Algorithms for In-Place Matrix Transposition

David W. Walker
School of Computer Science & Informatics
Cardiff University, UK

Fred G. Gustavson
IBM T. J. Watson Research Centre, emeritus
Umea University

Matrix Transposition

- Important in linear algebra computations
- Example: finding Fourier transform of 3D array. Do this by:
  - 1D transform wrt each dimension in turn
  - Transpose wrt 2 of the dimensions after each 1D transform
  - Makes efficient use of cache
  - Facilitates parallelization

Column-Major Ordering

A

0	5	10
1	6	11
2	7	12
3	8	13
4	9	14

A^T

0	1	2	3	4
5	6	7	8	9
10	11	12	13	14

- In CMO for matrix A elements are stored:

  0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14
- In CMO for transpose of A elements are stored:

  0, 5, 10, 1, 6, 11, 2, 7, 12, 3, 8, 13, 4, 9, 14

Permutations

- Transposition of an nxm matrix can be viewed as a permutation, P, of the indices 0, 1, …, q, where q = nm-1.
- If P(k) is the index of the element originally at location k, then k=0 and k=q are invariant under P, and for 0<k<q:

$$P(k) = (km) \bmod (q)$$

Permutations and Cycles

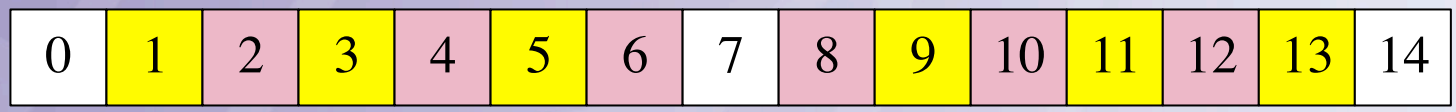
k	P(k)
0	0
1	3
2	6
3	9
4	12
5	1
6	4
7	7
8	10
9	13
10	2
11	5
12	8
13	11
14	14

- If P is applied repeatedly we end up at the starting index, i.e., $k=P^c(k)$.
- This defines a *cycle* of length c.
- Elements k=0, 7, 14 are cycles of length 1.
- (3,9,13,11,5,1) is a cycle.
- (6,4,12,8,10,2) is a cycle.

Cycle-Based In-Place Transposition

- Consider cycle (3,9,13,11,5,1).
- Make a copy, t, of A[1], and then move A[5] to A[1], A[11] to A[5], A[13] to A[11], A[9] to A[13], A[3] to A[9]. Then move t to A[3].
- Repeat the procedure for cycle (6,4,12,8,10,2).
- This completes the transposition.

Cycle-Based Transposition of 5x3 Matrix



- Requires O(nm) data moves.
- Requires O(1) additional memory location.

Pros and Cons

- Cycle-based matrix transposition is elegant and simple to implement.
- Has irregular memory access patterns so does not use cache efficiently.

Swap-Based In-Place Matrix Transposition

- Algorithm based on paper of Tretyakov and Tyrtyshnikov (J. of Complexity, vol. 25, pp. 377-384, 2009) for transpose of nxm matrix with n≥m.
- Basic idea is to partition matrix into sub-matrices that are easier to transpose.
- Partitioning naturally gives rise to parallelism.

