In-Place Transposition of Rectangular Matrices

Fred G. Gustavson[1] and Tadeusz Swirszcz[2]

[1] T. J. Watson Research Center, Yorktown Heights, NY 10598, USA,
fg2@us.ibm.com
[2] Faculty of Mathematics and Information Science, Warsaw University of Technology, Warsaw, Poland, Europe,
swirszcz@mini.pw.edu.pl

Abstract. We present Algorithms for In-Place Rectangular Transposition that are efficient. Some of the Algorithms are new. A main result is a new In-Place Transposition algorithm that uses no additional storage. Performance result are given and they are compared to Out-of-Place Transposition algorithms.

1 Description of In-Place Transpositions for Rectangular Matrices

We present algorithms that require little or no extra storage to transpose a m by n rectangular (non-square) matrix A in place. We assume that A is stored in the standard storage format of the Fortran and C programming languages. We remark that many other programming languages use this same standard format for laying out matrices. The first author has produced algorithms that require a bit vector of length mn extra storage to do the transpose inplace. The second author made a key observation that the bit vector could be removed; however, at the cost of extra computer operations. Hence this work exhibits an example of a classic programming principle: "Reducing storage at the cost of increasing computer operations".

Matrix A^T is an n by m matrix. Now both A and A^T are simultaneously represented by either A or A^T. Also, in Fortran, A is stored stride one by column and A^T is stored stride one by rows. A given application determines which format is best and frequently, for performance reasons, both formats are deemed necessary.

Currently, in-place transpose algorithms are not present in libraries for Dense Linear Algebra when $m \neq n$.

Our algorithms are based on following the cycles of a permutation P of length $q = mn - 1$. This permutation P is defined by the mapping of A_{ij} of A^T_{ij} that is induced by the standard storage layouts of Fortran and C. Thus, if one follows a cycle of P then one must eventually return to the beginning point of this cycle of P. By using a bit vector one can tag which cycles of P have been visited and then a starting point for each new cycle is easily determined. The cost of this algorithm is easily seen to be $O(q)$ which is minimal.

Now, we go further and remove the bit vector. Thus, we need a method to distinguish between a new cycle and a previous cycle (the original reason for the bit vector). Our key observation is that every new cycle has a starting value that is a minimum. If we transverse a proposed new cycle and we find an iterate whose value is less than the current starting value we know that the cycle we are generating has already been generated. We can therefore abort and go onto the next starting value. On the other hand, if we return to the original starting value, thereby completing a cycle, where every iterate is larger than this starting value we are assured that a new cycle has been found and we can therefore record it. We now give this basic algorithm.

```
ALGORITHM ITP (m,n,A)
DO cnt = 1, mn-2
  k = P(cnt)
  DO WHILE (k > cnt)
    k = P(k)
  END DO
  IF (k = cnt) then
    Transpose that part
    of A which is in the
    new cycle just found
  ENDIF
END DO
```

Clearly, the algorithm IPT just described contains redundant computation in the inner while loop. We have been able to remove much of the redundancy. In this regard, here are some observations. Many times the basic algorithm IPT completes while cnt is still on the first column of A; i.e. before cnt reaches m. However, the existence of small cycles causes cnt to become much larger before IPT completes. Other observations will now follow.

A standard programming technique called BABE (Burn At Both End) can be used to speed up the inner while loop of ITP. Use of BABE allows one to use functional parallelism, [1]. More importantly, no matrix elements are accessed during the inner while loop and so no cache misses occur. In fact the inner while loop consist entirely of register based fixed point instructions and hence the inner loop will perform at the peak rate of any processor. The actual transposition part of IPT runs relatively slow in comparison to the inner loop processing. Any cycle of P usually accesses the elements of A in a completely random fashion. Hence, a cache miss almost always occurs for each element of the cycle and thus the whole line of the element is brought into cache. The remaining elements in the line usually never get used. On the other hand, an out-of-place transpose algorithm allows one to bring the elements of A into and out of cache in a fairly structured way. This again illustrate the principle of "trading storage at the cost of increasing performance".

