Parallel and Cache-Efficient In-Place Matrix Storage Format Conversion

UMINF 10.05

FRED GUSTAVSON
IBM T.J. Watson Research Center, Emeritus, and Umeå University
and
LARS KARLSSON and BO KÅGSTRÖM
Umeå University

Techniques and algorithms for efficient in-place conversion to and from standard and blocked matrix storage formats are described. Such functionality is required by numerical libraries that use different data layouts internally. Parallel algorithms and a software package for in-place matrix storage format conversion based on in-place matrix transposition are presented and evaluated. A new algorithm for in-place transposition which efficiently determines the structure of the transposition permutation a priori is one of the key ingredients. It enables effective load balancing in a parallel environment.

1. INTRODUCTION

Deep memory hierarchies require both spatial and temporal locality of reference to amortize the relatively high cost of memory accesses. Explicitly blocked algorithms obtain high performance by arranging for a large number of floating point operations to be performed on a small block of the matrix with a technique known as *cache blocking* [IBM 1986; Gallivan et al. 1988; Lam et al. 1991]. The high arithmetic intensity pays off the cost of transferring the matrix blocks to and from main memory. When blocked algorithms are used in conjunction with blocked storage formats, instead of the standard column- and row-major formats, the memory hierarchy is better utilized and even higher performance can be obtained. Blocked, recursive, and hybrid data layouts are able to reduce the number of cache- and TLB-misses by storing the elements of a block contiguously [Gustavson 2000; Hong et al. 2003]. Ideally, blocked formats match the memory access patterns of blocked algorithms.

Financial support has been provided in part by the *Swedish Research Council* (the grant 2008-5243) and the *Swedish Foundation for Strategic Research* (the frame program grant A3 02:128). The work is also a part within an IBM Shared University Research (SUR) grant. This research was conducted using the resources of High Performance Computing Center North (HPC2N).
Authors' addresses: L. Karlsson and B. Kågström, Department of Computing Science and HPC2N, Umeå University, SE-901 87 Umeå, Sweden, email: {larsk,bokg}@cs.umu.se. F. Gustavson, 3 Welsh Court, East Brunswick, NJ 08816, email: fg2935@hotmail.com.

In this paper, a *blocked format* partitions the matrix into submatrices, so called *blocks*, of size $m_b \times n_b$ and stores each rectangular block contiguously in $m_b n_b$ consecutive memory locations. A key property is that a few such fixed-sized blocks fit perfectly into all caches.

Recursively blocked formats partition a matrix into blocks and stores the blocks in a specified order governed by the definition of the recursive scheme. The scheme is applied recursively to each block. Examples of recursive blocked formats include recursive blocked row and column orderings [Gustavson et al. 1998], which are variations of Morton, Hilbert, and Peano orderings [Elmroth et al. 2004; Bader and Zenger 2006; Bader and Heinecke 2008].

Both theoretical and empirical evidence shows that blocked storage formats may improve performance of blocked matrix computations, especially on multicore processors, see for example [Gustavson 2000; Hong et al. 2003; Chan et al. 2008; Gustavson et al. 2009; Dongarra and Kurzak 2009]. This suggests that numerical libraries should use blocked formats for algorithms that benefit from them. The choice of internal data layout should be hidden from the library users, e.g., by storage format conversion at the library interface. When it is undesirable or impossible to copy the matrix, e.g., due to memory limitations or performance concerns, then *in-place* conversion can be used.

This paper addresses the problem of parallel and cache-efficient in-place matrix storage format conversion and provides novel algorithms, techniques, and software. Our conversion algorithms are built on top of in-place matrix transposition and extends the work in [Gustavson 2008; Karlsson 2009]. A major contribution is our new algorithm GKKLEADERS, that efficiently determines the properties of the in-place transposition permutation between standard column and row major formats. These properties make load balancing very efficient in many parallel environments.

The organization of the paper is as follows. Notation and necessary facts from classical number theory are introduced in Section 1.1. Section 1.2 reviews standard blocked matrix formats and introduces parameterized storage mappings of blocked formats. Mixed radix number systems and some of their connections to matrix storage formats are addressed in Section 1.3. Section 1.4 introduces the problem of in-place matrix transposition and presents a generic in-place transposition algorithm based on following cycles. The connections between mixed radix number systems and matrix storage formats, conversion, and transposition are further elaborated in Section 2. Notably, important shared-memory parallelization issues are discussed in Section 2.3. The theory and most implementation issues related to our new algorithm GKKLEADERS that determines the cycle leaders of the transposition permutation a priori is explained in Section 3.

Section 4 briefly discusses the problem of in-place transposition of square matrices, for which efficient algorithms already exist. Generalizations of the standard blocked storage formats to arbitrary matrix dimensions and a conversion scheme (amenable to parallelization) between these new formats are presented in Section 5. Some computational experiments are discussed in Section 6. These include a comparison of the performance and overhead of different cycle-following algorithms, showing that our GKKLEADERS algorithm outperforms previous algorithms. In addition, performance results for conversion between different matrix storage for-

mats are presented, demonstrating the effectiveness of our algorithms and software. Related work on in-place matrix transposition that has most influenced our work is briefly reviewed in Section 7. Finally, some conclusions are given in Section 8. Various subalgorithms (called by a higher level algorithm) and proofs related to GKKleaders are deferred to Appendices A and B.

1.1 Preliminaries

1.1.1 *Notation and terminology.* When referring to elements in matrices, arrays, and memory we use *zero-based indexing*. In particular, the top left element of an $m \times n$ matrix A is denoted $A(0,0)$, or simply $(0,0)$ when A is implicit, and its bottom right element is denoted $A(m-1, n-1)$. The indices of A make up the domain

$$\mathcal{I} = \{(i,j) : i = 0, 1, \ldots, m-1,\ j = 0, 1, \ldots, n-1\}.$$

A matrix is stored in memory starting at some base location and occupies mn consecutive memory locations. Element $(i,j) \in \mathcal{I}$ is stored at some offset $k \in \mathcal{M} = \{0, 1, \ldots, mn-1\}$ from the base location. When referring to a matrix element $A(i,j)$ by its offset k, then we use the array notation $A[k]$. The notation $a : b$ refers to the sequence of integers $a, a+1, \ldots, b$.

A permutation algorithm which permutes the elements of an array A into another array B is called an *out-of-place* algorithm and it requires mn extra storage for B. In contrast, if only a constant amount of workspace, i.e., not depending on m and n, is required, then the algorithm is said to be *in-place*. Semi-in-place algorithms require more than a constant amount of workspace, for instance $\mathcal{O}(m+n)$, but still much less than mn. Semi-in-place algorithms are in common use, especially when the required workspace is relatively small.

1.1.2 *Number theory facts.* The set of integers is denoted by $\mathbb{Z}$ and the subset of nonnegative integers less than or equal to n is denoted by $\mathbb{Z}_n = \{0, 1, \ldots, n-1\}$.

