COFFEE: Cross-Layer Optimization for Fast and Efficient Executions of Sinkhorn-Knopp Algorithm on HPC Systems

Chengyu Sun, Huizhang Luo, *Member, IEEE*, Hong Jiang, *Fellow, IEEE*, Jeff Zhang, and Kenli Li, *Senior Member, IEEE*

***Abstract*—In this paper, we present COFFEE, cross-layer optimization for fast and efficient executions of the Sinkhorn-Knopp (SK) algorithm on HPC systems with clusters of compute nodes by exploring some architectural features of the system. By analyzing the performance of a typical implementation of the SK algorithm on such a system, a huge performance gap is observed between the row rescaling and column rescaling of the algorithm, where the latter requires much more time than the former. We also found that the costly MPI communication of the column rescaling seriously hinders the exploitation of parallelism. By observing and leveraging unique architectural characteristics across different system optimizations, such as column rescaling redesign, data blocking, micro-kernel design, enhanced intra-node and inter-node communication in MPI, etc., COFFEE is able to explore cross-layer optimization opportunities that enable fast and efficient execution of the SK algorithm. Our experimental results show that COFFEE provides up to 7.5X with an average of 2.0X performance improvement over the typical implementation on a single node, and up to 2.9X with an average of 1.6X performance improvement over the state-of-the-art MPI Allreduce algorithms on Tianhe-1 supercomputer.**

***Index Terms*—Data blocking, HPC system, MPI allreduce, micro-kernel design, sinkhorn-knopp algorithm.**

I. Introduction

SINCE the mid-2010 s, Sinkhorn's theorem [1] has attracted significant attention in different application domains, such as mathematics, economics, and computer science. It is widely used to find solutions for the optimal transport problem, since the speed of its matrix scaling algorithm is several orders of magnitude faster than that of traditional transport solvers [2]. Recently, Sinkhorn distance [3] based on Sinkhorn's theorem is further proposed to evaluate the difference between data distributions and permutations, as a proxy for the optimal transport distance (a.k.a Wasserstein or Earth Mover's Distance) to improve its speed [4]. This has in turn gained increasing traction in the Artificial Intelligence (AI) community. A key to the application of Sinkhorn's theorem is the Sinkhorn-Knopp (SK) algorithm, a simple but very useful iterative method to approach the double stochastic matrix of Sinkhorn's theorem by alternately rescaling all rows and all columns of the given matrix [5]. The SK algorithm has been applied to a wide range of theoretical and practical problems, including optimal transport problem [2], [6], [7], [8], [9], Google's PageRank [10], computer vision [11], seismic tomography and reflection seismology [12], latent permutations [13], word mover's distance (WMD) [14], and Cooperative Bayesian [15], etc.

However, most of the existing SK algorithm works focus on using it to enhance machine learning algorithms, such as optimal transport distance and WMD as mentioned above, or speeding up the convergence, such as Anderson's acceleration [16], [17]. Very few of them consider improving the algorithm from the view of computer system architecture [18], particularly high-performance computing (HPC) systems. HPC systems, on the other hand, with their unique compute, memory and communication capabilities, pose new challenges and opportunities for unlocking the applications' full potential at scale [19]. Moreover, as described in Section III, it is found that the proportion of time occupied by the SK algorithm is more than half in all four representative applications analyzed. So, it is critical to optimize this part of the program.

This paper aims to speed up the SK algorithm with the Message Passing Interface (MPI) communication support on multi-core machines and multi-node clusters. To that end, we propose COFFEE, cross-layer optimization for fast and efficient executions of the SK algorithm on HPC systems by exploring and exploiting both architectural and algorithmic characteristics. We first analyze the performance of a typical implementation of the SK algorithm on Tianhe-1 supercomputer. It is found that the column rescaling takes much longer time than the row rescaling. The reason is that the memory accesses by the column rescaling are highly non-sequential, which leads to a high cache miss rate. To solve this problem, COFFEE explores the optimization opportunities of redesigning column rescaling and data blocking

Manuscript received 15 November 2022; revised 12 May 2023; accepted 14 May 2023. Date of publication 19 May 2023; date of current version 5 June 2023. This work was supported by the National Natural Science Foundation of China under Grant 62102141. Recommended for acceptance by S. Chandrasekaran. *(Corresponding author: Huizhang Luo.)*

Chengyu Sun, Huizhang Luo, and Kenli Li are with the College of Computer Science and Electronic Engineering, Hunan University, Changsha, Hunan 410082, China (e-mail: sunchengyu@hnu.edu.cn; luohuizhang@hnu.edu.cn; lkl@hnu.edu.cn).

Hong Jiang is with the Computer Science and Engineering Department, University of Texas at Arlington, Arlington, TX 76019 USA (e-mail: hong.jiang@uta.edu).

Jeff Zhang is with the School of Electrical, Computer and Energy Engineering, Arizona State University, Tempe, AZ 85281 USA (e-mail: jeffzhang@asu.edu).

Digital Object Identifier 10.1109/TPDS.2023.3277915

to reduce the cache miss rate. Further, we design micro-kernel and reschedule the instructions to increase instruction-level parallelism. We also apply Foster's methodology to parallelize programs of the SK algorithm. While applying it enables all the processors to work perfectly for the row rescaling, it leads to poor performance for the column rescaling. Because for the column rescaling, we found that the parallel SK algorithm results in an MPI Allreduce problem. It is discovered that the state-of-the-art (SOTA) MPI Allreduce solutions cannot meet the requirement of the SK algorithm. COFFEE solves this problem by exploring optimization opportunities for the MPI environment, including specially modifying intra-node MPI Reduce and optimizing MPI Allreduce by overlapping inter-node communication, intra-node communication and intra-node calculation.

It is known that some optimization ideas are already well-known and widely used. For example, in matrix-related operations (e.g., general matrix multiplication (GEMM)) and other operations that require enhanced data locality in the field of HPC, blocking and mirco-kernel design are pretty standard optimizations [20], [21], [22]. We first apply it to our algorithm and proved to be effective. However, we redesigned the algorithm of column rescaling which is original and innovative according to the characteristics of the SK algorithm. In the algorithm optimization related to MPI Allreduce, the idea of layering is very common. Many studies have used this idea, and we choose SALaR (as described in Section II-B) as the benchmark for our Allreduce optimization scheme [23]. However, we optimized the Reduce algorithm which is original and innovative and used it to replace the standard MPI_Reduce function in SALaR. At the same time, according to the characteristics of the SK algorithm, we originally found that the implementation of Allreduce can overlap with other computing tasks of the SK algorithm, further improving the performance.

The main contributions of this paper are listed as follows:

- We analyze the execution behaviors of the SK algorithm on HPC clusters and observe two main performance challenges. The first is its column rescaling that exhibits highly nonsequential memory access patterns, which lead to a very high cache miss rate and thus substantially reduce overall performance. The second is that the column rescaling severely limits parallelism even when it is designed with Foster's methodology (Section III).
- We propose COFFEE,[1] a novel approach that implements a multi-level optimization design to optimize the processing of the SK algorithm at scale in an HPC system (Section IV). We improve the parallel efficiency by enhancing MPI Allreduce with an effective leader-worker mechanism to overlap inter-node communication, intra-node communication and intra-node calculation as much as possible (Section V).
- We evaluate a prototype implementation of COFFEE on Tianhe-1 supercomputer, which demonstrates its significant performance advantages over the SOTA solutions (Section VI). Our experimental results show that COFFEE brings up to 7.5X and 2.9X performance improvement for a single node and multi-node environment, respectively.

[1]The source code is available at https://github.com/sunchengyu1111/COFFEE.

II. Background and Related Work

A. Sinkhorn-Knopp Algorithm

A matrix $A = (A_{ij})$ is doubly stochastic if all of its elements are non-negative and it marginally sums to 1, i.e., $\sum_i A_{ij} = \sum_j A_{ij} = 1$. Sinkhorn's theorem states that every square matrix with positive entries can be written in doubly stochastic form. To approach such a form, Sinkhorn and Knopp presented the simple but efficient SK algorithm, and analyzed its convergence [1], [5]. The main idea of the SK algorithm is to alternately rescale all rows and all columns of A to sum to 1. Algorithm 1 presents a typical implementation of the SK algorithm. The reasons for choosing the SK algorithm rather than its convergence-accelerated versions [24] are twofold. First, for the matrix multiplication, modern mainstream linear algebra libraries are based on the most basic three-loops implementation without Strassen or Coppersmith-Winograd algorithm [25], [26]. Similar to the basic implementation of matrix multiplication, the original SK algorithm lends itself more naturally and easily to be optimized from the view of computer system architecture, as discussed later. Second, while existing works on the SK algorithm mainly focus on speeding up the convergence by reducing the number of matrix scaling iterations, we aim to reduce the time of each iteration. This means that our optimizations will be orthogonal and complementary to existing works, allowing our solution to work with and be implemented on top of the latter. Note that we introduce the idea of marginal distribution to generalize the SK algorithm, such that the sums of rows and columns are not limited to 1. We use row marginal distribution (RMD) and column marginal distribution (CMD) to represent the sum of each row or column of the final matrix. The original SK algorithm is a special case of our implementation, where all the RMD_i and CMD_j are 1.

