Efficient Adaptive Algorithms for Transposing Small and Large Matrices on Symmetric Multiprocessors.

Article *in* Informatica · January 2006

Source: DBLP

CITATIONS
2

READS
1,851

3 authors, including:

Seong-Moo Yoo
University of Alabama in Huntsville
114 PUBLICATIONS **1,056** CITATIONS

SEE PROFILE

INFORMATICA, 2006, Vol. 17, No. 4, 535–550

Efficient Adaptive Algorithms for Transposing Small and Large Matrices on Symmetric Multiprocessors

Rami Al NA'MNEH, W. David PAN, Seong-Moo YOO
Department of Electrical and Computer Engineering, University of Alabama in Huntsville
301 Sparkman Drive, Huntsville, Alabama 35899, USA
e-mail: dwpan@ece.uah.edu

Received: November 2005

Abstract. Matrix transpose in parallel systems typically involves costly all-to-all communications. In this paper, we provide a comparative characterization of various efficient algorithms for transposing small and large matrices using the popular symmetric multiprocessors (SMP) architecture, which carries a relatively low communication cost due to its large aggregate bandwidth and low-latency inter-process communication. We conduct analysis on the cost of data sending / receiving and the memory requirement of these matrix-transpose algorithms. We then propose an adaptive algorithm that can minimize the overhead of the matrix transpose operations given the parameters such as the data size, number of processors, start-up time, and the effective communication bandwidth.
Key words: matrix transpose, SMP, MPI, all-to-all communication.

1. Introduction

Matrix transpose, which basically depends on all-to-all communication, is widely used in implementations of Fast Fourier Transforms (FFT) (Calvin and Trystram, 1996; Choi *et al.*, 1995; Portnoff, 1999) and other applications such as seismic imaging (Claerbout, 1985) and synthetic aperture radars (Curlander and McDonough, 1991; Jakowatz *et al.*, 1996). For example, parallel implementation of the 1-D FFT using the classical six-step algorithm requires matrix transpose three times (Bailey, 1990). Nonetheless, matrix transpose on parallel computers requires intensive all-to-all communication, which adversely affects the performance of the FFT, since it is difficult to scale pure all-to-all communication (Shan *et al.*, 2003). For instance, matrix transpose in the 1-D FFT running on 32 processors is responsible for 50% of the overall run time compared to only 16% of the sequential case (Shan *et al.*, 2003). Therefore, matrix transpose is definitely a factor that limits the scalability of FFT in parallel systems.

There has been extensive work on optimizing all-to-all communication in parallel systems (Bala *et al.*, 1995; Barnnet *et al.*, 1993; Barnnet *et al.*, 1994; Bokhari, 1991; Bokhari and Berryman, 1992; Bruck *et al.*, 1997; Johnsson and Ho, 1991; Scott, 1991; Suh and

Shin, 2001; Thakur and Gropp, 2001; Thakur and Gropp 2003; Traff, 2002), including optimizing all-to-all communication in mesh systems (Barnnet *et al.*, 1993; Bokhari, 1991; Bokhari and Berryman, 1992; Gupta and Kumar, 1993; Scott, 1991; Suh and Shin, 2001), torus (Suh and Shin, 2001; Swarztrauber and Hammond, 2001), and hypercube systems (Gupta and Kumar, 1993; Johnsson and Ho, 1991; Swarztrauber and Hammond, 2001). For example, (Scott, 1991) proposed efficient algorithms that minimize link contentions in hypercube and mesh topologies. (Johnsson and Ho, 1991) introduced optimal broadcasting and personalized communication in hypercube systems. In (Suh and Shin, 2001), an optimal all-to-all personalized communication was proposed for multidimensional torus and mesh networks.

Nowadays, Symmetric Multiprocessing (SMP) may be one of the most popular architectures. SMP typically has a low communication cost since it provides large aggregate bandwidth and low latency inter-process communication. Furthermore, networks of SMP machines have long been viewed as an effective platform for obtaining supercomputer performance on parallel applications (Culler, *et al.*, 1995). Moreover, SMP can support two programming models: message passing and shared address space. Shared address space offers ease of programming, particularly for irregularly structured computations, but suffers from performance limitation due to the protocol overhead and poor spatial locality (Shan *et al.*, 2003). On the other hand, there are performance advantages in integrating message passing in cache coherent multiprocessors (Woo *et al.*, 1993). These advantages include the ability to overlap the communication and computation, and the ability to effectively move data with large sizes. Therefore, there have been efforts that attempt to efficiently implement all to-all communication on clusters of SMP's (Sistare *et al.*, 1999; Traff, 2002). Furthermore, the performances of some applications with intensive all-to-all communication (including the FFT) were compared in (Karlsson and Brorsson, 1998; Shan *et al.*, 2003). It was found that the message-passing model outperformed the shared address space model for most of the applications considered.

The goal of this paper is to develop fast parallel algorithms that transpose matrices on SMP using the message-passing model. To this end, we first provide a comparative characterization of several existing algorithms that can transpose matrices efficiently in hypercube systems (Johnsson and Ho, 1991; Scott, 1991). We then propose a matrix-transpose algorithm that is adaptive with varying data sizes, number of processors, start-up time, and effective bandwidths.

The rest of the paper is organized as follows. Section 2 provides an introduction to matrix transpose in parallel systems. In Section 3, we first give a survey on the existing algorithms for transposing matrices, and then introduce an algorithm suitable for transposing large matrices. Section 4 discusses the Butterfly algorithms for transposing small matrices. In Section 5, we introduce the adaptive algorithm for transposing matrices of varying sizes. In Section 6, we present the experimental results of the proposed algorithms, along with discussions. Section 7 contains the concluding remarks.