One can compute the number of one cycles in P for any matrix A. This is a gcd computation and there are $1 + gcd(m-1, n-1)$ one cycles. Since non-trivial one cycles always occur in the interior of A; ie, for large values of cnt, knowing their number can drastically reduce the cost of running IPT. To see this, note that the outer loop of IPT runs from 1 to $mn-2$. If one records the total cycle count tcc then one can leave the outer loop when tcc=mn. We now call the modification of IPT that uses the gcd logic and tcc count our basic algorithm IPT.

Let $\bar{k} = P(k)$ and $l = q - k$. One can show that $P(l) = q - \bar{k}$. Thus, let cnt generate a cycle and suppose that iterate $l = q - cnt$ does not belong to this cycle. Then $q - cnt$ also generates a cycle. This result shows that a duality principle exists for P. The criterion for cnt is $j > cnt$ for every j in the cycle. For $q - cnt$ the criterion is $j < q - cnt$ for every j in the companion cycle.

The value q is the key to understanding P and hence our algorithm. Let k be the storage location of $A_{ij}, 0 < k < q$. One can show $P(k) = mod(nk, q)$. $P(k)$ is the storage location of A_{ji}. Suppose d is a divisor of q. The, every iterate of d also divides q. Hence, when cnt starts at d one can alternatively look at $\bar{k} = mod(nk, q/d)$ where cnt begins at 1. So, we can partition $0 < k < q$ into a disjoint union over the divisors of d of q. For each d, we can apply a suitable variant of algorithm IPT, called algorithm ITP1, as a subroutine that is called for each d of q. This master algorithm drastically reduces the operation count of the inner while loop of algorithm ITP.

1.1 Mathematical Description of Algorithm IPT

In Fortran the (i,j) element of A is stored at memory location $k = i + j * m$. Likewise, the (i,j) element of A^T is stored at memory location $k = i + j * n$. In both cases k takes on values $0, 1, \ldots, q$ where $q = mn - 1$.

Let P be the permutation of elements of the set $\{0, 1, \ldots, q\}$ determined by the transposition of A. One can show that $P(k) = mod(kn, q)$ for $k = 0, 1, \ldots, q$ where $P(0) = 0$ and $P(q) = q$. Now $P = \sigma_1 \circ \sigma_2 \circ \ldots \circ \sigma_p$ where we have excluded one-element cycles.

Let s_i be the smallest element of the cycle σ_i. Since disjoint cycles are commutative, we can assume that $s_1 < s_2 < \ldots < s_p$. Now 0 is a one cycle so $s_1 = 1$. Each cycle is a sequence $\sigma_i = (s_i, P(s_i), \ldots, P^{\lambda_i - 1}(s_i))$, where λ_i is the smallest positive integer such that $P^{\lambda_i}(s_i) = s_i$. The number λ_i is the *length* of the cycle.

The first nontrivial cycle is σ_1 This is the longest cycle.

Any cycle looks as follows: $\sigma = \big(s,\ mod(sm, q),\ \ldots,\ mod(sm^{\lambda_s - 1}, q)\big)$. It can be proved that λ_s is a divisor of λ_1.

It follows immediately that $s_i < P^l(s_i)$ for $l = 1, \ldots \lambda_i - 1$. Hence a number $1 < k < q - 1$ is in a cycle σ_i, where $s_i < k$ if and only if there exists a number l such that $P^l(k) < k$. This allows us to skip previously obtained cycles.

It can be proved that the number of cycles with the length equal to 1 is $d+1$, where d is the greatest common divisor of $m-1$ and $n-1$. So if we want to transpose a matrix we can act as follows:

1° We perform the first cycle σ_1 starting from 1 and we evaluate its length λ_1.

2° For any integer $k > 1$ we verify whether $P(k) = k$ or there exists a number l such that $P^l(k) < k$. If the answer is 'yes', we skip k. Otherwise we perform the cycle starting from k and we evaluate its length. Then we add obtained length to the sum of all previously obtained lengths. If the sum reaches $q - d$, the transposition is finished.

References

1. R. C. Agarwal, F. G. Gustavson, M. Zubair. Exploiting functional parallelism of POWER2 to design high-performance numerical algorithms. *IBM Journal of Research and Development*, Vol. 38, No. 5, Sep. 1994, pp. 563,576.