Applications of Sinkhorn's theorem in machine learning usually deal with large-scale data sets with high processing requirements, which will no doubt benefit greatly from the computing performance in the HPC environment. Therefore, there is an urgent need to accelerate the SK algorithm in such an environment, with typical optimizations along the computation (e.g., CPU) and communication (e.g., MPI) dimensions, as reviewed next.

Existing Work: Few of the prior works explicitly explore architectural and system features for SK algorithm optimization. A recent work [18] from the Intel parallel computing team uses the OpenMP based shared memory programming model and targets Intel's emerging architecture PIUMA and XEON CPUs, while our work in this paper targets supercomputer systems, a more representative HPC scenario. In addition, the baseline (typical implementation) we choose is a C language implementation, which is a rewritten version of the SOTA Python implementation in their work, in order to run at a large scale on supercomputer with MPI environment. The baseline has optimizations such as loop unrolling and data-level parallelism, and has performance not inferior to the Python implementation.

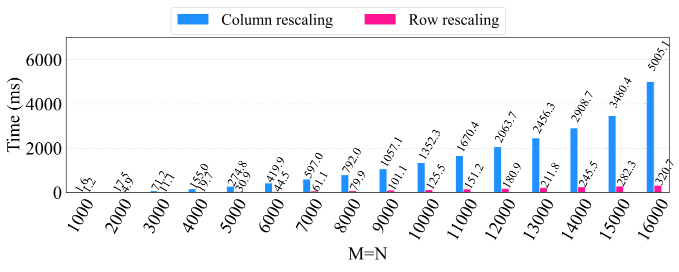
Algorithm 1: A Typical Implementation of SK Algorithm. α (β) is a Rescaling Factor for the Row (Column) Rescaling. M and N are the Numbers of Rows and Columns of the Matrix, Respectively. RMD and CMD are the Sum of Each Row or Column of the Final Matrix.

```
Require: $A_{M\times N}$, $CMD_{1\times N}$, $RMD_{1\times M}$, $\alpha = 0$, $\beta = 0$
Ensure: $A_{M\times N}$ after one iteration
 1:  //Row rescaling:
 2:  for $i$ from 0 to M-1 do
 3:      for $j$ from 0 to N-1 do
 4:          $\alpha += A[i][j]$;
 5:      end for
 6:      $\alpha = RMD[i]/\alpha$;
 7:      for $j$ from 0 to N-1 do
 8:          $A[i][j] *= \alpha$;
 9:      end for
10:  end for
11:  //Column rescaling:
12:  for $j$ from 0 to N-1 do
13:      for $i$ from 0 to M-1 do
14:          $\beta += A[i][j]$;
15:      end for
16:      $\beta = CMD[j]/\beta$;
17:      for $i$ from 0 to M-1 do
18:          $A[i][j] *= \beta$;
19:    end for
20:  end for
```

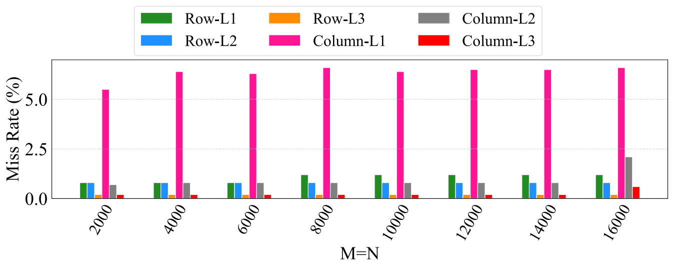
B. Optimization of MPI Allreduce

When performing row/column rescaling across different processors in a cluster environment, MPI communication among all processors[2] is required. The problem of calculating the sum of a row/column is a standard MPI Allreduce problem. The Allreduce operation is in fact one of the most widely used MPI primitives [27], [28], [29]. In recent years, in machine learning and other fields, the Ring algorithm [30], which divides the data in the num nodes equally into num parts, has been increasingly used and achieved good results. Through $num - 1$ steps, it allows each node to get one part of the complete Allreduce result. In the second stage, $num - 1$ steps are also performed, and the final result retained by each node is passed to all nodes, which can be approximated as an Allgather operation. When implementing multi-node Allreduce, the existing MPI library usually adopts a two-level design [23], [31]. It consists of three parts, including intra-node Reduce, inter-node Allreduce and intra-node Bcast. These three parts are equally important and are indispensable. In particular, Bayatpour et al. [23] proposed a MPI Allreduce design named SALaR. SALaR uses a two-level Allreduce design, and by dividing the data into small chunks, the inter-node Allreduce and intra-node Reduce operate in the form of a pipelining to overlap the running time. It shows significant performance improvement in deep learning frameworks.

[2]In this paper, we assume that one processor only runs one process.



(a) Running time



(b) Cache miss rate

Fig. 1. Difference between the row rescaling and column rescaling in serial environment. The y-axis shows the average results of running the iterations 10,000 times.

III. Motivation

We first introduce a typical parallel implementation of the SK algorithm in MPI environment with Foster's design methodology as the basis of our cross-layer optimizations. A key point is that how to divide the data and tasks, such that communications are as few as possible, and computation tasks are as balanced as possible. To that end, the matrix is divided into several submatrices by rows. By this way, there is no communication requirement during the row rescaling phase and all the communications happen during the column rescaling phase, which can be optimized together. So the execution of the algorithm becomes four steps. First, each process independently does the row rescaling of the submatrix, similar to Algorithm 1, Line 1-10. Second, each process calculates the sum of columns of the submatrix. Third, each process gets the sum of the columns of the matrix by calling the MPI_Allreduce function. Finally, each process independently does the remaining steps of column rescaling, that is, modifying the submatrix according to the column rescaling factors (the column rescaling factors are obtained from the results of the third step).

We evaluate the performance of the typical implementations of the SK algorithm on Tianhe-1 supercomputer [32] and report several observations. Without loss of generality, the matrices are assumed to be stored originally in the row-major format (See Section VI for more details).

A. SK Algorithm Profiling

Computation: Fig. 1(a) shows the running time comparison between the row rescaling and column rescaling for different matrix sizes in serial environment. It is observed that as the matrix size increases, the running time of a column rescaling gradually diverges from that of a row rescaling. For example,

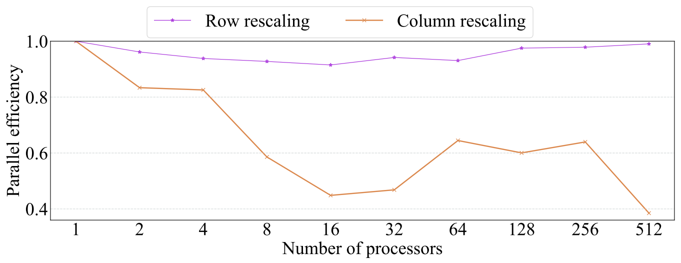


Fig. 2. Parallel efficiency comparison between different number of processors in Tianhe-1 supercomputer. We use a matrix of the same size (20,480 × 20,480), and spread the matrix across different numbers of processors. The experimental setup is 8 processors per node. Therefore, from 16 processors onwards, it will run on multiple nodes.

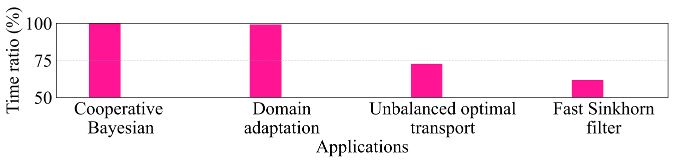


Fig. 3. The proportion of time occupied by the SK algorithm among four representative types of applications [2], [7], [8], [11], [15].

when the matrix size is 16,000 × 16,000, the running time of a column rescaling is 5,005.1 ms, which is 15.6X longer than that of a row rescaling. To explain the reason for the difference, we then use Valgrind [33] to analyze the cache miss rate of the row rescaling and column rescaling and the results are shown in Fig. 1(b). As we would expect, due to the nature of the column rescaling, it is less likely to exploit cache locality, especially for the L1 cache. For example, when the matrix size is 16,000 × 16,000, the L1 cache miss rate of the column rescaling is 6.6%, which is 5.5X larger than that of the row rescaling.

Communication: Fig. 2 shows the comparison of parallel efficiency when using different numbers of processors to run the row rescaling and column rescaling in the MPI environment. Parallel efficiency is computed as $T_1/(P \times T(P))$, where T_1 represents the time for optimal serial algorithm on one processor, P is the number of processors, and $T(P)$ is the time for parallel algorithm on P processors. We can observe that based on the partitioning strategy by rows mentioned previously, the row rescaling has high parallel efficiency. For example, the running time of 512 processors for a row rescaling is basically the same as that of a single processor. However, as the number of processors increases, the parallel efficiency of the column rescaling gradually decreases. For example, the parallel efficiency is reduced to 40% as the number of processors increases to 512. The reason is that the row rescaling is communication free and all the processors are load balanced. While for the column rescaling, the Allreduce used to find the column sum performs a lot of inter-node and intra-node communication, so that some processors are in the process of waiting for data.

It is noted that the parallel efficiency from 16 processors to 64 processors has an upward trend, and after 64 processors, the parallel efficiency begins to decline again. Because the variation of parallel efficiency in a multi-node environment is related to many factors. As the number of processors increases, the size of the submatrix allocated to each processor decreases, and so does the time for column rescaling. However, the rate at which the running time decreases is not proportional to the rate at which the matrix size decreases. As shown in Fig. 1(a), when the matrix size is changed from 4,000 × 4,000 to 2,000 × 2,000, the scale is reduced by 4 times and the running time is reduced by 8.9 (155.0 / 17.5) times. This has played a positive role in improving parallel efficiency. At the same time, when the scale of nodes increases, the time overhead caused by inter-node communication gradually dominates, resulting in a decline in parallel efficiency.

