3) A synchronisation-free scheduling strategy. The three components of the LU numeric factorisation are described in detail in the following.



(a) First layer sparse storage format

(b) Second layer sparse storage format

(c) LU factorisation of five time slices before and after load balancing.

(d) Sub-matrix block dependency graph before and after load balancing.

Figure 6: Data layout and mapping. (a) and (b) are the sparse storage formats of the first and second layers, respectively. In the figure, sub-matrix blocks are allocated according to the two-dimensional process grid under four processes, with different colours representing different processes. The upper and lower subplots of (c), respectively, represent the five time slices of LU factorisation before and after load balancing, with examples of kernel weight calculations included at the top. (d) represents the matrix block dependency graph and the weights of each process before and after load balancing.

Firstly, PanguLU splits the original matrix into several blocks of equal size and stores them by using the Compressed Sparse Column (CSC) format. To balance the computational load between processes, we develop a mapping method to redistribute the computational load in Section 4.2. Secondly, in the numeric factorisation of PanguLU, since the matrix blocks are sparse, we develop four dedicated sparse kernels: general triangular factorisation (GETRF), sparse lower triangular solve (GESSM), sparse upper triangular solve (TSTRF), and Schur complement with sparse-sparse matrix multiplication (SSSSM)[3]. The performance of sparse BLAS can vary widely for sparse blocks with different structures and different methods. Therefore, we design adaptive sparse BLAS to speed up the computation by selecting the appropriate algorithm according to the structure of the input sparse matrices, as described in Section 4.3. Finally, to better reduce synchronisation overhead on distributed systems, PanguLU proposes a synchronisation-free scheduling strategy. This strategy uses sparse kernels as the smallest scheduling unit. A synchronisation-free array is used to perform scheduling and maintain correctness. Each process executes the highest priority sparse kernel first in Section 4.4.

4.2 Data Layout and Mapping

PanguLU uses regular two-dimensional blocking with equal block sizes, which are distributed to each process according to a two-dimensional process grid. Each process uses a two-layer sparse structure to store the matrix. At the block level, we use a block-based CSC format to compress nonzero blocks. Within each block, sub-matrix blocks are also stored by using the CSC format.

The sparse structure of the first layer is shown in Figure 6(a). The sub-matrix blocks result from the regular two-dimensional blocking method, and the grey blocks, numbered consecutively, denote sub-matrix blocks containing nonzeros. The empty blocks are represented by blank spaces. The non-empty blocks are allocated across four processes within a two-dimensional process grid, with each one being identified by a unique colour. Each process only stores the essential sub-matrix blocks for computation and employs position informations of sub-matrix blocks in the original matrix to facilitate communication. This work uses three auxiliary arrays, *blk_ColumnPointer*, *blk_RowIndex* and *blk_Value*, to store the positions of the sub-matrix blocks. They store the prefix sum of the nonzero sub-matrix blocks within each column, the row index of each nonzero sub-matrix block after matrix blocking, and its sub-matrix block pointer, respectively.

The sparse structure of the second layer is shown in Figure 6(b). The nonzeros within a sub-matrix block are stored with the CSC format to compress, and we can see the storage result for the sixth sub-matrix block. From the above Section, the four main kernels are used for the sub-matrix blocks to compute. GETRF factorises the input matrix A into a lower triangular matrix L and an upper triangular matrix U; GESSM and TSTRF perform the lower or upper triangular solve; and SSSSM performs the Schur complement operation. The weight of the corresponding task can be obtained by calculating the FLOPs of the kernel.

It is worth noting that this two-layer sparse structure has no significant additional overhead, as we only need three additional arrays to represent and access the block-level sparse structure efficiently. In addition, PanguLU allocates the required memory for the sub-matrix blocks owned by each process during preprocessing and also allocates space for the matrices L and U required for the computation, in order to minimise memory consumption by reusing the space.

Based on the two-layer sparse structure, we implement a static load balancing strategy, which is a preprocessing before the numeric factorisation to balance the load by calculating the weight of the different processes on each time slice, where each weight corresponds to a task. The Figures 6(c) and 6(d) show five time slices of

LU numeric factorisation under multiprocess and the dependency graph of sub-matrix blocks, respectively, where the coloured sub-matrix blocks represent those to be calculated and the number next to the kernel icon indicates the weight of the particular calculation. For example, in the first time slice, sub-matrix blocks 1, 2, 9, 12, 13 and 16 need to be calculated, where sub-matrix block 1 performs GETRF, sub-matrix blocks 2, 9 and 13 perform GESSM or TSTRF, and sub-matrix blocks 12 and 16 perform SSSSM, and the dependencies between these sub-matrix blocks are shown in the dependency graph. The whole LU factorisation is completed by running in the time slice order.

The upper and lower subplots of Figures 6(c) and 6(d), respectively, represent the sub-matrix block distributions before and after load balancing. We perform fine-grained load balancing on the first layer structure, based on the total weight of each process and the weight of each process on the different time slices considered. In the second time slice, we balance the load between the process 1 with the highest total weight and the process 2 with the smallest number of tasks. We accomplish this by swapping all tasks assigned to these two processes in this time slice, resulting in a visualisation of sub-matrix block 4's GESSM migrating from process 2 to process 1, as seen in the lower subplots of Figures 6(c) and 6(d).

This load balancing causes the weight of process 1 to increase by 4, while the weight of process 2 decreases by the same amount, due to the GESSM weight of sub-matrix block 4. Similarly, in the third time slice, we perform load balancing, changing the total computational weight of each process. As shown in the figure, the total weight of each process is counted for comparison, and the result shows that the load in this example has reached a certain level of balance. Such a static strategy is completed by preprocessing, and the main overhead is the calculation of weights, which has a small time overhead compared to numeric factorisation.

4.3 Sparse Kernels and Algorithm Selection

The performance of sparse kernels is critical for numeric factorisation and is affected by a variety of factors such as the density, structure and size of the matrix. Therefore, optimizing sparse kernels and selecting a better algorithm for each case becomes an essential task to achieve overall performance gains. To this end, we implement 17 sparse kernels in PanguLU (three GETRF, five GESSM, five TSTRF and four SSSSM, as shown in Table 1), and then construct sparse kernels algorithm selection strategies according to a large amount of performance data, so as to select the kernel with better performance.

In Table 1, "C_Vi" and "G_Vi" denote CPU and GPU versions of the kernel, respectively. The "Addressing Method" indicates how the calculation value is located and updated, where "Direct" indicates direct data update in a dense space, "Bin-search" indicates that data update by searching in a sparse space, and "Merge" represents that data update by logically merging two sets of sparse spaces. In addition, the "Dense Mapping" corresponds to "Direct" and represents that the sparse structure is mapped to a dense space, and the computation traverses the sparse structure through dense space addressing assignments, where SSSSM maps only the result matrix C to dense space.

Kernel	Version	Addressing Method	Parallelising Method	Dense Mapping
GETRF	C_V1	Direct	Row	✓
	G_V1	Bin-search	Un-sync SFLU	-
	G_V2	Direct	Un-sync SFLU	✓
TSTRF/ GESSM	C_V1	Merge	Column	-
	C_V2	Direct	Column	✓
	G_V1	Bin-search	Warp-level column	-
	G_V2	Bin-search	Un-sync warp-level row	-
	G_V3	Direct	Warp-level column	✓
SSSSM	C_V1	Direct	Approximate equal load column block	✓
	C_V2	Bin-search	Adaptive split-bin type	-
	G_V1	Bin-search	Adaptive multi-level	-
	G_V2	Direct	Warp-level column	✓

Table 1: Details of sparse BLAS kernels in PanguLU.

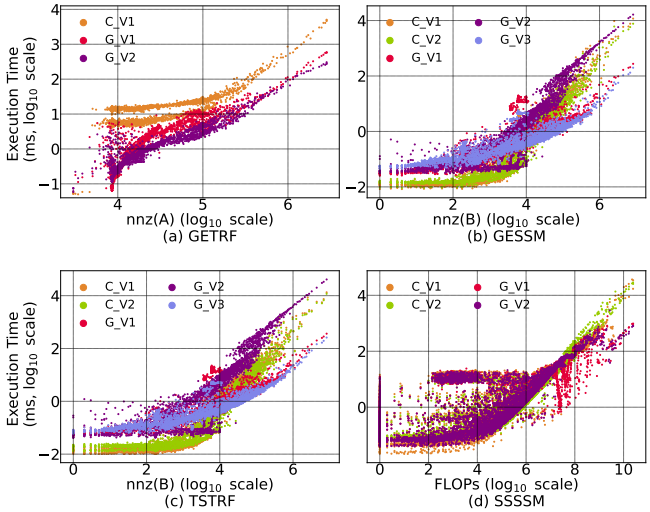


Figure 7: Performance comparison of sparse BLAS kernels.