Main Phases of TT Algorithm

1. Partition into sub-matrices based on base-m representation of n.
2. Transpose each sub-matrix.

Partition Phase for Matrix A

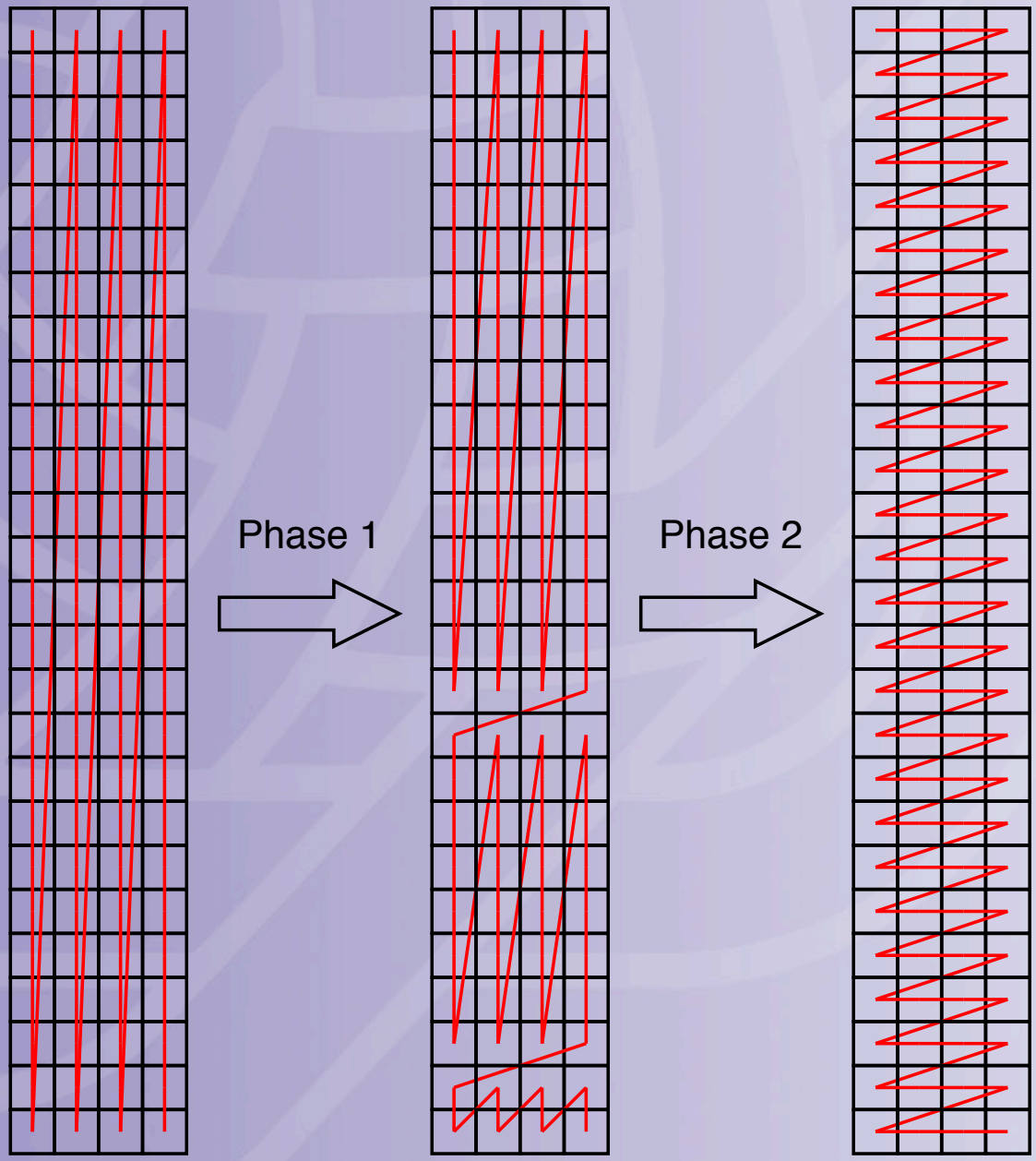
- Write n as as base-m number:

$$n = n_k m^k + n_{k-1} m^{k-1} + \ldots + n_1 m + n_0$$

- Partition A into k+1 contiguous sub-matrices:

  $A_k, A_{k-1}, \ldots, A_1, A_0$

  where A_k consist of the first $n_k m^k$ rows of A; A_{k-1} the next $n_{k-1} m^{k-1}$ rows of A; and so on.



26x4 matrix

- $26=1x4^2+2x4+2$
- So there are 3 sub-matrices:

  16x4

  8x4

  2x4
- Each sub-matrix is transposed independently.
- Need way to transpose pm^qxm matrix.

Partitioning Algorithm

- Can partition A in-place using k applications of the unshuffle operation, resulting in k+1 sub-matrices.
- Unshuffle just uses swaps.
- The partitioning process exposes parallelism.

Unshuffle Operation

- The unshuffle operation takes a shuffled sequence of items and unshuffles them:

$$a_1 b_1 a_2 b_2 \ldots a_m b_m \rightarrow a_1 a_2 \ldots a_m b_1 b_2 \ldots b_m$$

where each a_i is a contiguous vector of ℓ_a items, and each b_i is a contiguous vector of ℓ_b items.

- Each $a_i b_i$ pair represents one column of the matrix.
- The unshuffle operation partitions the matrix over rows.

Partitioning a 26 by 4 Matrix

- Unshuffle 1:

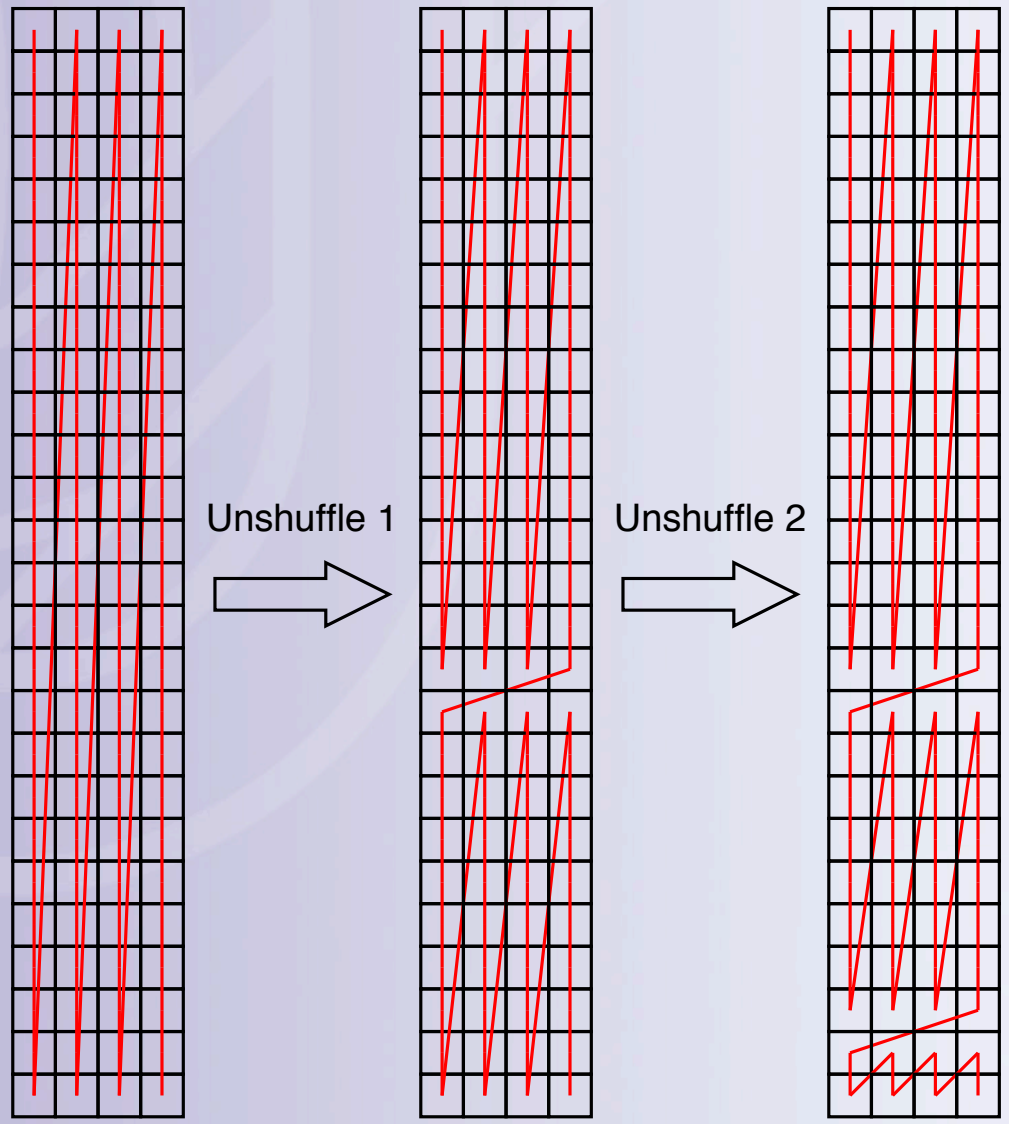
  $\ell_a = 16, \ell_b = 10$

  26x4 partitioned as16x4 and 10x4.

- Unshuffle 2:

  $\ell_a = 8, \ell_b = 2$

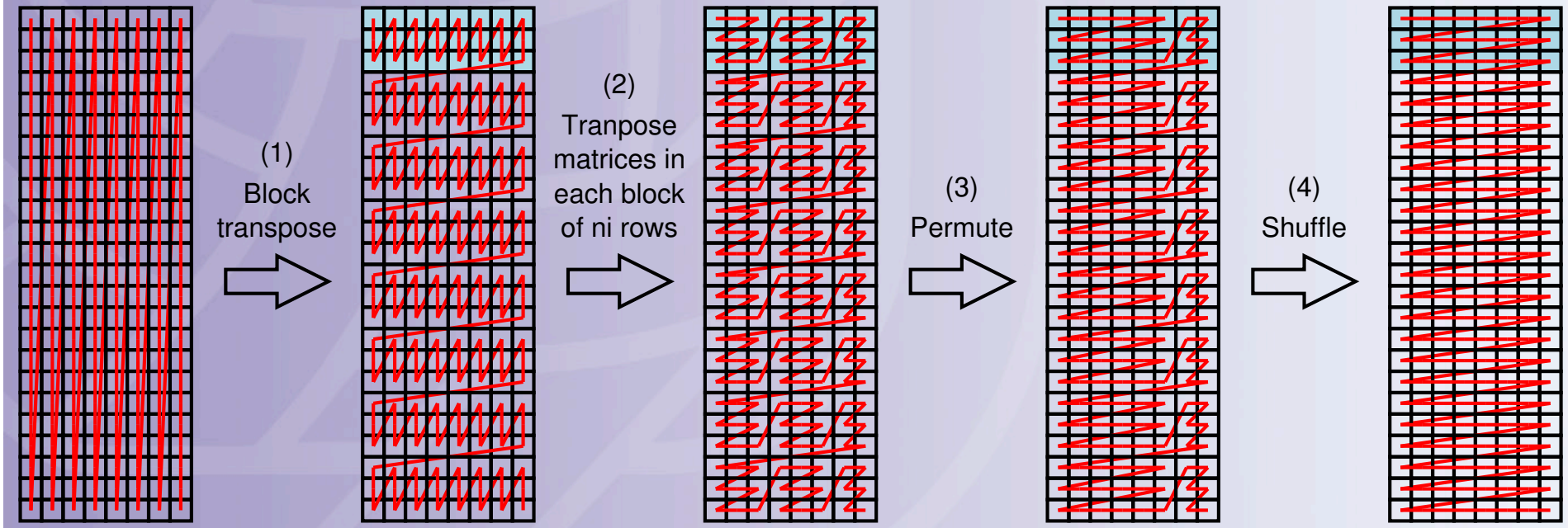
  10x4 partitioned as 8x4 and 2x4.