SK Algorithm in End-to-end Applications: We further examine several representative SK algorithm based applications and observe that the SK kernel consistently dominates each application's end-to-end execution time.[3] As shown in Fig. 3, the SK algorithm's execution alone occupies 99.99%, 99.39%, 72.88% and 62.00% of the end-to-end runtime respectively, of all four applications we ran. According to Amdahl's law, optimizing this dominant part of runtime can achieve the greatest effect in improving the overall performance of the applications.

B. Motivation of COFFEE

Our Insights: Motivated by the observations and analysis above, we found optimization opportunities in both computation and communication of the SK algorithm running on the state of the art HPC systems. However, *to the best of our knowledge*, we are the first to propose optimization strategies from these two factors affecting the running time of the SK algorithm.

For computation, COFFEE improves the column rescaling, and uses data blocking to maximize cache locality. It also leverages techniques such as instruction reordering to optimize micro-kernel operations. Moreover, the observation on the communication from the inherent characteristics of MPI motivates us to find a better way to balance the communication overhead by taking advantage of the different characteristics of communication bandwidth of inter-node and intra-node and trying to overlap computing and communication to further improve performance.

IV. CPU-Oriented Optimization

Both the row rescaling and column rescaling of the SK algorithm have a complexity of $O(n^2)$. Its rescaling time increases significantly as the matrix size increases. However, since the processed matrix is stored in the row-major format, it is much more difficult for a column rescaling than for a row rescaling to use the cache mechanism to obtain performance benefit from cache locality, as evidenced by experimental results presented in Section III. To address this problem, we first redesign the column rescaling method to reduce the cache miss rate by improving spatial locality. Then, based on the above redesign, we present how to design a data blocking strategy, how to design a

[3]We refer to the following repositories or libraries: [Online]. Available: https://github.com/PythonOT/POT, https://github.com/Jeston-de-Anda/SCBI_all, https://michielstock.github.io/posts/2017/2017-11-5-OptimalTransport/, https://github.com/paigautam/CVPR21_FastSinkhornFilters.

Algorithm 2: Improving Spatial Locality of Column Rescaling.

```
Require: $sum_{1\times N}$ initialized as 0, $temp_{1\times N}$, $CMD_{1\times N}$, $A_{M\times N}$
Ensure: $A_{M\times N}$ after column rescaling
1:  for $i$ from 0 to M-1 do
2:    for $j$ from 0 to N-1 do
3:      $sum[j] += A[i][j]$;
4:    end for
5:  end for
6:  for $j$ from 0 to N-1 do
7:    $temp[j] = CMD[j]/sum[j]$;
8:  end for
9:  for $i$ from 0 to M-1 do
10:   for $j$ from 0 to N-1 do
11:     $A[i][j] *= temp[j]$;
12:   end for
13: end for
```

micro-kernel and how to use pipelining technique to hide latency based on the SK algorithm for extreme performance.[4]

A. Column Rescaling Redesign

As shown in Algorithm 1, it is found that the main reason that the typical implementation of the column rescaling incurs a high cache miss rate is that sequentially accessing $A[i][j]$ and $A[i+1][j]$ will reference different cache lines. To address this problem, we first reschedule the accessing order of the column rescaling to improve the spatial locality.

As shown in Algorithm 2, we redesign the column rescaling by separating it's processing into the following two steps.

Step 1: Calculating the sum of columns concurrently (Lines 1–5). Unlike the typical implementation that calculates the sum of a column by adding all column elements at once, we reorganize the procedure so that each time only one element is added for all sums. After M iterations, we have the full sum of each column. This brings together $A[i][j]$ and $A[i][j+1]$, which are likely to be referenced together in a cache line. In doing so, the spatial locality of the program is improved.

Step 2: Modifying the matrix in row-wise order (Lines 6–13). We use an intermediate variable array $temp$ to store the rescaling factors, which are used to support the row-wise order matrix modification. Note that, the loops of the matrix modification in typical implementation cause a miss on each memory access because of the long stride given by index i in the inner loop. We switch the order of the loops, the stride is changed to 1, allowing the elements to be accessed in sequential order. By switching the order in which loops execute, misses can be significantly reduced by improving spatial locality.

[4]These optimizations are well adopted for HPC computation-intensive tasks such as general matrix multiplication (GEMM).

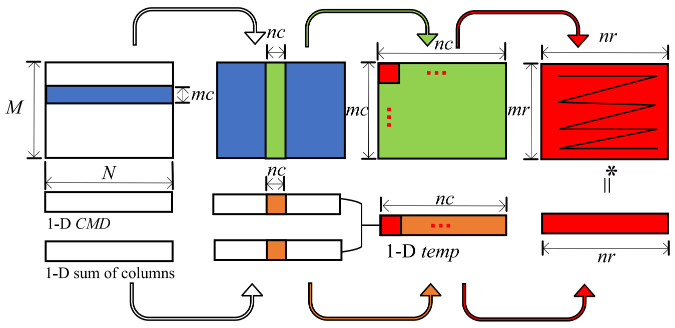


Fig. 4. Schematic of data blocking design.

B. Blocking Design

It is known that capacity misses of cache can occur for large matrices since it may not be possible to store all the elements of a matrix in the cache. We give the implementation of the blocking design to divide the matrix into several submatrices that can fit in the cache appropriately.

Fig. 4 shows the blocking algorithm, with the modifying part of the column rescaling as an example. The algorithm works similarly with other parts. Our design first divides the row dimension into several partitions. That is, we set the parameter mc in the outermost loop at Layer 1, expecting to partition the matrix of mc row panels. Second, we set the parameter nc at Layer 2, aiming to further partition a $mc \times N$ submatrix into several $mc \times nc$ submatrices. Then, we preprocess the two arrays of length nc required for column rescaling, namely, *CMD* and sum. The specific method is to create a 1-D array *temp* of length nc to store the rescaling factors needed for the current loop of nc column panels. There are two advantages to this method. First, CMD and sum are preprocessed to warm up the cache for $temp$. Second, the *temp* array length is nc rather than N, which is easier to be exploited by the L1 cache because it can be used in the mc rows of nc column panels. Note that *temp* does not need to be written back to main memory when column rescaling ends. Thus, using the *temp* array does not introduce extra overhead. In summary, blocking enables matrix A to benefit from spatial locality and array $temp$ to benefit from temporal locality.

Here we do not choose to carry out packing operation to the $mc \times nc$ submatrix for two reasons. First, unlike the matrix multiplication that needs to reuse the packed data (submatrices of A and B) many times, a row/column rescaling of the SK algorithm only needs to read each data once. Second, the column rescaling result needs to be written back to matrix A, which incurs significant overhead for packing. Therefore, the packing of the matrix cannot improve performance. At last, in the innermost layers, we set the parameters mr and nr, expecting to partition the $mc \times nc$ submatrix into $mr \times nr$ micro-kernels. The process of blocking and computing is shown within nested loops outlined in Algorithm 3. Note that we exchange the two outermost loops, such that the $temp[0 : nc - 1]$ is only calculated once.

C. Micro-Kernel Design

We further implement a micro-kernel to optimize the algorithm. As shown in Algorithm 3, the innermost loop (Lines

Algorithm 3: Blocking Algorithm for Matrix Modification of Column Rescaling.

```
1: for $j = 0 \rightarrow N-1$ step $= nc$ do
2:     $temp[0:nc-1] = CMD[j:j+nc-1]/sum[j:j+nc-1]$;
3:     for $i = 0 \rightarrow M-1$ step $= mc$ do
4:         for $ii = i \rightarrow i+mc-1$ step $= mr$ do
5:             for $jj = j \rightarrow j+nc-1$ step $= nr$ do
6:                 $l = jj \bmod nc$;
7:                 for $k = ii \rightarrow ii+mr-1$ step $= 1$ do
8:                     $A[k, jj:jj+nr-1] *= temp[l:l+nr-1]$;
9:                 end for
10:            end for
11:        end for
12:    end for
13: end for
```

```
movq %0, %%rax
vmovupd (%%rax), %%ymm0
movq %1, %%rbx
vmovupd (%%rbx), %%ymm1
movq %2, %%rcx
vmovupd (%%rcx), %%ymm2
movq %3, %%rdx
vmovupd (%%rdx), %%ymm3
movq %4, %%rax
vmovupd (%%rax), %%ymm4
vmulpd  %%ymm0, %%ymm1, %%ymm1
vmulpd  %%ymm0, %%ymm2, %%ymm2
vmulpd  %%ymm0, %%ymm3, %%ymm3
vmulpd  %%ymm0, %%ymm4, %%ymm4
vmovupd %%ymm1, (%%rbx)
vmovupd %%ymm2, (%%rcx)
vmovupd %%ymm3, (%%rdx)
vmovupd %%ymm4, (%%rax)
```

after schedule →

```
movq %0, %%rax
movq %1, %%rbx
movq %2, %%rcx
vmovupd (%%rax), %%ymm0
movq %3, %%rdx
vmovupd (%%rbx), %%ymm1
vmovupd (%%rcx), %%ymm2
movq %4, %%rax
vmulpd  %%ymm0, %%ymm1, %%ymm1
vmovupd (%%rdx), %%ymm3
vmovupd (%%rax), %%ymm4
vmulpd  %%ymm0, %%ymm2, %%ymm2
vmulpd  %%ymm0, %%ymm3, %%ymm3
vmulpd  %%ymm0, %%ymm4, %%ymm4
vmovupd %%ymm1, (%%rbx)
vmovupd %%ymm2, (%%rcx)
vmovupd %%ymm3, (%%rdx)
vmovupd %%ymm4, (%%rax)
```

Fig. 5. 4×4 processing micro-kernel and a better instruction schedule.