We demonstrate the performance of all these sparse kernels in PanguLU running the matrices shown in Table 3, which produces 4,550 sub-matrices for GETRF, 18,786 sub-matrices for both GESSM and TSTRF, and 86,982 sub-matrices for SSSSM (details of the experimental platform are given in Section 5.1). The results are shown in Figure 7, demonstrating various trends of performance variation for these sparse kernels. These kernels vary in performance, and none of them can always maintain the best performance, but when these kernels are combined in an appropriate way based on matrix characteristics, the overall performance can be greatly improved. Drawing from a vast pool of data and with a keen focus on selecting a more appropriate sparse kernel for each matrix block, we develop four decision trees to guide our algorithm selection process. These decision trees are illustrated in Figure 8. For GETRF, GESSM and TSTRF, which belong to panel factorisation, necessitaty the selection of the most appropriate algorithm primarily based on the number of nonzeros (nnz) present in the matrix. For SSSSM, it is primarily based on the FLOPs involved in the computation.

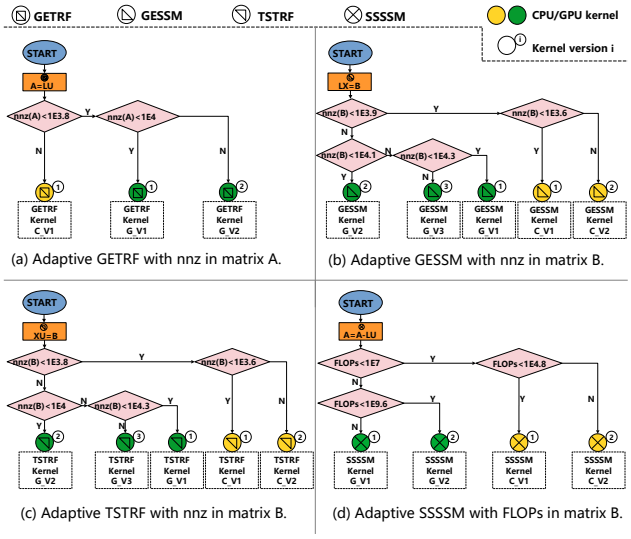


Figure 8: The decision trees constructed for the four kernels.

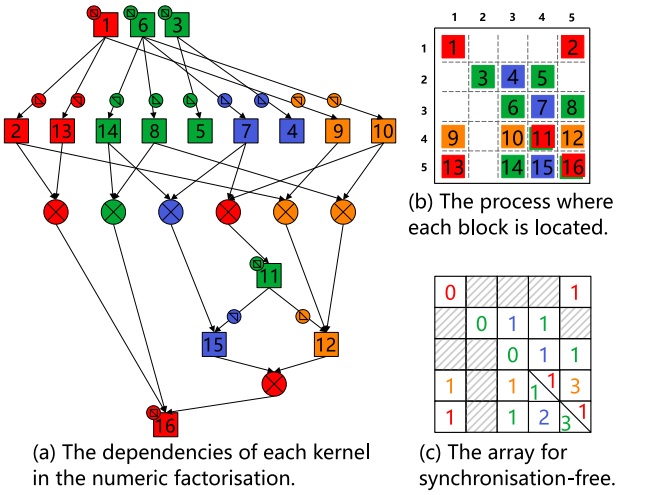


Figure 9: An example of synchronisation-free array preprocessing process. (a) shows the dependencies of each kernel in the numeric factorisation, where the different icons in the circles indicate four different kernels and the arrows indicate the dependencies. Different colours in (b) indicate the process on which each sub-matrix block resides. (c) shows the synchronisation-free array corresponding to sub-matrix blocks in (b).

4.4 Synchronisation-Free Scheduling

In this section, we introduce the synchronisation-free scheduling strategy proposed by PanguLU. It uses sparse kernels as the minimum scheduling unit to optimise the efficiency of parallelism within a distributed system. We achieve fine-grained process scheduling by transferring the values of the synchronisation-free array between processes, so that as many processes as possible are in a working state. The synchronisation-free scheduling strategy of this work is designed to reduce redundant synchronisation overhead other than the implicit synchronisation in the Schur complement caused by the data dependency of the LU synchronisation, i.e. the synchronisation between the tree nodes, in order to achieve higher parallelism. This strategy consists of two main parts.

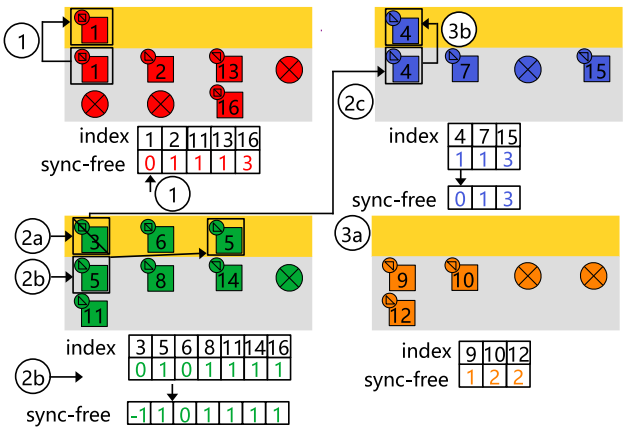
The first part is the construction of a synchronisation-free array. Several kernels can be executed for a sub-matrix block, including GETRF, GESSM, TSTRF and SSSSM, of which SSSSM may be executed multiple times. The working state of the process responsible for a sub-matrix block is influenced by the state of other related sub-matrix blocks. We construct a synchronisation-free array to record the remaining workload of each sub-matrix block with a value equal to the number of times that sub-matrix block still needs to execute GESSM, TSTRF or SSSSM, with the value subtracted by 1 for each execution. Note that due to the special nature of the diagonal matrix, when the diagonal sub-matrix block corresponds to a value of 0 for the array data, the GETRF operation needs to be executed, and the value is subtracted by 1 upon completion. If the value of the array corresponding to the diagonal sub-matrix block is -1, the TSTRF dependency of the corresponding row sub-matrix block and the GESSM dependency of the corresponding column will be broken. If the value of the array of the non-diagonal sub-matrix block is 0, the SSSSM dependency of its corresponding row or column sub-matrix blocks will be broken. The process corresponding to the sub-matrix block whose dependency is broken adds the corresponding kernel to the computation queue. As shown in Figure 9, the direction of the arrow represents the dependency between the sub-matrix blocks, the number indicates the remaining workload of the sub-matrix blocks, and the colour of the number indicates the process to which it belongs. The workload here represents the number of times that sub-matrix block still needs to execute GESSM, TSTRF or SSSSM. GETRF is performed for sub-matrix block 1 without arrows and a value of 0 on the array. For sub-matrix block 16 with three arrows, the variable on the synchronisation-free array is 3.

The second part is to update the synchronisation-free array and manage the calculation and communication of the processes, as shown in Figure 10.

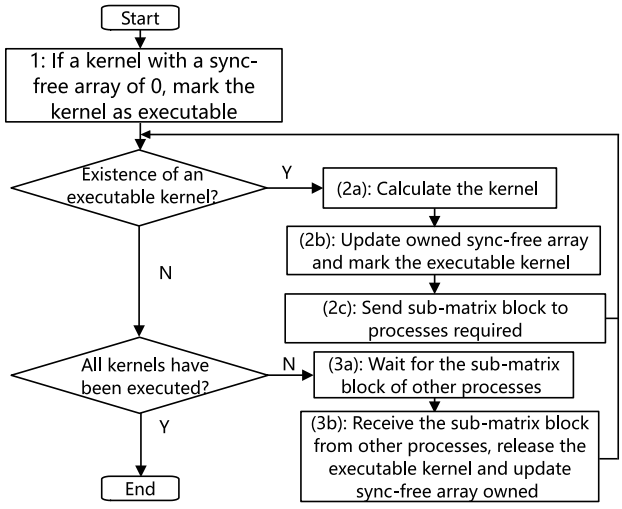
Step 1. Each process starts by marking the kernels with a variable 0 in the synchronisation-free array as executable and adding the kernels to the task queue. The process then checks if an executable kernel is currently available. If so, it will start the calculation of the sparse kernel. Otherwise, it waits for other processes to send the required sub-matrix block.