Every positive integer k has a unique prime power factorization:

$$k = p_1^{e_1} p_2^{e_2} \cdots p_t^{e_t}, \quad p_1 < p_2 < \cdots < p_t, \quad e_i > 0.$$

p_i is a prime number and $p_i^{e_i}$ is a *prime power* of k.

A *divisor* d of a nonnegative integer k is a positive integer which evenly divides k. The notation $d \mid k$ expresses that d is a divisor of k, or equivalently that k is a multiple of d. In particular, every integer divides 0.

The *greatest common divisor* of two nonnegative integers a and b, written $\gcd(a,b)$, is the largest positive integer c such that $c \mid a$ and $c \mid b$. In particular, $\gcd(a,0) = a$. The integers a and b are said to be *relatively prime* or *coprime* if they have no factor in common, i.e., if $\gcd(a,b) = 1$. If a and b are coprime, then $\gcd(ac,b) = \gcd(c,b)$.

The *least common multiple* of two positive integers a and b, written $\operatorname{lcm}(a,b)$, is the smallest positive integer c such that $a \mid c$ and $b \mid c$. The least common multiple generalizes to any number of integers.

The *remainder operator*, written $a \bmod b$, gives the remainder of a after division by b and is defined as $a \bmod b = a - \left\lfloor \frac{a}{b} \right\rfloor b$. The remainder operator is closely related to the notion of *congruency modulo* m. The notation $a \equiv b \pmod{m}$ means that $m \mid (a-b)$ and is read as "a is congruent to b modulo m".

The *linear congruence* $ax \equiv b \pmod{m}$ is solvable for x if and only if $\gcd(a, m) \mid b$. If the congruence is solvable, then the infinite set of solutions is $\{x_* + km/\gcd(a, m)\}$, where x_* is an arbitrary solution and k is an integer. In particular, with $b = 0$ the congruence $ax \equiv 0 \pmod{m}$ is *always* solvable and the set of solutions is $\{km/\gcd(a, m) : k \in \mathbb{Z}\}$. Both sides of a congruence and its modulus can be divided by any of their common divisors, i.e., $ax \equiv b \pmod{m}$ is equivalent to $(a/d)x \equiv (b/d) \pmod{m/d}$ where d is any common divisor of a, b, and m. Both sides of a congruence can be divided by any of their common divisors if it is coprime to the modulus, i.e., $ax \equiv b \pmod{m}$ is equivalent to $(a/d)x \equiv (b/d) \pmod{m}$ where d is a common divisor of a and b and d and m are coprime.

The *totient function*, or *Euler's ϕ-function*, is written as $\phi(k)$ and gives the number of positive integers less than or equal to k which are coprime to k. For a prime power, $\phi(p^e) = p^{e-1}(p-1)$. If $h = ab$ with $\gcd(a, b) = 1$, then $\phi(h) = \phi(a)\phi(b)$.

1.2 Storage Formats and Mappings

A *storage mapping* f of an $m \times n$ matrix is an invertible mapping from the domain $\mathcal{I}$ to the set of memory offsets $\mathcal{M}$, i.e.,

$$f : \mathcal{I} \mapsto \mathcal{M}.$$

A matrix *storage format* is a parameterized set of storage mappings where the parameters typically include the matrix sizes m and n, and possibly some block sizes such as m_b and n_b. The two standard formats Column-Major (CM) and Row-Major (RM) have the parameterized mappings

$$f_{\mathrm{CM}}(i, j) = i + jm \quad \text{and} \quad f_{\mathrm{RM}}(i, j) = in + j,$$

respectively. Note that a matrix is sometimes stored with a *leading dimension* which is larger than m (CM) or n (RM). In these cases, replace m (or n) with the leading dimension throughout the paper. See also Section 5 for a discussion on how we deal with blocked data layouts of matrices with arbitrary dimensions.

In the following, we are assuming that $m = Mm_b$ and $n = Nn_b$, i.e., that all blocks are of size $m_b \times n_b$. A blocked format is constructed from two independent formats: the *intra-block format* specifies the order of the elements of a block and the *inter-block format* specifies the order of the blocks. Choosing between CM and RM for the inter- and intra-block formats result in four combinations that we refer to here as the *standard blocked formats*:

Mnemonic	Inter	Intra
CCRB	CM	CM
CRRB	CM	RM
RCRB	RM	CM
RRRB	RM	RM

The suffix RB stands for Rectangular Block. The first letter designates the inter-block format and the second letter designates the intra-block format as shown above.

The parameters of the CM and RM formats are m and n. The standard blocked formats have two additional parameters, m_b and n_b, specifying the block size.

A row index i maps into a *block index* $0 \le i_2 < M$ and a *block offset* $0 \le i_1 < m_b$ by

$$i_2 = \left\lfloor \frac{i}{m_b} \right\rfloor \quad \text{and} \quad i_1 = i \bmod m_b.$$

Similarly, a column index j maps into a block index j_2 and a block offset j_1 by

$$j_2 = \left\lfloor \frac{j}{n_b} \right\rfloor \quad \text{and} \quad j_1 = j \bmod n_b.$$

With block indices and offsets, the parameterized storage mappings of the four standard blocked formats can be defined as

$$\begin{aligned}
f_{\mathrm{CCRB}}(i,j) &= (i_2 + j_2 M)\, m_b n_b + i_1 + j_1 m_b, \\
f_{\mathrm{CRRB}}(i,j) &= (i_2 + j_2 M)\, m_b n_b + i_1 n_b + j_1, \\
f_{\mathrm{RCRB}}(i,j) &= (i_2 N + j_2)\, m_b n_b + i_1 + j_1 m_b, \\
f_{\mathrm{RRRB}}(i,j) &= (i_2 N + j_2)\, m_b n_b + i_1 n_b + j_1.
\end{aligned}$$

1.3 Mixed Radix Representation of Storage Mappings

A *mixed radix number system* is a positional number system in which the radix (or base) is position dependent [Knuth 1998]. The radices of a mixed radix system with t positions are denoted by $\mathbf{r} = (r_t, r_{t-1}, \ldots, r_1)$ where $r_i > 0$. The digits are denoted by $\mathbf{d} = (d_t, d_{t-1}, \ldots, d_1)$ with $0 \le d_i < r_i$. The least significant position has radix r_1 and digit d_1. The invertible function $\psi_{\mathbf{r}}$ below defines what we mean by a mixed radix number:

$$\psi_{\mathbf{r}}(\mathbf{d}) = d_t \cdot r_{t-1} r_{t-2} \cdots r_1 + d_{t-1} \cdot r_{t-2} r_{t-3} \cdots r_1 + \cdots + d_2 \cdot r_1 + d_1.$$

Note that $\psi_{\mathbf{r}}$ is a generalization of the fixed-base expansion of decimal numbers, e.g., $123 = 1 \cdot 10^2 + 2 \cdot 10 + 3$, to mixed radices. The inverse of $\psi_{\mathbf{r}}$ gives the mixed radix digits of an integer k:

$$\psi_{\mathbf{r}}^{-1}(k) = \mathbf{d}, \quad d_i = \left\lfloor \frac{k}{r_{i-1} r_{i-2} \cdots r_1} \right\rfloor \bmod r_i.$$

In particular, $d_1 = k \bmod r_1$.