6-8) can be implemented as a $mr \times nr$ micro-kernel by unrolling the innermost loop to reduce loop overhead. We further accelerate the micro-kernel with SIMD, e.g., vectorization instructions [34]. To illustrate this in an example, we use AVX2 instructions for the x86 architecture to implement a 4×4 micro-kernel. The kernel is unrolled into four repeated operations with AVX2 instructions. Each operation handles four double-precision floating-point data, which is divided into three parts: load (i.e., _mm256_load_pd), multiply (i.e., _mm256_mul_pd), and store (i.e., _mm256_store_pd). There exists data dependency among the three parts, i.e., read-after-write hazard between load and multiply, and read-after-write hazard between multiply and store. We first solve data dependency by reordering the micro-kernel at the C/C++ statement level. Then, we enhance the implementation of the micro-kernel at the assembly level through instruction reordering. As shown in Fig. 5, on the left is the assembly code of C/C++ statement reordering, on the right is the enhanced assembly code with instruction reordering. For example, the load instruction consists of two steps including movq and vmovupd (the part highlighted in red in Fig. 5). These two instructions have data dependencies on the registers because of both using rax and there is insufficient instruction distance

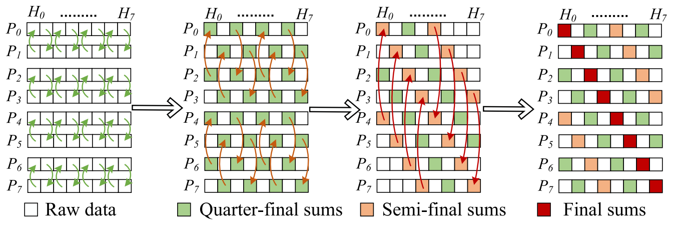


Fig. 6. Schematic of intra-node multi-processor Reduce optimization specially designed for the column rescaling of the SK algorithm.

to hide memory latency because the movq instruction has a latency of 3 [35]. So, we insert independent instructions between them to hide memory latency and improve the throughput of the instruction pipeline.

V. MPI-Oriented Optimization

In this section, we will describe our MPI Reduce and Allreduce design for the SK algorithm optimization on HPC systems. All the code of our design is implemented using MPI's standard primitives and can be used in any MPI-compliant library. In this paper, we use MPICH for experiments. The proposed optimization technique is based on the collective algorithms with hierarchical topology, with a focus on the bandwidth difference between intra-node communication and inter-node communication based on the key observation made in Section III and Fig. 2. We assign one processor within each node as the leader and the others as workers, where the leader receives partial sums from workers and then performs an inter-node Allreduce and finally broadcasts the global sums to workers [31], [36]. In what follows, we first introduce the optimization for intra-node Reduce algorithm, and then the optimization for inter-node Allreduce algorithm, with our optimization of overlapping communication and computation at last.

A. Optimization of Intra-Node Reduce

SOTA MPI Reduce implementations, such as Binary tree [37], have low efficiency for the SK algorithm. The main reason is that the overhead incurred on each processor is different, especially when the sum is reduced in the root processor that is heavily loaded while all the other processors are idle. To counter this serious problem of unbalanced load, we redesign a Reduce implementation for the SK algorithm that divides the local sum array into several parts. After intra-node Reduce, each worker keeps one part of the local final-sum and sends it to the leader in a non-blocking manner. Our intra-node Reduce implementation is based on the MPI standard primitives MPI_Send and MPI_Recv, consistent with the implementation of Reduce in the MPICH library. We did not use packing technology, because the data to be passed is almost continuous, and the extra overhead caused by packing outweighs the performance improvement brought by using it. Like the mainstream MPI message passing mechanism, we use additional buffers to receive data and save intermediate results.

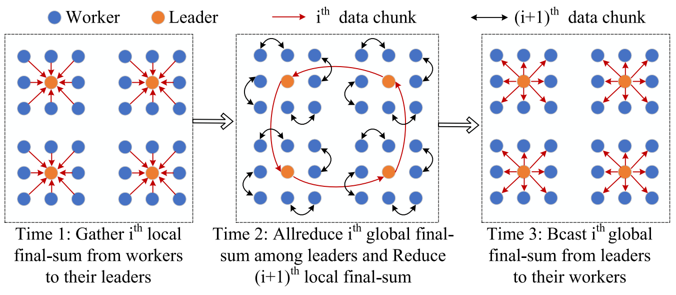

● Worker
● Leader
i th data chunk
(i+1) th data chunk
Time 1: Gather i th local final-sum from workers to their leaders
Time 2: Allreduce i th global final-sum among leaders and Reduce (i+1) th local final-sum
Time 3: Bcast i th global final-sum from leaders to their workers

Fig. 7. Enhanced hierarchical Ring Allreduce for inter-node communication [23], [31].

Fig. 6 shows the data movement of this multi-processor Reduce method with eight processors as an example. The Reduce is done in three rounds ($log 8 = 3$), where the quarter-final sums, semi-final sums and final sums are generated, respectively. In the first round, worker i exchanges data with worker $i+1$, $i = 0, 2, 4, 6$. In particular, worker i sends the jth part to and receives the $(j-1)$th part from worker $i+1$, $j = (i \mod 2) + 1, (i \mod 2) + 3, (i \mod 2) + 5, (i \mod 2) + 7$. In the second round, worker i exchanges data with worker $i+2$, $i = 0, 1, 4, 5$. In particular, worker i sends the jth part to and receives the $(j-2)$th part from worker $i+2$, $j = (i \mod 4) + 2, (i \mod 4) + 6$. In the third round, worker i exchanges data with worker $i+4$, $i = 0, 1, 2, 3$. In particular, worker i sends the jth part to and receives the $(j-4)$th part from worker $i+4$, $j = i+4$. By this way, the loads of all the processors are well balanced through all the three rounds.

B. Optimization of Inter-Node Allreduce

We implement Allreduce using the most popular Ring algorithm to generate the global final sum for the column rescaling. A disadvantage of Ring Allreduce is that it does not consider the hierarchy of nodes. In general, the bandwidth between nodes is much lower than that within nodes. Thus, hierarchical Ring Allreduce [31] is recently proposed.

Fig. 7 shows our optimization based on the hierarchical Ring. The main idea is to overlap time for intra-node Reduce and inter-node Allreduce. We divide the local sum array into several data chunks. As mentioned previously, there are three sequential steps to allreduce a data chunk. First, the workers perform intra-node Reduce on chunks of the local sum and send the local final sums to the leader (The red arrows of Time 1 in Fig. 7). Next, the leader performs inter-node Allreduce for the global final sums (The red arrows of Time 2 in Fig. 7). At last, a leader broadcasts the global final sums to its workers (The red arrows of Time 3 in Fig. 7). It is found that the sequential steps of different data chunks can be overlapped. For example, Time 2 in Fig. 7 shows that inter-node Allreduce of the ith data chunk (The red arrows) and intra-node Reduce of the $(i+1)$th data chunk (The black arrows) can be processed at the same time. Thus, we combine the idea of pipeline to the hierarchical Ring when we implement the Allreduce for the SK algorithm.

C. Optimization of Overlapping Communication and Computation

In a distributed environment with many nodes, it is reported that inter-node communication usually takes more time [38]. In other words, the workers have to be stalled when the leaders are doing Allreduce. We take advantage of this stall time for workers to modify the matrix within nodes. In our optimized Allreduce design of the SK algorithm, the computation of the matrix modification is added to the pipeline. Based on this, the workers can start to perform matrix modification once the global final sums of the previous data chunks are received. By this way, our design effectively combines inter-node communication, intra-node communication and intra-node calculation in a pipelined phase, further reducing the overall overhead of all processors.

It is noted that within this optimization, we need to make adjustments to the allocation of the submatrix. We add a preprocessing step before the third step of the parallel implementation of the SK algorithm (as described in Section III) starts. We distribute the submatrix of leader equally among all workers by rows. The advantage of this is that the computation work of modifying the submatrix of the leader will not slow down the progress of the entire program. Correspondingly, this increases the communication overhead and the computation overhead of the workers. It is believed that this optimization is worthwhile, and the experimental results (as shown in Section VI) also confirm this.

Moreover, based on our MPI-oriented optimization, we make some modifications to the typical parallel implementation of the SK algorithm. That is, we combine the third step and the fourth step into one function, so as to complete the communication task of Allreduce and the calculation task of modifying the submatrix at the same time.

VI. Experimental Evaluation

We present our experimental setup and evaluations on the CPU and MPI oriented optimizations in this section.

