An Efficient Algorithm for Large-Scale Matrix Transposition

Jinwoo Suh, Santosh Narayanan
and Viktor K. Prasanna

CENG 99-08

Department of Electrical Engineering - Systems
University of Southern California
Los Angeles, California 90089-2562
(213-740-4481)
December 1999

CENG Technical Report No. 99-08
An Efficient Algorithm for Large-Scale Matrix Transposition*

Jinwoo Suh, Santosh Narayanan and Viktor K. Prasanna

Department of EE-Systems, EEB-200C
University of Southern California
Los Angeles, CA 90089-2562
prasanna@*usc.edu*
http://ceng.usc.edu/~prasanna

* Work funded wholly by the DoD High Performance Modernization Program CEWES Major Shared Resource Center through Programming Environment and Training (PET) supported by Contract Number DAHC 94-96-C0002, and Subcontract Number NRC-CR-98-0002.

Abstract

Efficient transposition of large-scale matrices has been widely studied. These efforts have focused on reducing the number of I/O operations. However, in the state-of-the-art architectures, data transfer time and index computation time are also significant components of the overall time. In this paper, we propose an algorithm that considers all these costs and reduces the overall execution time.

Our algorithm reduces the number of I/O operations significantly by using two techniques: (1) writing the data onto disk in predefined patterns and (2) balancing the numbers of read and write operations to disk. The reduction of the number of I/O operations has a significant impact on overall time for I/O, which is measured by elapsed wallclock time. The reason for this is that the startup time for an I/O operation is several orders of magnitude than the time for transferring an actual data byte, in state-of-the-art disk systems. The idea behind balancing the I/O operations is to reduce the number of write operations at the expense of an increased number of read operations, so that the total number of I/O operations is reduced compared with the state-of-the-art. The index computation time, which is an expensive operation involving two divisions and a multiplication, is eliminated by partitioning the memory into two buffers. The expensive in-processor permutation is replaced by data collection operations. Our algorithm is analyzed using the well-known Linear Model and the Parallel Disk Model.

The experimental results on a Sun Enterprise and a DEC Alpha show that our algorithm reduces the execution time by about 50%, compared with the best known algorithms in the literature.

1 Introduction

Matrix transpose is a key primitive in a wide variety of data-intensive applications such as scientific computations, media processing, automatic target recognition systems, signal processing among others [5, 8, 12, 17, 18, 20, 21, 23, 24]. In many data-intensive applications, the typical data size that is stored on the disk is of the order of TeraBytes. High-performance computing platforms employ RAID [4] systems, which have very high storage capacities.

Two models have been widely used in the literature to abstract the behavior of disk systems: the Parallel Disk Model (PDM)[25] and the Linear Model (LM)[15]. The PDM is well suited to model I/O systems such as the RAID [4]. In PDM, the data access cost is represented as $\lceil m/(DB) \rceil \times T_b$, where m is the data size, D is the number of disks, and T_b is time to transfer a block of data (B) between memory and disk. In the Linear Model, the cost is represented as $T_s + m\tau$, where the T_s is startup time, m is data size, and τ is data transfer time per unit data.

In this paper, we propose an efficient algorithm for transposing large-scale matrices (out-of-core matrix transpose). An input matrix of size $N \times N$ initially resides on the disk, $N = \prod_{s=0}^{t-1} r_s$, where r_s is a positive integer. The matrix is to be transposed and stored in another array. The size of the available main memory, M, is assumed to be smaller than the matrix size.

Several researchers have studied the out-of-core matrix transpose problem. A straightforward algorithm performs matrix transpose using $O(N^{3/2})$ I/O operations when $M = O(N)$. Eklundh [13] proposed an algorithm that has $O(N \lg N)$ I/O complexity assuming $B = N$. Ari et al. [2] modified the algorithm in [13] to reduce the number of I/O operations at the expense of increased number of stages (passes). Floyd [14] derived the upper and lower bounds on the number of I/O operations when $M = 2B$. Aggarwal et al. [1] derived a lower bound on the number of I/O operations for the general case using PDM. Kaushik et al. [15] reduced the number of I/O operations by a factor of 25% by combining two read operations compared with the algorithm in [13].

All these efforts focus on reducing the number of I/O operations only. However, the main costs in the state-of-the-art architectures consist of not only the time for I/O but also the in-processor data transfer time and index computation time. Figure 1 depicts the breakdown of the various costs in a typical transpose operation.

Our out-of-core matrix transpose algorithm reduces the total execution time by reducing both the number of I/O operations and the index computation time. The reduction in the number of I/O operations is achieved by using efficient data layout on disk and balancing the number of read and write operations. We analyze the complexity of our algorithm using the well-known Parallel Disk Model (PDM) and the Linear Model (LM). A comparison of the algorithms with respect to the number of I/O operations is shown in Table 1.

Table 1: Comparison of the number of I/O operations (D=1)

Algorithm	Linear Model	Parallel Disk Model	
		$B \leq \frac{M}{r_s}$	$B > \frac{M}{r_s}$
Aggarwal et al. [1]	-	$\frac{2N^2}{B} \lg_{M/B} \min(\frac{N^2}{B}, B)$	-
Kaushik et al. [15]	$\frac{N^2}{M} \sum_{s=0}^{t-1} (1 + r_s)$	$\frac{2N^2 t}{B}$	$\frac{N^2}{B} \sum_{s=0}^{t-1} (\frac{Br_s}{M} + 1)$
This Paper	$\frac{N^2}{M} \sum_{s=0}^{t-1} \min(r_s, 2(\sqrt{2r_s} + 1))$	$\frac{2N^2}{B} \lg_{M/B} \min(\frac{N^2}{B}, B)$	$\frac{2N^2}{B} \sum_{s=0}^{t-1} (\sqrt{\frac{Br_s}{M}} + \frac{B}{M})$

To eliminate the index computation cost, our algorithm partitions the available memory into two buffers (read and write buffers). The expensive in-processor permutation is replaced by collect operations. The write operations and collect operations are scheduled efficiently to reduce the overall time. The size of each buffer is determined by the available memory size and the factorization of N. By using these techniques, the index computation is replaced by inexpensive do-loops (see Section 4.2.2).

We implemented the algorithm on a single node of a SGI/Cray T3E based on DEC Alpha 21164 at the San Diego Supercomputing Center (NPACI/SDSC) and a Sun Enterprise 4000 based on UltraSPARC at the University of Southern California. The experiments were carried out for available main memory sizes

Table 2: Example numbers of I/O operations ($D = 1, N = 2^{20} = 128 \times 128 \times 64, t = 3, M = 64$ MBytes, and each data is 8 Bytes)

Algorithm	Linear Model	Parallel Disk Model	
		$B = 512$ KBytes	$B = 8$ MBytes
Aggarwal et al. [1]	-	96×2^{20}	-
Kaushik et al. [15]	40×2^{20}	96×2^{20}	43×2^{20}
This Paper	12×2^{20}	96×2^{20}	24×2^{20}

ranging from 16 MB to 64 MB and data sizes ranging from 512 MB to 2 GB. The results show that our algorithm reduces the execution time by up to 50%.

The organization of the rest of this paper is as follows. In Section 2, two well-known disk models are briefly described. In Section 3, previous algorithms are discussed. Our algorithm is described in detail in Section 4. Experimental results as well as comparisons with previous algorithms are presented in Section 5. Section 6 discusses a further extension of our algorithm to perform Bit-Matrix-Multiply/Complement (BMMC) and Section 7 concludes the paper.

2 Disk Models

State-of-the-art disk systems employ sophisticated hardware and perform several optimizations to reduce the I/O time. For example, many of these systems employ a disk buffer, a library buffer, and a controller, and perform access reordering. Each of the above system features needs several parameters to describe its behavior and such a model will be too complex to be useful.

Two models of disk systems that capture the key characteristics of such systems have been widely used in the literature. One of them is the Parallel Disk Model (PDM) [25] (see Figure 3). It models the low level behavior of disk systems using several parameters: block size (B) which is the size of data that is transferred between disk and memory in one I/O operation, number of disks (D), memory size (M), number of processors (P), and amount of data transferred (m). This model has been used to study RAID systems. The total time for data transfer between disk and memory is proportional to the number of blocks transferred and is inversely proportional to the number of disks on which the data resides. Thus, the cost can be represented as $\lceil m/(DB) \rceil \times T_b$, where T_b is the time to transfer a block of data between memory and disk.