Step 2. The process begins to calculate the executable kernel. If there are multiple executable kernels, they are selected in order of priority. After the calculation, the process updates its own synchronisation-free array and sends the sub-matrix block to the other required process. In Figure 10, (2a) indicates the calculation of the kernel, (2b) indicates the updating of the synchronisation-free array on this process and the addition of new kernels, and (2c) indicates the sending of the sub-matrix block to other processes.

Step 3. The process will wait for the sub-matrix block from other processes. When it receive the sub-matrix block from other processes, the new executable kernels can be added, and then the



(a) The procedure of a synchronisation-free scheduling strategy.



(b) The flow chart of our synchronisation-free algorithm.

Figure 10: An example of our synchronisation-free algorithm, where the icons in the circles in (a) correspond to the operations in (b) labeled as (1), (2a), (2b), (2c), (3a), (3b). Each process stores a synchronisation-free array and a task queue containing kernels. Kernels are added to the queue once they have been freed from dependencies. The process uses its own synchronisation-free array and sends sub-matrix block to corresponding processes upon completion of the kernel.

synchronisation-free array can be updated. As shown in Figure 10, (3a) indicates that the process is waiting for the sub-matrix block from other processes, and (3b) indicates that the process is receiving the sub-matrix block from other processes, releasing the executable kernel and updating the synchronisation-free array.

The synchronisation-free strategy proposed by PanguLU can reduce the synchronisation cost to some extent. In the computation, each process always selects the most critical of the tasks to be computed to perform the computation, making the computation of the tasks on the critical path as fast as possible, so that more processes are in the state of computation.

5 EXPERIMENTAL RESULTS

5.1 Experimental Setup

We conduct experiments on two different 32-node 128-GPU distributed clusters, one with NVIDIA A100 GPUs and the other with AMD MI50 GPUs. Hardware information for both platforms is provided in Table 2. NVIDIA GPU platform uses CUDA 11.3.0 and driver 510.85.02. AMD GPU platform uses the ROCm-4.3.1. For both platforms, we compile SuperLU_DIST and PanguLU with gcc-9.3.0, OpenMPI-4.1.2, and cmake-3.23.1. We employ four MPI processes per node, each individual process independently occupying an NVIDIA A100 or an AMD MI50 GPU. Our experiments in Sections 5.2, 5.4, 5.5 and 5.6 are based on the A100 GPU platform, while the experiments in Section 5.3 are based on both the A100 GPU and MI50 GPU platforms. Regarding the data set, we use 16 representative sparse matrices from the SuiteSparse Matrix Collection [22] in various domains, and most of the test matrices are selected from those commonly used in SuperLU_DIST papers [67, 69]. Details of the matrices are shown in Table 3.

NVIDIA GPU Platform	AMD GPU Platform
4 * NVIDIA A100 GPUs with 40 GB, B/W 1555GB/s 2 * Intel Xeon 8180 CPUs @ 2.5 GHz 512GB DDR4	4 * AMD MI50 GPUs with 16 GB, B/W 1024GB/s 1 * AMD Epyc 7601 CPU @ 2.2 GHz 128GB DDR4

Table 2: Information of the two test platforms.

Matrix	n(A) (10^5)	nnz(A) (10^6)	SuperLU nnz($L+U$) (10^8)	PanguLU nnz($L+U$) (10^8)	PanguLU FLOPs (10^{11})
apache2	7.15	4.82	3.19	2.61	3.46
ASIC_680k	6.83	2.64	1.51	1.18	7.68
audikw_1	9.44	77.65	25.39	24.74	117.65
cage12	1.30	2.03	5.83	5.70	42.30
CoupCons3D	4.17	17.28	5.26	4.99	9.05
dielFilterV3real	11.03	89.31	11.33	10.77	20.08
ecology1	10.00	5.00	1.31	0.72	0.30
G3_circuit	15.85	7.66	2.83	1.81	0.91
Ga41As41H72	2.68	18.49	46.11	45.74	813.72
Hook_1498	14.98	59.37	30.59	29.44	163.27
inline_1	5.04	36.82	3.59	3.28	2.52
ldoor	9.52	42.49	3.23	2.80	1.45
nlpkkt80	10.62	28.19	38.52	33.92	329.49
Serena	13.91	64.13	55.33	54.31	598.51
Si87H76	2.40	10.66	39.42	39.08	662.30
SiO2	1.55	11.28	20.69	20.47	267.25

Table 3: The matrices tested in this paper. The matrix order, the number of nonzeros before and after SuperLU_DIST's symbolic factorisation, the number of nonzeros before and after PanguLU's symbolic factorisation, as well as the total number of floating-point operations in PanguLU's numeric factorisation are listed.

5.2 Symbolic Factorisation Time

We first compare the symbolic factorisation time of SuperLU_DIST and PanguLU. Both solvers use a serial symbolic factorisation algorithm. The difference is that SuperLU_DIST uses a combination of pruning and supernodes for symbolic factorisation, whereas PanguLU symmetrises the matrix and uses symmetric pruning to speed up the symbolic factorisation. Compared to SuperLU_DIST, PanguLU is on geometric mean 4.45x faster, with a maximum of 6.80x

in the cage12 matrix. In particular, on some structured matrices such as audikw_1 and nlpkkt80, PanguLU has a significant speedup, with improvements of 3.51x and 4.59x, respectively. We also note that although symmetric pruning introduces additional fill-ins, especially for highly non-symmetric matrices, our method achieves an average reduction in fill-ins of about 11% on the test matrices compared to SuperLU_DIST, demonstrating a lower number of fill-ins in Table 3.

5.3 Scalability of Numeric Factorisation

The performance of SuperLU_DIST and PanguLU in numeric factorisation is evaluated in experiments using 1, 2, 4, 8, 16, 32, 64 and 128 A100 GPUs and MI50 GPUs. Figure 12 shows the experimental results. Compared to SuperLU_DIST, PanguLU achieves an average speedup of 2.53x and 2.79x on the NVIDIA GPU platform and the AMD GPU platform, with speedups ranging from 1.10x - 11.70x and 1.12x - 17.97x, respectively.

In addition, PanguLU achieves a performance advantage of 1.10x and 1.12x on the NVIDIA and AMD GPU platforms respectively for structured audikw_1 matrices, which are well handled by SuperLU_DIST. In particular, PanguLU achieves significant performance advantages for irregular matrices such as ASIC_680k with speedups of up to 11.70x and 17.97x on the two GPU platforms, respectively.

This experiment also demonstrates the good scalability of PanguLU on distributed heterogeneous systems. For example, PanguLU achieves good performance on the Ga41As41H72 matrix as the number of GPUs increases. Compared to a single GPU, PanguLU can scale to 47.51x and 74.84x on 128 A100 GPUs and 128 MPI50 GPUs, respectively. But the scalability of SuperLU_DIST and PanguLU decreases with 128 GPUs for a few matrices, such as the apache2 and the ecology1. This reduction is mainly due to the increase in communication costs, despite the faster computation by using more GPUs.

5.4 Kernel Time on a Single GPU

We compare the kernel time of SuperLU_DIST and PanguLU on a single A100 GPU. Table 4 shows that PanguLU has a geometric mean of 6.54x compared to SuperLU_DIST in the whole computation kernel. Both SuperLU_DIST and PanguLU perform panel factorisation based on the original blocks, and the performance difference is mainly due to the computational method. In the Schur complement, SuperLU_DIST needs to gather the sub-matrix blocks together, perform the matrix multiplication, scatter them to the corresponding positions, and perform the subtraction. However,

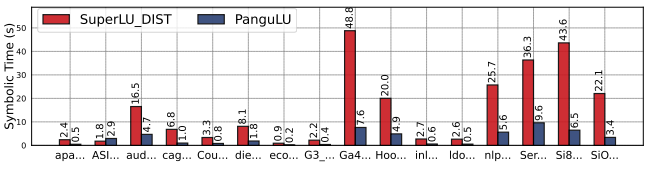


Figure 11: Symbolic factorisation time comparison of SuperLU_DIST and PanguLU.