2. Matrix Transpose

After a matrix (array) is transposed, a row will become a column and vice versa. Hierarchal transpose (Portnoff, 1999) is widely used, since it creates more opportunities for parallelization. In hierarchal transpose, an array is first divided into smaller square arrays. These smaller sub arrays are then transposed. In a parallel system, the jth block sent from process i is received by process j and is placed in the ith block of the receive buffer. Inside each of these arrays, sub arrays are then transposed until the entire array is transposed.

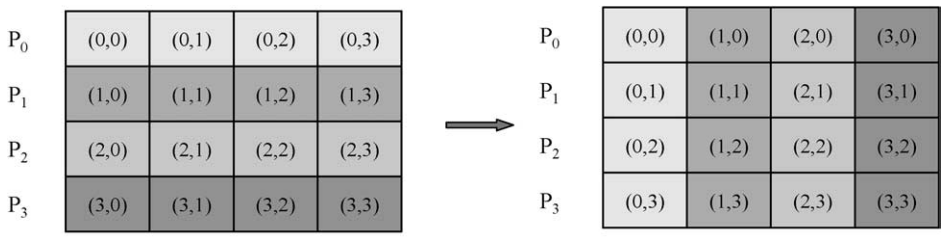
Matrix transpose requires all-to-all communication between processors. Assume that the data size is N and there are a total of P processors. Before the transpose, each processor initially has P blocks of data, each block of size $\frac{\sqrt{N}}{P} \times \frac{\sqrt{N}}{P}$. Hence, each processor has $\frac{\sqrt{N}}{P} \times \sqrt{N}$ of data. The goal is to exchange the ith block of processor j with the jth block of processor *i*. Fig. 1 shows an example of transposing the matrix using all-to-all communication for $P = 4$, where each processor transposes one block locally and sends one block to each other processor in the system. The ith block of processor j will be exchanged with the jth block of processor i.

3. Transposing Large Matrices

In this section we present the algorithms for transposing large matrices, where matrix transpose can be completed in $P - 1$ steps, with each processor exchanging a square block with size $\frac{\sqrt{N}}{P} \times \frac{\sqrt{N}}{P}$ to all other processors.

3.1. *The MPICH-1.2.4 Algorithm*

The MPICH-1.2.4 algorithm implements all-to-all in $P-1$ steps by simply posting all the non-blocking receives and sends, and then calling MPI_Waitall on all of them (Thakur



(a) Before transpose (b) After transpose

Fig. 1. Transposing the matrix using all-to-all communication for $P = 4$. (a) The entry (i, j) represents the jth data block located in processor P_i. The jth block of processor P_i will be exchanged with the ith block of processor P_j. (b) the entry (i, j) represents the ith data block located in processor P_j.

and Gropp, 2001). Each processor exchanges a square block with size $\frac{\sqrt{N}}{P} \times \frac{\sqrt{N}}{P}$ to all other processors. So the overhead associated with this algorithm is

$$H(N, P) = \left(Ts + Tw\frac{N}{P^2}\right)(P-1), \tag{1}$$

where Tw is the effective per 8-byte communication time supported by the inter-processor communication network, and Ts is the start-up time associated with all basic communication operations. This algorithm requires memory space to hold its data of size $\frac{N}{P}$ and two temporary buffers (to send and receive) each of size $\frac{N}{P}$.

The disadvantage of this algorithm is that messages are not ordered. All messages are first sent to process 0, then to process 1, and so forth, resulting in a single-node bottleneck. Fig. 2 shows the pseudo code for the MPICH-1.2.4 algorithm.

```
For (i = 0; i < P; i ++)
{
   Destination = i
   Source      = i
   None blocking receive (from destination)
   Pack the data to be sent of size ( √N/P × √N/P )
   None blocking send (to source)
}
MPI_Waitall ()
Unpack the data of size ( √N/P × √N/P )
```

Fig. 2. The pseudo code for the MPICH-1.2.4 algorithm.

3.2. *The Bruck Algorithm*

The Bruck algorithm consists of three steps (Bruck *et al.*, 1997). In the first step, each processor P_i independently rotates its n data blocks i steps upwards in a cyclic manner. In the second step, each processor rotates its jth data block j steps to the right in a cyclic manner. The third step is the same as step one except that the rotation is downwards in a cyclic manner. More specifically, step 2 consists of w sub-phases, where $w = \lceil \log_r P \rceil$. During each sub-phase, each processor needs to perform send/receive operations $r-1$ times, except for the last sub-phase, where each processor needs to send/receive $\lceil \frac{P}{r^{w-1}} \rceil - 1$ times. During step z of sub-phase x, where $0 \leqslant x \leqslant w-1$ and $1 \leqslant z \leqslant r-1$, all data blocks, for which the xth digit of their block-id is z, are rotated $z \times r^x$ steps to the right. If we use the Bruck algorithm with radix-P, the overhead associated with this algorithm is the same as that in (1). The total memory space required by the transpose algorithm includes that for the data with size $\frac{N}{P}$, two temporary buffers (to send and receive) each of size $\frac{N}{P^2}$, and one extra buffer for local data movement (rotation) of size $\frac{N}{P}$. Fig. 3 shows an example of the Bruck algorithm using three processors.