Note that $\psi_{\mathbf{r}}$ can be seen as a generalization of the CM and RM mappings to higher dimensions and it is a standard mapping used by compilers when storing multi-dimensional arrays. Thus, a matrix stored in any of the storage formats we consider can be accessed using a four-dimensional array.

Choose some radices $\mathbf{r}$ such that their product equals mn. Define an invertible mapping h from the matrix elements (i, j) to the numbers in the mixed radix number system defined by the chosen radices $\mathbf{r}$. Then the composed function $f = \psi_{\mathbf{r}} \circ h$ is a storage mapping. For example, choose $\mathbf{r} = (N, M, m_b, n_b)$ and define the mapping

$$h(i,j) = (j_2, i_2, i_1, j_1).$$

After composition with $\psi_{\mathbf{r}}$ we get

$$\begin{aligned}
f(i,j) &= (\psi_{\mathbf{r}} \circ h)(i,j) = \psi_{\mathbf{r}}(h(i,j)) \\
&= j_2 \cdot M m_b n_b + i_2 \cdot m_b n_b + i_1 \cdot n_b + j_1 \\
&= (i_2 + j_2 M) m_b n_b + i_1 n_b + j_1 = f_{\mathrm{CRRB}}(i,j).
\end{aligned}$$

In other words, element (i, j) of a matrix in CRRB format is represented by the number $\mathbf{d} = (j_2, i_2, i_1, j_1)$ in a mixed radix system with radices $\mathbf{r} = (N, M, m_b, n_b)$. After repeating the procedure above for the other formats we end up with different permutations of the same basic components i_1, i_2, j_1, j_2 and M, N, m_b, n_b:

Mnemonic	Radices, $\mathbf{r}$	Digits, $\mathbf{d}$
CM	(N, n_b, M, m_b)	(j_2, j_1, i_2, i_1)
CCRB	(N, M, n_b, m_b)	(j_2, i_2, j_1, i_1)
CRRB	(N, M, m_b, n_b)	(j_2, i_2, i_1, j_1)
RCRB	(M, N, n_b, m_b)	(i_2, j_2, j_1, i_1)
RRRB	(M, N, m_b, n_b)	(i_2, j_2, i_1, j_1)
RM	(M, m_b, N, n_b)	(i_2, i_1, j_2, j_1)

1.4 Cycle-Following In-Place Matrix Transposition

We are given an $m \times n$ matrix A stored in CM format. Thus, element (i, j) is stored at offset $k = f_{\mathrm{CM}}(i, j) = i + jm$. The *in-place transposition problem* is to rearrange the storage of A, using only a constant amount of workspace, so that element (i, j) is moved to offset $\hat{k} = f_{\mathrm{RM}}(i, j) = in + j$. The storage format of A thus changes from CM to RM. Alternatively, we can interpret the output as A^T stored in CM format. The latter interpretation explains the heading of this section. We emphasize the former interpretation since the topic of this paper is storage format conversion. Note that the mapping from k to $\hat{k}$, denoted $\hat{k} = P(k)$, is a permutation of $\mathbb{Z}_{mn}$.

Set $q = mn - 1$ and notice that for all k in the range $0 < k < q$, i.e., excluding 0 and q which are fixed points of the permutation, we have

$$P(k) = kn \bmod q = (in + jmn) \bmod q = (in + j + j(mn - 1)) \bmod q = \hat{k}.$$

The inverse of P is $P^{-1}(\hat{k}) = \hat{k}m \bmod q = k$. The transposition permutation associated with CM and RM is clearly related to multiplication of integers modulo q. The structure of the transposition permutation is analyzed in detail in Section 3.

Since transposition is a permutation it can be expressed as a combination of simple permutations called cycles. A cycle shifts a subset of the elemenets one step in a cyclic fashion while the remaining elements are left untouched. For example, $m = 5$ and $n = 3$ has $q = 14$ and $\mathbb{Z}_{15}$ is permuted according to the product of cycles

$$(0)(1\ 3\ 9\ 13\ 11\ 5)(7)(2\ 6\ 4\ 12\ 8\ 10)(14).$$

For example, the elements at offset 2 is moved to offset 6, i.e., $P(2) = 6$. There are three *singleton cycles*, or *fixed points*, in this example: (0), (7), and (14). Since singleton cycles usually have no effect, they are sometimes omitted. The two nontrivial cycles in this example both have length six.

The notion of a *cycle leader*, i.e., a unique representative of a cycle, is an important concept. In this paper, cycle-following in-place transposition algorithms consist of two phases. In the first phase, a complete leader set is constructed. Such a set contains one and only one cycle leader for each (nonsingleton) cycle. In the second phase, each cycle is shifted backwards starting from its cycle leader. In the example above, $\mathcal{S} = \{0, 1, 2, 7, 14\}$ is one of many possible leader sets.

Algorithm 1 ShiftCycle(A, s, L, P)

Input: An array A, a cycle leader s, a permutation P^{-1}, and a chunk size L.
Purpose: Shifts contiguous memory chunks of length L in A around the cycle of P that contains s.

```
1: B ← A[sL : sL + L − 1]
2: k1 ← s
3: k2 ← P^{-1}(s)
4: while k2 ≠ s do
5:     A[k1L : k1L + L − 1] ← A[k2L : k2L + L − 1]
6:     k1 ← k2
7:     k2 ← P^{-1}(k2)
8: end while
9: A[k1L : k1L + L − 1] ← B
```

Algorithm 1 takes an array A, a cycle leader s, a chunk size L, and a permutation P as input and shifts contiguous chunks of size L around the cycle of P that contains s. The cycle of P is traversed backwards via P^{-1} since this leads to a more efficient implementation than shifting forwards using P.

Note that the transposition permutation P is similar to a pseudo-random number generator. Furthermore, there is typically no spatial locality of reference within a cycle and pairs of cycles do not tend to stay close to one another. The first memory access to a chunk is therefore most likely a cache miss in all levels of the cache hierarchy. When L is small, this is a disaster which makes shifting the cycle very inefficient. However, when L is large, then the cost of the first cache miss is amortized over the large contiguous chunk, which is being streamed in from main memory at the peak rate of the machine. The techniques described in this paper obtain high performance by expressing storage format conversion as in-place transpositions with a large chunk size, e.g., L is typically m_b, n_b, or $m_b n_b$, where m_b and n_b are block sizes [Gustavson 2008]. Some conversions lead to transpositions of size $m_b \times n_b$ with $L = 1$. This is generally no problem since $m_b n_b$ is small enough to allow all of the transposed elements to reside in cache simultaneously. Thus, for these cases an out-of-place algorithm can be used to obtain high performance.