In another model [15], two costs are considered: startup time and data transfer time. The startup time is a fixed time for setting up the data transfer between memory and disk. The rest of the cost is proportional to the amount of data transferred. Thus, it can be represented as $T_s + m\tau$, where T_s is the startup time, m

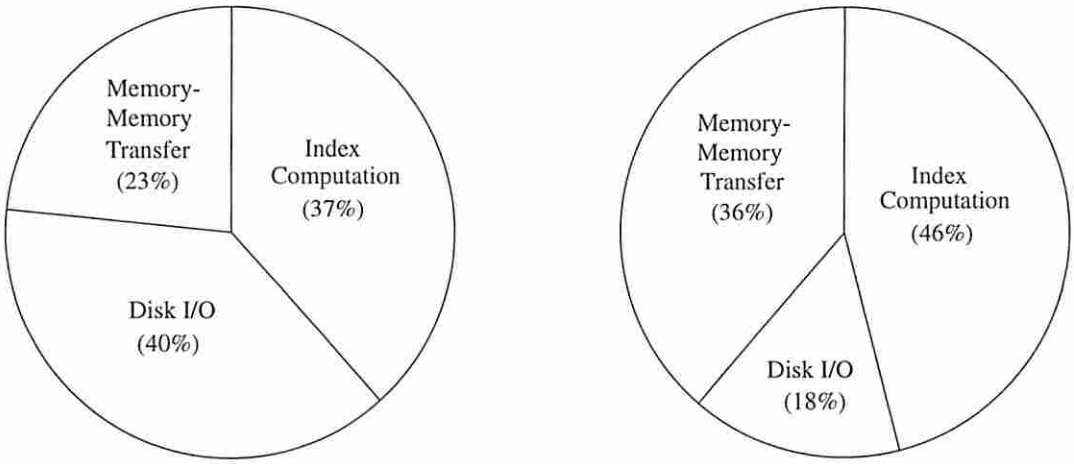


Figure 1: Breakdown of total execution time for matrix transpose (a) On SGI/Cray T3E (DEC Alpha), M = 16 MBytes, data size = 128 Mbytes (b) On Sun Enterprise, M = 64 MBytes, data size = 2 Gbytes

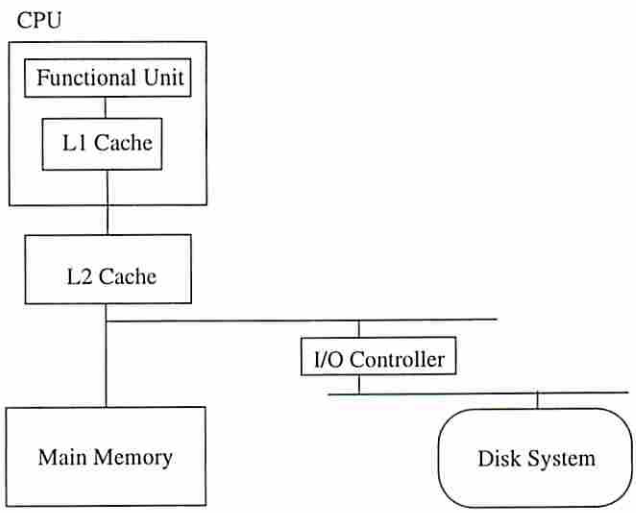


Figure 2: Typical Disk system

is the data size, and τ is the time to transfer unit data. Typically, T_s is in msec range, and τ is in tens of nsec/byte range.

3 Previous Algorithms

In this section, for the sake of completeness, two well-known algorithms are briefly described. These two algorithms provide the best performance among many other algorithms. The algorithm in [1] has been designed using the Parallel Disk Model (PDM) and the algorithm in [15] has been designed using the Linear

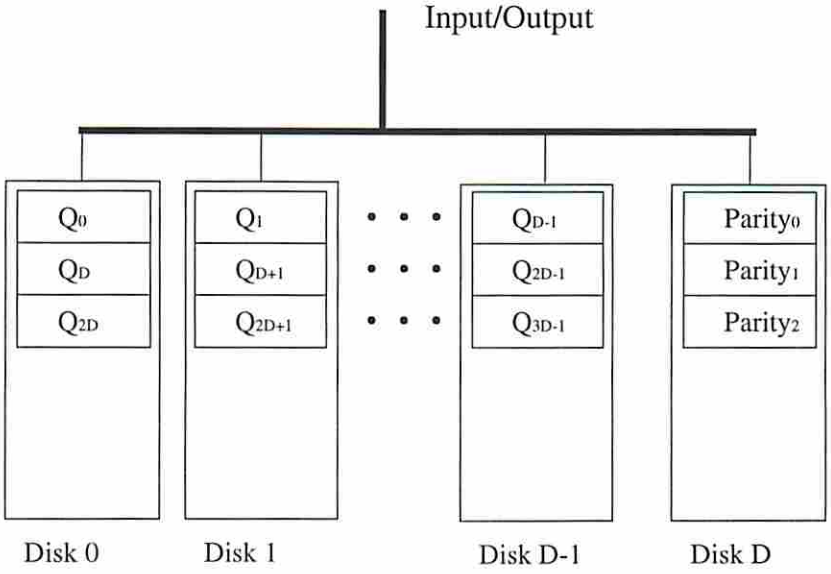


Figure 3: Typical RAID system (RAID 3)

Model. In Section 4, our algorithm is compared with these algorithms.

3.1 Matrix Transpose

In the matrix transpose problem, an input matrix of size $N \times N$ initially resides on the disk. $N = \prod_{s=0}^{t-1} r_s$, where r_s is a positive integer. If N is a prime number, we can add dummy rows to make N nonprime. The input matrix is to be transposed and stored in the other array. The available memory size, M, is assumed to be smaller than the input matrix size. Throughout this paper, to illustrate the key ideas, we use square matrices. However, the algorithms can be easily extended to rectangular matrices as well using the technique in [15]. For the sake of simplicity, throughout this paper, we assume that all the ratios are integers.

3.2 Aggarwal's Algorithm

Aggarwal et al. [1] showed a lower bound on the number of I/O operations to perform matrix transpose. A pseudo code for the algorithm is shown in Figure 4. In this algorithm, as many blocks as the size of the available memory are read into memory. Then, the data is permuted and written onto the disk. The number of I/O operations for this approach is shown in Table 1 (page 3).

In this algorithm, r_s is restricted to be $\leq M/B, 0 \leq s \leq t-1$. In our algorithm, we relax this restriction by developing a technique to use a larger block size. Also, this algorithm does not consider index computation time. Index computation is needed to perform permutation of the data in memory.

```
1   for s = 0 to lg_{M/B} min(B, N/B) - 1 // for each stage
2       for j = 0 to N^2/M − 1 // for each step
3           Read M/B blocks;
4           Perform permutation of data in memory;
5           Write M/B blocks;
```

Figure 4: Pseudo-code for Aggarwal et al.'s algorithm

```
1   for s = 0 to t-1 // for each stage
2       for j = 0 to M/B − 1 // for each step
3           Read M amount of data;
4           Perform permutation of data in memory;
5           for k = 0 to r_s − 1
6                   Write M/B amount of data;
```

Figure 5: Pseudo-code for Kaushik et al.'s algorithm

3.3 Kaushik's Algorithm

In this algorithm [15], there are t stages, where $N = \prod_{s=0}^{t-1} r_s$. Each stage consists of N^2/M steps. In each step, M/N rows are read into memory and a permutation of the data is performed in the memory. Then, the data is written back to the disk in $r_i, 0 \leq s \leq t-1$, write operations. Thus, the number of read (write) operations in each step is 1 (r_s). A pseudo code for the algorithm is shown in Figure 5. The total number of I/O operations using the Linear Model and the Parallel Disk Model are shown in Table 1 (see page 3).

Although the number of I/O operations and the time to transfer data between memory and disk are considered, the total number of read and write operations are not optimized. Also, the index computation time is not considered.

4 An Efficient Algorithm

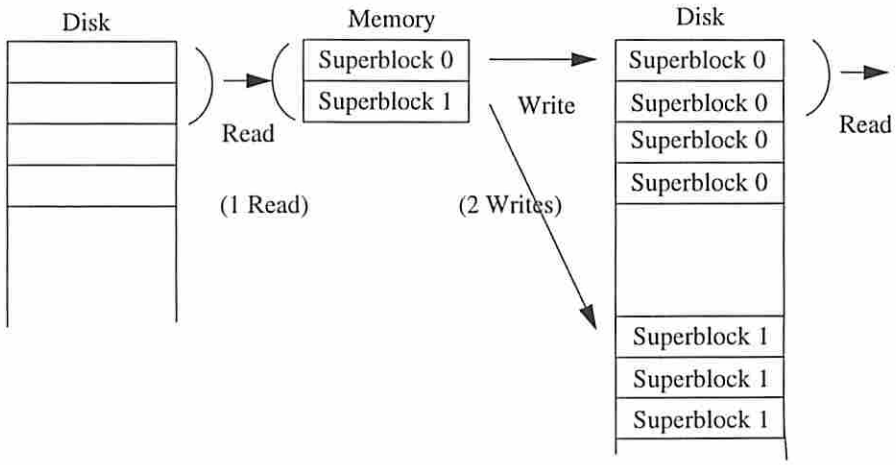
We present an overview of our approach in Section 4.1. The subsequent sections provide all the details of our approach and analyses using the PDM and LM.