PanguLU performs the Schur complement directly on the original matrix blocks, greatly reducing data movement overhead. In addition, PanguLU uses sparse kernels for computation, while SuperLU_DIST uses dense kernels. Therefore, when dealing with irregular matrices such as ASIC_680k, the benefit of sparse computation in PanguLU is very obvious, resulting in a more significant speedup.

Matrix	Panel Fact. Time (s)		Schur Time (s)		All Time (s)		
	SuperLU	PanguLU	SuperLU	PanguLU	SuperLU	PanguLU	Speedup
apache2	9.4	6.32	27.36	3.86	36.76	10.18	3.61x
ASIC_680k	3.39	3.21	982.36	17.8	985.76	21.01	46.92x
audikw_1	103.44	47.94	764.83	159.35	868.27	207.29	4.19x
cage12	15.8	13	3570.76	70.65	3586.56	83.65	42.88x
CoupCons3D	15.69	11.11	55.55	8.5	71.24	19.61	3.63x
dielFilterV3real	41.62	24.63	127.92	25.27	169.54	49.9	3.40x
ecology1	2.26	1.8	2.88	0.57	5.14	2.37	2.17x
G3_circuit	5.69	4.98	14.71	1.35	20.4	6.34	3.22x
Ga41As41H72	979.85	80.91	21893.96	1275.41	22873.81	1356.32	16.86x
Hook_1498	117.77	66.3	1208.79	230.17	1326.56	296.47	4.47x
inline_1	10.15	6.15	19.66	2.04	29.82	8.19	3.64x
ldoor	7.2	4.55	13.87	1.05	21.07	5.59	3.77x
nlpkkt80	173.69	80.48	1816.94	537.19	1990.63	617.68	3.22x
Serena	238.6	115.98	3551.09	899.36	3789.69	1015.35	3.73x
Si87H76	152.21	74.88	17245.73	1034.96	17397.94	1109.84	15.68x
SiO2	77.78	38.57	9762.91	410.6	9840.69	449.17	21.91x
Geometric Mean	-		-		-		6.54x

Table 4: Kernel time comparison of SuperLU_DIST and PanguLU for 16 test matrices on an A100 GPU.

5.5 Synchronisation Cost on 128 GPUs

We also compare the synchronisation time of SuperLU_DIST and PanguLU on 128 A100 GPUs. PanguLU uses the synchronisation-free scheduling strategy to reduce the synchronisation overhead, and the experimental results (Figure 13) show that it is effective for most matrices ,with an average of 2.20x compared to SuperLU_DIST. Besides, for structured matrices such as the Hook_1498 and the audikw_1, SuperLU_DIST can more easily form supernodes to make the computation more regular, and the synchronisation overhead is equal to that of PanguLU.

5.6 Effects of Different Optimisations

In Sections 4.3 and 4.4 we describe the two optimisations used in PanguLU in order to improve performance, the sparse kernel selection and the synchronisation-free scheduling strategy. Here, we analyse the effect of our optimisations through experiments. Figure 14 shows a performance comparison of the three versions of PanguLU on 128 A100 GPUs. Compared to the baseline version, PanguLU achieves speedups ranging from 1.0x to 2.2x (1.7x on average) by selecting more efficient kernels based on the matrix structure. For the cage12 and Hook_1498 matrices, the sparse kernel selection is very effective and speeds up by a factor of 2.2x and 2.1x, respectively. In particular, the ASIC_680k matrix has a speedup of 1.0x. Due to the inherent sparsity and irregular distribution of this matrix, high computational performance can be achieved with the base sparse kernel. We also test the performance of PanguLU on 128 A100 GPUs using both the sparse kernel selection and the synchronisation-free scheduling strategy. Eventually, PanguLU achieves speedups ranging from 2.3x to 5.4x (3.8x on average) over the baseline version.

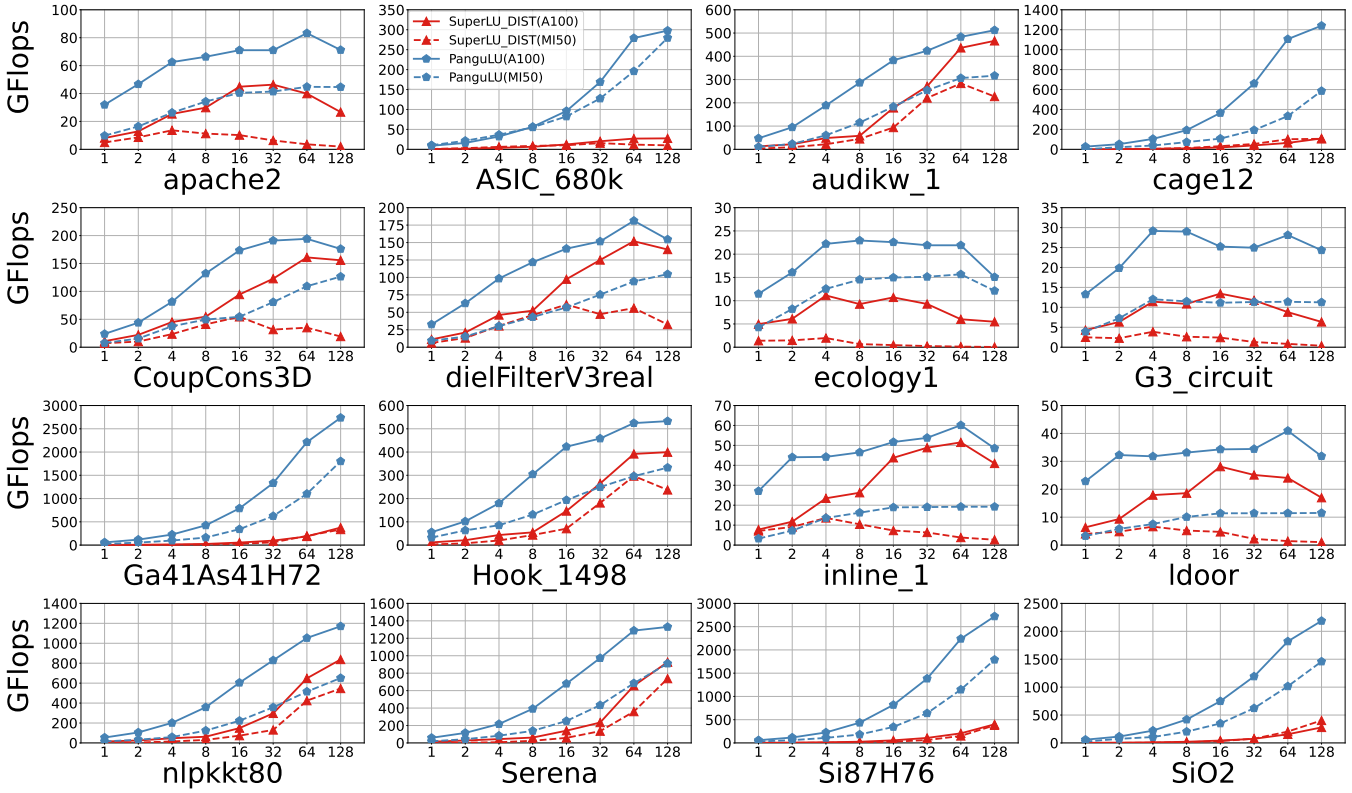


Figure 12: Performance comparison between SuperLU_DIST and PanguLU on 128 A100 GPUs and 128 MI50 GPUs platforms. The red lines and blue lines indicate the performance of SuperLU_DIST and PanguLU, respectively, while the solid lines and dotted lines indicate the performance on the A100 GPU and MI50 GPU platforms, respectively.

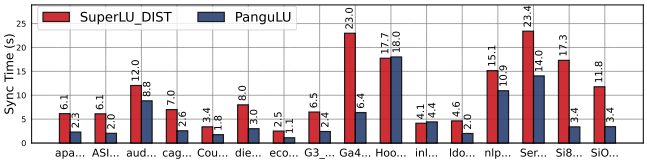


Figure 13: Synchronisation time comparison of SuperLU_DIST and PanguLU on 128 A100 GPUs.

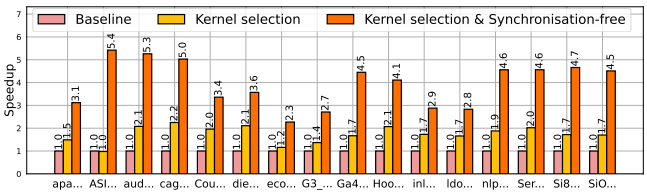


Figure 14: Relative performance improvement of using the sparse kernel selection and the synchronisation-free scheduling strategy on 128 A100 GPUs.