Algorithm 2 GenericInPlaceTransposition(A, m, n, L)

Input: An array A with mnL elements where m, n, and L are positive integers.
Purpose: Applies the permutation $P(k) = kn \bmod q$ where $q = mn - 1$ to A with chunk size L.

```
1: Define P : k ↦ kn mod q where q = mn − 1
2: S ← CycleLeaders(m, n)
3: while S ≠ ∅ do
4:     Select an s ∈ S and let S ← S \ {s}
5:     ShiftCycle(A, s, L, P)
6: end while
```

A generic in-place transposition algorithm can be conceptually formulated as in Algorithm 2. A leader set is constructed on line 2 and each cycle is shifted starting from its leader on lines 3 to 6. Thus, even though the algorithm conceptually consists of two phases, in practice they have to be interleaved since the leader set is sometimes $\mathcal{O}(q)$ and can not be explicitly stored. One of the challenges is to construct an efficient algorithm that implements the so far undefined function

CycleLeaders. We develop a new algorithm called GKKleaders in Section 3 and recall some previously published algorithms in Section 7.

2. IN-PLACE MATRIX STORAGE FORMAT CONVERSION

We now continue our discussion of the connection between mixed radix number systems and matrix storage formats that we started in Section 1.2. The notion of a *digit permutation* [Fraser 1976] plays an important role. Since the mixed radix representations of the storage mapping in Section 1.2 use the same digits and radices, they are connected via digit permutations.

If we permute the digits in the mixed radix representation of the storage mapping f, we obtain another storage mapping $\hat{f}$. By rearranging the stored matrix so that the element at offset $f(i,j)$ is moved to offset $\hat{f}(i,j)$, we have effectively converted the storage format of the matrix. For example, by swapping the second and third digits in the representation of the CM mapping, we obtain the representation of the CCRB mapping. By reflecting this change in the stored matrix, the storage format is converted from CM to CCRB.

2.1 Adjacent Digit Swap

In the following, we remark that m and n refer to the size of a transposition problem, which normally concerns (only) a submatrix of the original A. To simplify notation, we still use A for this submatrix (array).

note that m and n do not refer to the size of the matrix A any longer. Instead, they refer to the size of a transposition problem.

An *adjacent digit swap* is a special type of digit permutation in which two adjacent digits are swapped. We pay special attention to this type of digit permutation since it is possible to reflect it in a stored matrix using only in-place transpositions.

Suppose we want to swap the digits d_i and d_{i+1} in a t-radix system. Define

$$\begin{aligned} n_{\text{ind}} &= r_t r_{t-1} \cdots r_{i+2}, \\ n &= r_{i+1}, \\ m &= r_i, \\ L &= r_{i-1} r_{i-2} \cdots r_1. \end{aligned}$$

Algorithm 3 reflects the adjacent digit swap in a stored matrix A. Each call to GenericInPlaceTransposition (Algorithm 2) picks up a chunk of size L starting at offset $(jm+i)L$ and moves it to a chunk of size L starting at offset $(in+j)L$. Thus, the element that originally resided at offset

Algorithm 3 AdjacentSwap(A, n_{ind}, m, n, L)

Input: An array A with $n_{\text{ind}}mnL$ elements, where n_{ind}, m, n, and L are positive integers.
Purpose: Moves $A[k]$ to $A[\hat{k}]$ where

$$k = \psi_{(n_{\text{ind}},n,m,L)}(d_4, j, i, d_1) \quad \text{and} \quad \hat{k} = \psi_{(n_{\text{ind}},m,n,L)}(d_4, i, j, d_1).$$

1: **for** $r = 0, 1, \ldots, n_{\text{ind}} - 1$ **do**
2: GenericInPlaceTransposition($A[rnmL]$, m, n, L)
3: **end for**

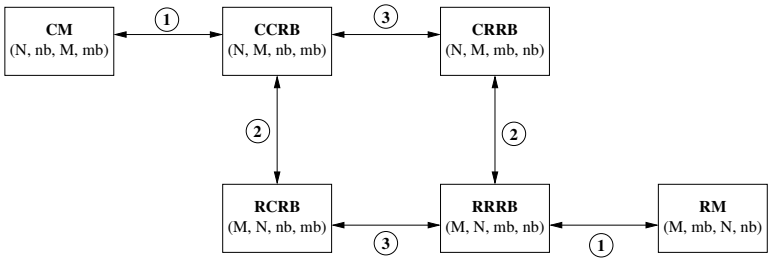


Fig. 1. Conversion graph for six matrix storage formats. An adjacent digit swap converts between two neighbouring formats.

$$d_4nmL + (jm + i)L + d_1 = \psi_{(n_{\text{ind}},n,m,L)}(d_4, j, i, d_1)$$

is moved to offset

$$d_4nmL + (in + j)L + d_1 = \psi_{(n_{\text{ind}},m,n,L)}(d_4, i, j, d_1),$$

by Algorithm 3.

2.2 Conversion Graph

Conversion between any pair of the matrix storage formats in Section 1.2 can be realized by one or more adjacent digit swaps by following the edges of the conversion graph in Figure 1. Each edge in the conversion graph represents an adjacent digit swap:

(1) swaps the second and third digits with chunk size $L = m_b$ or $L = n_b$,
(2) swaps the third and fourth digits with chunk size $L = m_b n_b$, and
(3) swaps the first and second digits with chunk size $L = 1$.

Each swap requires reading and writing the matrix only once. Assuming that the bottleneck is the memory bandwidth, each of the three swaps above should take roughly the same time to complete.

According to Figure 1, converting from CM to RM, i.e., transposing the matrix, can be implemented by four digit swaps. However, note that swaps (2) and (3) can be *fused*, meaning that both are performed without reading and writing the matrix more than once. Fusion is possible since the *chunks* in swap (2) correspond to the transposition of block in swap (3). Hence, these subtranspositions can be performed while shifting the cycles of swap (2). Note that during a fused swap operation the singleton cycles of swap (2) can no longer be ignored since they are still affected by swap (3).

2.3 Parallelization

Algorithm 3 and the subalgorithms that it calls form the building blocks of matrix storage format conversion. In this section, we describe a shared-memory parallelization suitable for implementation using OpenMP [OpenMP.org], threads, or similar technologies.

The aim is to utilize all of the available memory bandwidth when shifting cycles. Typically, at least on modern computer architectures, p threads need to execute in

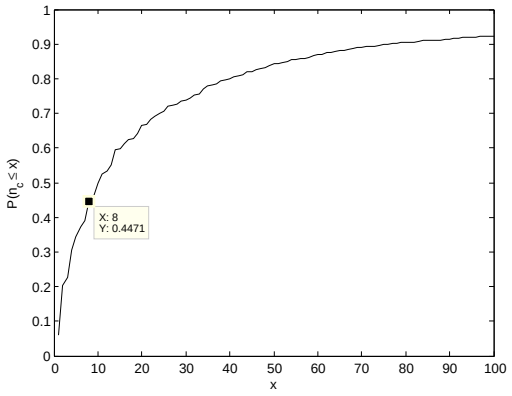


Fig. 2. The probability, $P(n_{\text{cycles}} \leq x)$, that the number of cycles, n_{cycles}, is less than or equal to x for $x = 1, 2, \ldots, 100$ for random problem parameters $m, n \in \{2, 3, \ldots, 100\}$, excluding $m = n$.

parallel to saturate the memory bandwidth. Usually, the optimal p is smaller than the total number of processor cores since memory bandwidth eventually becomes the bottleneck. Algorithm 3 can be parallelized at three different levels described below. To be effective on all possible inputs, an implementation should exploit all of them.