Transposing a pm^q by m Matrix

- New algorithm by TT, although q=1 case is known and uses half the swaps.
- Can be done in-place using swaps.
- TT algorithm requires additional m memory locations to store a permutation vector, P.
- P is used in the shuffle and unshuffle operations, and in a vector permute operation.

Case: m = 8, p = 3, q = 1



- The shuffle operation in stage 4 un-partitions, or joins, sub-matrices over columns.

Avoiding Permutations

- The permutation algorithm of TT is actually a swap-based form of cycle following.
- Can avoid O(m) memory cost by:
  - Perform the permutation in step 3 with multiple shuffle or unshuffle operations.
  - Perform shuffle and unshuffle operations using divide-and-conquer algorithm.

Divide-And-Conquer Unshuffle

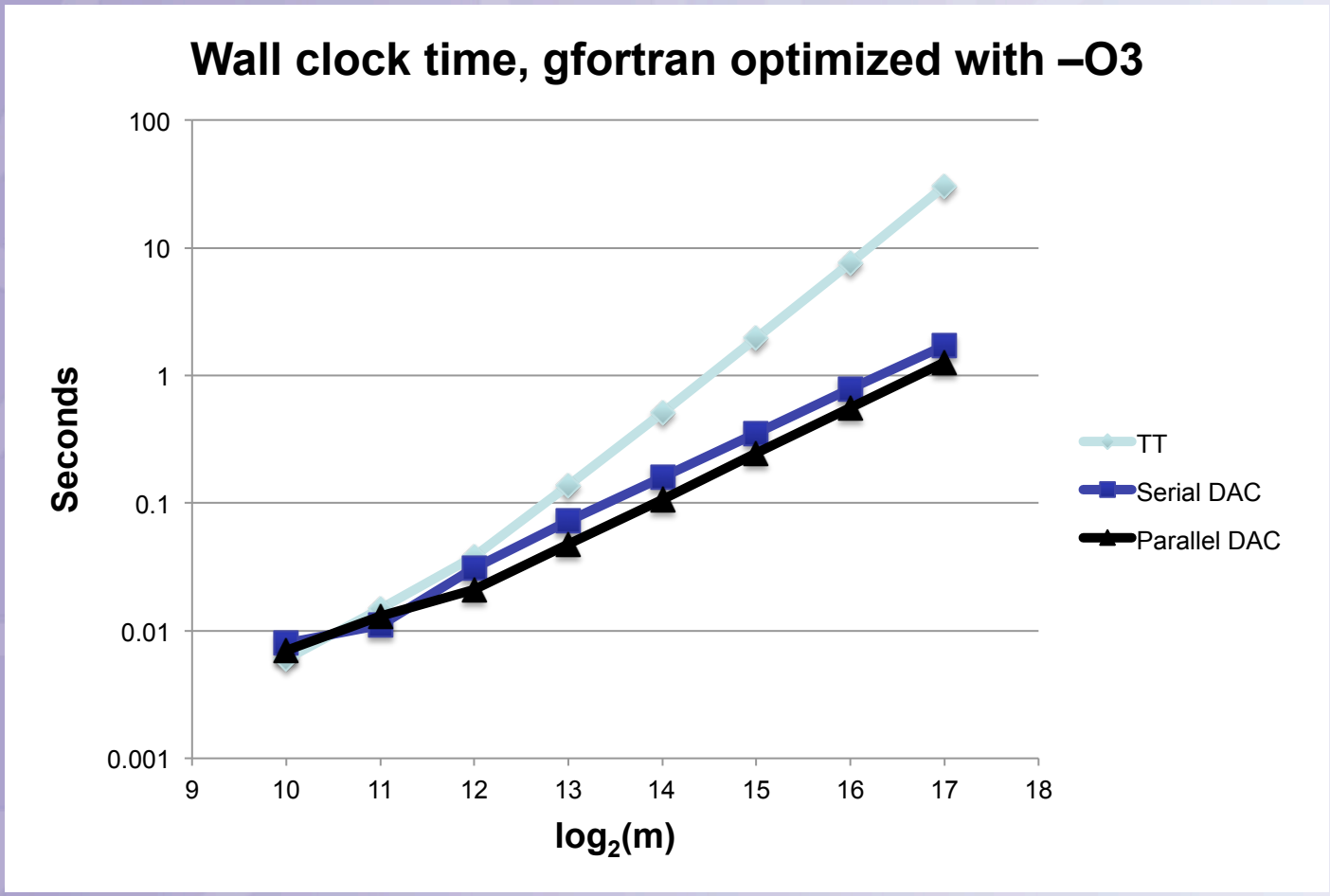
- Suppose m=8

1. Group as: $(a_1b_1a_2b_2)(a_3b_3a_4b_4)(a_5b_5a_6b_6)(a_7b_7a_8b_8)$
2. Swap first b vector with second a vector in each group: $(a_1a_2b_1b_2)(a_3a_4b_3b_4)(a_5a_6b_5b_6)(a_7a_8b_7b_8)$