A. Experimental Setup

To assess the effectiveness of COFFEE, we compare two versions of it, the CPU-oriented optimization (Section IV), denoted as *COFFEE-CPU*, and the MPI-oriented optimization (Section V), denoted as *COFFEE-MPI*, against two existing implementations of the SK algorithm, one using the Ring Allreduce algorithm (*MPICH-Ring*) [30] and the other using SALaR (*MPICH-SALaR*) [23] in the MPICH environment. Among them, MPICH-Ring uses the Ring algorithm to perform the third step of the typical parallel implementation of the SK algorithm, while MPICH-SALaR uses the SALaR algorithm (as described in Section V-B) to perform the third step of the typical parallel implementation of the SK algorithm. COFFEE-MPI adds two optimizations on the basis of SALaR, namely optimization of intra-node Reduce in Section V-A and optimization of overlapping communication and computation in Section V-C. All three implementations (MPICH-Ring, MPICH-SALaR and

TABLE I
HARDWARE EVALUATION PLATFORM. THE AMD AND ARM V8 ARE ONLY USED FOR SINGLE NODE TESTING

	AMD	Intel	ARM V8
Number of Cores	8	12	10
Frequency	2.9 GHz	2.93 GHz	3.2 GHz
L1 cache	512 KB	64 KB	320 KB
L2 cache	4 MB	256 KB	12 MB
L3 cache	8 MB	12 MB	None
RAM	16 GB	48 GB	16 GB
Cache line size	64 B	64 B	64 B
Compiler	GCC9.4.0	GCC5.4.0	Clang12.0.0

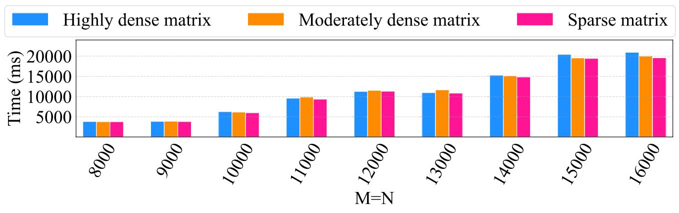


Fig. 8. Performance of typical SK algorithm implementation on matrices of different density under serial execution.

COFFEE-MPI) combine our CPU-oriented optimization. We implement COFFEE with the open-source MPICH-3.4 library. All the code was compiled with -O2 flag. Moreover, because column rescaling redesign can largely reduce cache miss rate, we also present it as COFFEE-C2R to clearly evaluate the performance of COFFEE on column rescaling. For the parameters, we set $mr = 4$, $nr = 4$, which is also the default setting of OpenBlas [39]. Then we set $mc = 8$, $nc = 1000$. For the experimental environment, we evaluate COFFEE on single node and multi-node clusters, respectively. Single node experimental results are obtained using three representative platforms: AMD Ryzen7 4800H on Lenovo R9000X, Intel Xeon Westmere EP on Tianhe-1 supercomputer, and ARM V8 on Mac M1 pro. Multi-node results are obtained using Intel Xeon Westmere EP platform on Tianhe-1 supercomputer. Table I lists the main configuration parameters of the above hardware platforms.

The workload characteristics of the SK algorithm are likely application dependent, which we think can be largely reflected in the density of the matrices representing these applications that the SK algorithm operates on. Since COFFEE is designed to optimize the time spent on each iteration of the SK algorithm, we then evaluate the average time spent on each iteration for typical SK algorithm with matrices of different densities, namely, highly dense matrix (Non-zero elements account for 95%), moderately dense matrix (Non-zero elements account for 50%), sparse matrix (Non-zero elements account for 5%), each matrix is generated by a random number generation function in C language. As indicated in Fig. 8, the average running time of an iteration is insensitive to the density of a matrix, suggesting that our algorithm is likely applicable to any matrix stored in the array format. Note that some AI applications using the SK algorithm are based on sparse matrices and others are based on dense matrices. A sparse matrix will be divided into zero submatrices and non-zero submatrices. Then, the compute kernels will skip computations of zero submatrices and only focus on non-zero submatrices [40]. Non-zero submatrices are usually stored in the array format in the main memory which COFFEE can handle with them.[5] Note that in this paper we are not concerned with the engineering implementation of any specific end-to-end applications, instead, we focus on the characteristics of the SK algorithm itself so that our proposed solution is suitable for all kinds of applications. Therefore, we only focus on a representative application, Cooperative Bayesian [15], which is a tool to analyze the consistency, rate of convergence and stability of a cooperative Bayesian model in human-human and human-machine cooperation learning. It is based on a randomly generated dense matrix, so our experiments use the same setting. Note that the number of iterations between different methods in the following experiments is the same, so the running time difference is only related to that of a single iteration.

B. CPU-Oriented Optimization

In this section, we evaluate the effectiveness of CPU-oriented optimization of COFFEE on a single node.

1) SK Algorithm Performance on a Single Core: In this experiment, we evaluate the average running time of column rescaling on the three platforms individually to reflect the direct effect of single node optimization. Fig. 9 shows the performance comparison of COFFEE-CPU, COFFEE-C2R and the typical SK algorithm implementation on a single core. We observe that as the matrix size increases, the column rescaling time of the typical SK algorithm tends to rise much higher than that of COFFEE-C2R and COFFEE-CPU. With data blocking and micro-kernel design, COFFEE-CPU has a further performance improvement over COFFEE-C2R. For example, on the Intel platform, when $M = N = 16,000$, the performance of COFFEE-CPU is 9.3X higher than that of the typical implementation, 2.1X higher than COFFEE-C2R.

2) SK Algorithm Performance on Multiple Cores: In this experiment, we show the effect of COFFEE-C2R and COFFEE-CPU optimizations for the SK algorithm on a single node with multiple cores on the three platforms using MPICH-Ring. As shown in Fig. 10, the performance of COFFEE-C2R and COFFEE-CPU with MPICH-Ring is always better than that of the raw MPICH-Ring (In this subsection, raw MPICH-Ring does not contain our CPU-oriented optimization) on all the three platforms. As the matrix size increases, COFFEE-C2R and COFFEE-CPU show more pronounced advantages. For example, when $M = N$, $P = 2$, the performance improvement of COFFEE-CPU over MPICH-Ring increases from about 1.5X to 2.4X on the Intel platform. When the number of processors increases, the size of the matrix assigned to each processor becomes smaller, and the performance improvement is less obvious. Nevertheless, when $M = N = 20,000$, $P = 8$, COFFEE-CPU still outperforms MPICH-Ring by 1.27X on ARM V8. In general, the performance of COFFEE-CPU is 1.14 to 7.5X, with an average of 2.0X, higher than the raw MPICH-Ring in a single node MPI environment.

[5][Online]. Available: https://developer.nvidia.com/blog/accelerating-matrix-multiplication-with-block-sparse-format-and-nvidia-tensor-cores/

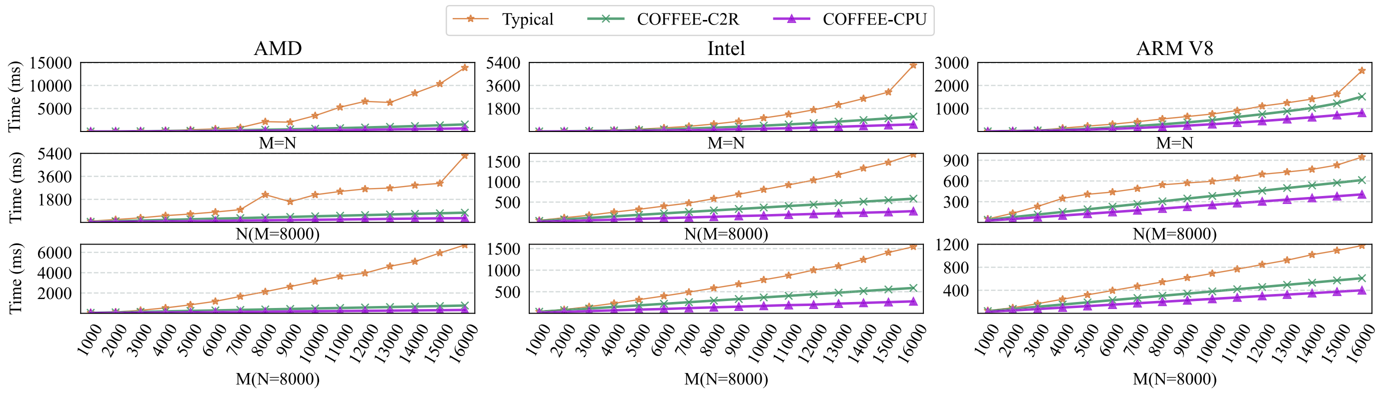


Fig. 9. Performance of column rescaling matrices under serial execution on three platforms.

