Adaptive Matrix Transpose Algorithms for Distributed Multicore Processors

John C. Bowman and Malcolm Roberts

Abstract An adaptive parallel matrix transpose algorithm optimized for distributed multicore architectures running in a hybrid OpenMP/MPI configuration is presented. Significant boosts in speed are observed relative to the distributed transpose used in the state-of-the-art adaptive FFTW library. In some cases, a hybrid configuration allows one to reduce communication costs by reducing the number of MPI nodes, and thereby increasing message sizes. This also allows for a more slab-like than pencil-like domain decomposition for multidimensional Fast Fourier Transforms, reducing the cost of, or even eliminating the need for, a second distributed transpose. Nonblocking all-to-all transfers enable user computation and communication to be overlapped.

1 The matrix transpose

The matrix transpose is an essential primitive of high-performance parallel computing. In contrast to the situation on serial and shared-memory parallel architectures, where the use of memory strides in linear algebra and Fast Fourier Transform (FFT) libraries allows matrices to be accessed in transposed order, many distributed computing algorithms rely on a global matrix transpose. This requires so-called all-to-all communication, where every process must communicate with all of the other processes to swap each matrix column with its corresponding row. For example, multi-dimensional FFT algorithms use a matrix transpose to localize the computation within individual processes. For efficiency, all data corresponding to a given

John C. Bowman
University of Alberta, Edmonton, Alberta, T6G 2G1 Canada, e-mail: bowman@ualberta.ca

Malcolm Roberts
University of Strasbourg, e-mail: malcolm.i.w.roberts@gmail.com

direction must be made available locally for processing with the divide-and-conquer subdivision strategy of the FFT.

Writing an efficient implementation of a matrix transpose is surprisingly difficult. Even on serial and shared-memory machines there are implementation issues. While the storage savings afforded by in-place matrix transposition is often desirable, in-place matrix transposition on a serial machine is nontrivial for nonsquare matrices. For example, transposing $\begin{bmatrix} 0 & 1 & 2 & 3 \\ 4 & 5 & 6 & 7 \end{bmatrix}$ requires that the elements, stored linearly in memory, be permuted according to the cycles (0), $(1,4,2)$, $(3,5,6)$, and (7).

Algorithms for out-of-place matrix transposition are much simpler. Nevertheless, efficient implementation of out-of-place transposes still requires detailed knowledge of the cache size and layout, unless a recursive cache-oblivious algorithm is used [1]. For a review of serial in- and out-of-place matrix transposition algorithms, see [2].

On distributed memory architectures, a number of different matrix transposition algorithms have been proposed. For instance, Choi *et al.* [3] identified, in order of increasing speed, the *rotation*, *direct communication*, and *binary exchange* algorithms. However, the relative performance of these transposition algorithms depends on many factors, including communication latency, bandwidth, network congestion, packet size, local cache size, and network topology. Since it is hard to estimate the relative importance of these factors at compilation time, an adaptive algorithm, dynamically tuned to take advantage of these specific architectural details, is desirable. Al Na'Mneh *et al.* [4] have previously described an adaptive transposition algorithm for symmetric multiprocessors that share a common memory pool and exhibit low-latency interprocess communication. At the other extreme are adaptive algorithms optimized for distributed memory architectures with high latency communication, like those implemented in the widely used FFTW library [5].

Modern high-performance computer architectures consist of a hybrid of the shared and distributed paradigms: distributed networks of multicore processors. The hybrid paradigm marries the high bandwidth low-latency interprocess communication featured by shared memory systems with the massive scalability afforded by distributed computing. In this work, we describe recent efforts to exploit modern hybrid architectures, using the popular MPI message passing interface to communicate between distributed nodes and the OpenMP multithreading paradigm to communicate between the individual cores of each processor.

One of the obvious advantages of exploiting hybrid parallelism is the reduction in communication relative to the pure-MPI approach since messages no longer have to be passed between threads sharing a common memory pool. Another advantage is that some algorithms can be formulated, through a combination of memory striding and vectorization, so that local transposition is not required within a single MPI node (e.g. the multi-dimensional FFT[1]). The hybrid approach also allows smaller problems to be distributed over a large number of cores. This is particularly advantageous for 3D FFTs: the reduced number of MPI processes allows for a more slab-like than pencil-like domain decomposition, reducing the cost of, or even eliminat-

[1] However, the recent availability of serial cache-oblivious in-place transposition algorithms in some cases tips the balance in favour of local transposition, if transposed output is acceptable.

ing the need for, a second transpose. A final reason in favour of the hybrid paradigm is that it is compatible with the modern trend of decreasing memory/core: the number of cores on recent microchips is growing faster than the total available memory. This restricts the memory available to individual pure-MPI processes.