3. Re-group as: $(a_1a_2b_1b_2a_3a_4b_3b_4)(a_5a_6b_5b_6a_7a_8b_7b_8)$
4. Swap first pair of b's with second pair of a's in each group: $(a_1a_2a_3a_4b_1b_2b_3b_4)(a_5a_6a_7a_8b_5b_6b_7b_8)$
5. Re-group as: $(a_1a_2a_3a_4b_1b_2b_3b_4a_5a_6a_7a_8b_5b_6b_7b_8)$
6. Swap first set of 4 b's with second set of 4 a's: $(a_1a_2a_3a_4a_5a_6a_7a_8b_1b_2b_3b_4b_5b_6b_7b_8)$

Unshuffle Trade-Offs: TT vs DAC

ℓ_a=1152, ℓ_b=640, number of swaps			
m	TT	DAC serial	DAC parallel
32	93,122	133,120	51,584
64	191,106	319,488	104,832
128	398,604	745,472	211,328
256	817,686	1,703,936	424,320
512	1,708,336	3,833,856	850,304
1024	3,682,140	8,519,680	1,702,272
2048	8,427,462	18,743,296	3,406,208
4096	21,057,168	40,894,464	6,814,080
8192	58,902,826	88,604,672	13,629,824
16384	184,926,814	190,840,832	27,261,312
32768	638,302,408	408,944,640	54,524,288

Unshuffle: ℓ_a=1152, ℓ_b=640



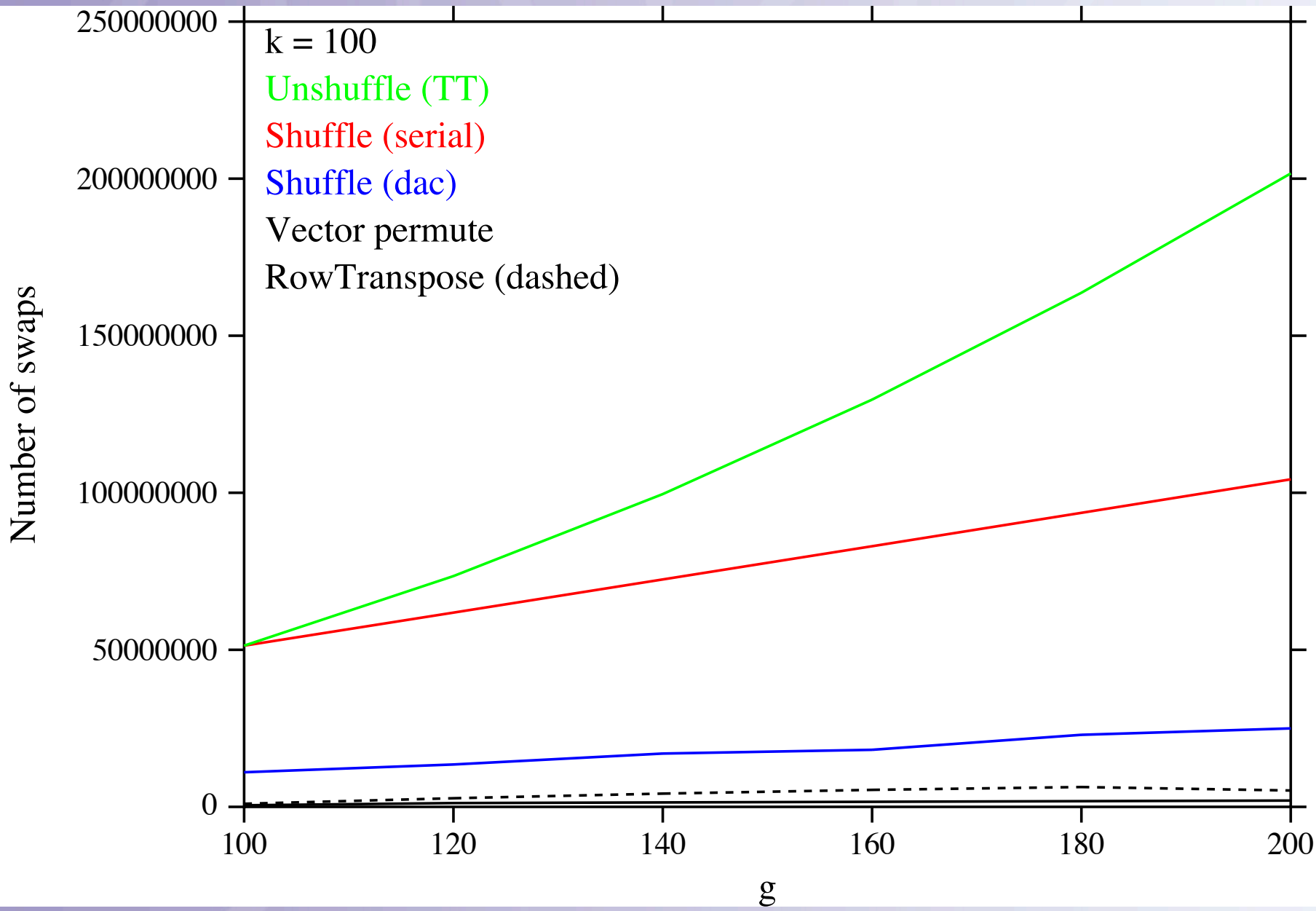
- Results on multicore system with 12 Intel Xeon E5649 CPUs. DAC parallelized with OpenMP