4.1 Overview

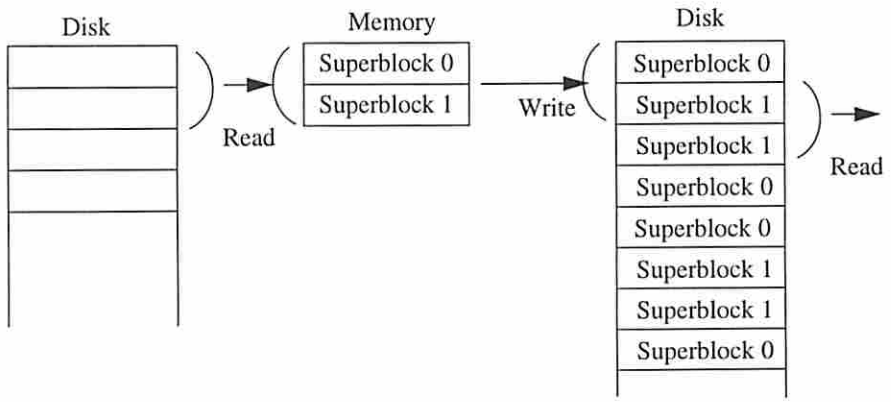
One of the key features of our algorithm is the reduction in the total number of I/O operations, which is achieved by means of an efficient data layout scheme on the disk. For example, in [15], there are three I/O operations (one read operation and two write operations) in each step when $M = 2N$ and $B = N$ (see Figure 6). The concept of a step is explained in detail in Section 4.2. Our algorithm requires only a single write operation in each step as against two write operations in the case of the previous algorithm in [1, 15]. Since

our algorithm consists of only the same number of steps as the previous algorithms, there is a considerable reduction in the total number of write operations.

This reduction in write operations is a consequence of the efficient data layout scheme ($L_s, 0 \leq s \leq t-1$) employed. The proposed layout scheme provides a means for reducing the number of write operations while maintaining the same number of read operations. Thus, the number of I/O operations is reduced from three to two in each step which leads to a 33% reduction in the total number of I/O operations.



(a) Previous Approach [12]



(b) Our Approach

Figure 6: An illustrative example ($r_s = 2$)

Another technique used in our algorithm is the balancing the numbers of read and operations. In balancing the numbers of read and write operations, the key idea is that the total number of I/O operations can be reduced by reducing the number of write operations at the expense of an increased number of read operations. For example, when $r_s = 32$, in each step, the number of read (write) operations in [15] is 1 (32),

where r_s is explained in Section 4.2. In our algorithm, we increase the number of read operations to 9 in order to reduce the number of write operations to 9. This results in a 45% reduction in the total number of I/O operations. Note that a straightforward method to balance the number of read and write operations reduces the total number of I/O operations by only one (see Section 4.2). The data that is written onto the disk in z write operations in the previous algorithm is written in one write operation in our algorithm (see Figure 7). Thus, there is a reduction in the number of write operations by a factor of z. However, this writing causes the data to be "scattered": consecutive superblocks are not contiguous as shown in Figure 7. The superblock is a chunk of memory of size M/r_s at s^{th} stage, $0 \leq s \leq t-1$, and is explained in detail in Section 4.2. In a subsequent read operation, to read the data that is scattered, z read operations are needed. By choosing an optimal value of z, the total number of I/O operations is reduced.

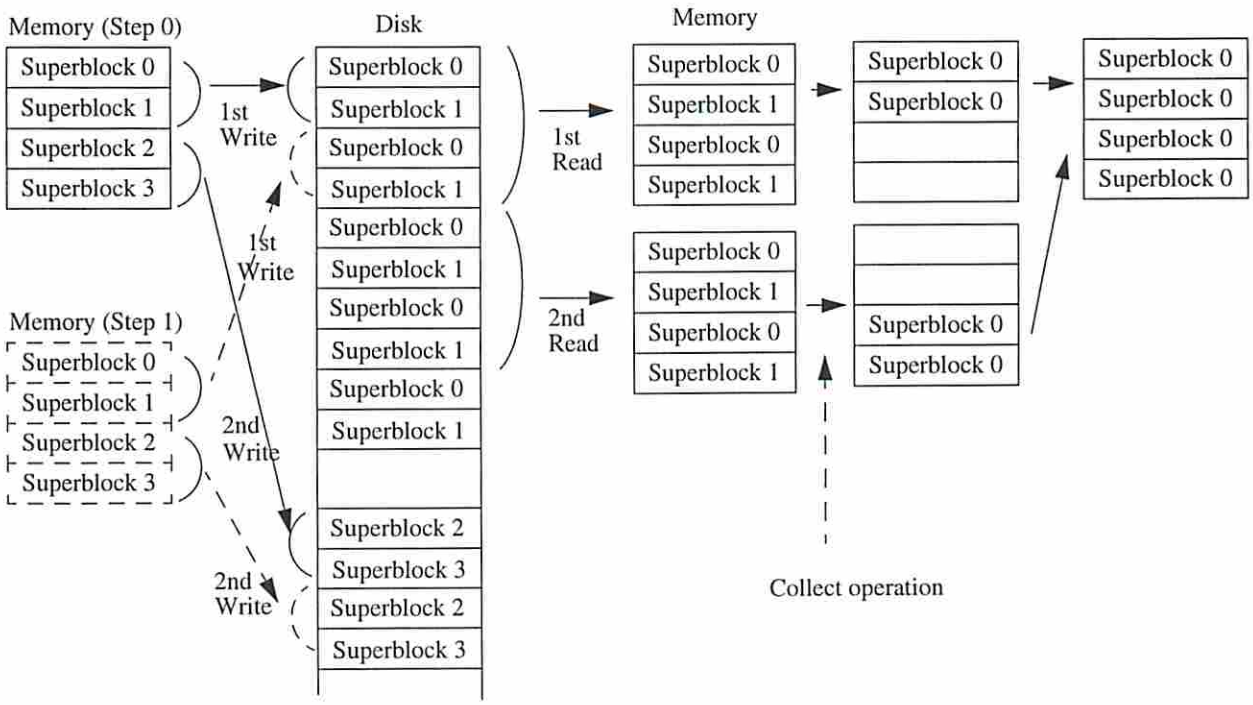


Figure 7: An illustrative example ($r_s = 4$ and $z = 2$)

As shown in Figure 1 (see page 5), the index computation takes up a significant portion of the total execution time. In the previous algorithms [1, 15], the entire available memory is used for reading data from disk. Even though this approach maximizes the memory utilization, it results in excessive index computation cost. (Index computation refers to computing the source or destination addresses of each data.)

To eliminate the index computation cost, the available memory is partitioned into two different-size buffers (read and write buffers). Instead of performing a permutation before every write operation, only the data needed for each write operation is moved into the write buffer. This is denoted as a *collect* operation. The stride of the data access for the collect operation is constant. Thus, it can be performed using inexpensive

```
1  for s = 0 to t-1 // for each stage
2      for step = 0 to N^2/M-1 // for each step
3          Read data from disk using the layout L_{s-1};
4          Permute the data on memory;
5          Write data to disk using the layout L_s;
```

Figure 8: Reducing the number of I/O operations

do-loops.

If the same schedule as in the previous algorithms is used (collect operations followed by write operations), then the size of the write buffer should be $M/2$. However, in our algorithm, the utilization of the write buffer is increased using our schedule which results in a smaller write buffer. In our schedule, a write operation follows each collect operation. Since the read buffer size is less than the available memory size, the number of I/O operations is increased slightly. However, as shown in Section 5, the total execution time is reduced significantly due to reduction in the index computation time.

4.2 Details

Additional details of our algorithm as well as the analysis are presented in this section. However, due to space limitation, proofs of the theorems are not included.

4.2.1 Reducing the Number of I/O Operations

Our algorithm to reduce the number of I/O operations is elaborated here (see Figure 8). Note that the matrix size is $N \times N$ and $N = \prod_{s=0}^{t-1} r_s$.

The algorithm consists of t stages (Line 1). In the s^{th} stage, $0 \leq s \leq t-1$, a *submatrix* is defined as follows. Let us denote the data at row i and column j in the original input matrix as $d_{i,j}$. A submatrix, $S_{k,l}$, $0 \leq k,l \leq R_s - 1$, consists of $d_{i,j}, kN/R_s \leq s \leq (k+1)N/R_s - 1, lN/R_s \leq j \leq (l+1)N/R_s - 1$, where $R_s = \prod_{s=0}^{s} r_s$.

In each stage, there are N^2/M steps (Line 2). In each step, the data is first read into memory (Line 3). The data in the memory that is in the same submatrix is moved to a contiguous chunk of the memory (Line 4). Let us denote this contiguous chunk of memory as a *superblock*. There are r_s superblocks of size M/r_s. The superblocks are written onto the disk (Line 5). The layout, $L_s, 0 \leq s \leq t-1$, specifies the locations of the superblocks on the disk.

The layout, the schedule of reading data from the disk, and the schedule of writing data onto the disk are explained in the following four cases. Case 1 and Case 2 pertain to the cases where as much data as the memory size can be read from the disk or written onto the disk in one I/O operation ($B = M$). Our analysis shows that efficient data arrangement reduces the number of I/O operations by a factor of $(r_s + 1)/r_s$ (Case 1). In addition to this, if $r_s \geq 8$ (Case 2), balancing the number of I/O operations further reduces the total

number of I/O operations.

If $M/r_s < B < M$ (Case 3), our algorithm provides the best performance compared with the previous algorithms. In the other case, $B \leq M/r_s$ (Case 4), our algorithm has the same performance as the previous algorithm in [1] with respect to the number of I/O operations.

Note that reducing the index computation time (discussed in Section 4.2.2) further improves the performance in all the cases.

Case 1: ($B = M$ **and** $1 < r_s < 8$) The key idea here is data arrangement on the disk, L_s. The matrix is first partitioned into R_{s-1} areas. Each area includes N/R_{s-1} rows. The layout and schedules of reading and writing data in each area are explained for two cases.