Since the multicore nodes in modern hardware are typically connected to the distributed network via a single socket, message passing typically does not directly benefit from multithreading. However, we show in this work that message passing can benefit from the increased communication block lengths associated with the hybrid model. In addition, the necessary local transposes into and out of the communication buffer can benefit somewhat from multithreading.

The most popular algorithms for transposing an $N \times N$ matrix distributed over P processes are the *direct communication* (all-to-all) and recursive *binary exchange* (butterfly) algorithms. Direct communication transmits each block of data directly to its final destination in the matrix transpose, without any intermediate steps. It is most efficient for $P \ll N$, when the message sizes are large. However, its performance degrades for $P \approx N$, when the message size N^2/P^2 becomes small. To avoid this degradation, the binary exchange algorithm first groups messages together to reduce communication latency, by recursively subdividing the transpose into smaller block transposes.

The FFTW [5] library contains algorithms for both direct communication and binary exchange. However, the FFTW implementation of an adaptive matrix transpose has been optimized for distributed memory architectures with high latency communication. It does not effectively exploit the larger communication block sizes that are available with hybrid decompositions. It is also not multithreaded.

We have developed an efficient hybrid algorithm in the open-source library FFTW++ [6]. It uses direct communication when the message sizes are large and a two-stage block transpose in latency bound cases. In the latter case, we divide the total number of processes P into a blocks each containing b processes. A block transpose expresses an $N \times M$ matrix as an $a \times a$ matrix of $N/a \times M/a$ blocks. Here we only discuss the case where $P = ab$ divides N and M; the general case can be handled with minor technical modifications. The transpose of each $N/a \times M/a$ blocks is computed first, followed by the transpose of the $a \times a$ matrix of blocks. Grouping is used to increase the message sizes in the first transpose from NM/P^2 to aNM/P^2.

The binary exchange algorithm performs recursive block transposes. In practice, only one level (at most) of recursion is actually necessary to increase the communication message sizes. After that single recursion, direct communication typically becomes optimal since the message sizes have now been multiplied by a factor of a in the first phase and b in the second phase. We show theoretically in Section 2 that the communication costs are minimized for $a = b = \sqrt{P}$. In practice, the optimal value will lie somewhere near this value, but may vary due to other considerations, such as cache configuration and network topology.

Block transposition is illustrated for the case $N = M = 8$, $a = 4$, and $b = 2$ in Fig. 1. In (a), the transpose of each 2×2 block is computed. The communications between pairs $(2n, 2n+1)$ of processes are grouped together by first doing an out-of-place local transpose of the data, considered as a 4×2 matrix, on each process.

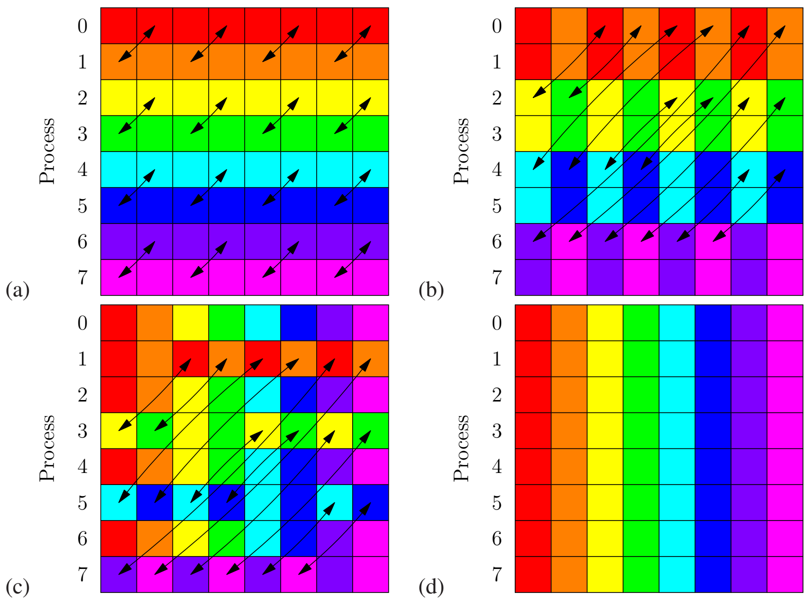

Fig. 1 An 8×8 block transpose over 8 processes for the case $a = 4$, $b = 2$.