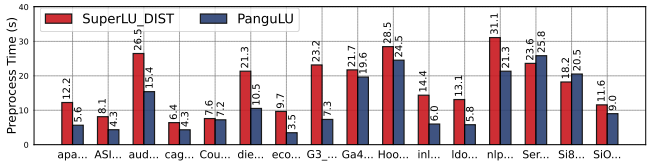


Figure 15: Preprocessing time comparison of SuperLU_DIST and PanguLU on 128 A100 GPUs.

5.7 Preprocessing Cost

Finally, to fully demonstrate the overall performance of our work, we also compared the preprocessing time of SuperLU_DIST and PanguLU, and the results are shown in Figure 15. Compared with SuperLU_DIST, PanguLU has faster preprocessing stage for most of the test matrices, on average by a factor of 1.61x and up to a factor of 3.16x. In particular, for sparse matrices such as G3_circuit and inline_1, PanguLU's preprocessing speed is improved by 3.16x and 2.40x, respectively. However, for the Serena and Si87H76 matrices, PanguLU is slightly slower than SuperLU_DIST due to the matrix structure and the higher overhead of converting the 2D block layout, with speedups of 0.91x and 0.89x, respectively.

6 RELATED WORK

Much work has been done over the past decades to accelerate sparse direct solvers. For example, maintaining the numerical stability of the LU factorisation by pivoting [24, 31, 35, 37, 38, 54], reducing the amount of computations of the sparse LU factorisation by reordering [5, 6, 19, 20, 51], accelerating symbolic factorisation [40, 41, 44, 45], improving the parallelism of the numeric factorisation [14, 16, 29, 56, 63, 64, 70, 76], and achieving a higher performance of the sparse triangular solve [12, 26, 58–60]. However, the parallelism of the sparse direct solvers still has room to improve, mainly due to the following three challenges.

The first challenge for sparse direct solvers is **how to maintain the sparse properties of the sparse LU factorisation and fully exploit the scalability of modern supercomputers**. Duff and Reid [36] proposed the multifrontal methods, and Demmel et al. [24] and Li et al. [53, 55] developed the supernodal method. These methods require the input matrices to be transformed into a relatively regular pattern by collecting similar columns, combining them into a dense sub-matrix and using dense BLAS for the calculation. Much work [4, 10, 21, 28, 39, 42, 43, 77, 80] has been optimised on the basis of these two methods because they have the advantage of not only preserving the sparse properties, but also effectively improving the scalability when dealing with regular matrices. Furthermore, Gupta [47, 48] achieved better load balancing by stealing tasks using a task parallelism engine. Amestoy et al. [7] utilised dynamic task scheduling based on a multifrontal algorithm to achieve load balancing in a distributed system. Duff et al. [34] exploited a combination of new load balancing and communication minimisation techniques to improve scalability. Sao et al. [65–67] proposed a communication-avoiding 3D algorithm to balance the load of the supernodes and to make the solver more scalable. However, multifrontal and supernodal methods rely heavily on the matrix structure, which can lead to suboptimal performance on irregular matrices. In this paper, 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.

The second challenge is **how to make better use of heterogeneous processors to speed up sparse LU factorisation.** Ren et al. [62] optimised work partitioning based on GPU architecture to accelerate LU factorisation, and Chen et al. [13] combined the characteristics of task-level and data-level parallelism on GPU to optimise sparse LU factorisation for circuit simulation. He et al. [49], Lee et al. [52], Peng and Tan [61] developed GLU, which performs sparse LU factorisation using a level-set approach on GPU. However, this approach often requires a lot of time for synchronisation between kernel calls. To reduce the synchronisation costs, Zhao et al. [81] developed SFLU, which uses a synchronisation-free algorithm to improve the parallelism on GPU. However, these optimisations are based on a single GPU for optimisation in LU factorisation, which is usually memory constrained. For this reason, Xia et al. [78] proposed an end-to-end approach to address memory limitations on a single GPU. With the development of large-scale supercomputers, this has motivated much work to scale the LU factorisation to distributed heterogeneous systems, such as Gaihre et al. [44], who exploit the high throughput of GPUs to accelerate symbolic factorisation. Sao et al. [69] developed the ability to aggregate small dense BLAS operations into larger ones to utilise GPU. Tian et al. [75] optimise computational kernels for better parallel efficiency and cache utilisation on the hierarchy of the Sunway many-core architecture. All of these studies design better parallel approaches based on heterogeneous architectures. Compared with them, we design a block-wise sparse BLAS with 17 sparse kernels and a decision tree approach that selects sparse kernels based on the matrix structure to improve parallel efficiency on GPUs.

The third challenge is **how to reduce the synchronisation overhead in large-scale distributed systems.** Communication costs have been one of the main bottlenecks on the performance of distributed solvers. Existing distributed solvers use many methods to reduce communication costs. Such as, Amestoy et al. [9] used a dynamic scheduling strategy that utilised asynchronous communication, effectively avoiding communication costs by overlapping communication and computation in MUMPS. Schenk et al. [71] designed a dynamic two-level scheduling scheme to reduce cache conflicts and interprocessor communication costs in PARDISO. In SuperLU_DIST, Sao et al. [68] proposed an HALO algorithm to hide communication costs in distributed Xeon Phi systems. In addition, Agullo et al. [2] and Tan et al. [74] used a pipelined approach to overlap computation and communication to improve the performance of distributed solvers. Grigori et al. [46] proposed CALU, a communication-avoiding LU factorisation algorithm that avoids communication costs as much as possible. However, little work has focused on effectively reducing the synchronisation overhead of LU factorisation in distributed systems. In this work, we propose the synchronisation-free scheduling strategy that allows each process to compute as much as possible and reduces the synchronisation overhead to improve the performance of distributed solvers.

7 CONCLUSION

In this paper, we have proposed PanguLU, a scalable regular 2D block-cyclic sparse direct solver on distributed heterogeneous platforms. In PanguLU, a mapping approach was designed for load balancing, a variety of block-wise sparse BLAS methods were selected for higher efficiency on GPU, and a synchronisation-free communication strategy was developed to reduce the overall latency cost. Our experimental results showed that PanguLU achieved up to 11.70x and 17.97x speedups over the latest SuperLU_DIST with 128 NVIDIA A100 GPUs and 128 AMD MI50 GPUs, and brought up to 47.51x and 74.84x speedups over a single GPU, respectively.

ACKNOWLEDGMENTS

We thank all reviewers for their invaluable comments. This work was supported by the National Key R&D Program of China (Grant No. 2021YFB0300600, No. 2022YFB4400400), the National Natural Science Foundation of China (Grant No. 62372467, No. 61972415, No. 62204265), the State Key Laboratory of Computer Architecture (ICT, CAS) (Grant No. CARCHA202115) and the GHfund A (Grant No. 202302017546). Weifeng Liu is the corresponding author of this paper. We thank Beijing Super Cloud Computing Centre and Wuhan Supercomputing Centre for providing the computing resources and excellent technical support. We also thank Dr. Weile Jia for very helpful discussion, as well as Shuhui Song for suggestions on plotting the figures.

REFERENCES

[1] https://www.top500.org/. 2023.

[2] E. Agullo, P. R. Amestoy, A. Buttari, A. Guermouche, J.-Y. L'Excellent, and F.-H. Rouet. Robust Memory-Aware Mappings for Parallel Multifrontal Factorizations. *SIAM Journal on Scientific Computing*, 38(3), 2016.

[3] E. Agullo, C. Augonnet, J. Dongarra, M. Faverge, J. Langou, H. Ltaief, and S. Tomov. LU Factorization for Accelerator-Based Systems. In *AICCSA '11*. IEEE, 2011.

[4] P. R. Amestoy, C. Ashcraft, O. Boiteau, A. Buttari, J.-Y. L'Excellent, and C. Weisbecker. Improving Multifrontal Methods by Means of Block Low-rank Representations. *SIAM Journal on Scientific Computing*, 37(3), 2015.

[5] P. R. Amestoy, T. A. Davis, and I. S. Duff. An Approximate Minimum Degree Ordering Algorithm. *SIAM Journal on Matrix Analysis and Applications*, 17(4), 1996.