2.3.1 *Independent subtranspositions.* Recall that AdjacentSwap performs n_{ind} independent and identical in-place transpositions of size $m \times n$ with chunk size L. At the highest level is a loop over the n_{ind} independent transpositions. The input parameter n_{ind} may be small or large. For matrix storage format conversion, it is typical to have $n_{\text{ind}} \in \{1, M, N, MN\}$. The independent transpositions can be performed in parallel and load balancing is simplified since the subproblems are identical. This is the coarsest level of parallelization and is enough to obtain high efficiency in many cases. However, when n_{ind} is small, such as when converting between CCRB and RRRB, a second level of parallelism becomes necessary.

2.3.2 *Independent cycles.* Another level of parallelism is found in the independent cycles within one transposition [Gustavson 2008]. All cycles can be shifted in parallel, but the number of cycles and the length of each cycle vary greatly with the problem parameters m and n. In many cases there are only a couple of long cycles, perhaps not enough to utilize all the bandwidth. Figure 2 shows the cumulative distribution function of the number of cycles over a wide range of problems. The marker highlights that almost 45% of the tested problems have eight or fewer cycles. For problems with only a few long cycles, a third level of parallelism is required.

2.3.3 *Long cycles.* For a chunk size $L > 1$ it is possible to partition each chunk and form several independent cycles with smaller chunk sizes. These cycles can then be shifted in parallel without further synchronization. Since splitting the chunks reduces the chunk size, this approach might actually degrade the overall performance.

A different approach, which keeps the chunk size intact, is to split the cycle into

several linear sequences [Gustavson 2008]. A cycle is broken into a sequence by copying one of its chunks to workspace. A sequence can be further broken up by copying another chunk. In general, a cycle of length $\ell \geq p$ with leader s can be broken into p sequences, i.e., one sequence per thread, in the following way. Choose p integers

$$x_i \in \{0, 1, \ldots, \ell - 1\}, \quad i = 0, 1, \ldots, p - 1, \quad x_0 < x_1 < \cdots < x_{p-1}$$

and define the p sequence leaders

$$s_i = sn^{x_i} \bmod q, \quad i = 0, 1, \ldots, p - 1.$$

Copy the first chunk of each sequence in parallel, i.e.,

$$B_i \leftarrow A[s_i L : s_i L + L - 1],$$

and synchronize the threads. Thread i shifts the $\ell_i = x_i - x_{(i-1) \bmod p}$ elements starting at s_i backwards along the sequence. The final chunk at $sn^{x_i - \ell_i + 1} \bmod q$ is stored in $B_{(i-1) \bmod p}$ and was initialized by a different thread. This approach requires pL elements of workspace for the copies B_i for $i = 0, 1, \ldots, p - 1$ and a barrier synchronization of the p threads.

2.3.4 *Load balancing.* Given a leader set and associated cycle lengths, the load balancing problem is to decide how best to assign the cycles to the p threads in order to minimize the execution time. Several simplifications are necessary to reduce the complexity of the problem.

We need a performance model in order to estimate the load assigned to each thread. Basically, the only information that is available a priori is the length of each cycle. Thus, let $T(i)$ for $i = 0, 1, \ldots, p - 1$ be the total number of chunks assigned to thread i. The parallel execution time, according to our model, is proportional to

$$T_p = \max_{0 \leq i < p} \{T(i)\}.$$

The sum of the *idle times* is proportional to

$$T_o = \sum_{i=0}^{p-1} (T_p - T(i)) = pT_p - \sum_{i=0}^{p-1} T(i).$$

The *relative overhead* is defined as

$$\frac{T_o}{pT_p} = 1 - \frac{\sum_{i=0}^{p-1} T(i)}{pT_p}$$

and it quantifies the load imbalance in the following sense. The relative overhead is the relative reduction in pT_p obtained by perfectly balancing the load. For instance, if $T_o/pT_p = 0.05$, then pT_p can not be reduced by more than 5% by a perfect load balancing scheme.

It is difficult to find an optimal solution. A simple and good approximation is obtained by greedily assigning each cycle to the thread with the currently smallest $T(i)$. If the cycles are assigned to threads with the longest cycles first, then the greedy algorithm produces even better approximations. For our purposes, the first approach, i.e., without sorting, is sufficient. If the relative overhead becomes too

large after the greedy algorithm terminates, long cycles can be split into smaller cycles until the relative overhead becomes sufficiently small.

3. A PRIORI DETERMINATION OF CYCLE LEADERS AND LENGTHS

After having established the usefulness of in-place transposition to the problem of matrix storage format conversion in the previous section, we now focus on the development of an efficient algorithm called GKKLEADERS that determines cycle leaders and cycle lengths. Such an algorithm makes it possible to cheaply balance the load in a parallel environment.

The problem of finding a leader set naturally partitions into one subproblem for each divisor v of $q = mn - 1$ [Pall and Seiden 1960]. The set of elements associated with v is $\{k : 1 \leq k \leq q,\ \gcd(k, q) = v\}$, the size of which is $\phi(q/v)$. These elements are spread over $\phi(q/v)/\ell$ cycles of the same length ℓ.

What we call the *main subproblem* or the *coprime case* corresponds to $v = 1$. It includes the $\phi(q)$ integers which are coprime to q. For all the remaining subproblems, $v > 1$. Recall that the k-th element in the cycle starting at s is $sn^k \bmod q$. Suppose $v = \gcd(s, q) > 1$ and factor $s = v\tilde{s}$. Rewrite $sn^k \bmod q$ as $v(\tilde{s}n^k \bmod (q/v))$. Thus, any subproblem reduces to a coprime case with q replaced by q/v. After obtaining a leader set for the reduced subproblem, the solution to the unreduced subproblem is obtained by multiplying each leader by v.

The basic idea of GKKLEADERS is inspired by the pen-and-paper algorithm proposed by Pall and Seiden [1960]. We include several nontrivial improvements and identify and resolve a number of implementation issues.

The material presented here is based on elementary number theory, as described in introductory textbooks such as Niven et al. [1991].