The pairs of processors then exchange data, as indicated by the arrows. This is followed by separate all-to-all communications between the even processes (b) and odd processes (c), again grouping the data bound for identical processors, to obtain the transposed matrix in (d).

The block transposition algorithm may be stated as follows:

1. Inner transpose:
   a. Locally transpose $N/b \times b$ matrix of blocks of M/P elements.
   b. All-to-all communicate over teams of b processes, using block size aNM/P^2.
2. Outer transpose:
   a. Locally transpose $N/a \times a$ matrix of blocks of M/P elements.
   b. All-to-all communicate over teams of a processes, using block size bNM/P^2.
3. Locally transpose $N \times M/P$ matrix (optional).

Step 2 is omitted when $a = 1$ (the non-latency bound case); the algorithm then reduces to direct communication. Step 3 can be omitted if local transposition of the output data is not required. We designed our algorithm to use nonblocking communications (`MPI_Ialltoall`, available in MPI 3.0), to allow the user to overlap computation with one or even both communication phases. Overlapping computation with communication has been observed to yield modest speedups (roughly 10%) for computing 3D implicitly dealiased convolutions [6, 7], where a natural parallelism between communication and computation arises.

2 Communication costs

Direct transposition of an $N \times M$ matrix distributed over P processes, involves $P-1$ communications per process, each of size NM/P^2, for a total per-process data transfer of $(P-1)NM/P^2$. For large P, this cost asymptotically approaches NM/P.

For a block transpose, one exploits a factorization $P = ab$ to perform the transform in two stages. First, one groups the processes into a teams of b according to the quotient of their rank and b. Over each team of b processes, one computes the inner transpose of the a individual $N/a \times M/a$ matrices, grouping all a communications with the same source and destination together. This requires $b-1$ messages per process, each of size $(NM/a)/b^2 = aNM/P^2$, for a total per-process data transfer of $(b-1)aNM/P^2$. One then regroups the processes into b teams of a according to their rank modulo b. Over each team of a processes, the outer transpose of the $a \times a$ matrix of $N/a \times M/a$ blocks requires $a-1$ communications per process, each of size $(NM/b)/a^2 = bNM/P^2$, for a total per-process data transfer of $(a-1)bNM/P^2$.

Each process performing a block transpose must therefore send $(a-1)+(b-1) = a + P/a - 2$ messages, for a total per-process transfer of

$$[(b-1)a+(a-1)b]\frac{NM}{P^2} = \left(2P - a - \frac{P}{a}\right)\frac{NM}{P^2}.$$

Let τ_ℓ be the typical latency of a message and τ_d be the time required to send each matrix element. The time required to perform a direct transpose is

$$T_D = \tau_\ell(P-1) + \tau_d \frac{P-1}{P^2} NM = (P-1)\left(\tau_\ell + \tau_d \frac{NM}{P^2}\right),$$

whereas a block transpose requires

$$T_B(a) = \tau_\ell\left(a + \frac{P}{a} - 2\right) + \tau_d\left(2P - a - \frac{P}{a}\right)\frac{NM}{P^2}.$$

Since

$$T_D - T_B = \tau_d\left(P + 1 - a - \frac{P}{a}\right)\left(L - \frac{NM}{P^2}\right),$$

where $L = \tau_\ell/\tau_d$ is the effective communication block length, we see that a direct transpose is preferred when $NM \geq P^2 L$, while a block transpose should be used when $NM < P^2 L$. To determine the optimal value of a for a block transpose, consider

$$T_B'(a) = \tau_\ell\left(1 - \frac{P}{a^2}\right) + \tau_d\left(-1 + \frac{P}{a^2}\right)\frac{NM}{P^2} = \tau_d\left(1 - \frac{P}{a^2}\right)\left(L - \frac{NM}{P^2}\right).$$