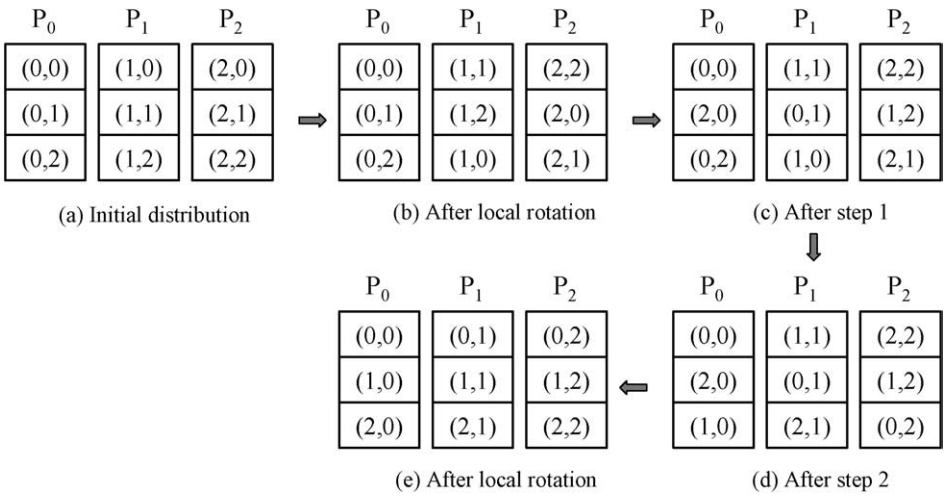


Fig. 3. The Bruck algorithm using $P = 3$.

3.3. *The Transpose Algorithm*

In this section, we introduce the transpose algorithm that can be completed in $P-1$ steps. The transpose algorithm stems originally from the hypercube systems (Scott, 1991), hence it is limited to systems with an even number of processors. It has been shown that in hypercube systems, the transpose algorithm minimizes the link contention at each step (Scott, 1991). In each step of the algorithm, each processor exchanges its data block with the destination (the processor rank exclusive-ored with the step number i, where $1 \leqslant i \leqslant P-1$). This all-to-all communication can be accomplished by overlapping the communication (sending the data) and computation (packing the data of size $\frac{\sqrt{N}}{P} \times \frac{\sqrt{N}}{P}$).

The transpose algorithm can be generalized to any number of processors. In step k, each processor receives from the $k-1$ processor and sends to the $k+1$ processor in

```
For (i =1; i < P; i ++)
{
  Source = mod (processor-rank – i + P, P)
  Destination = mod (processor-rank + i, P)
  None blocking receive (from destination)
  Pack the data be sent ( √N/P × √N/P )
  Send (to source)
  Unpack the data ( √N/P × √N/P )
}
```

Fig. 4. The pseudo code for the general transpose algorithm.

the round-robin fashion. This is illustrated in Fig. 4. The mod (x, y) function gives the remainder of dividing x by y. The overhead associated with this algorithm is given by (1). The transpose algorithm requires memory space to hold its data of size $\frac{N}{P}$ and two temporary buffers (to send and receive), each of size $\frac{N}{P^2}$.

4. Transposing Small Matrices

In parallel systems, the start-up time Ts is usually larger than the transfer time Tw. For example, in our SMP system, *Ts* = 300 *Tw*. From (1), we notice that Ts is multiplied by P (number of communication steps required by the algorithm), and Tw is multiplied by $\frac{N}{P}$ (if we assume that $P - 1 \approx P$). Therefore, for relatively small data sizes, it is important to reduce the number of communication steps required by an algorithm.

4.1. *The Butterfly Algorithm*

In the following, we will discuss the butterfly algorithm suitable for transposing small matrices. The butterfly algorithm was based on the hypercube system (Johnsson and Ho, 1991). In the butterfly algorithm, the data array of size $\frac{\sqrt{N}}{P} \times \sqrt{N}$ in each processor is divided into P blocks with size of $\frac{\sqrt{N}}{P} \times \frac{\sqrt{N}}{P}$. The array is then transposed in $\log_2 P$ phases. In phase k, all nodes swap half of their data ($\frac{\sqrt{N}}{P} \times \frac{\sqrt{N}}{2}$) with their peers along the dimension k of the hypercube (swap half of the data with myproc (processor rank) XOR k, where $k = 1, 2, \ldots, \log_2 P$). Since some blocks will not be sent at all and some blocks may be sent more than once, we must ensure that all the blocks are transposed in the first phase. Therefore, blocks are sent row/column wise and received column/row wise. Other blocks that are not sent in the first phase should be transposed internally. In all other phases, the data are sent row/column wise and received row/column wise. This all-to-all communication can be carried out by overlapping the communication (sending the data) with the computation (packing the data $\frac{\sqrt{N}}{P} \times \frac{\sqrt{N}}{2}$). Fig. 5 gives an example of transposing the array in $\log_2 P$ steps using 4 processors on the SMP.

In order to transpose the array using the butterfly algorithm, each node must swap $\log_2 P$ messages of size ($\frac{\sqrt{N}}{P} \times \frac{\sqrt{N}}{2}$). Thus, the total overhead is given by:

$$H(N, P) = \left(Ts + Tw\frac{N}{2P}\right) \log_2 P. \tag{2}$$

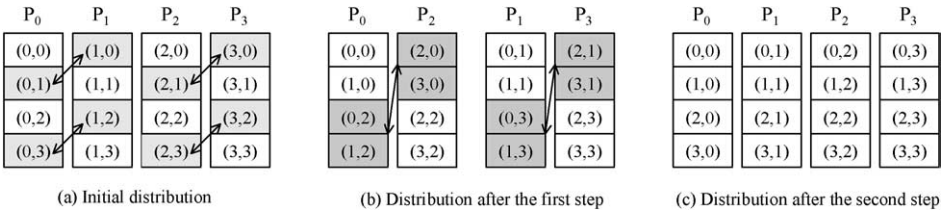


Fig. 5. Transpose the array in $\log_2 P$ steps on the SMP.