[6] P. R. Amestoy, T. A. Davis, and I. S. Duff. Algorithm 837: AMD, An Approximate Minimum Degree Ordering Algorithm. *ACM Transactions on Mathematical Software*, 30(3), 2004.

[7] P. R. Amestoy, I. S. Duff, J.-Y. L'Excellent, and J. Koster. A Fully Asynchronous Multifrontal Solver Using Distributed Dynamic Scheduling. *SIAM Journal on Matrix Analysis and Applications*, 23(1), 2001.

[8] P. R. Amestoy, I. S. Duff, J.-Y. L' Excellent, and J. Koster. MUMPS: A General Purpose Distributed Memory Sparse Solver. In *International Workshop on Applied Parallel Computing*, 2000.

[9] P. R. Amestoy, I. S. Duff, and C. Vömel. Task Scheduling in an Asynchronous Distributed Memory Multifrontal Solver. *SIAM Journal on Matrix Analysis and Applications*, 26(2), 2004.

[10] P. R. Amestoy and C. Puglisi. An Unsymmetrized Multifrontal LU Factorization. *SIAM Journal on Matrix Analysis and Applications*, 24(2), 2002.

[11] E. Anderson, Z. Bai, C. Bischof, L. S. Blackford, J. W. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, and A. McKenney. *LAPACK Users' Guide*. SIAM, 1999.

[12] S. Cayrols, I. S. Duff, and F. Lopez. Parallelization of the Solve Phase in A Task-based Cholesky Solver Using A Sequential Task Flow Model. *The International Journal of High Performance Computing Applications*, 34(3), 2020.

[13] X. Chen, L. Ren, Y. Wang, and H. Yang. GPU-Accelerated Sparse LU Factorization for Circuit Simulation with Performance Modeling. *IEEE Transactions on Parallel and Distributed Systems*, 26(3), 2015.

[14] X. Chen, Y. Wang, and H. Yang. An Adaptive LU Factorization Algorithm for Parallel Circuit Simulation. In *ASP-DAC '12*, 2012.

[15] X. Chen, Y. Wang, and H. Yang. NICSLU: An Adaptive Sparse Matrix Solver For parallel Circuit Simulation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 32(2), 2013.

[16] X. Chen, Y. Wang, and H. Yang. A Fast Parallel Sparse Solver For SPICE-based Circuit Simulators. In *DATE '15*, 2015.

[17] T. A. Davis. Algorithm 832: UMFPACK V4.3—an Unsymmetric-Pattern Multifrontal Method. *ACM Transactions on Mathematical Software*, 30(2), 2004.

[18] T. A. Davis. *Direct Methods for Sparse Linear Systems*. SIAM, 2006.

[19] T. A. Davis, J. R. Gilbert, S. I. Larimore, and E. G. Ng. A Column Approximate Minimum Degree Ordering Algorithm. *ACM Transactions on Mathematical Software*, 30(3), 2004.

[20] T. A. Davis, J. R. Gilbert, S. I. Larimore, and E. G. Ng. Algorithm 836: COLAMD, A Column Approximate Minimum Degree Ordering Algorithm. *ACM Transactions on Mathematical Software*, 30(3), 2004.

[21] T. A. Davis and W. W. Hager. Dynamic Supernodes in Sparse Cholesky Update/Downdate and Triangular Solves. *ACM Transactions on Mathematical Software*, 35(4), 2009.

[22] T. A. Davis and Y. Hu. The University of Florida Sparse Matrix Collection. *ACM Transactions on Mathematical Software*, 38(1), 2011.

[23] T. A. Davis and E. Palamadai Natarajan. Algorithm 907: KLU, A Direct Sparse Solver for Circuit Simulation Problems. *ACM Transactions on Mathematical Software*, 37(3), 2010.

[24] J. W. Demmel, S. C. Eisenstat, J. R. Gilbert, X. S. Li, and J. W. H. Liu. A Supernodal Approach to Sparse Partial Pivoting. *SIAM Journal on Matrix Analysis and Applications*, 20(3), 1999.

[25] J. W. Demmel, J. R. Gilbert, and X. S. Li. An Asynchronous Parallel Supernodal Algorithm for Sparse Gaussian Elimination. *SIAM Journal on Matrix Analysis and Applications*, 20(4), 1999.

[26] N. Ding, Y. Liu, S. Williams, and X. S. Li. A Message-driven, Multi-GPU Parallel Sparse Triangular Solver. In *ACDA21 '21*, 2021.

[27] I. S. Duff. A Survey of Sparse Matrix Research. *Proceedings of the IEEE*, 65(4), 1977.

[28] I. S. Duff. Parallel Implementation of Multifrontal Schemes. *Parallel computing*, 3(3), 1986.

[29] I. S. Duff. Parallel Algorithms for Sparse Matrix Solution. *Parallel Computing*, 2020.

[30] I. S. Duff, A. M. Erisman, and J. K. Reid. *Direct Methods for Sparse Matrices*. Oxford University Press, Inc, 2nd edition, 2017.

[31] I. S. Duff, J. D. Hogg, and F. Lopez. A New Sparse Symmetric Indefinite Solver Using A Posteriori Threshold Pivoting. *NLAFET Working Note*, 2018.

[32] I. S. Duff and J. Koster. 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), 1999.

[33] I. S. Duff and J. Koster. On Algorithms for Permuting Large Entries to the Diagonal of a Sparse Matrix. *SIAM Journal on Matrix Analysis and Applications*, 22(4), 2001.

[34] I. S. Duff, P. Leleux, D. Ruiz, and F. S. Torun. Improving the Scalability of the ABCD Solver with a Combination of New Load Balancing and Communication Minimization Techniques. In *Parco '19*, 2019.

[35] I. S. Duff and S. Pralet. Strategies for Scaling and Pivoting for Sparse Symmetric Indefinite Problems. *SIAM Journal on Matrix Analysis and Applications*, 27(2), 2005.

[36] I. S. Duff and J. K. Reid. The Multifrontal Solution of Indefinite Sparse Symmetric Linear. *ACM Transactions on Mathematical Software*, 9(3), 1983.

[37] I. S. Duff and J. K. Reid. MA48, a Fortran code for direct solution of sparse unsymmetric linear systems of equations. *Science and Engineering Research Council, Rutherford Appleton Laboratory*, 1993.

[38] I. S. Duff and J. K. Reid. The Design of MA48: A Code for The Direct Solution of Sparse Unsymmetric Linear Systems of Equations. *ACM Transactions on Mathematical Software*, 22(2), 1996.

[39] I. S. Duff and J. A. Scott. A Parallel Direct Solver for Large Sparse Highly Unsymmetric Linear Systems. *ACM Transactions on Mathematical Software*, 30(2), 2004.

[40] S. C. Eisenstat and J. W. H. Liu. Exploiting Structural Symmetry in Unsymmetric Sparse Symbolic Factorization. *SIAM Journal on Matrix Analysis and Applications*, 13(1), 1992.

[41] S. C. Eisenstat and J. W. H. Liu. Exploiting Structural Symmetry in a Sparse Partial Pivoting Code. *SIAM Journal on Scientific Computing*, 14(1), 1993.

[42] K. Eswar, P. Sadayappan, C.-H. Huang, and V. Visvanathan. Supernodal Sparse Cholesky Factorization on Distributed-memory Multiprocessors. In *ICPP '93*, volume 3, 1993.

[43] K. Eswar, P. Sadayappan, and V. Visvanathan. Multifrontal Factorization of Sparse Matrices on Shared-Memory Multiprocessors. In *ICPP '91*, 1991.

[44] A. Gaihre, X. S. Li, and H. Liu. gSoFa: Scalable Sparse Symbolic LU Factorization on GPUs. *IEEE Transactions on Parallel and Distributed Systems*, 33(4), 2022.

[45] L. Grigori, J. W. Demmel, and X. S. Li. Parallel Symbolic Factorization for Sparse LU with Static Pivoting. *SIAM Journal on Scientific Computing*, 29(3), 2007.

[46] L. Grigori, J. W. Demmel, and H. Xiang. CALU: A Communication Optimal LU Factorization Algorithm. *SIAM Journal on Matrix Analysis and Applications*, 32(4), 2011.

[47] A. Gupta. WSMP: Watson Sparse Matrix Package (Part-I: Direct Solution of Symmetric Sparse Systems). *IBM TJ Watson Research Center, Yorktown Heights, NY*, 2000.