If $r_s = 2$, the layout, L_s is as follows: the first superblock is stored in the $j^{th}, \lfloor (j+1)/2 \rfloor \bmod 2 = 0$, superblock and the second superblock is stored in the rest of the superblocks (see Figure 9). The number in each small square denotes a data element. Notice that the data is in row-major order in the initial matrix at stage 0 and in column-major order in the last matrix in stage 2. Using this layout, in the first step, the first and second superblocks are stored on the disk in one write operation since they are contiguous in the memory as well as on the disk. This is illustrated in Figure 9. In the figure, at each stage, the left (right) matrix is initial (final) matrix. In the second step, the third and fourth superblocks are saved on disk, and so on. In the next stage, the data in the second and third superblocks are read into the memory using one read operation, and data in the fourth and fifth superblocks are read into the memory in the next step.

If $r_s > 2$, in the st^{th} step, two superblocks, $(st+1) \bmod r_s$ and $(st+2) \bmod r_s$, are stored in one write operation and the rest of the data is written in $(r_s - 2)$ write operations which results in $r_s - 1$ write operations (see Figure 10). The figures in the middle show the data in memory after permutation.

A comparison of the numbers of I/O operations in the algorithm in [15] and our algorithm is shown in Table 3.

Table 3: Number of I/O operations in each step

r_s	2	3	4
Kaushik's Algorithm [15]	3	4	5
This Paper	2	3	4
Reduction	33%	25%	20 %

Case 2: ($B = M$ **and** $r_s \geq 8$) In this case, the total number of I/O operations can be further reduced by balancing the numbers of read and write operations in addition to the data layout and the schedule explained in Case 1. In Kaushik et al.'s algorithm, the difference between the numbers of read and write operations

is large. That is, in each step, the number of read operations is 1 and the number of write operations is r_s. In our algorithm, we develop a technique that reduces the number of write operations at the expense of an increased number of read operations.

Note that if a straightforward method is used, the number of write operations is reduced to $r_s - z$, where z is the number of the new read operations. Then, the new total number of I/O operations is $(r_s - z) + z = r_s$. The total number of I/O operations is reduced by only one. In our algorithm, we decrease the number of write operations to approximately r_s/z. Then, the total number of I/O operations can be reduced by choosing an optimal value of z.

In the previous algorithms, each superblock is stored in one disk write operation. In our algorithm, z blocks are stored on the disk in one write operation. Thus, the number of write operations is reduced by a factor of z. In each read operation, to read data that is "scattered" in noncontiguous locations, we need to perform z read operations. It can be shown that the optimal value of z is $\sqrt{2r_s}$ in the s^{th} stage, $0 \leq s \leq t-1$.

The total number of I/O operations in the algorithm in [15] and in our algorithm are compared in Table 4. The algorithm in [1] is not compared here since it cannot be used in this case. The following Theorem 1 applies to Case 1 and Case 2.

Theorem 1 *In the Linear Model, the total number of I/O operations in our algorithm is* $\frac{N^2}{M} \sum_{s=0}^{t-1} \min(r_s, \sqrt{2r_s} + 1)$.

Case 3: $(M/r_s \leq B < M)$ This is similar to Case 2; the ony difference is the size of the block. It relaxes the restriction $(r_s \leq M/B)$ that was imposed in [1]. In our algorithm, we can increase the value of r_s to be larger than M/B so that the number of stages is decreased. The optimal value of z is Br_s/M.

Theorem 2 *In the Parallel Disk Model, the total number of I/O operations in our algorithm is* $\frac{2N^2}{M} \sum_{s=0}^{t-1} (\sqrt{\frac{r_s M}{B}} + 1)$, *where* $\frac{M}{r_s} < B < M$, $0 \leq s \leq t-1$.

Case 4: $(B \leq M/r_s)$ In this case, our algorithm is the same as in [1].

Table 4: Number of I/O operations in each step

r_s	$r_s = 32$		$r_s = 128$	
	Kaushik's Algorithm	Our Algorithm	Kaushik's Algorithm	Our Algorithm
# of Read operations	1	9	1	17
# of Write operations	32	9	128	17
Total	33	18	129	34

4.2.2 Reducing Index Computation Time

In the previous algorithms, the available memory is fully utilized to reduce the number of I/O operations. In other words, in a read operation, as much data as the size of the memory is read from disk. However, this results in a large index computation time. Permuting the data within the memory requires destination location of each data element to be computed.

To reduce the total execution time, we eliminate the expensive index computation by using the algorithm shown in Figure 11. In our algorithm, we partition the memory into two different-size buffers: one with size M_r (Read buffer) and the other with size M_w (Write buffer). The read buffer is used for reading data from disk. After reading the data, there are r_s/z sets of collect and write operations, where z is a positive integer and explained in Section 4.2.2. In each collect operation, data in z supermatrices is collected in the write buffer. The sizes of the write and read buffers are determined as $Mz/(r_s + z)$ and $MR_s/(r_s + z)$, respectively.

In collect operation, the data in the z superblocks is located in $M_r R_{s-1}/N$ chunks of data. The amount of the data in each chunk is N/R_{s-1}, where $R_s = \prod_{i=0}^{s}$. Thus, to collect the data in z superblocks, multiple-level do-loops are necessary. In each do-loop, the required computations are simple additions to compute loop-variables. Note that, in the previous algorithms [1, 15], the required computations for permutation of the data consist of both the index computations and the loop-variable computations. In our algorithm, since the loop-variables are used to collect data to the write buffer, the index computation is eliminated.

The collected data in the write buffer is written onto the disk in a write operation (Line 6). Even though the number of I/O operations increases by a factor of M/M_r, the total execution time is reduced significantly due to the elimination of index computation time.

5 Experimental Results

We implemented the algorithms on a DEC Alpha system (SGI/Cray T3E) at the San Diego Supercomputing Center (SDSC) and a Sun Enterprise 4000 system at the University of Southern California. For comparison purposes, Kaushik et al.'s algorithm described in Section 3.3 was also implemented.

Note that Aggarwal et al.'s algorithm described in 3.2 has the same total execution time as the Kaushik et al.'s algorithm in our experiments. Even though the two algorithms perform permutation using different methods and the data being permuted are different, the permutation times are the same. If the block size is smaller than M/r_s at the s^{th} stage, $0 \leq s \leq t-1$, then both the algorithms require the same I/O time, where t is the number of stages. I/O time is different for the two algorithms when the amount of data transferred in one I/O operation is smaller than B. The amount of the data transferred in one I/O operation in our experiments ranges from 128 KBytes to 2 MBytes and the typical size of B in state-of-the-art platforms is 4 KBytes. Thus, the performance of the two algorithms are the same in our experiments. Therefore, the execution time reported under the heading "previous algorithm" refers to both the algorithms.

The system parameters of the computing platforms in which our experiments were conducted are summarized in Table 5.

Table 5: Computing Platforms on which experiments were conducted

Platform	SGI/Cray T3E	Sun Enterprise
Processor	DEC Alpha (300 MHz)	UltraSPARC (336 MHz)
OS	UNICOS/mk 2.0.3.39	SunOS Release 5.6 Version Generic_105181-11
Compiler	Cray Standard C Version 6.1.0.1	SC4.0

The amount of main memory allocated to the data was varied from 16 MBytes to 64 MBytes and the data size was varied from 512 MBytes to 2 GBytes. For each parameter value, the algorithms were executed 5 times and the maximum, average, and minimum values were calculated. The speedup of our algorithm over the previous algorithms was calculated for each parameter setting. The results of our experiments are shown in the figures from Figure 12 to Figure 17. The results show that our algorithm reduces the execution time by about 50%.

The execution times correlate well with our analysis. From our experiment in implementing the previous matrix transpose algorithm on the Sun Enterprise, the time for I/O is 10 nsec/byte, the time for index computation is 25 nsec/byte, and the time for data movement is 20 nsec/byte. In this case, the time for I/O is measured as the elapsed wallclock time, which is the most tangible method of measuring I/O performance. The time for I/O is obtained by dividing the total elapsed time by the data size. The time for index computation involves the computation of either the source or the destination address. The destination address y of a data located in x is based on the equation is $y = \frac{x}{a} + xmodb$, where a and b are variables calculated based on the step and stage. Thus, a single index computation involves two divisions and a multiplication, rendering it an expensive operation. For example, on the DEC Alpha 21264, an integer multiplication takes 13 cycles and integer divide is not directly supported. Thus, when the data size is 2 GBytes, the expected total execution time is 2 G $\times$ 3 stages $\times$ 2 (read and write) $\times$ (10+25+20) nsec = 660 sec which is similar to the actual 642 sec. For the size of 512 (128) MBytes, the expected time is 165 (41) sec and actual time is 148 (37) sec.

The execution time is increased about four times as the data size is increased four times. This is reasonable since the three major costs (I/O time, index computation time, and memory-memory transfer time) are proportional to the data size. Thus, we expect the same speedup for the larger data sizes than 2 GBytes.