The butterfly algorithm requires memory space to hold its data of size $\frac{N}{P}$ and two temporary buffers (to send and receive) each of size $\frac{N}{2P}$.

4.2. *The Generalized Butterfly Algorithm*

The generalized algorithm for the butterfly algorithm is based on the Bruck algorithm (radix-2) (Bruck *et al.*, 1997). If the number of processors is even, then the overhead of the Bruck algorithm is the same as that given by (2). However, if the number of processors is odd, then this algorithm can be completed in $\lceil \log_2 P \rceil$ steps. In the first $\lfloor \log_2 P \rfloor$ steps, the overhead is given by:

$$H(N, P) = \left(Ts + Tw \frac{N}{2P} \right) \lfloor \log_2 P \rfloor. \tag{3}$$

In the last step, each processor should send/receive the following data size

$$DS_1 = \frac{\sqrt{N}}{P} \times \frac{\sqrt{N}}{P} (P - 2^{\lfloor \log_2 P \rfloor}). \tag{4}$$

The generalized butterfly algorithm requires memory space to hold its data of size $\frac{N}{P}$, two temporary buffers (to send and receive), each of size $(\frac{\sqrt{N}}{P} \times \frac{\sqrt{N}}{2})$, and an extra buffer for local data movement (rotation) of size $\frac{N}{P}$.

5. The Adaptive Algorithm

(Bruck *et al.*, 1997) introduced a general algorithm (the index algorithm) for all-to-all communication. The index algorithm (Bruck *et al.*, 1997) consists of three phases. Phase1 and phase 3 involve only local data movement. Phase 2 consists of a sequence of point-to-point communication operations, which can be divided into w sub-phases, where $w = \lceil \log_r P \rceil$. During each sub-phase, each processor needs to perform send/receive operations $r - 1$ times, except for the last sub-phase, where each processor needs to send/receive $\lceil \frac{P}{r^{w-1}} \rceil - 1$ times. During step z of sub-phase x, where $0 \leqslant x \leqslant w - 1$ and $1 \leqslant z \leqslant r - 1$, all data blocks, for which the xth digit of their block-id is z, are rotated $z \times r^x$ steps to the right. In (Bruck *et al.*, 1997) (pp. 1148), the upper bound of the overhead can be expressed as follows:

$$H(r, P) = (r - 1) \lceil \log_r P \rceil Ts + b(r - 1) \left\lceil \frac{P}{r} \right\rceil \lceil \log_r P \rceil Tw, \tag{5}$$

where $2 \leqslant r \leqslant P$and b is the block size. Since we divide the array into P blocks, each of the size $(\frac{\sqrt{N}}{P} \times \frac{\sqrt{N}}{P})$, the overhead can be given by:

$$H(r, P, N) = (r - 1) \lceil \log_r P \rceil T_s + \frac{\sqrt{N} \times \sqrt{N}}{P^2} (r - 1) \left\lceil \frac{P}{r} \right\rceil \lceil \log_r P \rceil Tw. \tag{6}$$

In (6), Ts is multiplied by $(r-1)\lceil \log_r P \rceil$(which represents the number of communication operations), and Tw is multiplied by $\frac{N}{P^2}(r-1)\lceil \frac{P}{r} \rceil \lceil \log_r P \rceil$(which represents the total amount of data to be transferred). As we increase r from 2 to P, $\lceil \log_r P \rceil$ will decrease and $r-1$ will increase. But the increasing rate of $r-1$ is higher than the decreasing rate of $\lceil \log_r P \rceil$. In other words, $r-1$ will be the dominant term in $(r-1)\lceil \log_r P \rceil$. Therefore, if r = 2, then the number of communication operations will be minimal. On the other hand, if we assume, for simplicity, that $\frac{P}{r}$ is an integer and $\frac{r-1}{r} \approx 1$, then $\frac{N}{P^2}(r-1)\lceil \frac{P}{r} \rceil \lceil \log_r P \rceil$ can be simplified into $\frac{N}{P}\lceil \log_r P \rceil$. By tuning r (between 2 and P), we can find the minimum overhead based on the data size N, the number of processors P, the start-up time Ts, and the effective communication time Tw.

In order to get the exact overheads rather than the upper bound of the overheads, we assume, for simplicity, that $r^2 \geqslant P$, then $2 \geqslant \log_r P$, and thus $w = \lceil \log_r P \rceil \leqslant 2$. In other words, we have at most 2 sub-phases, and the block with ID j $(0 \leqslant j \leqslant P-1)$, can be encoded using $\lceil \log_r P \rceil = w \leqslant 2$ digits. We know from the index algorithm that in the first sub-phase we need to perform send/receive operations $r-1$ times. In the first send/receive operation, all the data blocks, for which the 0th digit of their block is 1, are rotated to the right. In the $r-1$ send/receive operations, all those data blocks, whose 0th digit of their block ID is $r-1$, are rotated to the right. Therefore, if $\frac{P}{r}$ is an integer, then each processor must send/receive the following data with size being