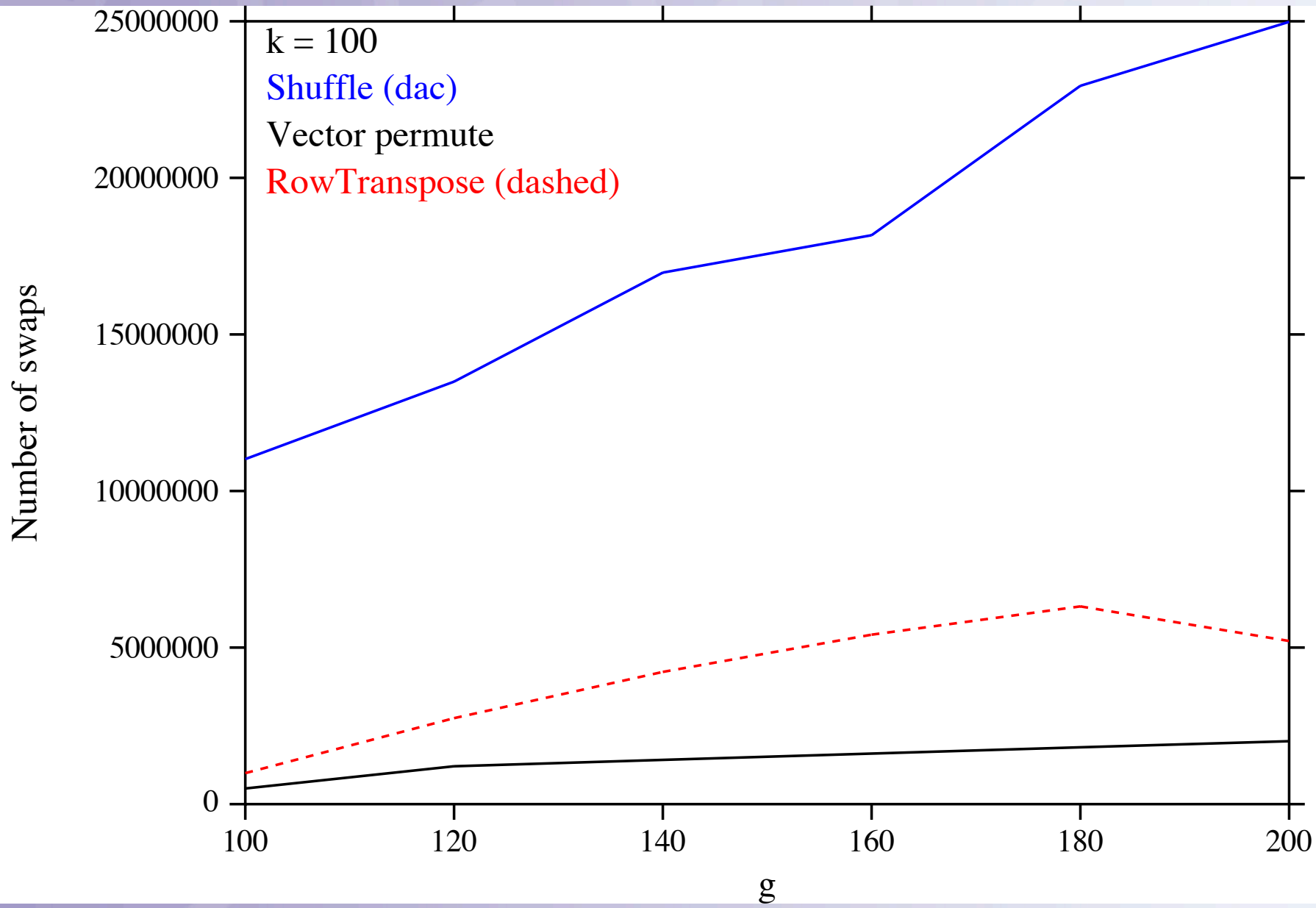
Replace vector permute with multiple unshuffle operations

- Perform the permutation in step 3 with multiple shuffle or unshuffle operations.