[48] A. Gupta. WSMP: Watson Sparse Matrix Package (Part-II: Direct Solution of General Sparse Systems). *IBM TJ Watson Research Center, Yorktown Heights, NY*, 2000.

[49] K. He, S. X.-D. Tan, H. Wang, and G. Shi. GPU-Accelerated Parallel Sparse LU Factorization Method for Fast Circuit Analysis. *IEEE Transactions on Very Large Scale Integration Systems*, 24(3), 2015.

[50] G. Karypis and V. Kumar. METIS: A Software Package for Partitioning Unstructured Graphs, Partitioning Meshes, and Computing Fill-reducing Orderings of Sparse Matrices. 1997.

[51] B. Kumar, P. Sadayappan, and C.-H. Huang. On Sparse Matrix Reordering for Parallel Factorization. In *ICS '94*, 1994.

[52] W.-K. Lee, R. Achar, and M. S. Nakhla. Dynamic GPU Parallel Sparse LU Factorization for Fast Circuit Simulation. *IEEE Transactions on Very Large Scale Integration Systems*, 26(11), 2018.

[53] X. S. Li. An Overview of SuperLU: Algorithms, Implementation, and User Interface. *ACM Transactions on Mathematical Software*, 31(3), 2005.

[54] X. S. Li and J. W. Demmel. Making Sparse Gaussian Elimination Scalable by Static Pivoting. In *SC '98*, 1998.

[55] X. S. Li and J. W. Demmel. SuperLU_DIST: A Scalable Distributed-Memory Sparse Direct Solver for Unsymmetric Linear Systems. *ACM Transactions on Mathematical Software*, 29(2), 2003.

[56] X. S. Li, P. Lin, Y. Liu, and P. Sao. Newly Released Capabilities in the Distributed-Memory SuperLU Sparse Direct Solver. *ACM Transactions on Mathematical Software*, 49(1), 2023.

[57] J. W. H. Liu. The Role of Elimination Trees in Sparse Factorization. *SIAM Journal on Matrix Analysis and Applications*, 11(1), 1990.

[58] W. Liu, A. Li, J. D. Hogg, I. S. Duff, and B. Vinter. A Synchronization-Free Algorithm for Parallel Sparse Triangular Solves. In *Euro-Par '16*, 2016.

[59] Y. Liu, N. Ding, P. Sao, S. Williams, and X. S. Li. Unified Communication Optimization Strategies for Sparse Triangular Solver on CPU and GPU Clusters. In *SC '23*, 2023.

[60] Y. Liu, M. Jacquelin, P. Ghysels, and X. S. Li. Highly Scalable Distributed-memory Sparse Triangular Solution Algorithms. In *CSC '18*, 2018.

[61] S. Peng and S. X.-D. Tan. GLU3.0: Fast GPU-based Parallel Sparse LU Factorization for Circuit Simulation. *IEEE Design & Test*, 37(3), 2020.
[62] L. Ren, X. Chen, Y. Wang, C. Zhang, and H. Yang. Sparse LU Factorization for Parallel Circuit Simulation on GPU. In *DAC '12*, 2012.
[63] P. Sadayappan and S. K. Rao. Communication Reduction for Distributed Sparse Matrix Factorization on a Processor Mesh. In *SC '89*, 1989.
[64] P. Sadayappan and V. Visvanathan. Efficient Sparse Matrix Factorization for Circuit Simulation On Vector Supercomputers. In *DAC '98*, 1989.
[65] P. Sao, R. Kannan, X. S. Li, and R. Vuduc. A Communication-Avoiding 3D Sparse Triangular Solver. In *ICS '19*, 2019.
[66] P. Sao, X. S. Li, and R. Vuduc. A Communication-Avoiding 3D LU Factorization Algorithm for Sparse Matrices. In *IPDPS '18*, 2018.
[67] P. Sao, X. S. Li, and R. Vuduc. A Communication-Avoiding 3D Algorithm for Sparse LU Factorization on Heterogeneous Systems. *Journal of Parallel and Distributed Computing*, 131, 2019.
[68] P. Sao, X. Liu, R. Vuduc, and X. S. Li. A Sparse Direct Solver for Distributed Memory Xeon Phi-Accelerated Systems. In *IPDPS '15*, 2015.
[69] P. Sao, R. Vuduc, and X. S. Li. A Distributed CPU-GPU Sparse Direct Solver. In *Euro-Par '14*, 2014.
[70] O. Schenk and K. Gärtner. Solving Unsymmetric Sparse Systems of Linear Equations with PARDISO. *Future Generation Computer Systems*, 20(3), 2004.
[71] O. Schenk and K. Gärtner. Two-level Dynamic Scheduling in PARDISO: Improved scalability on Shared memory Multiprocessing Systems. *Parallel Computing*, 28(2), 2002.
[72] O. Schenk, K. Gärtner, and W. Fichtner. Efficient Sparse LU Factorization with Left-right Looking Strategy on Shared Memory Multiprocessors. *BIT Numerical Mathematics*, 40, 2000.
[73] O. Schenk, K. Gärtner, W. Fichtner, and A. Stricker. PARDISO: a High-Performance Serial and Parallel Sparse Linear Solver in Semiconductor Device Simulation. *Future Generation Computer Systems*, 18(1), 2001.
[74] G. Tan, C. Shui, Y. Wang, X. Yu, and Y. Yan. Optimizing the LINPACK Algorithm for Large-Scale PCIe-Based CPU-GPU Heterogeneous Systems. *IEEE Transactions on Parallel and Distributed Systems*, 32(9), 2021.
[75] M. Tian, J. Wang, Z. Zhang, W. Du, J. Pan, and T. Liu. swSuperLU: A Highly Scalable Sparse Direct Solver on Sunway Manycore Architecture. *The Journal of Supercomputing*, 78(9), 2022.
[76] T. Wang, W. Li, H. Pei, Y. Sun, Z. Jin, and W. Liu. Accelerating Sparse LU Factorization with Density-Aware Adaptive Matrix Multiplication for Circuit Simulation. In *DAC '23*, 2023.
[77] J. Xia, S. Chandrasekaran, M. Gu, and X. S. Li. Superfast Multifrontal Method for Large Structured Linear Systems of Equations. *SIAM Journal on Matrix Analysis and Applications*, 31(3), 2010.
[78] Y. Xia, P. Jiang, G. Agrawal, and R. Ramnath. End-to-End LU Factorization of Large Matrices on GPUs. In *PPoPP '23*, 2023.
[79] Z. Yan, B. Xie, X. Li, and Y. Bao. Exploiting Architecture Advances for Sparse Solvers in Circuit Simulation. In *DATE '22*, 2022.
[80] C. Yu, W. Wang, and D. Pierce. A CPU–GPU Hybrid Approach for the Unsymmetric Multifrontal Method. *Parallel Computing*, 37(12), 2011.
[81] J. Zhao, Y. Wen, Y. Luo, Z. Jin, W. Liu, and Z. Zhou. SFLU: Synchronization-Free Sparse LU Factorization for Fast Circuit Simulation on GPUs. In *DAC '21*, 2021.

Appendix: Artifact Description/Artifact Evaluation

ARTIFACT DOI

https://doi.org/10.5281/zenodo.8252752

ARTIFACT IDENTIFICATION

PanguLU is an open source package for solving the linear system $Ax = b$. The package uses a sparse matrix block LU factorisation algorithm, which partitions the sparse matrix into multiple sparse matrix blocks and then uses the sparse basic linear algebra subprograms to execute between the sparse matrix blocks. The algorithm can run accurately and efficiently on heterogeneous distributed platforms.

We have evaluated PanguLU on a 32-node distributed GPU cluster. Each node contains two Kunpeng 920 7265 CPUs and four NVIDIA A100 GPUs. Four MPI processes are used on each node, with each process independently occupying one NVIDIA A100 GPU. We used GPUs for acceleration in all experiments as described in the paper. To facilitate the reproduction of our experiments, this artifact includes the source code, dataset, all test scripts and test data.