6 Further Extensions

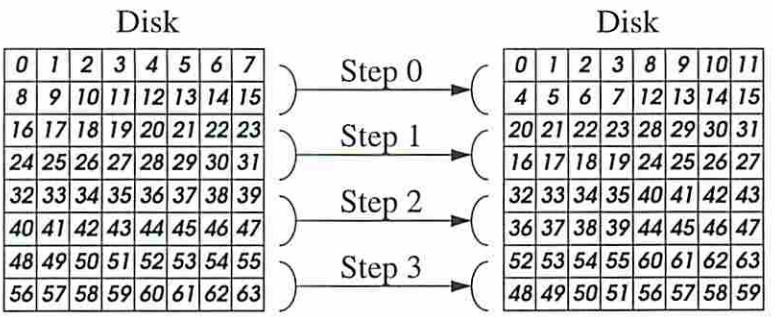
Our matrix transpose algorithm can be extended to the more general problem of Bit Matrix Multiply Complement (BMMC) [7]. In this problem, we consider a matrix X of size $N \times N$. The permutation of the data is represented using a nonsingular matrix A of size $2 \lg N \times 2 \lg N$ and a vector c, where each of the entries of A and c is a binary number. The address of an element is represented in bit notation. A target address is $y = (y_0, y_1, ..., y_{2 \lg N - 1})$, and source address is $x = (x_0, x_1, ..., x_{2 \lg N - 1})$, where the first $\lg N$ bits represent the row number and next $\lg N$ bits represent the column number. The target address is obtained from the source address by $y_i = (\bigoplus_{j=0}^{2 \lg N - 1} a_{i,j} x_j) \oplus c_i, 0 \leq i \leq 2 \lg N - 1$, where the $\oplus$ denotes an exclusive-or operation. A specific case of BMMC is a matrix transpose.

The BMMC consists of many steps. In each step, there are three basic operations as in the case of matrix transpose: read data from disk, permutation of the data on memory, and write data onto disk. Since the key ideas in our matrix transpose algorithm (reducing the number of I/O operations and index computation) are independent of the permutation method of the data in memory, our algorithm will greatly enhance the computational efficiency of the BMMC problem.

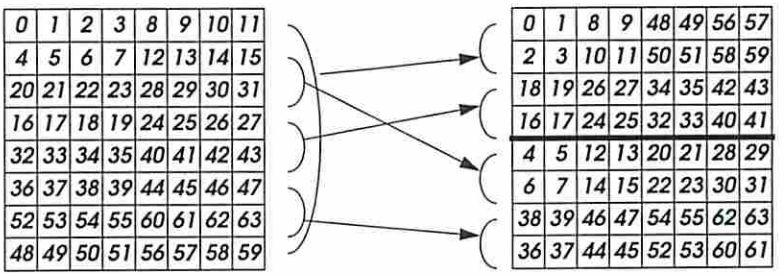
To reduce the number of I/O operations, the algorithm in Figure 8 is used to employ the two techniques: efficient data layout scheme and balancing the number of I/O operations. The permutation of the data in memory is performed as in [7]. To reduce the index computation time, the algorithm in Figure 11 is used: the available memory is partitioned into two buffers, the permutation is replaced by collect operations, and the collect operations and write operations are scheduled to maximize the memory utilization.

7 Conclusion

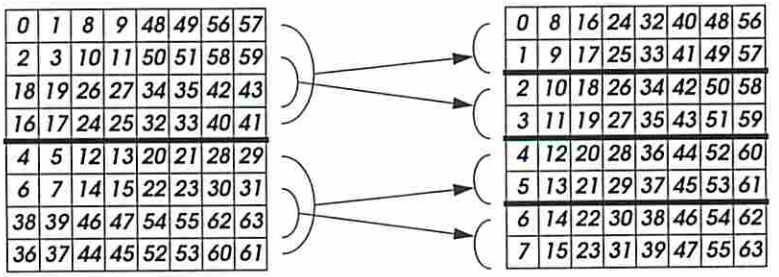
In this paper, we presented an efficient algorithm for large-scale matrix transposition. One of the key achievements of our technique is that it takes into account all aspects of matrix transposition that are computationally intensive in the state-of-the-art computing platforms. Contrary to the previous works that have focused on the reduction of the number of I/O operations, we identified major costs in state-of-the-art computing platforms and these costs are reduced in addition to the number of I/O operations. The generality of the main ideas that constitute our algorithm offer an immense potential for further research into applying these ideas to a variety of algorithms that operate on large data sets. These include problems that have recursive structure such as merge sort, FFT, and graph problems.



(a) Stage 0

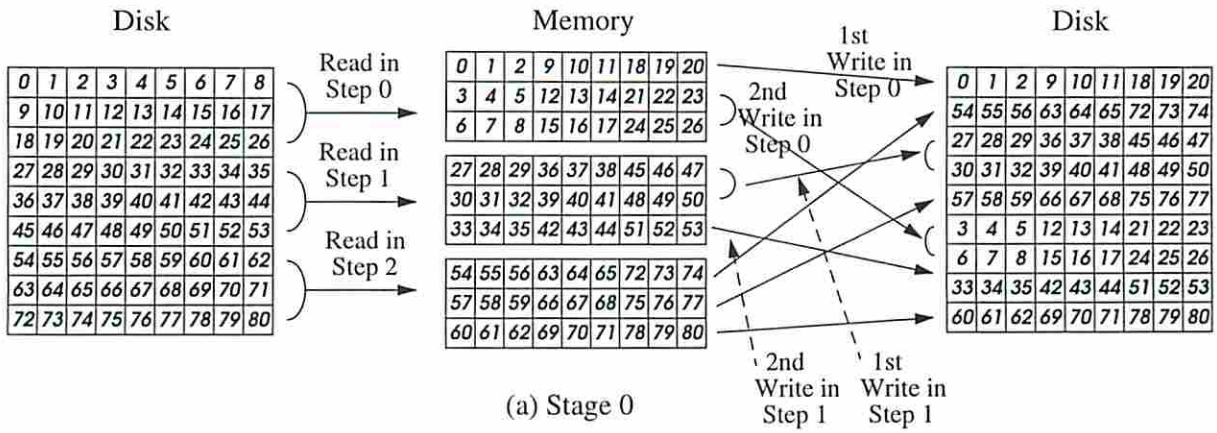


(b) Stage 1

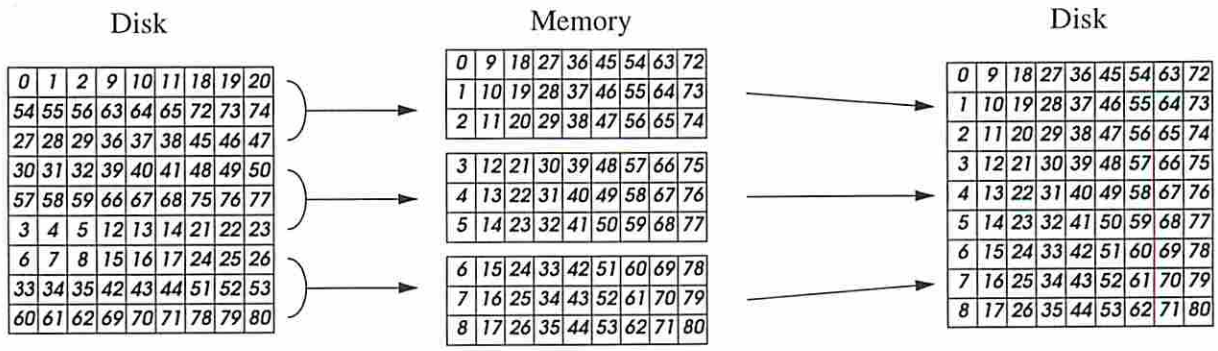


(c) Stage 2

Figure 9: An illustrative example ($N = 8 = \prod_{s=0}^{2} 2$, and $M = 16$)



(a) Stage 0



(b) Stage 1

Figure 10: An illustrative example ($N = 9 = \prod_{s=0}^{1} 3$, and $M = 27$)

```
1   for s = 0 to t - 1 // for each stage
2       for step = 0 to N^2/M_r - 1 // for each step
3           Read data from disk;
4           for i = 0 to r_s/z - 1
5               Move data that has the i^th supermatrix to the write buffer;
6               Write data in write buffer to disk;
```

Figure 11: Pseudo-code for our algorithm

References

[1] A. Aggarwal and J. S. Vitter, "The Input/Output complexity of sorting and related problems," Communications of the ACM, Vol. 31, No. 9, pp. 1116-1127, 1988.

[2] M. B. Ari, "On transposing large $2^n \times 2^n$ matrices," IEEE Trans. Computers, Vol. C-27, No. 1, pp. 72-75, 1979.

[3] L. Carter, J. Ferrante and S. F. Hummel, "Hierarchical Tiling for Improved Superscalar Performance," Proceedings of IPPS '95, 1995.

[4] P. M. Chen, E. K. Lee, G. A. Gibson, R. H. Katz, and D. A. Patterson, "RAID: high performance, reliable secondary storage," ACM Computing Surveys, Vol. 26, No. 2, pp. 145-185, June 1994.

[5] A. Choudhary, W. K. Liao, P. Varshney, D. Weiner, R. Linderman and M. Linderman "Design, Implementation and Evaluation of Parallel Pipelined STAP on Parallel Computers," 12th International Parallel Processing Symposium, Orlando, Florida, 1998.