$$DS_2 = \frac{\sqrt{N}}{P} \times \frac{\sqrt{N}}{P}(r-1)\frac{P}{r}. \tag{7}$$

If $\frac{P}{r}$ is not an integer, then each processor must send/receive the following data size

$$DS_3 = \frac{\sqrt{N}}{P} \times \frac{\sqrt{N}}{P}\left\{\left(r-1-(P\%r-1)\right)\left\lfloor \frac{P}{r} \right\rfloor + (P\%r-1)\left\lceil \frac{P}{r} \right\rceil\right\}, \tag{8}$$

where $P\%r$ gives the remainder of dividing P by r. Consequently, the total overhead for the first sub-phase (assuming that $\frac{P}{r}$ is an integer) is

$$H_1(r,P,N) = (r-1)Ts + \frac{\sqrt{N}}{P} \times \frac{\sqrt{N}}{P}(r-1)\frac{P}{r}Tw. \tag{9}$$

In the second sub-phase we need to perform send/receive operations $\lceil \frac{P}{r^{w-1}} \rceil - 1$ times, where we send all the blocks with the 1st digit of the radix-r representation of their blocks IDs is equal to z, where $1 \leqslant z \leqslant r-1$. Therefore, in the second sub-phases each processor must send/receive the following data size

$$DS_4 = \frac{\sqrt{N}}{P} \times \frac{\sqrt{N}}{P}(P - r^{\lfloor \log_r P \rfloor}). \tag{10}$$

The total overhead for the second sub-phase is

$$H_2(r,P,N) = \left(\left\lceil \frac{P}{r^{w-1}} \right\rceil - 1\right)Ts + \frac{\sqrt{N}}{P} \times \frac{\sqrt{N}}{P}\left(P - r^{\lfloor \log_r P \rfloor}\right)Tw. \tag{11}$$

Hence, the total overhead is the sum of (9) and (11). If we assume that $r = 2$, both $\frac{P}{r}$ and $\log_2 P$ are integers (e.g., $P = 8$), then from (6) the overhead of the radix-2 algorithm can be found as:

$$H_{radix-2}(N, P) = \left(Ts + \frac{N}{2P}Tw\right) \log_2 P, \tag{12}$$

which is the same as (2).

For the case of $r = 4$, if we again assume that $\frac{P}{r}$ is an integer, and $P \leqslant r^2 = 16$ (i.e., $w = \lceil \log_4 P \rceil = 2$), then by summing (9) and (11), the total overhead of the radix-4 algorithm can be found as

$$H_{radix-4}(N, P) = \left(\left\lceil \frac{P}{4} \right\rceil + 2\right) Ts + \frac{N}{P^2}\left(\frac{7}{4}P - 4^{\lfloor \log_4 P \rfloor}\right)Tw. \tag{13}$$

From (12) and (13), we can determine the following threshold N_{th}:

$$N_{th} = \frac{\left(\left\lceil \frac{P}{4} \right\rceil - \log_2 P + 2\right) Ts}{\left(\frac{\log_2 P}{2P} - \frac{7}{4P} + \frac{4^{\lfloor \log_4 P \rfloor}}{P^2}\right) Tw} \tag{14}$$

such that

$$H_{radix-2}(N_{th}, P) = H_{radix-4}(N_{th}, P). \tag{15}$$

If the data size is greater than the threshold N_{th}, then the radix-4 algorithm will have less overhead, and vice versa. For the parallel system with eight processors ($P = 8$) and *Ts* = 300 *Tw*, from (14), we have $N_{th} = 9600$. Therefore, we expect the radix-4 algorithm will outperform the radix-2 algorithm for a data size $N > 9600$.

Next, we consider a special case where $r = P$. We have $w = \lceil \log_r P \rceil = \lceil \log_P P \rceil = 1$. Consequently, there is only one sub-phase (due to $w = 1$ digit) for this special case. Therefore, we need to use only (9) to find the total overhead for the radix-P algorithm:

$$H(N, P)_{radix-P} = H_1(P, P, N) = (P - 1)Ts + (P - 1)\frac{N}{P}Tw. \tag{16}$$

Similar to the comparison between the radix-2 and radix-4 algorithms, from (13) and (16), we can find the following threshold of data size

$$N_{th}^P = \frac{\left(P - \left\lceil \frac{P}{4} \right\rceil - 3\right) Ts}{\left(\frac{3}{4P} + \frac{1 - 4^{\lfloor \log_4 P \rfloor}}{P^2}\right) Tw} \tag{17}$$

such that

$$H_{radix-P}(N_{th}^P, P) = H_{radix-4}(N_{th}^P, P). \tag{18}$$

block located in processor P_j.