More details about the artefacts are given below:

- **Relevant Hardware Details**: 32-node distributed GPU cluster. Each node contains two Kunpeng 920 7265 CPUs and four NVIDIA A100 GPUs. The nodes are interconnected through Cisco CX-6 NICs and four 100G ports to switches.
- **Operating Systems and Versions**: CentOS 7
- **Compilers and Versions**: gcc 9.3.0
- **Libraries and Versions**: OpenMPI-4.1.2; CUDA-11.3.0;
- **Input Datasets and Versions**: 16 square matrices in the SuiteSparse Matrix Collection.
- **Key Algorithms**: Sparse LU factorisation.
- **Publicly available?**: Yes

Author-Created or Modified Artifacts:
Persistent ID:
https://doi.org/10.5281/zenodo.8252752
Artifact name: PanguLU

REPRODUCIBILITY OF EXPERIMENTS

- **Download**
  First, the artifact can be delivered in this link:
  https://doi.org/10.5281/zenodo.8252752
  Second, at the time of writing, PanguLU only supports input files in matrix market format (*.mtx). The matrix list (the dataset we used in our experiments) is included in our package. These matrices are publicly available in the SuiteSparse Matrix Collection.
  We can run the python script *matrix16.py* in the <PanguLU_dir>/matrix folder to retrieve these matrices automatically. These matrices will be stored in the <PanguLU_dir>/matrix folder.
  The whole process is possible to take up to half an hour depending on the network conditions.
- **Installing and Compiling**
  Next we need to compile the artifact. This artifact requires the GNU GCC 9.3.0, OpenMPI-4.1.2 and CUDA-11.3.0 compilation environments, which have been compiled and running on CentOS 7 and GPU driver 510.85.02. It is also expected to compile and run successfully on other Linux distributions. We configure the appropriate environment by entering it in *make.inc* and use GNU make to compile and build the executable file. See the *README* for more detailed instructions. Installation and compilation will take about 10 minutes.
- **Experiments**
  There are five shell scripts and five python scripts in the <PanguLU_dir>/script folder to reproduce the performances and figures in our paper. Figure 7, Figure 11, Figure 12, Figure 13 and Figure 14 can be repeated by running *figureX.sh* in script to obtain the corresponding performance and plotting the corresponding figure with *figureX.py*. All performances and figures will be saved in the <PanguLU_dir>/data folder and <PanguLU_dir>/figure folder after all scripts have been completed, which will take about four days in total.

ARTIFACT DEPENDENCIES REQUIREMENTS

- Hardware resources required and utilized: The solver needs to be built on a heterogeneous distributed platform, and the algorithm will use GPUs for computing, so a distributed cluster of NVIDIA A100 GPUs must be used, with each process occupying a single GPU. The hardware for our experiments is a 32-node distributed GPU cluster. Each node contains two Kunpeng 920 7265 CPUs and four NVIDIA A100 GPUs. The nodes are interconnected through Cisco CX-6 NICs and four 100G ports to switches.
- Operating systems: **CentOS 7**.
- Software libraries: **GCC 9.3.0**; **OpenMPI-4.1.2**; **CUDA-11.3.0**; **metis-5.0.2**.
- Input dataset: 16 square matrices from the SuiteSparse Matrix Collection, the dataset needed to execute the code, which are freely and openly available to the public for research purposes. The motivation for selecting the dataset for the experiments was: the most of our test matrices were selected from those commonly used in SuperLU_DIST papers, and the rest of the matrices were from the same, using these matrices for testing ensures a fair and representative experiment.
- Any other dependencies or requirements: The test requires a total of 128 GPUs, 4 GPUs and two Kunpeng 920 7265 CPUs on each node. The nodes are interconnected through Cisco CX-6 NICs and four 100G ports to switches. Carefully, we employ four MPI processes per node, with each individual process independently occupying an A100 GPU. The test requires the installation of the running system CentOS 7, as well as the installation of dependent packages. These include GCC 9.3.0, OpenMPI-4.1.2, CUDA-11.3.0 and metis-5.0.2.

ARTIFACT INSTALLATION DEPLOYMENT PROCESS

- Install and compile the libraries and the code:
  You can download the latest version from the link below:
  https://doi.org/10.5281/zenodo.8252752
  Second, at the time of writing, PanguLU only supports input files in matrix market format (*.mtx). The matrix list (the dataset we used in our experiments) is included in our package. These matrices are publicly available in the SuiteSparse Matrix Collection.
  Step 1. You need to place the zip package we provided in a shared directory on a distributed system and then run matrix16.py in matrix/ to download the matrix.
  Step 2. We need to compile the artifact. This artifact requires the GNU GCC 9.3.0, OpenMPI-4.1.2 and CUDA-11.3.0 compilation environments, which have been compiled and running on CentOS 7 and GPU driver 510.85.02. It is also expected to compile and run successfully on other Linux distributions. We configure the appropriate environment by entering it in *make.inc* and use GNU make to compile and build the executable file. See the *README* for more detailed instructions. We provide a *new.sh* to install the 64-bit METIS package we provided, our own program and the code needed to draw the figure.
  Installation and compilation will take about 10 minutes.
  Step 3. You need to modify run.sh located in the script folder, we provide two ways to run in run.sh:
  (1) mpirun -np [NP] <PanguLU_dir>/test/$numerical_file -F < PanguLU_dir > /matrix/$Smatrix_name
  With this command we do not specify the node to be used in the distributed system, you can add a host file to specify the location where the MPI program is assigned, depending on how your distributed system runs, e.g:
  mpirun -np [NP] -hostfile host <PanguLU_dir>/test/$numerical_file -F < PanguLU_dir > /matrix/$Smatrix_name
  where the host file is used to specify the node used in the distributed system.
- process description to deploy the code:
  In the <PanguLU/dir>/script folder there are 5 shell scripts and 5 python scripts to reproduce the performance and figures in our paper. The actual performance of figures 7, 11, 12, 13 and 14 can be reproduced by running *figureX.sh* in the scripts to obtain the corresponding performance.
  Step 4. In our experimental replication, there are five main scripts in the script, they correspond to figure7, figure11, figure12, figure13, and figure14 in the thesis. here is the specific time for testing: figure7.sh (3 minutes), figure7_origin.sh (3 days), figure11.sh (30 minutes), figure12.sh (3 hours), figure12_origin.sh (16 hours), figure13.sh (30 minutes), figure14.sh (1 hour and 30 minutes).
  In order to reduce the exact amount of time needed for the tests, we have reduced the tests for figure7 and added a new script *figure7.sh*, which allows you to reproduce the data from our figure7 experiments with *figure7_origin.sh*.
  In addition, since MPI programs run very slowly at low process levels, we have also shortened the tests in *figure12.sh* by reducing the number of tests at 1, 2, and 4 processes, so that you can reproduce the performance more quickly.
  When all the scripts have been completed, all the performance and figures are saved in the <PanguLU_dir>/data folder. We also provide the corresponding drawing script *figureX.py* to draw the corresponding figures, which will be saved in the <PanguLU_dir>/figure folder when the drawing is finished.
  It is particularly important to note that the output data from *figureX.sh* is only stored in the <PanguLU_dir>/data folder. All the output files can be obtained from *configureX_getedata.py* file in the <PanguLU_dir>/script folder and stored in <PanguLU_dir>/data folder. We provide the complete script *figureX.sh* in <PanguLU_dir>/script folder to help you with the following operations: run the code, get the data, get the figure.
  Step 5. We also provide an all_figures.sh in <PanguLU_dir>/script to plot all the figures in our paper directly. You can run it with the command : " bash all_figures.sh " and get figure7, figure11, figure12, figure13 and figure14 in <PanguLU_dir>/figure folder.
  Note that this script uses different data to the data you used to run the code in step 4. It only uses data from our tests, whereas step 4 only used data from your system tests. We are just providing a reference for how to plot the figures in our paper.
  Step 6. We provide an all.sh to run all matrix16.py, new.sh, figure7.sh, figure11.sh, figure12.sh, figure13.sh, figure14.sh .
  Note that you must check that run.sh in <PanguLU_dir>/script is able to send the process to the distributed system for execution before running all the duplicate new experiments, otherwise unknown errors may occur.

OTHER NOTES

In particular, it is important to note that using mpirun for MPI programmes may have specific MPI parameters for each machine, and we have included them in . /script/run.sh file, it may not be suitable for your machine, in particular, you need to add the -hostfile parameter to specify the number of MPIs to be sent to different nodes in the distributed cluster.

be aware that if a node has X A100 Nvidia GPUs then you need to map X processes to that node to do the calculations. For example, if a node has 4 A100 Nvidia GPUs, 4 processes need to be mapped to a single node.