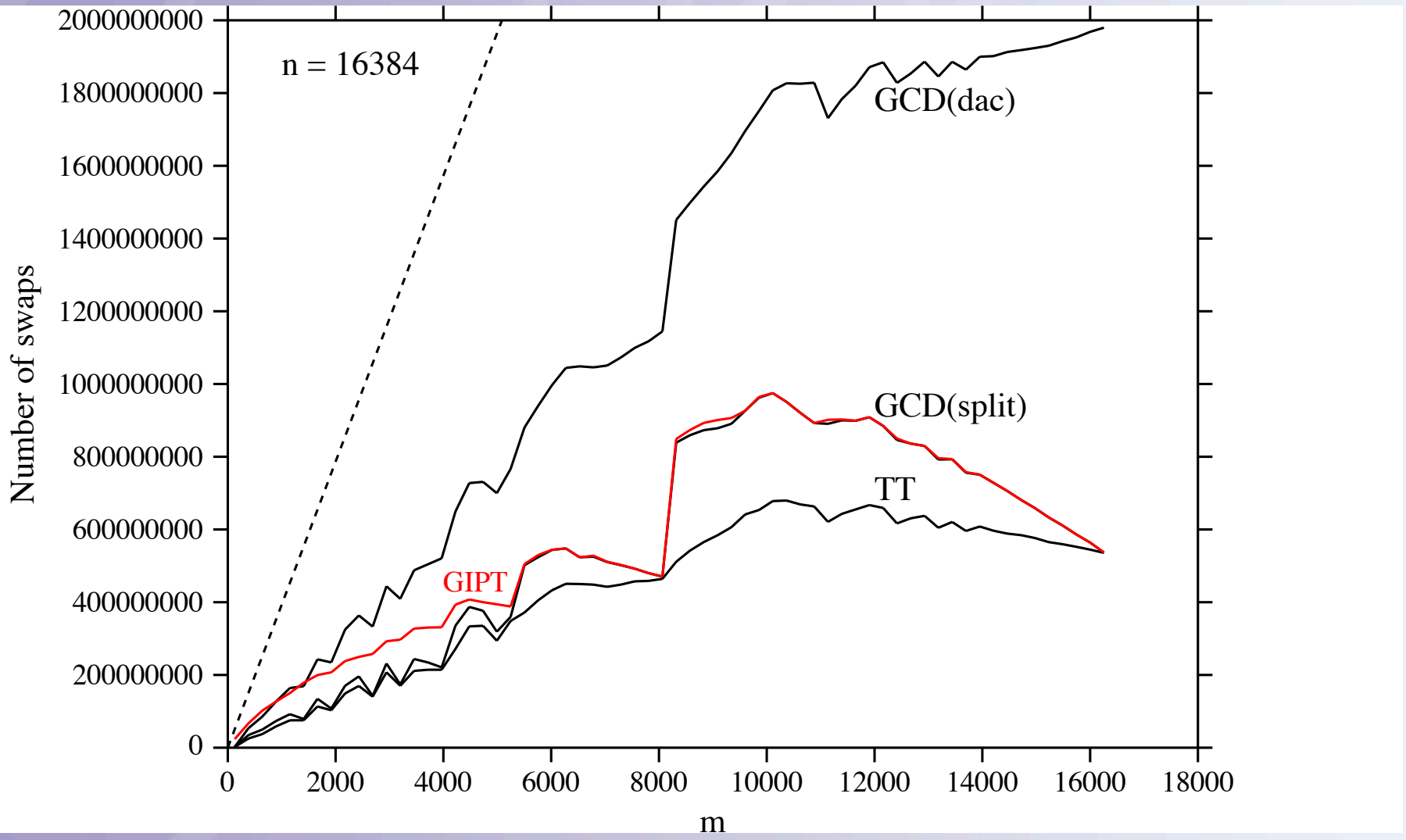
$$w_1^1 \ldots w_k^1 w_1^2 \ldots w_k^2 \ldots w_1^g \ldots w_k^g \rightarrow w_1^1 w_1^2 \ldots w_1^g \ldots w_k^1 w_k^2 \ldots w_k^g$$

- This results in an increased number of operations.


k = 100
Unshuffle (TT)
Shuffle (serial)
Shuffle (dac)
Vector permute
RowTranspose (dashed)
Number of swaps
250000000
200000000
150000000
100000000
50000000
0
100
120
140
160
180
200
g

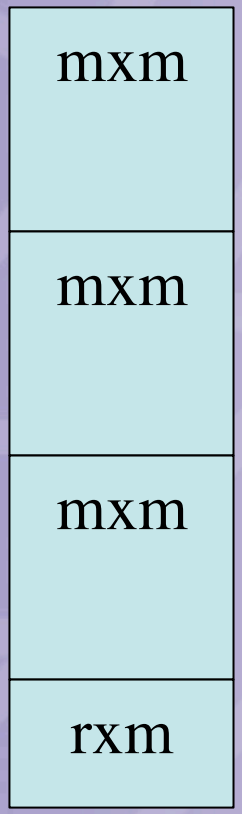

k = 100
Shuffle (dac)
Vector permute
RowTranspose (dashed)
Number of swaps
25000000
20000000
15000000
10000000
5000000
0
100
120
140
160
180
200
g