Table 1

The cost of sending/receiving and the memory required by the algorithms

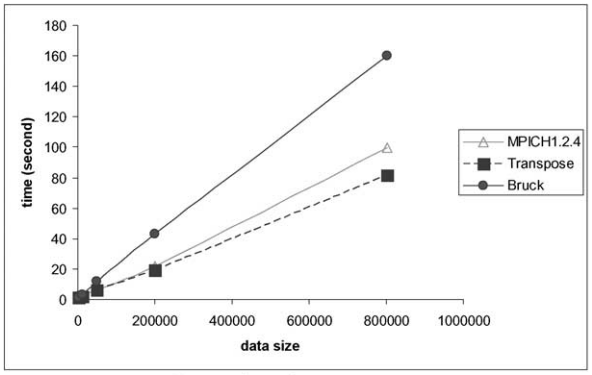
Algorithm	Number of steps	Transmission rate	Memory space
MPICH-1.2.4	$P-1$	$\frac{\sqrt{N}}{P} \times \frac{\sqrt{N}}{P}$	$2\frac{N}{P}$
Bruck (radix = P)	$P-1$	$\frac{\sqrt{N}}{P} \times \frac{\sqrt{N}}{P}$	$\frac{N}{P} + 2\frac{N}{P^2}$
Transpose	$P-1$	$\frac{\sqrt{N}}{P} \times \frac{\sqrt{N}}{P}$	$2\frac{N}{P^2}$
Butterfly	$\log_2 P$	$\frac{\sqrt{N}}{P} \times \frac{\sqrt{N}}{2}$	$\frac{N}{P}$
Generalized Butterfly ($\log_2 P =$ integer)	$\log_2 P$	$\frac{\sqrt{N}}{P} \times \frac{\sqrt{N}}{2}$	$\frac{N}{P}+\frac{N}{P}=2\frac{N}{P}$
Adaptive algorithm	$(r-1)+\lceil \frac{P}{r^{w-1}} \rceil - 1$	$\frac{\sqrt{N}\times\sqrt{N}}{P^2}[(r-1)\frac{P}{r}+(P-r^{\lfloor \log_r P \rfloor})]$	$\lceil \frac{N}{P\times r} \rceil + \frac{N}{P}$

If data size $N > N_{th}^P$, the radix-P algorithm will have less overhead than that of the radix-4 algorithm, and vice versa. For the parallel system with eight processors ($P = 8$) and *Ts* = 300 *Tw*, from (17), we have $N_{th}^P = 19200$. Therefore, we expect the radix-8 algorithm will outperform the radix-4 algorithm for a data size $N > 19200$.

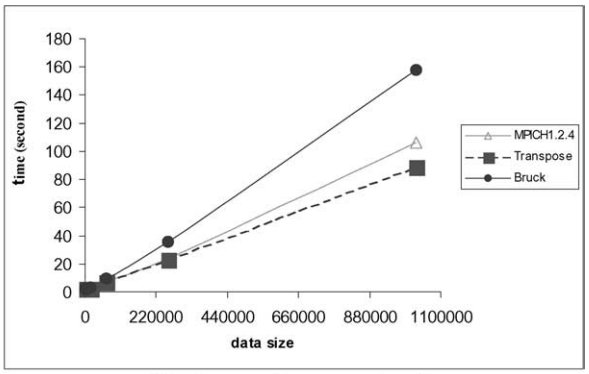
From the foregoing discussions, we know that the index r can be chosen adaptively (between 2 and P) in order to minimize the overhead given the data size N, the number of processors P, the start-up time Ts, and the effective communication time Tw. The total memory space required by the adaptive algorithm is $\lceil \frac{\sqrt{N}\times\sqrt{N}}{P\times r} \rceil = \lceil \frac{N}{P\times r} \rceil$, which is the maximum data size to be sent in both the first sub-phase and the second sub-phase. Moreover, we need an extra buffer for local data movement (rotation) of size $\frac{N}{P}$ (Bruck *et al.*, 1997). Table 1 summarizes the cost of sending/receiving (in terms of number of steps and transmission rate) and the memory space required (including the extra buffering and temporary buffers) for the six algorithms discussed in this paper.

6. Experimental Results

All the above parallel matrix-transpose algorithms were implemented on an 8-node SMP. The size of the input data vector varied between 64 (8×8) and 4 MB (2 KB $\times$2 KB). Each entry of this input vector is a complex number. The MPI function, MPI_Wtime(), was used to measure the time for the parallel code in micro seconds. In order to measure Ts and *Tw*, we send the data between two processors 10000 times. In addition, by changing the data size and measuring the time for each run, we plot the measured time as a function of the data size. The intersection between the time vs. data size curve and the Y-axis is the startup time *Ts* and the slope is *Tw*. The results showed that the *Ts* (24 μsec) is approximately 300 times of *Tw* (0.08 μsec). In order to get accurate results, for each data size we transpose the array 1000 times, and then we measured the overhead associated with each algorithm.



(a) Number of processors P = 7.



(b) Number of processors P = 8.

Fig. 6. Running times of three transpose algorithms for large matrices.

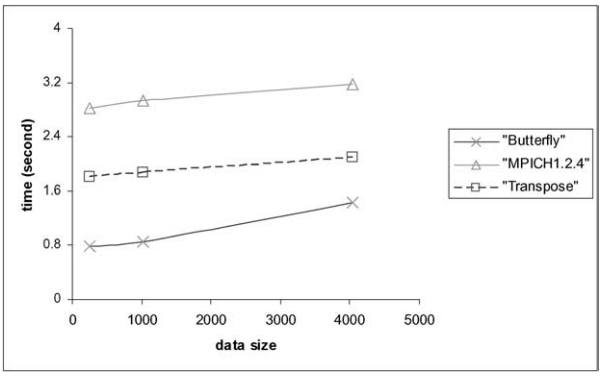
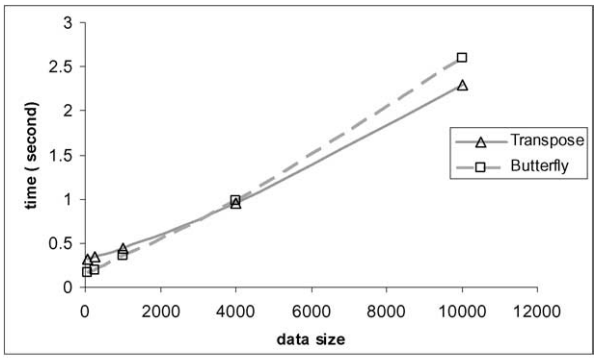
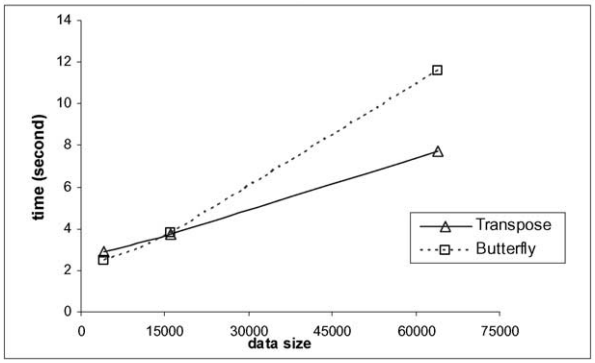