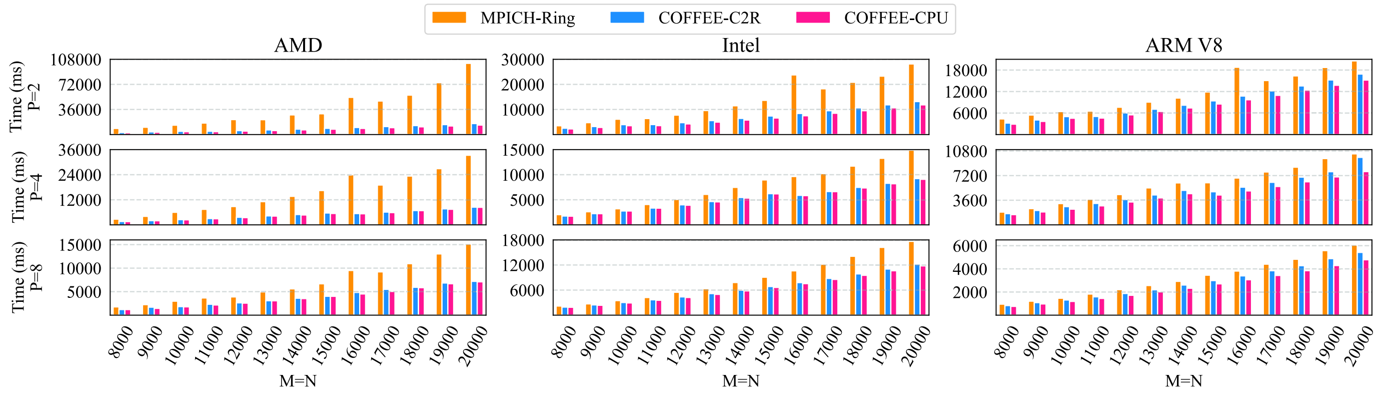


Fig. 10. Performance of MPI multi-core parallel SK algorithm implementations on single node on three platforms. In this figure, MPICH-Ring does not contain our CPU-oriented optimization.

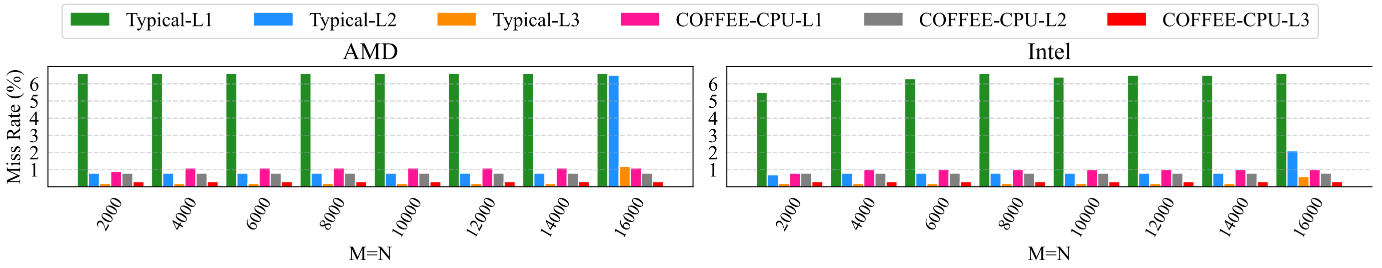


Fig. 11. Cache miss rate for column rescaling. We are not able to demonstrate the ARM V8 results because macOS does not support an appropriate cache miss analysis tool.

3) Cache Locality: In this experiment, we use Valgrind [33] to compare column rescaling cache miss rate with and without COFFEE-CPU optimization on two platforms. As shown in Fig. 11, the column rescaling optimized by COFFEE-CPU has improved greatly over the typical SK algorithm implementation in L1 cache locality on all matrix sizes. For example, on the Intel platform, when $M = N = 16,000$, the L1 cache miss rate drops from 6.6% to 1.0%. In general, COFFEE-CPU improves the cache locality of column rescaling by adding COFFEE-C2R, data blocking, and micro-kernel optimizations.

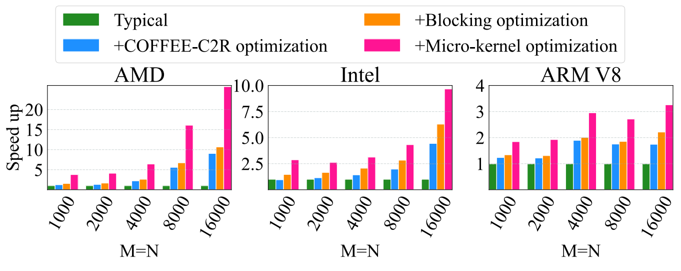


Fig. 12. Breakdown of optimizations for column rescaling on three platforms.

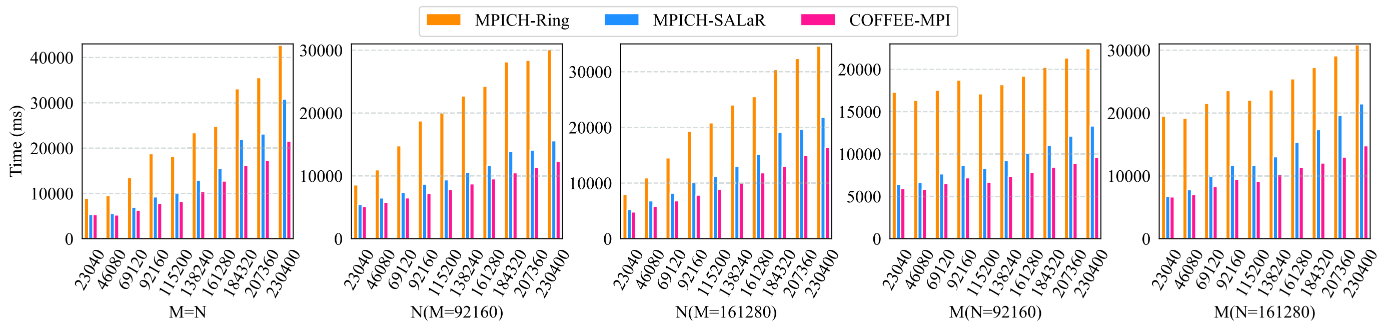


Fig. 13. Performance of SK algorithm implementations on Tianhe-1 supercomputer with a distributed environment (64 nodes with 9 processors per node, bandwidth between nodes is limited to 1 Gb/s).

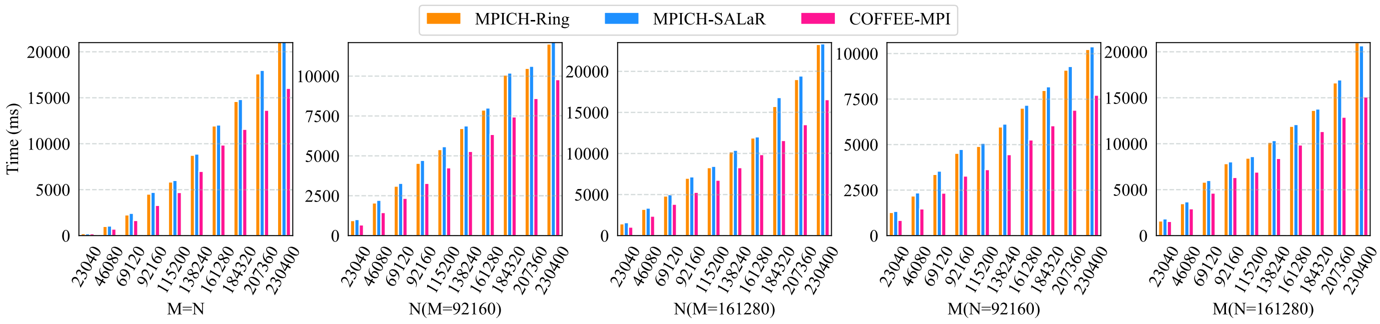


Fig. 14. Performance of SK algorithm implementations on Tianhe-1 supercomputer with a centralized environment (64 nodes with 9 processors per node, without limiting the bandwidth between nodes).

4) Breakdown of COFFEE-CPU Optimizations: In this experiment, we evaluate the performance impact of COFFEE-CPU on the three platforms. Our experiments evaluate the speedup of the three components of COFFEE-CPU for column rescaling over the typical SK algorithm implementation. Experiments are implemented in a single node serial execution environment. It can be seen from Fig. 12 that the reduction in running time is significant after using COFFEE-C2R, because it makes full use of cache locality and reduces memory access latency. For example, on the Intel platform, when $M = N = 16,000$, after using COFFEE-C2R, the running time is reduced from about 5,181 ms of the typical implementation to 1,167 ms with a speedup of 4.4X. On this basis, the data blocking and micro-kernel design further improve the performance. The speedup is improved to 6.2X with data blocking plus COFFEE-C2R and 9.8X with all of them. We observe that the speedup of AMD is much higher than Intel and ARM V8. This is related to the different cache mechanisms of the processors. For Intel and ARM V8, L1 cache retrieves data from the L2 cache, meaning that data stored in the L1 cache is a part of L2 cache. For AMD, fills from the DRAM layer go directly into the appropriate L1 cache [41]. The L1 cache of the Intel and ARM V8 processors has a higher chance of retrieving data from the L2 cache, because once a cache line is loaded from the main memory, the relevant cache lines will be also prefetched to L2 cache. As shown in Fig. 11, it is verified that AMD's L2 cache miss rate of the typical implementation is much higher, resulting in poor performance of the SK algorithm. COFFEE can significantly reduce L1 and L2 cache miss rates of the AMD platform, which leads to a higher speedup. To sum up, the SK algorithm running on the AMD platform with GCC compiler has achieved the greatest improvement through our optimization. The performance of the typical implementation of the SK algorithm is the best on the ARM platform with Clang compiler among all other platforms, but our optimization still achieves a speedup of 3.3X when $M = N = 16,000$.