3.1 Moving to an Additive Domain

The cycle structure of the transposition permutation in the coprime case is not readily apparent in the multiplicative domain modulo q. Therefore, we first establish an isomorphism from the multiplicative domain to an additive domain for each divisor v of q. The analysis in the additive domain turns out to be much clearer and leads to a compact description of a leader set. The leader set is finally translated back into the multiplicative domain where cycle shifting takes place. The set of leaders, $\mathcal{S}$, in the additive domain is a direct product of sets of integers of $\{0, 1, \ldots, k_i - 1\}$. Thus, there is no need to explicitly store these sets and the GKKLEADERS algorithm could generate the leaders one by one and require no workspace. However, for efficiency reasons, we use a small, bounded amount of workspace.

The prime power factorization

$$q = mn - 1 = p_1^{e_1} p_2^{e_2} \cdots p_t^{e_t}, \quad p_1 < p_2 < \cdots < p_t, \quad e_i \geq 1$$

has t prime powers $p_i^{e_i}$. It is worth noting that the theory, and an implementation in particular, becomes more technical when $p_1 = 2$, i.e., q is even and hence both m and n are odd. We advise readers first to fully understand the special case of q odd, since the general case does not alter the big picture.

3.1.1 *Modular arithmetic.* By using the t prime powers $p_i^{e_i}$ of q as *moduli* in a *modular arithmetic system*, each integer $s \in \mathbb{Z}_q$ is uniquely represented by its so

called *modular components*, which we denote by $\hat{s}_i$ for $i = 1, 2, \ldots, t$. The Chinese Remainder Theorem (CRT) ensures that the *CRT mapping*

$$\mathbb{Z}_q \to \mathbb{Z}_{p_1^{e_1}} \times \cdots \times \mathbb{Z}_{p_t^{e_t}} : s \mapsto (\hat{s}_1, \hat{s}_2, \ldots, \hat{s}_t), \quad \hat{s}_i = s \bmod p_i^{e_i}$$

is invertible. The *inverse CRT mapping* transfers $(\hat{s}_1, \hat{s}_2, \ldots, \hat{s}_t)$ to $s \in \mathbb{Z}_q$, where

$$s = \left(\sum_{i=1}^{t} M_i z_i \hat{s}_i \right) \bmod q, \quad M_i = \frac{q}{p_i^{e_i}}, \quad M_i z_i \equiv 1 \pmod{p_i^{e_i}}. \tag{1}$$

Modular multiplication in $\mathbb{Z}_q$, i.e., $c = ab \bmod q$, turns into component-wise modular multiplication of the modular components $\hat{a}_i$ and $\hat{b}_i$, i.e., $\hat{c}_i = \hat{a}_i \hat{b}_i \bmod p_i^{e_i}$.

3.1.2 *The additive domain.* For an element s associated with the main subproblem we have $\gcd(s, q) = 1$. The modular components $\hat{s}_i$ of s are coprime to their respective moduli $p_i^{e_i}$. This allows us to relate $\hat{s}_i$ to so called *indices* s_i in an additive domain.

When p_i is odd, there exist integers g_i which are called *primitive roots* for all powers of p_i. A primitive root g_i of $p_i^{e_i}$ has the property that the multiplicative order of g_i is $\phi(p_i^{e_i})$. Thus, relative to a primitive root g_i, each integer coprime to $p_i^{e_i}$ is associated with a unique *index* between 0 and $N_i - 1$ where $N_i = \phi(p_i^{e_i})$. In particular, we denote the index of $\hat{s}_i$ by s_i and it is defined by

$$\hat{s}_i \equiv g_i^{s_i} \pmod{p_i^{e_i}}.$$

Note that computing the index s_i appears to require a linear search through the set of all indices. Hence, it can be prohibitively expensive to do so. We use indices only as a theoretical tool and do not compute them explicitly.

As an example of primitive roots and indices, take $\hat{s}_i = 227$ modulo $p_i^{e_i} = 7^3$. Given the primitive root $g_i = 3$, the index of $\hat{s}_i$ is $s_i = 175$ since $227 \equiv 3^{175} \pmod{7^3}$.

When p_i is even, i.e., $i = 1$ and $p_1 = 2$, a complication arises. No power of two except for 2^1 and 2^2 has primitive roots. However, there exists a unique pair of indices, s_0 and s_1, such that

$$\hat{s}_1 \equiv (-1)^{s_0} 5^{s_1} \pmod{2^{e_1}}.$$

The integers -1 and 5 take on a similar role as the primitive roots. The powers of 5 trace out the set of integers which are congruent to 1 modulo 4. Multiplying this set by -1 gives the set of integers which are congruent to 3 modulo 4. Together these two sets cover all the $\phi(2^{e_1})$ odd integers. The index $s_0 \in \{0, 1\}$ is uniquely determined modulo $N_0 = \min\{e_1, 2\}$. It equals zero if and only if $\hat{s}_1 \equiv 1 \pmod 4$. The index s_1 is uniquely determined modulo $N_1 = \max\{2^{e_1-2}, 1\}$. Note that $N_0 N_1 = \phi(2^{e_1}) = 2^{e_1-1}$.

As an example, consider $\hat{s}_1 = 43$ modulo $2^6 = 64$. From $43 \equiv (-1)5^{13} \pmod{2^6}$ it follows that the indices of $\hat{s}_1$ are $s_0 = 1$ and $s_1 = 13$.

Primitive roots turn modular multiplication into modular addition in much the same way as logarithms turn multiplication into addition. For instance, if p_i is odd, then

$$\hat{a}_i \hat{b}_i \equiv g_i^{a_i} g_i^{b_i} \equiv g_i^{a_i + b_i} \pmod{p_i^{e_i}}$$

and if p_1 is even, then

$$\hat{a}_1\hat{b}_1 \equiv (-1)^{a_0}5^{a_1}(-1)^{b_0}5^{b_1} \equiv (-1)^{a_0+b_0}5^{a_1+b_1} \pmod{2^{e_1}}.$$

To handle general q, we set $h = 1$ if q is odd, and $h = 0$ if q is even. Thus, we say that for any q, an integer s which is coprime to q has t modular components $\hat{s}_i$ for $i = 1, 2, \ldots, t$ and t or $t+1$ indices s_i for $i = h, h+1, \ldots, t$.

3.2 A Leader Set for the Main Subproblem in the Additive Domain

We are now ready to describe a leader set for the main subproblem in the additive domain. The cycle of a contains $b = an^x \bmod q$ for some x. The components of this equation in the additive domain expands to

$$b_i = (a_i + n_i x) \bmod N_i, \quad \text{for all } h \le i \le t,$$

where a_i, b_i, and n_i are the indices of a, b, and n, respectively. Consider the sequence $x = 0, 1, \ldots$ and note that b_i is periodic and takes on values in a subset of $\mathbb{Z}_{N_i}$. Specifically, define

$$d_i = \gcd(n_i, N_i) \quad \text{and} \quad K_i = \frac{N_i}{d_i} \tag{2}$$

and observe from

$$n_i x \equiv 0 \pmod{N_i} \Leftrightarrow (n_i/d_i)x \equiv 0 \pmod{K_i} \Leftrightarrow x \equiv 0 \pmod{K_i} \tag{3}$$

that b_i has a period of K_i. The first equivalence in (3) comes from factoring out d_i and the second equivalence comes from the fact that n_i/d_i and K_i are coprime. Note that $b = a$ precisely when x is a multiple of *all* the periods. Thus, the cycle length is

$$\ell = \operatorname{lcm}(K_h, K_{h+1}, \ldots, K_t). \tag{4}$$

The leader set which forms the basis for GKKleaders is stated in Theorem 3.1. It is simply the direct product of integers in the range $(0 : k)$.

