Array Transposition in SSD

David H. Bailey

October 3, 1989

Abstract

One obstacle to running very large two- and three-dimensional codes on the Cray X-MP and Y-MP systems is to efficiently perform array transpositions using SSD storage. This article discusses how such transpositions can be performed by means of algorithms that feature exclusively unit stride, long vector transfers between main memory and SSD, and which only require a single pass through the data (provided sufficient main memory buffers are available).

The author is with the Numerical Aerodynamic Simulation (NAS) Systems Division at NASA Ames Research Center, Moffett Field, CA 94035.

Introduction

The limited main memory on X-MP and Y-MP (compared to the Cray-2) has led some users of very large 2-D and 3-D simulation codes to consider using the SSD (solid state device) memory system. For some of these codes, not all of the necessary arrays need be present in main memory at any one time, and so such codes can run fairly efficiently using SSD. For other codes, however, it is not convenient to segregate the data in this fashion.

One situation that is particularly challenging to handle in SSD is a very large 2-D or 3-D array that must be accessed by each dimension. Accessing such arrays in SSD by the first dimension is not difficult, since in Fortran this is the natural storage order. Accessing by the second or third dimension amounts to accessing the array with a very large stride, which renders SSD access inefficient. This is because SSD access, like disk access, is only efficient when done with large blocks of contiguous data.

One way to solve this problem is to perform an array transposition after accessing the array by the first dimension. Then subsequent computation can again be done with large vector, unit stride data accesses. Depending on the circumstances, additional array transpositions may be necessary, including one to restore the array to its original ordering. What this means is that if an efficient means can be found to transpose an array in an external memory such as SSD, then large 2-D and 3-D codes could be executed on the X-MP and Y-MP without prohibitive I/O cost. This article will describe two methods for array transposition in an external memory such as SSD. Each of these methods employs exclusively large block, contiguous data transfers between main and external memory, and each requires only a single read-write pass through the data, provided sufficient main memory buffer space is available.

Fraser's Algorithm

Probably the most efficient algorithm currently known to transpose data in external storage is due to Fraser [1, 2]. A particularly attractive aspect of this algorithm is that it can easily be tuned for maximum efficiency on a given system. It is easier to exhibit an example of Fraser's algorithm than to precisely state it. Suppose one wishes to transpose a $n_1 \times n_2$ matrix on an external random access dataset into a $n_2 \times n_1$ matrix, where $n_1 = 128 = 2^7$ and $n_2 = 256 = 2^8$. Suppose also that the size of an efficient I/O block is $64 = 2^6$ (on a Cray SSD system it is 512). Finally, suppose that two main memory buffers of size $512 = 2^9$ are available, and that an external scratch dataset of size 2^{15} is available. Let the notation (14 13 12 ... 2 1 0) denote the binary digit positions in the binary expansion of an index in the 2^{15}-long input array. Then the steps required to transpose this array can be compactly presented in Table 1.

The notation at the beginning of each line in Table 1 indicates the source of the data in each operation: E1 denotes external dataset number 1, M2 denotes main memory buffer number 2, etc. Note that the transfers between external memory and main memory only alter locations 6 through 14, and leave locations 0 through 5 unchanged (i.e., 64-long contiguous blocks are preserved), and that transfers between two main memory buffers only alter locations 0 through 8 (i.e. only affect data within a single 512-long main memory

	Permutations of Disk Blocks														
							Permutations in Main Memory								
E1	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
M1	7	6	14	13	12	11	10	9	8	5	4	3	2	1	0
M2	7	6	14	13	12	11	5	4	3	2	1	0	10	9	8
E2	7	6	5	4	3	14	13	12	11	2	1	0	10	9	8
M1	7	6	5	4	3	14	13	12	11	2	1	0	10	9	8
M2	7	6	5	4	3	14	2	1	0	13	12	11	10	9	8
E1	7	6	5	4	3	2	1	0	14	13	12	11	10	9	8

Table 1: Transposing a 128×256 Array Using Fraser's Algorithm

buffer).

The first step, from external to main, involves fetching contiguous blocks of size 64 from disk with a block stride of four (i.e. fetch the first 64-long block, skip three blocks, fetch the fifth 64-long block, etc.). The first step is done in batches of 8 blocks, so that 512 words are fetched to one of the main memory buffers before proceeding. The second step, which is performed between the two main memory buffers, is to transpose the resulting 512-long array, considered as a 64×8 matrix, into a 8×64 matrix. In the third step, the eight 64-long blocks in the main memory buffer are stored out to external memory, this time with a block stride of eight. This completes one pass through the external data set.

In the next pass, eight contiguous 64-long blocks are fetched into main memory, and the resulting 512-long array is transposed in a block fashion that preserves 8-long contiguous sections. Finally, the resulting 64-long blocks are stored back to external memory, again in a manner that achieves a certain block permutation. The array has now been transposed in just two passes. With an adjustment of the parameters (for example, with a block size of 32 and a memory buffer size of 1024), the transposition could be achieved in a single pass.