C. MPI-Oriented Optimization

We evaluate COFFEE-MPI's performance of the SK algorithm implementation in two environments on Tianhe-1 supercomputer, namely, a distributed environment, where the bandwidth between nodes is limited to 1 Gb/s, and a centralized environment, where there is no bandwidth limit between nodes.

1) Distributed Cluster Environment: Fig. 13 compares COFFEE-MPI against the two SOTAs. COFFEE-MPI outperforms both SOTAs in all cases and its performance advantages are more pronounced when the matrix is large. For example, when $M = N = 230,400$, the performance of COFFEE-MPI is 2.0X higher than MPICH-Ring and 1.4X higher than MPICH-SALaR. We found that COFFEE-MPI has better scalability than both SOTAs. For example, when the matrix is expanded from $M = N = 23,040$ to $M = N = 230,400$, the running time of MPICH-Ring increases by 4.8X, MPICH-SALaR increases by 5.8X, and COFFEE-MPI increases by 4.0X. The

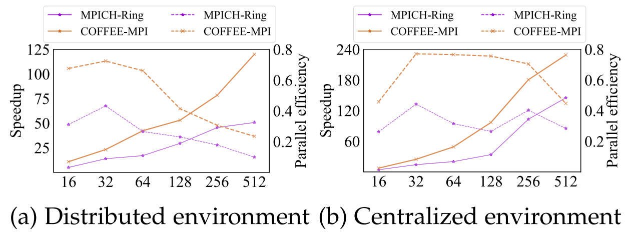


Fig. 15. Scalability of SK algorithm implementations on Tianhe-1 supercomputer. The solid lines refer to speedup, and the dotted lines refer to parallel efficiency.

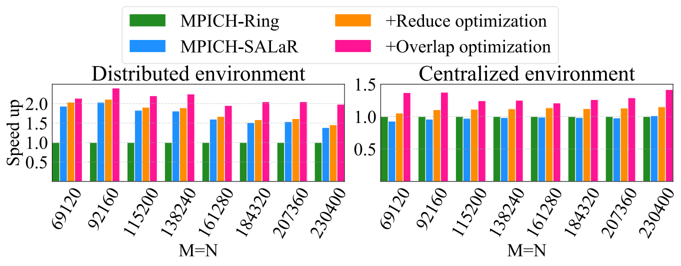


Fig. 16. Breakdown of MPI-oriented optimizations on Tianhe-1 supercomputer.

performance improvement is not as obvious as that in the single node experiments. The reason is that as the number of nodes increases, the size of the matrix assigned to each processor becomes smaller leading to shorter calculation time, and the time of Allreduce becomes dominant. We found that compared to COFFEE-MPI, MPICH-Ring is more sensitive to change in column length than row length. For example, with matrix size $M = 23,040$, $N = 230,400$, MPICH-Ring takes 2.4X longer time than $M = 230,400$, $N = 23,040$, in contrast to COFFEE-MPI's 1.4X. This is because the Allreduce time is only related to the column length. In general, COFFEE-MPI provides a performance improvement of up to 2.9X, with an average of 1.8X, over MPICH-Ring and MPICH-SALaR in a distributed cluster environment.

2) Centralized Cluster Environment: As shown in Fig. 14, COFFEE-MPI outperforms both SOTAs in all cases and the running time gap is gradually increasing. For example, when $M = N = 230,400$, the performance of COFFEE-MPI is 1.4X higher than MPICH-Ring and 1.4X higher than MPICH-SALaR. We found that MPICH-Ring performs slightly better than MPICH-SALaR. It shows that raw Ring Allreduce has relatively better applicability for small and medium-sized data. In general, COFFEE-MPI provides a performance improvement of up to 1.5X, with an average of 1.3X, over MPICH-Ring and MPICH-SALaR in a centralized cluster environment.

3) Scalability and Parallel Efficiency: Fig. 15 shows the scalability and the parallel efficiency of the multi-node implementation of the SK algorithm. Here we choose different numbers of processors to process the same matrix of size $61,440 \times 61,440$. As the number of processors increases, COFFEE exhibits the best performance, the best scalability, and the best parallel efficiency. The experimental results show that the acceleration can reach up to 120X (23.4% parallel efficiency) in the distributed cluster environment and 229X (43.0% parallel efficiency) in the centralized cluster environment.

4) Breakdown of COFFEE-MPI Optimizations: In this experiment, we evaluate the performance impact of COFFEE-MPI on Tianhe-1 supercomputer. On the basis of MPICH-SALaR, we experimentally evaluate two parts of MPI-oriented optimization. As shown in Fig. 16, the two parts of the optimization are referred to as Reduce optimization in Section V-A and overlap optimization in Section V-C, the experimental results are normalized by MPICH-Ring implementation. It is found that after using our Reduce optimization, the speedup increased by an average of 5.0% in distributed environment and an average of 14.5% in centralized environment over MPICH-SALaR. On this basis, our overlap optimization further improves performance significantly. The speedup can reach up to 2.4X in distributed environment and up to 1.4X in centralized environment over MPICH-Ring with all of our MPI-oriented optimizations.

VII. Conclusion and Future Work

The SK algorithm is showing increasing importance in machine learning and other fields. In this paper, we proposed and implemented a cross-layer optimization design for both computation and communication of the SK algorithm implementation, called COFFEE. Unlike most of the existing works that focus on speeding up the convergence by reducing the number of rescaling iterations, COFFEE focuses on speeding up the convergence by shortening each rescaling iteration. We have conducted an in-depth study of the issues affecting performance in the implementation of the SK algorithm. It is found that the column rescaling incurs a high cache miss rate and low parallel efficiency. We use cross-layer optimizations such as column rescaling redesign, data blocking and micro-kernel design to speed up the column rescaling. We also optimize MPI Reduce and Allreduce to enhance parallel efficiency based on the SK algorithm characteristic. Finally, we verify the effectiveness of COFFEE on Tianhe-1 supercomputer. In the future, we plan to further explore and exploit the correlation between the row rescaling and column rescaling. In addition, we plan to combine GPUs to make full use of the heterogeneous parallel computing architecture to further improve the performance of COFFEE. Finally, we plan to study the performance of COFFEE on sparse matrices of the SK algorithm in which data are not stored in the array format.

Acknowledgments

The authors sincerely acknowledge the computing resources provided on National Supercomputing Center in Changsha of China.

References

[1] R. Sinkhorn, “A relationship between arbitrary positive matrices and doubly stochastic matrices,” *Ann. Math. Statist.*, vol. 35, no. 2, pp. 876–879, 1964.

[2] R. Flamary et al., “POT: Python optimal transport,” *J. Mach. Learn. Res.*, vol. 22, no. 78, pp. 1–8, 2021.

[3] M. Cuturi, “Sinkhorn distances: Lightspeed computation of optimal transportation distances,” 2013, *arXiv:1306.0895*.

[4] G. Mena, D. Belanger, G. Munoz, and J. Snoek, “Sinkhorn networks: Using optimal transport techniques to learn permutations,” in *Proc. NIPS Workshop Optimal Transport Mach. Learn.*, 2017, vol. 3, pp. 1–10.

[5] R. Sinkhorn and P. Knopp, “Concerning nonnegative matrices and doubly stochastic matrices,” *Pacific J. Math.*, vol. 21, no. 2, pp. 343–348, 1967.

[6] G. Peyré et al., “Computational optimal transport: With applications to data science,” *Found. Trends Mach. Learn.*, vol. 11, no. 5/6, pp. 355–607, 2019.

[7] N. Courty, R. Flamary, D. Tuia, and A. Rakotomamonjy, “Optimal transport for domain adaptation,” *IEEE Trans. Pattern Anal. Mach. Intell*, vol. 39, no. 9, pp. 1853–1865, Sep. 2017.

[8] K. Pham, K. Le, N. Ho, T. Pham, and H. Bui, “On unbalanced optimal transport: An analysis of sinkhorn algorithm,” in *Proc. Int. Conf. Mach. Learn.*, PMLR, 2020, pp. 7673–7682.

[9] J. Feydy, T. Séjourné, F.-X. Vialard, S.-I. Amari, A. Trouvé, and G. Peyré, “Interpolating between optimal transport and MMD using sinkhorn divergences,” in *Proc. 22nd Int. Conf. Artif. Intell. Statist.*, PMLR, 2019, pp. 2681–2690.

[10] P. A. Knight, “The sinkhorn–knopp algorithm: Convergence and applications,” *SIAM J. Matrix Anal. Appl.*, vol. 30, no. 1, pp. 261–275, 2008.

[11] G. Pai, J. Ren, S. Melzi, P. Wonka, and M. Ovsjanikov, “Fast sinkhorn filters: Using matrix scaling for non-rigid shape correspondence with functional maps,” in *Proc. IEEE/CVF Conf. Comput. Vis. Pattern Recognit.*, 2021, pp. 384–393.

[12] L. Métivier, R. Brossier, Q. Mérigot, E. Oudet, and J. Virieux, “Measuring the misfit between seismograms using an optimal transport distance: Application to full waveform inversion,” *Geophysical Suppl. Monthly Notices Roy. Astronomical Soc.*, vol. 205, no. 1, pp. 345–377, 2016.

[13] T. Shen, H. Zhang, D. Yuan, J. Xiao, and Y. Yang, “Foresight of graph reinforcement learning latent permutations learnt by gumbel sinkhorn network,” 2021, *arXiv:2110.12144*.