[6] A. Choudhary, M. Kandemir, H. Nagesh, J. No, X. Shen, V. Taylor, S. More, and R. Thakur, "Data Management for Large-Scale Scientific Computations in High Performance Distributed Systems," in High Performance Distributed Computing Conference '99, San Diego, CA, August 1999.

[7] T. H. Cormen, T. Sundquist, and L. F. Wisniewski, "Asymptotically Tight Bounds for Performing BMMC Permutations on Parallel Disk Systems," Technical Report PCS-TR94-223, Dartmouth College Department of Computer Science, 1994.

[8] J. C. Curlander and R. N. McDonough, Synthetic Aperture Radar-Systems and Signal Processing, Wiley, 1991.

[9] L. G. Delcaro and G. L. Sicuranza, "A method on transposing externally stored matrices," IEEE Trans. on Computers, Vol. C-23, No. 9, pp. 801-803, 1974.

[10] M. Kallahalla and P. Varman, "Optimal Read-Once Parallel Disk Scheduling," Proc. ACM Workshop on I/O in Parallel and Distributed Systems, April 1999.

[11] M. Kallahalla and P. Varman, "An Improved Parallel Prefetching Algorithm," Proc. of Intl. Conference on High Performance Computing, Dec. 1998.

[12] D. E. Dudgen and R. M. Mersereau, Multidimensional Signal Processing, Prentice-Hall, 1984.

[13] J. O. Eklundh, "A fast computer method for matrix transposing," IEEE Transactions on Computers, Vol. 20, Number 7, pp. 801-803, 1972.

[14] R. W. Floyd, "Permuting information in idealized two-level storage, " Complexity of Computer Computations, pp. 105-109, Plenum, 1972.

[15] S. D. Kaushik, C.-H. Huang, J. R. Johnson, R. W. Johnson, and P. Sadayappan, "Efficient Transposition Algorithms for Large Matrices", Supercomputing, 1993.

[16] C. Koelbel, D. Loveman, R. Schreiber, G. Steele Jr., and M. Zosel, The High Performance Fortran Handbook, The MIT Press, 1994.

[17] V. Kumar, A. Grama, A. Gupta, and G. Karypis, Introduction to Parallel Computing, The Benjamin/Cummings Publishing Company, Inc., 1994.

[18] W. Liao, A. Choudhary, D. Weiner, and Pramod Varshney, "Multi-Threaded Design and Implementation of Parallel Pipelined STAP on Parallel Computers with SMP Nodes, IPPS/SPDP, San Juan, Puerto Rico, 1999.

[19] Y. W. Lim, P. B. Bhat, and V. K. Prasanna, "Efficient Algorithms for Block-Cyclic Redistribution of Arrays," 24:298-330, Algorithmica, 1999.

[20] http://www.darpa.mil/ito/research/dis/index.html, 1999.

[21] H. Park, J. Suh, V. K. Prasanna, and M. Ung, "Parallel Implementation of 2D FFT on High Performance Computing Platforms," DoD HPC User's Conference '98, Houston, Texas, June 1998.

[22] H. K. Ramapriyan, "A generalization of Eklundh's algorithm for transposing large matrices," IEEE Trans. on Computers, Vol. C-24, No. 12, pp. 1221-1226, 1975.

[23] S. Ranka and S. Sahni, Hypercube Algorithms for Image Processing and Pattern Recognition, Springer-Verlag, New York, New York, 1990.

[24] J. Suh and V. K. Prasanna, "Portable Implementation of Real Time Signal Processing Benchmarks on HPC Platforms," International Workshop on Applied Parallel Computing in Large Scale Scientific and Industrial Problems '98, Umea, Sweden, June 1998.

[25] J. S. Vitter and E. A. M. Shriver, "Algorithms for parallel memory I: Two-level memories," Algorithmica, Vo. 12 No. 2-3, pp. 110-147, 1994.

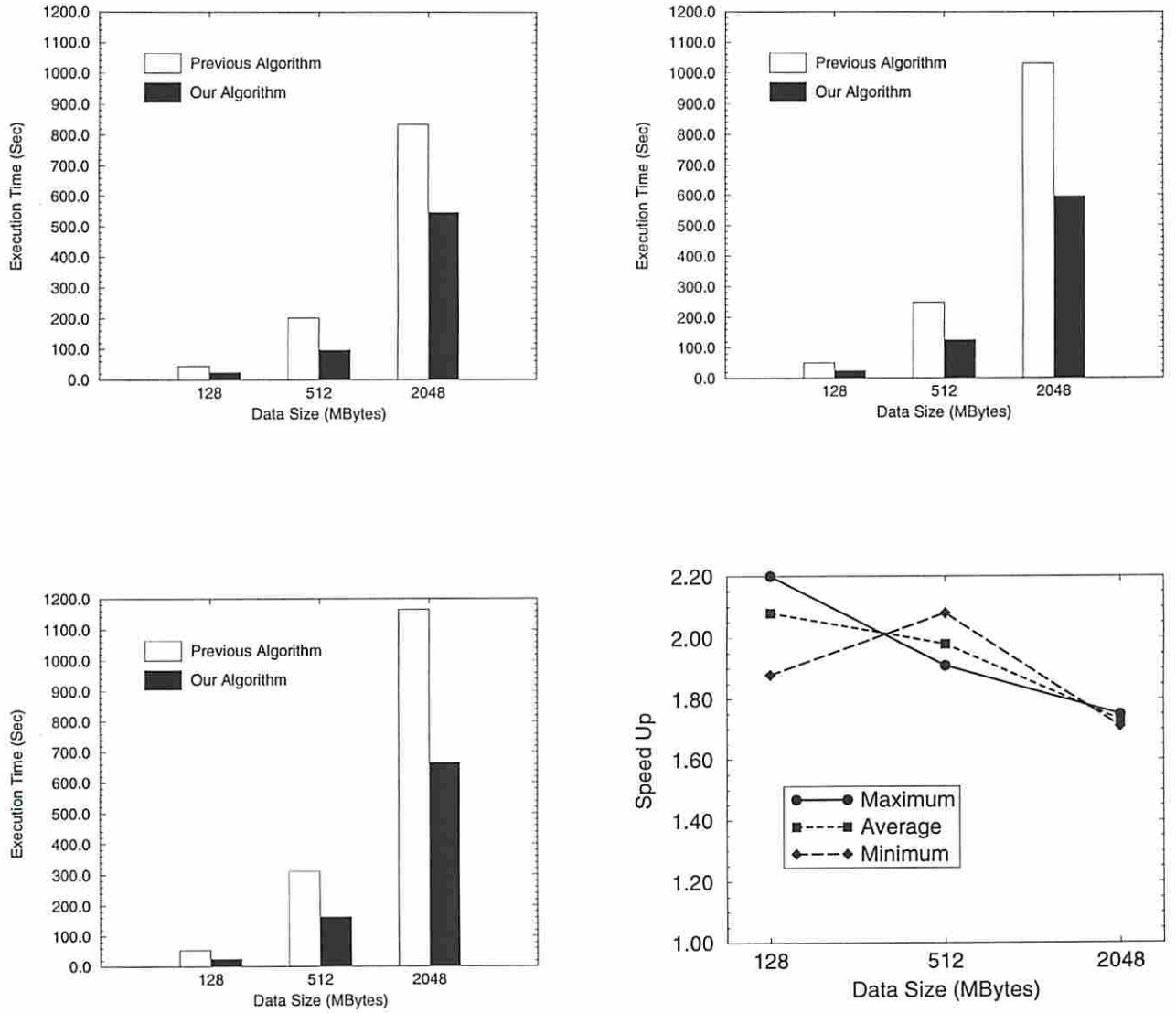


Figure 12: Experimental Results on DEC Alpha (T3E) (a) Minimum (b) Average (c) Maximum (d) Speedup, $M = 16$ MBytes