Theorem 3.1 Leader set for the main subproblem. *Define for* $i = h, h+1, \ldots, t$

$$L_i = \gcd(K_i, \operatorname{lcm}(K_h, K_{h+1}, \ldots, K_{i-1})). \tag{5}$$

The set

$$\mathcal{S} = \{(s_h, s_{h+1}, \ldots, s_t) : 0 \le s_i < d_i L_i, \text{ for all } i = h, h+1, \ldots, t\} \tag{6}$$

is a leader set for the main subproblem.

Proof. See Section B.1. □

The following Corollary allows us to produce a more efficient GKKleaders algorithm by setting $z_i = 1$ in the inverse CRT map (1) of Section 3.1.1.

Corollary 3.2. *The set*

$$\widetilde{\mathcal{S}} = \{(c_h + s_h, c_{h+1} + s_{h+1}, \ldots, c_t + s_t) : 0 \le s_i < d_i L_i, \text{ for all } i = h, h+1, \ldots, t\}$$

with arbitrary constants c_i *is a leader set.*

3.2.1 *Avoiding index calculations.* It is possible to obtain the crucial quantities without computing the indices of n. This crucial observation is in large parts what makes the GKKLEADERS algorithm so efficient.

The period K_i defined in (2) is closely related to the multiplicative order ℓ_i of n modulo $p_i^{e_i}$. In fact, from $\hat{n}_i \equiv g_i^{n_i} \pmod{N_i}$ it follows that $K_i = \ell_i$ for all odd primes p_i. Thus, it is possible to first determine N_i, then compute K_i, e.g., by the fast algorithm given in Section A.2, followed by setting $d_i = N_i/K_i$. Thus, the index n_i need not be computed.

For $p_1 = 2$ we do not even have to compute the multiplicative order of n. Instead we determine N_1, then compute d_1 followed by K_1, where d_1 is obtained using Lemma 3.3, again without computing an index.

LEMMA 3.3. *Either* $\hat{n}_1 = 1 + 4k$ *or* $\hat{n}_1 = 2^{e_1} - (1 + 4k)$. *In both cases,* $d_1 = \gcd(k, N_1)$.

PROOF. See Section B.2 □

We also need to determine N_0, d_0, and K_0. Since n_0 is just 0 or 1, it is trivial to compute. From n_0 we then get $d_0 = \gcd(n_0, N_0)$ and $K_0 = N_0/d_0$.

3.2.2 *Example.* It is best to describe how to apply Theorem 3.1 by using an example.

The $m \times n = 68 \times 227$ transposition problem leads to $q = mn - 1 = 15435$. The prime power factorization of q is given in the second to fourth columns of the table below. The entries in the fifth row are the products of all the entries above.

i	p_i	e_i	$p_i^{e_i}$	N_i	$\hat{n}_i$	n_i	K_i	d_i	L_i	d_iL_i	g_i	M_i	z_i
1	3	2	9	6	2	1	6	1	1	1	2	1715	2
2	5	1	5	4	2	1	4	1	2	2	2	3087	3
3	7	3	343	294	227	175	42	7	6	42	3	45	61
			15345	7056						84			

Since q is odd, we get $h = 1$ and the additive domain of indices becomes $\mathbb{Z}_{N_1} \times \mathbb{Z}_{N_2} \times \mathbb{Z}_{N_3}$, where $N_i = \phi(p_i^{e_i})$. The multiplicative orders of n give K_i and $d_i = N_i/K_i$. Since $d_i = 1$ for $i \in \{1, 2\}$, we get $g_1 = n \bmod p_1^{e_1} = 2$ and $g_2 = n \bmod p_2^{e_2} = 2$. For $p_3^{e_3} = 7^3$ we find $g_3 = 3$ after a search, see Section A.3. After computing L_i we obtain d_iL_i and the leader set in Theorem 3.1 becomes

$$\mathcal{S} = \{(s_1, s_2, s_3) : s_1 = 0,\ s_2 = 0, 1,\ s_3 = 0, 1, \ldots, 41\},$$

which can also be expressed as the Cartesian product $\mathcal{S} = \{0\} \times \{0, 1\} \times \{0, 1, \ldots, 41\}$. The cycle length is $\ell = \operatorname{lcm}(K_1, K_2, K_3) = 84$. The number of leaders is $N_1N_2N_3/\ell$, which happens to be 84 as well.

Take $(s_1, s_2, s_3) = (0, 1, 19)$ as an example of the translation of a leader from the additive to the multiplicative domain modulo q:

$$s = (M_1 g_1^{s_1} + M_2 g_2^{s_2} + M_3 g_3^{s_3}) \bmod q = (1715 + 3087 \cdot 2 + 45 \cdot 3^{19}) \bmod 15435 = 14009.$$

3.3 A Complete Set of Leaders

In Section 3.2, we outlined the GKKLEADERS algorithm for the main subproblem.

In the following, assume that the prime power factorization of q/v is

$$q/v = p_1^{f_1} p_2^{f_2} \cdots p_t^{f_t}, \quad 0 \le f_i \le e_i,$$

which means that the prime power factorization of v must be

$$v = p_1^{e_1 - f_1} p_2^{e_2 - f_2} \cdots p_t^{e_t - f_t}.$$

In Section 3.3.1, we let $f_0 = f_1$ when $h = 0$.

3.3.1 *Computing subproblem quantities.* The central quantities M_i, N_i, K_i, d_i, and L_i are dependent on v. Notably, the primitive roots g_i are constant with respect to v (see Section A.3). The notation $M_i(f_i)$, $N_i(f_i)$, $K_i(f_i)$, and $d_i(f_i)$ distinguishes the different values of the quantities and emphasizes that their dependence on v is restricted to the exponent f_i. From (5) we observe that L_i depends on $K_j(f_j)$ for $h \le j \le i$. We therefore use the notation $L_i(v)$ to distinguish the different values of L_i. Without an argument, the quantities refer to those of the main subproblem.

Since $K_i(f_i)$ is related to the multiplicative order of n modulo $p_i^{f_i}$ it appears to be costly to compute. However, Lemma 3.4 establishes a connection between $K_i(f_i)$ and $K_i = K_i(e_i)$, which makes the computation of $K_i(f_i)$ trivial when K_i is known.