[14] G. Huang, C. Guo, M. J. Kusner, Y. Sun, F. Sha, and K. Q. Weinberger, “Supervised word mover’s distance,” in *Proc. Adv. Neural Inf. Process. Syst.*, 2016, pp. 4869–4877.

[15] J. Wang, P. Wang, and P. Shafto, “Sequential cooperative Bayesian inference,” in *Proc. Int. Conf. Mach. Learn.*, PMLR, 2020, pp. 10039–10049.

[16] P. Dvurechensky, A. Gasnikov, and A. Kroshnin, “Computational optimal transport: Complexity by accelerated gradient descent is better than by sinkhorn’s algorithm,” in *Proc. Int. Conf. Mach. Learn.*, PMLR, 2018, pp. 1367–1376.

[17] L. Chizat, P. Roussillon, F. Léger, F.-X. Vialard, and G. Peyré, “Faster wasserstein distance estimation with the sinkhorn divergence,” *Adv. Neural Inf. Process. Syst.*, vol. 33, pp. 2257–2269, 2020.

[18] J. J. Tithi and F. Petrini, “A new parallel algorithm for sinkhorn word-movers distance and its performance on PIUMA and xeon CPU,” 2021, *arXiv:2107.06433*.

[19] J. A. Kahle, J. Moreno, and D. Dreps, “2.1 summit and sierra: Designing AI/HPC supercomputers,” in *Proc. IEEE Int. Solid-State Circuits Conf.*, 2019, pp. 42–43.

[20] G. Frison, D. Kouzoupis, T. Sartor, A. Zanelli, and M. Diehl, “BLASFEO: Basic linear algebra subroutines for embedded optimization,” *ACM Trans. Math. Softw.*, vol. 44, no. 4, pp. 1–30, 2018.

[21] W. Yang, J. Fang, D. Dong, X. Su, and Z. Wang, “Libshalom: Optimizing small and irregular-shaped matrix multiplications on ARMv8 multi-cores,” in *Proc. Int. Conf. High Perform. Comput., Netw., Storage Anal.*, 2021, pp. 1–14.

[22] Y. Liu and X.-H. Sun, “CaL: Extending data locality to consider concurrency for performance optimization,” *IEEE Trans. Big Data*, vol. 4, no. 2, pp. 273–288, Jun. 2018.

[23] M. Bayatpour, J. M. Hashmi, S. Chakraborty, H. Subramoni, P. Kousha, and D. K. Panda, “SALaR: Scalable and adaptive designs for large message reduction collectives,” in *Proc. IEEE Int. Conf. Cluster Comput.*, 2018, pp. 12–23.

[24] J. Altschuler, J. Niles-Weed, and P. Rigollet, “Near-linear time approximation algorithms for optimal transport via sinkhorn iteration,” in *Proc. Adv. Neural Inf. Process. Syst.*, 2017, pp. 1961–1971.

[25] A. Schönhage and V. Strassen, “Schnelle multiplikation grosser zahlen,” *Computing*, vol. 7, no. 3, pp. 281–292, 1971.

[26] V. V. Williams, “Multiplying matrices faster than coppersmith-winograd,” in *Proc. 44th Annu. ACM Symp. Theory Comput.*, 2012, pp. 887–898.

[27] T. T. Nguyen, M. Wahib, and R. Takano, “Hierarchical distributed-memory multi-leader MPI-allreduce for deep learning workloads,” in *Proc. IEEE 6th Int. Symp. Comput. Netw. Workshops*, 2018, pp. 216–222.

[28] G. Wang, S. Venkataraman, A. Phanishayee, N. Devanur, J. Thelin, and I. Stoica, “Blink: Fast and generic collectives for distributed ML,” *Proc. Mach. Learn. Syst.*, vol. 2, pp. 172–186, 2020.

[29] T. T. Nguyen, M. Wahib, and R. Takano, “Topology-aware sparse allreduce for large-scale deep learning,” in *Proc. IEEE 38th Int. Perform. Comput. Commun. Conf.*, 2019, pp. 1–8.

[30] U. von Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, 2017. [Online]. Available: https://github.com/baidu-research/baidu-allreduce

[31] Y. Ueno and R. Yokota, “Exhaustive study of hierarchical allreduce patterns for large messages between GPUs,” in *Proc. IEEE/ACM 19th Int. Symp. Cluster, Cloud Grid Comput.*, 2019, pp. 430–439.

[32] X. Yang et al., “TH-1: China’s first petaflop supercomputer,” *Front. Comput. Sci. China*, vol. 4, pp. 445–455, 2010.

[33] N. Nethercote and J. Seward, “Valgrind: A framework for heavyweight dynamic binary instrumentation,” *ACM Sigplan Notices*, vol. 42, no. 6, pp. 89–100, 2007.

[34] H. Jeong, S. Kim, W. Lee, and S.-H. Myung, “Performance of SSE and AVX instruction sets,” 2012, *arXiv:1211.0820*.

[35] Intel, “Intel$^{®}$ intrinsics guide,” 2018. Accessed: May 25, 2023. [Online]. Available: https://software.intel.com/sites/landingpage/IntrinsicsGuide/

[36] M. Bayatpour, S. Chakraborty, H. Subramoni, X. Lu, and D. K. Panda, “Scalable reduction collectives with data partitioning-based multi-leader design,” in *Proc. Int. Conf. High Perform. Comput., Netw., Storage Anal.*, 2017, pp. 1–11.

[37] K. Hasanov and A. Lastovetsky, “Hierarchical redesign of classic MPI reduction algorithms,” *J. Supercomput.*, vol. 73, no. 2, pp. 713–725, 2017.

[38] K.-I. Sato, “Implication of inter-node and intra-node contention in creating large throughput photonic networks,” in *Proc. Int. Conf. Opt. Netw. Des. Model.*, 2014, pp. 144–149.

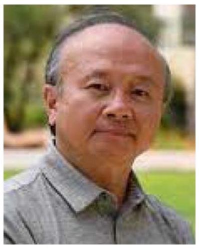
[39] W. Yang, J. Fang, and D. Dong, “Characterizing small-scale matrix multiplications on ARMv8-based many-core architectures,” in *Proc. IEEE Int. Parallel Distrib. Process. Symp.*, 2021, pp. 101–110.

[40] S. Gray, A. Radford, and D. P. Kingma, “GPU kernels for block-sparse weights,” 2017, *arXiv: 1711.09224*.

[41] P. Conway, N. Kalyanasundharam, G. Donley, K. Lepak, and B. Hughes, “Cache hierarchy and memory subsystem of the AMD opteron processor,” *IEEE Micro*, vol. 30, no. 2, pp. 16–29, Mar./Apr. 2010.

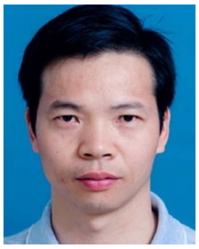
Chengyu Sun received the BS degree form the Ocean University of China, Qingdao, China, in 2021. Currently, he is working toward the MS degree with the College of Computer Science and Electronic Engineering, Hunan University, Changsha, China. His research interests include the parallel computing and distributed computing system.

Huizhang Luo (Member, IEEE) received the BS and PhD degrees in computer science from Chongqing University, China, in 2012 and 2017, respectively. He is currently an Associate Professor with Hunan University. Before that, he was a postdoctoral researcher with the Department of Electrical and Computer Engineering, NJIT. His research interests include memory systems, high-performance computing, and nonvolatile memory.



Hong Jiang (Fellow, IEEE) received the BSc degree in computer engineering from the Huazhong University of Science and Technology, Wuhan, China; the MASc degree in computer engineering from the University of Toronto, Toronto, Canada, and the PhD degree in computer science from the Texas A&M University, College Station, Texas, USA. He is currently chair and Wendell H. Nedderman Endowed professor of Computer Science and Engineering Department, University of Texas at Arlington. Prior to joining UTA, he served as a program director with National Science Foundation (2013.1-2015.8) and he was with University of Nebraska-Lincoln since 1991, where he was Willa Cather professor of computer science and engineering. His present research interests include computer architecture, computer storage systems and parallel I/O, high-performance computing, Big Data computing, cloud computing, performance evaluation. He is a Member of ACM.

Jeff Zhang received the PhD degree from New York University. He is an Assistant Professor with the School of Electrical, Computer and Energy Engineering, ASU. From 2020-2022, he was a postdoctoral fellow with Harvard University. His general research interests are in deep learning, computer architecture, embedded systems, and EDA, with particular emphasis on energy-efficient and fault-tolerant design for AI/ML systems and hardware accelerators.



Kenli Li (Senior Member, IEEE) received the PhD degree in computer science from the Huazhong University of Science and Technology, China, in 2003. He is currently a Cheung Kong professor of computer science and technology with Hunan University, the dean of the College of Computer Science and Electronic Engineering, Hunan University. His major research interests include parallel and distributed processing, high-performance computing, and Big Data management. He has published more than 250 research papers in international conferences and journals such as *IEEE Transactions on Computers*, *IEEE Transactions on Parallel and Distributed Systems*, *IEEE Transactions on Cloud Computing*, ICPP, ICDCS, etc. He serves on the editorial board of the *IEEE Transactions on Computers*. He is an outstanding member of the CCF.