Fig. 7. Comparison between the transpose algorithm, butterfly algorithm, and the MPICH 1.2.4 for small data size using 8 processors.



(a) Number of processors P = 4.



(b) Number of processors P = 8.

Fig. 8. The overheads of the transpose algorithm and the butterfly algorithm.

Fig. 6 shows the running times for the three algorithms for transposing large matrices. Among the three algorithms, the Transpose algorithm is the fastest, whereas the Bruck algorithm is the slowest. This is because of the extra data copying in phase one of the index algorithm and the extra rotation in phase three of the index algorithm.

Fig. 7 shows that the Butterfly algorithm outperforms the Transpose algorithm and the MPICH-1.2.4 algorithm in transposing small data arrays using 8 processors. Fig. 8 shows that, as the data size increases, the Transpose algorithm starts to outperform the Butterfly algorithm whenever the data size $N \geqslant 4000$ (for $P = 4$), or $N \geqslant 15000$ (for $P = 8$).

Fig. 9 illustrates the overhead of the adaptive algorithm. For small data sizes (less than approximately 10000) radix-2 is the best. For data sizes between approximately 10000 and 25000, radix-4 is the best, and for large data sizes radix-8 is the best. The minor difference between these empirically obtained values and the analytical values given in Section 5 is due to the fact that we have neglected the cost of data packing, unpacking and copying.

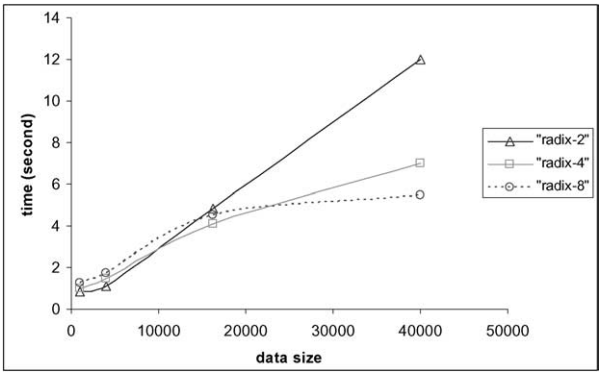


Fig. 9. Comparison between the radix-2, radix-4, and the radix-8 algorithms using 8 processors.

7. Conclusions

We have discussed several efficient algorithms for transposing matrices on symmetric multiprocessors (SMP). These algorithms are efficient since they have less overhead than transposing the array using the built–in collective communication function MPI_Alltoall(). Analysis and experimental results show that, on an SMP with gigaplane bus, the transpose algorithm is best suited for transposing large matrices and the butterfly algorithm is most suitable for transposing small matrices. We have also proposed an adaptive algorithm that can choose the radix adaptively so as to minimize the overhead, based on the data size, number of processors, start-up time, and the effective communication bandwidth.

References

Bailey, D.H. (1990). FFTs in external or hierarchical memory. *Journal of Supercomputing*, **1**(4), 23–35.

Bala, V., J. Bruck, R. Cyper, P. Elustondo, A. Ho, C.-T. Ho, S. Kipnis and M. Snir (1995). CCL: A Portable and tunable collective communication library for scalable parallel computers. *IEEE Transactions on Parallel and Distributed Systems*, **6**(2), 154–164.

Barnnet, M., R. Littlefield, D. Payne and R. van de Geijn (1993). Global combine on mesh architecture with wormhole routing. In *Proceedings of the 7th International Parallel Processing Symposium*.

Barnett, M., L. Shuler, R. van de Geijn, S. Gupta, G. Payne and J. Watts (1994). Inter-processor collective communication library (InterCom). In *Proceedings of the Scalable High-Performance Computing Conference*. pp. 357–364.

Bokhari, S. (1991). *Complete Exchange on the iPSC/860*. Technical Report 91–4, ICASE, NASA Langley Research Center.

Bokhari, S., and H. Berryman (1992). Complete exchange on a circuit switched mesh. In *Proceedings of the Scalable High Performance Conference*. pp. 300–306.

Bruck, J., C. Ho, S. Kipnis, E. Upfal, and D. Weathersby (1997). Efficient algorithms for all-to-all communications in multi-port message-passing systems. *IEEE Transactions on Parallel and Distributed Systems*, **11**(8), 1143–1155.

Calvin, C., and D. Trystram (1996). Matrix transpose of block allocations on torus and de bruijn networks. *Journal of Parallel and Distributed Computing*, **34**(1), 36–49.