For $NM < P^2 L$, we see that T_B is convex, with a global minimum value at $a = \sqrt{P}$ of

$$2\tau_d\left(\sqrt{P} - 1\right)\left(L + \frac{NM}{P^{3/2}}\right) \sim 2\tau_d\sqrt{P}\left(L + \frac{NM}{P^{3/2}}\right), \qquad P \gg 1.$$

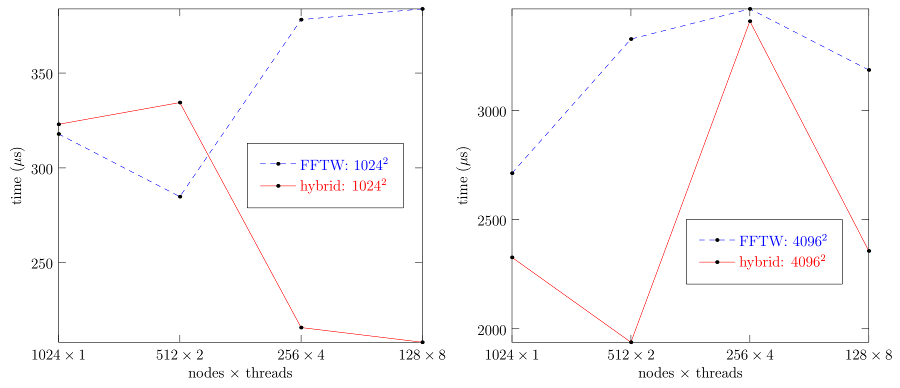

Fig. 2 Wall-clock times for distributed transposes with the FFTW library vs. our implementation.

The global minimum of T_B over both a and P is then seen to occur at $P \approx (2NM/L)^{2/3}$. If the matrix dimensions satisfy $NM > L$, as is typically the case, this minimum occurs above the transition value $(NM/L)^{1/2}$. For $P \gg 1$, we note that $T_D \sim \tau_d(PL + NM/P)$ has a global minimum of $2\tau_d(NML)^{1/2}$ at $P = (NM/L)^{1/2}$, precisely at the transition between the two algorithms. Provided $NM > L$, the optimal choice of P is thus $(2NM/L)^{2/3}$. On a multicore parallel run over S sockets, with C cores per socket, the optimal number of OpenMPI threads to use is then $T = \min(SC/(2NM/L)^{2/3}, C)$, with $P = SC/T$ MPI nodes. We benchmarked our hybrid implementation against the FFTW transpose for 1024×1024 and 4096×4096 complex matrices on the Dell Zeus C8220 Cluster at the Texas Advanced Computer Center, using $S = 128$ sockets and $C = 8$ cores. In Fig. 2, we see that our implementation typically outperforms FFTW, in some cases by nearly a factor of 2. We measured the value of L to be roughly 4096 bytes for this machine. This predicts that the optimal number of threads is $T = 8$ for Fig. 2 (a) and $T = 2$ for Fig. 2 (b), precisely as observed.

Acknowledgements The authors gratefully acknowledge Professor Wendell Horton for providing access to state-of-the-art computing facilities at the Texas Advanced Computer Center.

References

1. M. Frigo, C.E. Leiserson, H. Prokop, S. Ramachandran, in *Foundations of Computer Science, 1999. 40th Annual Symposium on* (IEEE, 1999), pp. 285–297
2. M. Dow, Transposing a matrix on a vector computer, Parallel computing **21**(12), 1997 (1995)
3. J. Choi, J.J. Dongarra, D.W. Walker, Parallel matrix transpose algorithms on distributed memory concurrent computers, Parallel Computing **21**(9), 1387 (1995)
4. R. Al Na'mneh, W.D. Pan, S.M. Yoo, Efficient adaptive algorithms for transposing small and large matrices on symmetric multiprocessors, Informatica **17**(4), 535 (2006)
5. M. Frigo, S.G. Johnson, The design and implementation of `FFTW3`, Proceedings of the IEEE **93**(2), 216 (2005)

6. J.C. Bowman, M. Roberts. `FFTW++`**:** A fast Fourier transform C^{++} header class for the `FFTW3` library. http://fftwpp.sourceforge.net (May 6, 2010)
7. J.C. Bowman, M. Roberts, Efficient dealiased convolutions without padding, SIAM J. Sci. Comput. **33**(1), 386 (2011)