The author has implemented Fraser's algorithm on a Cray Y-MP running UNICOS, using SDS space in SSD. The resulting implementation appears to be very effective, as can be judged by the performance figures in Table 2. In addition to the CPU times listed, some I/O time was logged as "system time". As in many applications that use I/O, this system time can vary wildly from run to run, depending on swapping and other system activity. When the system I/O is not excessively busy, this time is generally about the same as the user CPU time listed in the table.

The Block Interchange Scheme

Although Fraser's algorithm is normally performed with a separate scratch dataset as indicated above, it is possible to dispense with the scratch dataset if one is willing to relax the requirement for a physically transposed array (by utilizing pointers to index to the external data blocks). If such indexing is undesirable, there is another method that can

Size of matrix	Memory buffer	No. of Passes	CPU time
$2^{10} \times 2^9$	2^{14}	2	0.0449
	2^{18}	1	0.0177
$2^{10} \times 2^{10}$	2^{14}	2	0.0886
	2^{18}	1	0.0352
$2^{11} \times 2^9$	2^{14}	2	0.0885
	2^{18}	1	0.0343
$2^{11} \times 2^{10}$	2^{14}	2	0.1767
	2^{18}	1	0.0685
$2^{11} \times 2^{11}$	2^{14}	2	0.3506
	2^{18}	1	0.1353
$2^{12} \times 2^9$	2^{14}	2	0.1760
	2^{18}	1	0.0678
$2^{12} \times 2^{10}$	2^{14}	2	0.3518
	2^{18}	1	0.1349
$2^{12} \times 2^{11}$	2^{14}	2	0.6997
	2^{18}	1	0.2696

Table 2: Performance of Fraser's Algorithm on the Cray Y-MP

$$\begin{bmatrix} A_{11} & A_{12} & A_{13} & A_{14} \\ A_{21} & A_{22} & A_{23} & A_{24} \\ A_{31} & A_{32} & A_{33} & A_{34} \\ A_{41} & A_{42} & A_{43} & A_{44} \end{bmatrix}^t = \begin{bmatrix} A_{11}^t & A_{21}^t & A_{31}^t & A_{41}^t \\ A_{12}^t & A_{22}^t & A_{32}^t & A_{42}^t \\ A_{13}^t & A_{23}^t & A_{33}^t & A_{43}^t \\ A_{14}^t & A_{24}^t & A_{34}^t & A_{44}^t \end{bmatrix}$$

Table 3: The Block Interchange Algorithm for Transposing Square Arrays

be performed in-place — it does not require a scratch dataset — although it can only be used in certain cases.

Consider first the case where $n_1 = n_2$, so that the matrix is square. In that case a block interchange technique can be used to transpose the array in a single pass, in place. This can be done by simply considering the external $n_1 \times n_2$ matrix to be decomposed into square blocks of size b on a side, where b is the block size of an efficient I/O operation. The square blocks down the diagonal can be transposed simply by fetching the blocks one at a time into main memory, transposing them using any efficient main memory scheme, and storing the resulting matrices back in the same locations. The off-diagonal square blocks can be fetched in opposing pairs, transposed in main memory, and then stored back in opposite locations. This scheme is described in Table 3.

Transposing matrices whose dimensions are powers of two in main memory on Cray systems, using the straightforward scheme, results in performance reductions due to memory bank conflicts, especially on Cray-2 systems. However, such arrays can be transposed completely without bank conflicts by fetching and storing opposite diagonals.

When $n_1 \neq n_2$, the above scheme will not work. In cases where $n_1 = an_2$ or $an_1 = n_2$, where a is an integer, a more complicated version of the block interchange algorithm will still perform the transposition in-place, although it requires two passes. In general, however, for non-square matrices and for multidimensional arrays, it is better to use Fraser's algorithm.

Conclusion

Although it may at first seem intuitively "impossible" to do so, large arrays can be transposed in an external storage device such as SSD using exclusively long vector, unit stride data accesses, with only one read-write pass. The algorithms presented in this note have these attributes and are thus vastly more efficient than the straightforward schemes ordinarily employed.

At present, there are no readily available library implementations of these algorithms. Thus it is necessary for a programmer to code them and incorporate them into his or her applications. The author is willing to offer some assistance in such efforts. His address is the following:

David H. Bailey
NASA Ames Research Center
Mail Stop T045-1
Moffett Field, CA 94035, USA
Telephone: 415-694-4410
Internet: dbailey@ew11.nas.nasa.gov

References

1. Fraser, D., "Array Permutation by Index-Digit Permutation", *Journal of the Association for Computing Machinery*, vol. 23 (1976), p. 298 - 309.

2. Swarztrauber, P. N., "Transposing Large Arrays in Extended Memory", in *Multiprocessing in Meterological Models*, G. R. Hoffmann and D. F. Snelling eds., Springer-Verlag, 1988, p. 283 - 287.