Choi, J., J. Dongarra and D. Walker (1995). Parallel matrix transpose algorithms on distributed memory concurrent computers, *Parallel Computing*, **21**(9), 1387–1405.

Claerbout, F.J. (1985). *Imaging the Earth's Interior*, Blackwell Scientific.

Culler, D.E. *et al.* (1995). A Case for networks of workstations: NOW. *IEEE Micro.*

Curlander, J., and R. McDonough (1991). *Synthetic Aperture Radar*. Wiley.

Gropp, W., E. Lusk and A. Skjellum (1994). *Using MPI: Portable Parallel Programming with the Message Passing Interface*, MIT Press. pp. 88–92.

Gupta, A., and V. Kumar (1993). The Scalability of FFT on parallel computers. *IEEE Transactions on Parallel and Distributed Systems*, **4**(8), 922–932.

Jakowatz C.V., P. Thompson, D.E. Wahl, P.H. Eichel and D.C. Ghiglia (1996). *Spotlight-Mode Synthetic Aperture Radar*, Kluwer.

Johnsson, S., and C.-T. Ho (1991). Optimal all-to-all personalized communication with minimum span on boolean cubes. In *Proceedings of the Distributed Memory Computing Conference*. pp. 299–304.

Karlsson, S., and M. Brorsson (1998). A comparative characterization of communication patterns in applications using MPI and shared memory on an IBM SP2. In *Proceedings of 1998 Workshop on Communication, Architecture, and Applications for Network-based Parallel Computing*, Las Vegas. pp. 189–201.

Portnoff, M. (1999). An efficient parallel-processing method for transposing large matrices in place. *IEEE Transactions on Image Processing*, **9**(8), 1265–1275.

Scott, D. (1991). Efficient all-to-all communication patterns in hypercube and mesh topologies. In *Proceedings of the Distributed Memory Computing Conference*. pp. 398–403.

Shan, H., J. Singh, L. Oliker and R. Biswas (2003). Message passing and shared address space parallelism on an SMP cluster. *Parallel Computing*, **29**(2), 167–186.

Sistare, S., R. Vaart and E. Loh (1999). Optimization of MPI collectives on clusters of large-scale SMP's. In *Proceedings of the 1999 ACM/IEEE Conference on Supercomputing*, Portland.

Suh, Y., and K. Shin (2001). All-to-all personalized communication in multi-dimensional torus and mesh networks. *IEEE Transactions on Parallel and Distributed Systems*, **1**(12), 38–59.

Swarztrauber, P.N., and S.W. Hammond (2001). A comparison of optimal FFTs on torus and hypercube multicomputers. *Parallel Computing*, **6**(27), 847–859.

Thakur, R., and W. Gropp (2001). *Improving the Performance of MPI Collective Communication on Switched Networks*, Mathematics and Computer Science Division, Argonne National Laboratory, USA.

Thakur, R., and W. Gropp (2003). Improving the performance of collective operations in MPICH. In *Proc. of the 10th European PVM/MPI Users' Group Conference (Euro PVM/MPI 2003)*. pp. 257–267.

Thakur, R., R. Rabenseifner and W. Gropp (2005). Optimization of collective communication operations in MPICH. *International Journal of High Performance Computing Applications*, **1**(19), 49–66.

Traff, J.L. (2002). Improved MPI all-to-all communication on Giganet SMP cluster. *Lecture Notes in Computer Science*, **2474**, 392–400.

Woo, S.C., J.P. Singh and J.L. Hennessy (1993). *The Performance Advantages of Integrating Message Passing in Cache-Coherent Multiprocessors*. Technical Report, CSL-TR-93-593, Stanford University.

R. Al Na'mneh is a PhD student in the Department of Electrical and Computer Engineering, University of Alabama in Huntsville, USA. His research interests include parallel algorithms and digital signal processing.

W.D. Pan is an assistant professor in the Department of Electrical and Computer Engineering, University of Alabama in Huntsville, USA. He received his PhD degree in electrical engineering from the University of Southern California in 2002, and his MS degree in computer engineering from the University of Louisiana at Lafayette in 1998. His research interests include digital signal and image processing, video coding, and multimedia information assurance.

S.-M. Yoo received the BS degree in economics from Seoul National University, Seoul, Korea, and the MS and PhD degree in computer science from the University of Texas at Arlington in 1989 and 1995, respectively. Since September 2001, he is an associate professor in Electrical and Computer Engineering Department of the University of Alabama in Huntsville, Huntsville, Alabama, U.S.A. Dr. Yoo is the conference chair of ACM Southeast Conference 2004, April, 2004, Huntsville, Alabama, U.S.A. He was the co-program chair of ISCA 16th International Conference on Parallel and Distributed Computing Systems (PDCS-2003), August 2003, Reno, Nevada, USA. Dr. Yoo's research interests include wireless networks, parallel computer architecture, and computer network security. Dr. Yoo is a senior member of IEEE and a member of ACM.

Efektyvus adaptuotas algoritmas simetriniams mikroprocesoriams mažos ir didelės dimensijos matricoms transponuoti

Rami Al NA'MNEH, David W. PAN, Seong-Moo YOO

Straipsnyje dėstoma įvairių mažos ir didelės dimensijos matricų transponavimo algoritmų lyginamoji analizė. Siūlomas algoritmas, kuris gali minimizuoti atliekamų operacijų skaičių, priklausomai nuo tokių uždavinio parametrų, kaip duomenų kiekis, procesorių skaičius ir pan. Pateikti autorių atliktų eksperimentų rezultatai.