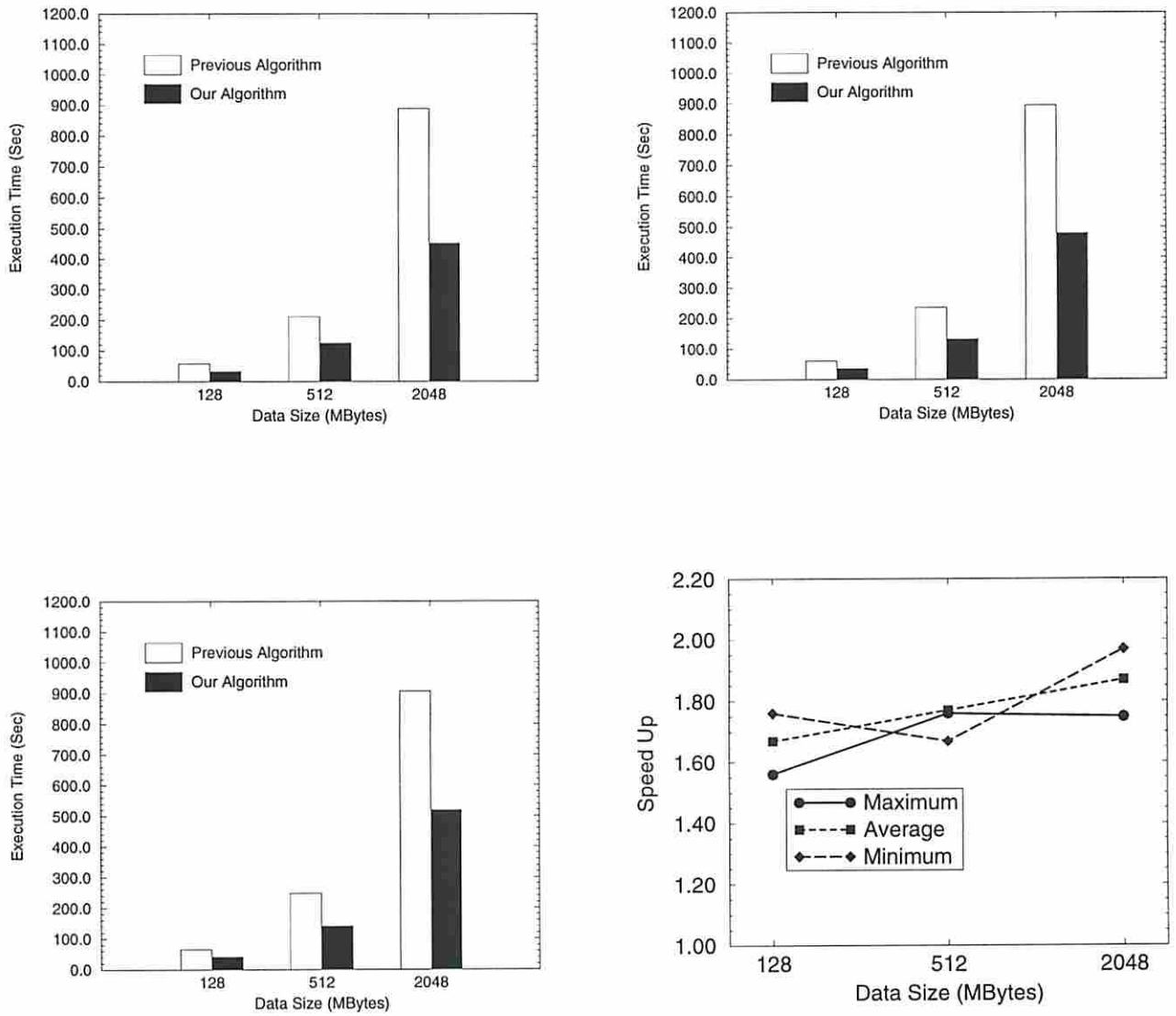


Figure 13: Experimental Results on DEC Alpha (T3E) (a) Minimum (b) Average (c) Maximum (d) Speedup, $M = 32$ MBytes

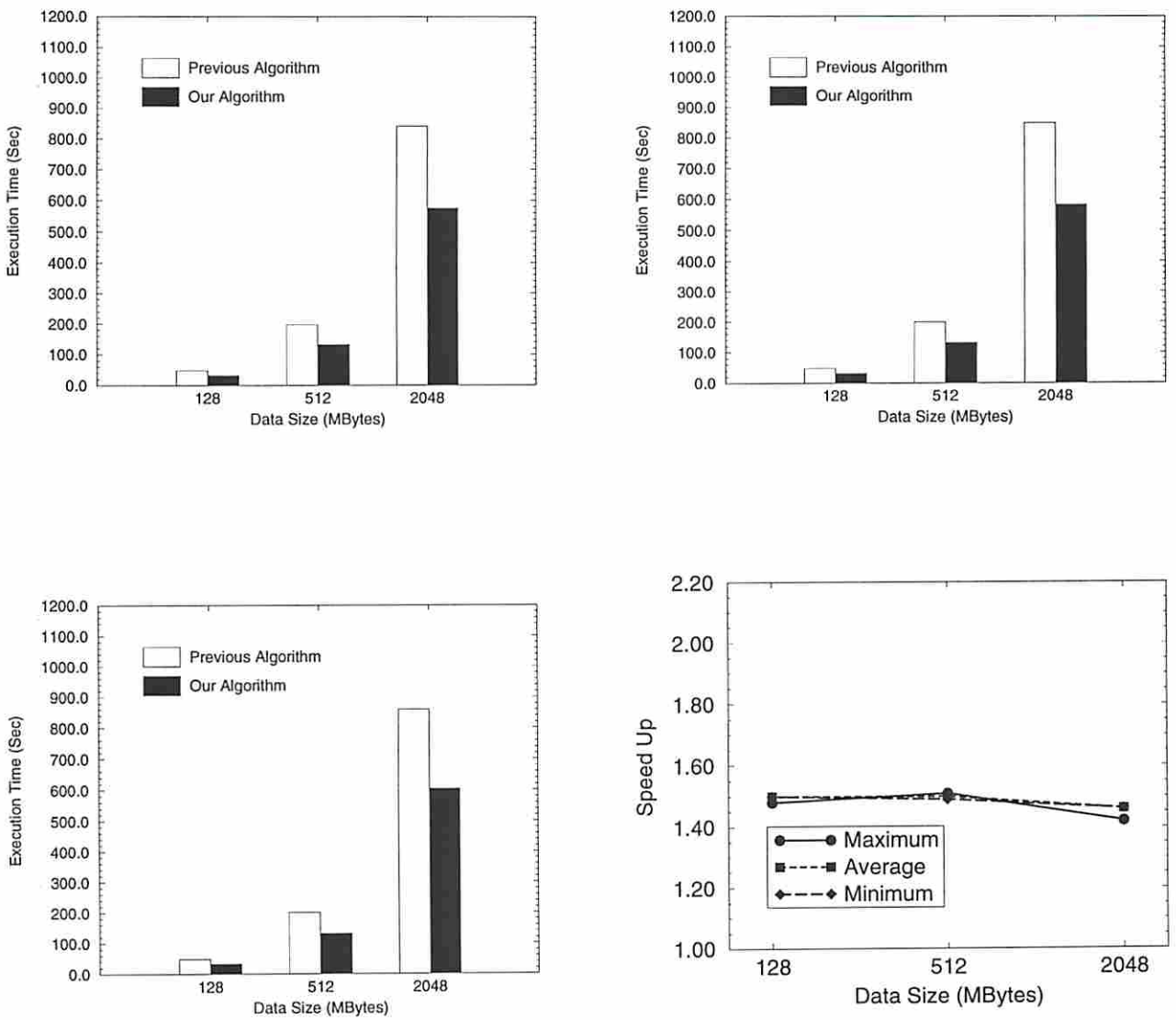


Figure 14: Experimental Results on DEC Alpha (T3E) (a) Minimum (b) Average (d) Speedup, $M = 64$ MBytes

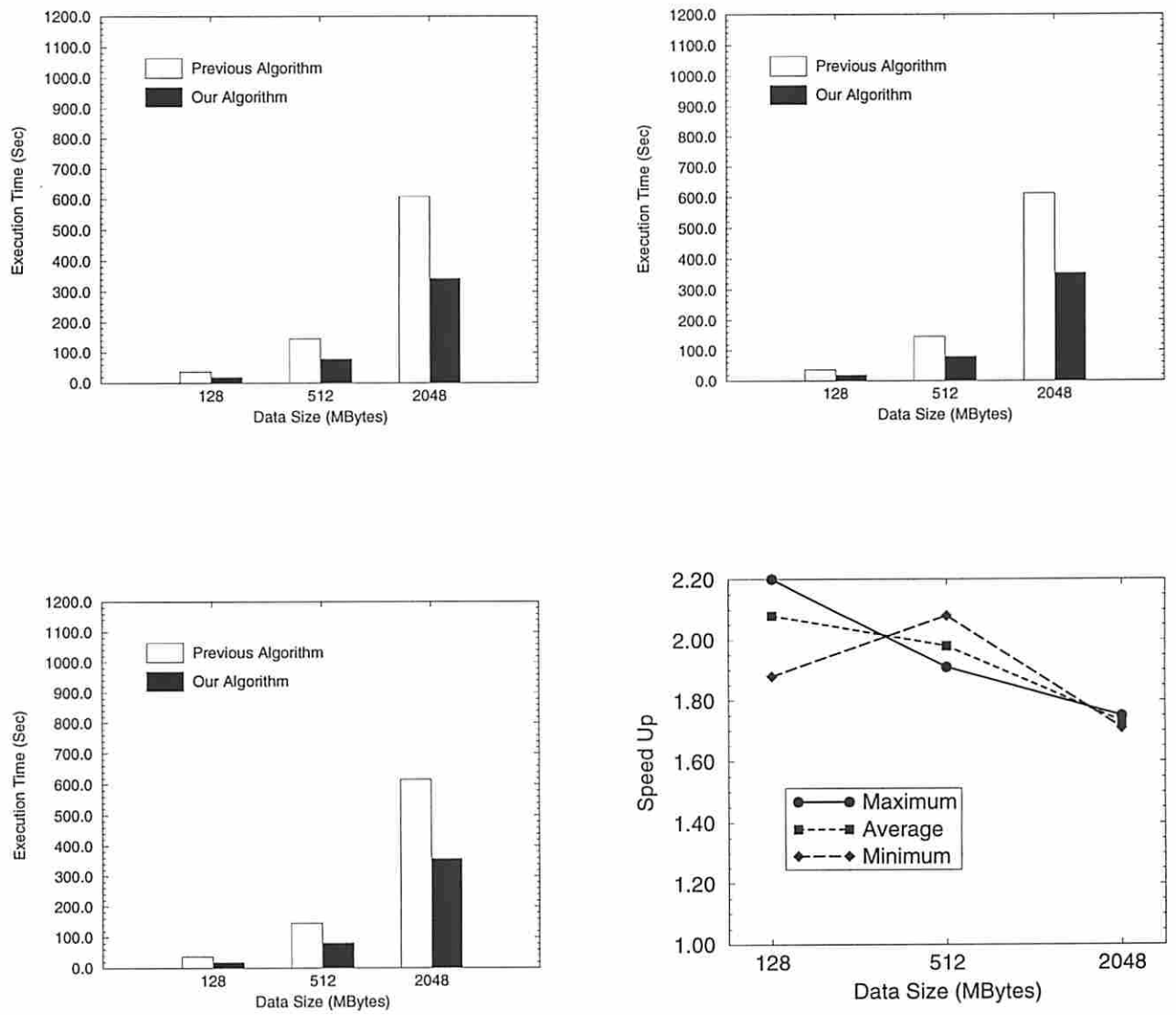


Figure 15: Experimental Results on Sun Enterprise (a) Minimum (b) Average (c) Maximum (d) Speedup, $M = 16$ MBytes