Overall Swaps

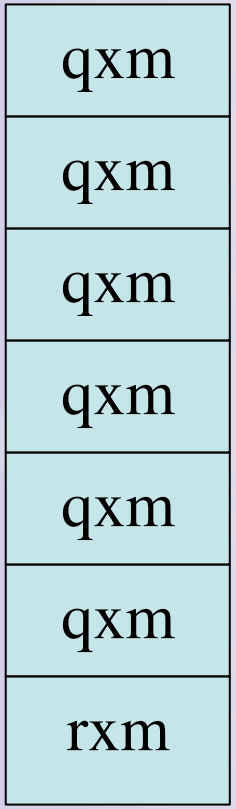


$n = qm+r, 0 \leq r < m$

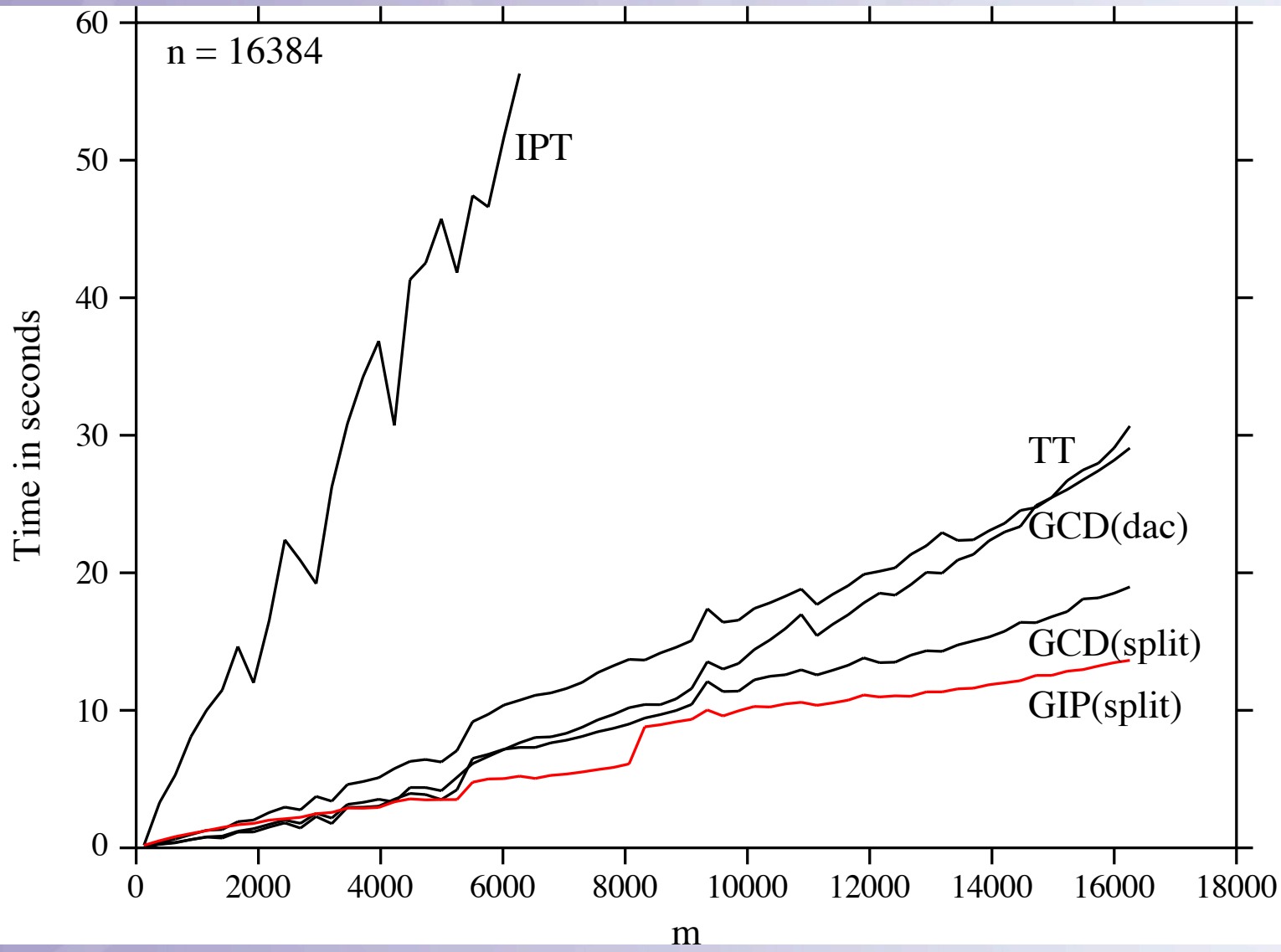
GIPT: q matrices of size mxm plus one of size r/m



GCDT: m matrices of size qxm plus one of size r/m



Overall Times



Concluding Remarks

- Can improve performance of TT algorithm by use of a divide-and-conquer algorithm that is readily parallelizable .
- Can avoid need for extra O(m) storage in TT algorithm, but at cost of extra swaps which degrades performance.

Thank you for your attention.

Any Questions?