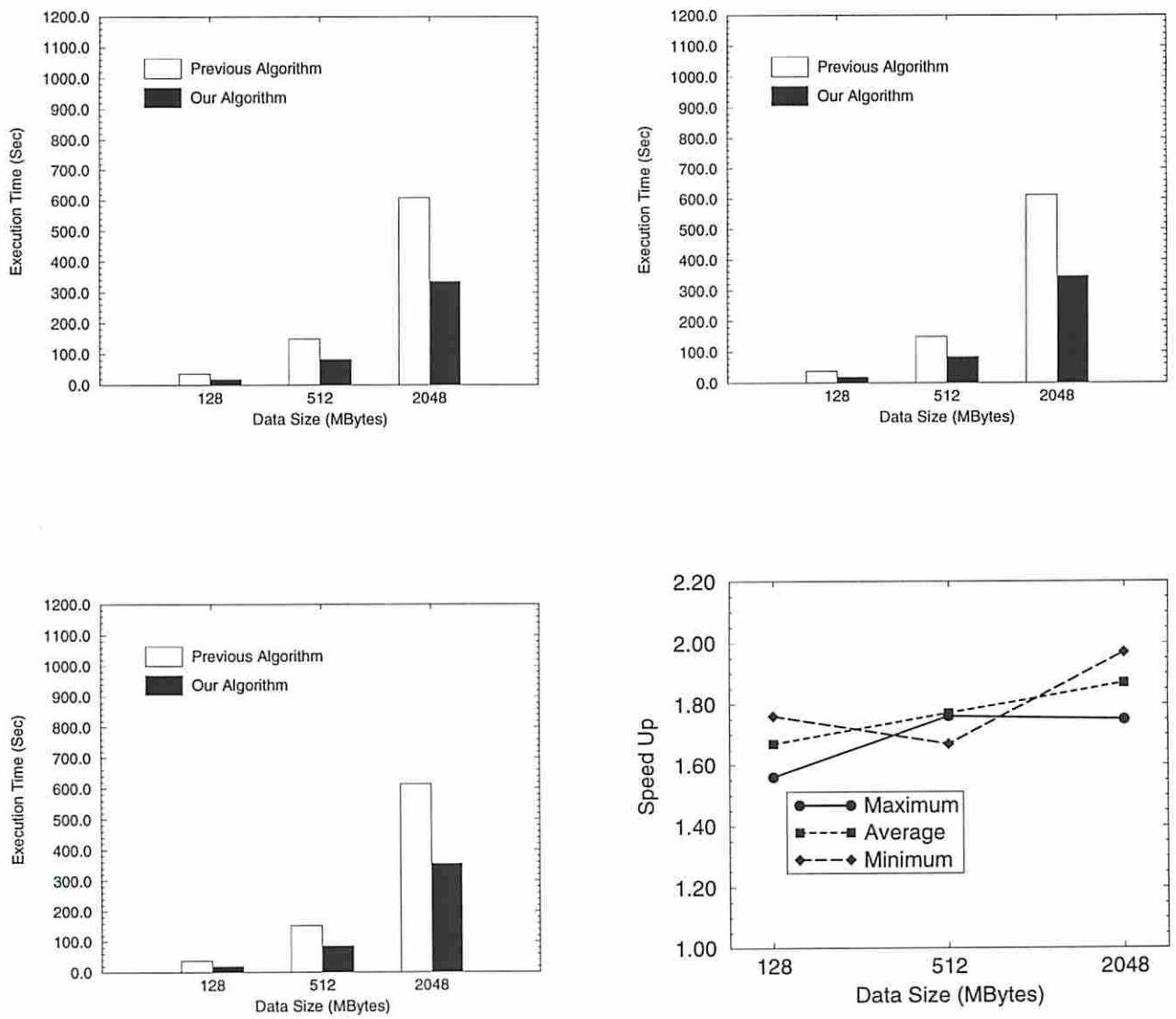


Figure 16: Experimental Results on Sun Enterprise (a) Minimum (b) Average (c) Maximum (d) Speedup, $M = 32$ MBytes

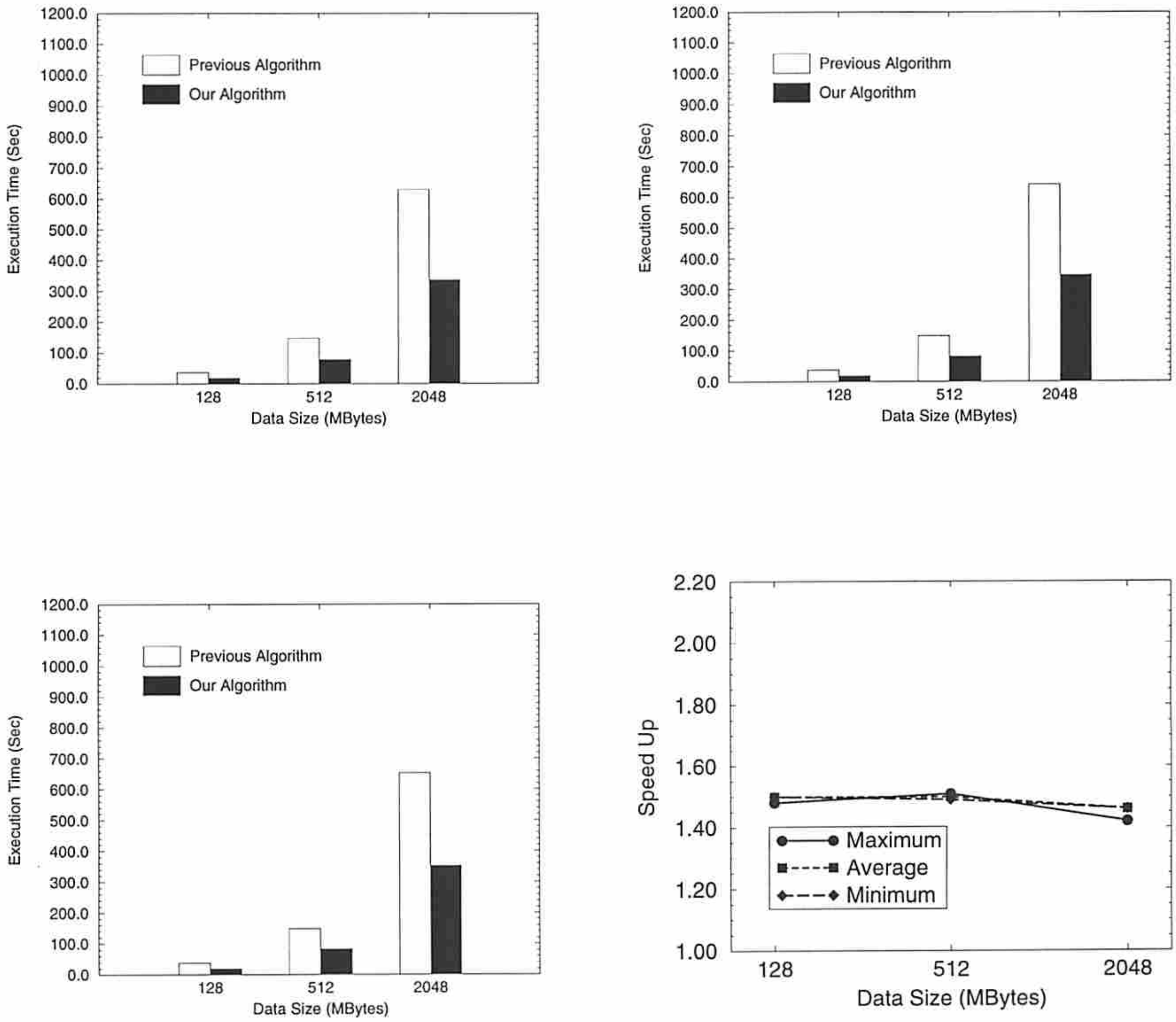


Figure 17: Experimental Results on Sun Enterprise (a) Minimum (b) Average (c) Maximum (d) Speedup, M = 64 MBytes