LEMMA 3.4. *If $i \in \{1, 2, \ldots, t\}$, then for any f_i in the range $0 \le f_i < e_i$*

$$K_i(f_i) = \begin{cases} 1 & \text{if } f_i = 0, \\ K_i(f_i + 1) & \text{if } 0 < f_i < e_i \text{ and } K_i(f_i + 1) < p_i, \\ K_i(f_i + 1)/p_i & \text{otherwise.} \end{cases}$$

PROOF. See Section B.3. □

The LCM computation (4), which computes the cycle length ℓ, is arranged in Algorithm 4 so that the L_i, defined in (5), fall out as byproducts.

Algorithm 4 LCM$(K_h(f_h), \ldots, K_t(f_t))$

Input: $K_i(f_i)$ for $i = h, h+1, \ldots, t$ where f_i are the exponents associated with the subproblem divisor v.

Purpose: Computes the cycle length

$$\ell = \operatorname{lcm}(K_h(f_h), K_{h+1}(f_{h+1}), \ldots, K_t(f_t))$$

and $L_i(v)$ for all $i = h, h+1, \ldots, t$.

1: $\ell \leftarrow 1$
2: **for** $i = h, h+1, \ldots, t$ **do**
3: $\quad L_i(v) \leftarrow \gcd(\ell, K_i(f_i))$
4: $\quad \ell \leftarrow \frac{K_i(f_i)\ell}{L_i(v)}$
5: **end for**

3.3.2 *Preparing for the generation of leaders.* For an arbitrary subproblem corresponding to v, the translation from the additive description of the leader set into the multiplicative domain involves computing the modular components of each leader using the primitive roots, then applying the inverse CRT map (1), and finally multiplying the generated leader by v. Let $\mathcal{S}_v$ denote the leader set,

given by Theorem 3.1, for the reduced subproblem corresponding to v. The indices $(s_h, s_{h+1}, \ldots, s_t) \in \mathcal{S}_v$ are mapped to the integer $s \in \mathbb{Z}_q$ according to

$$\begin{aligned} s &= v \left[\left(\sum_{i=1}^{t} M_i(f_i) \cdot (g_i^{s_i} \bmod p_i^{f_i}) \right) \bmod q/v \right] \\ &= \left(\sum_{i=1}^{t} p_i^{e_i - f_i} M_i g_i^{s_i} \right) \bmod q. \end{aligned} \tag{7}$$

If q is even, simply replace $g_1^{s_1}$ with $(-1)^{s_0} 5^{s_1}$. According to Theorem 3.1, the range of s_i is $0 \le s_i < d_i(f_i) L_i(v)$. The upper bound, $d_i(f_i)L_i(v)$ is maximal when $f_i = e_i$, i.e., for the main subproblem. Thus, the powers of g_i need to be computed only once and later reused for all subproblems. Specifically, when solving the main subproblem we compute and store

$$V_i(s_i) = M_i g_i^{s_i} \bmod q \quad \text{for } i = 1, 2, \ldots, t.$$

This simplifies (7) and reduces the number of operations considerably when there are multiple subproblems. When q is even, V_1 actually depends on both s_0 and s_1: $V_1(s_0, s_1) = M_1(-1)^{s_0} 5^{s_1} \bmod q$.

3.3.3 *Generating leaders.* When evaluating (7) for all possible index combinations in sequence, it is possible to further reduce the number of operations. Denote the partial sum of the first k terms in (7) by

$$S_k = \left(\sum_{i=1}^{k} p_i^{e_i - f_i} V_i(s_i) \right) \bmod q = \begin{cases} p_i^{e_i - f_i} V_i(s_i) \bmod q & \text{if } k = 1, \\ (S_{k-1} + p_i^{e_i - f_i} V_i(s_i)) \bmod q & \text{otherwise.} \end{cases}$$

Thus, $s = S_t$. After prescaling $V_i(s_i)$ with $p_i^{e_i - f_i}$, each leader is generated by little more than a single modular addition. This is a significant reduction compared to the $t-1$ modular additions, $2t$ modular multiplications, and t modular exponentiations suggested by (7).

3.4 The GKKleaders Algorithm

We are finally in a position to present GKKleaders (Algorithm 5). The loop over all divisors of v is cheap since the prime power factorization of q is already known.

The search for primitive roots is a relatively expensive operation. Fortunately, when $d_i = 1$, which it frequently is, it follows from $K_i = \phi(p_i^{e_i})$ that $n \bmod p_i^{e_i}$ is a primitive root of $p_i^{e_i}$.

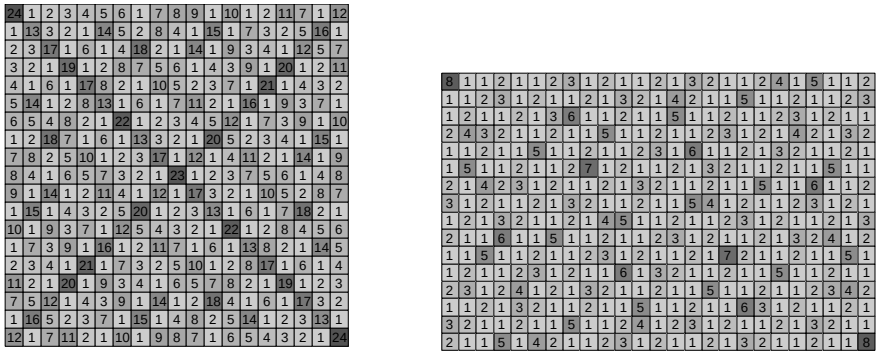
4. SQUARE IN-PLACE MATRIX TRANSPOSITION

When the matrix is square, i.e., $m = n$, the diagonal elements lead to singleton cycles. The off-diagonal element a_{ij} pairs with a_{ji}, forming a total of $n(n-1)/2$ cycles of length two. An algorithm such as GKKleaders is not required in this case since special-purpose cache-blocked in-place transposition algorithms for square matrices already exist.

It is interesting to note that the square case, although seemingly simple, still has many subproblems scattered in the domain. For instance, consider $m \times n = 19 \times 19$, which leads to $q = mn - 1 = 360 = 2^3 \cdot 3^2 \cdot 5$. There are 24 divisors of q and hence

Algorithm 5 GKKLEADERS(m, n)

Input: Transposition problem of size $m \times n$.
Purpose: Computes a complete leader set $\mathcal{S}$ for the transposition permutation.
1: Factor $q = mn - 1 = p_1^{e_1} p_2^{e_2} \cdots p_t^{e_t}$ (Section A.1).
2: Set $h = 0$ if q is even and $h = 1$ otherwise.
3: Factor $p_i - 1$ for $i = 1, 2, \ldots, t$ for later use in steps 4 and 6. subalgorithms (Section A.1).
4: Compute $K_i = K_i(e_i)$ for $i = h, h+1, \ldots, t$ (Section A.2).
5: Compute $K_i(f_i)$ for $f_i = 0, 1, \ldots, e_i - 1$ from K_i for $i = h, h+1, \ldots, t$ (Section 3.3.1).
6: Compute primitive roots g_i for all odd p_i for which $K_i \neq \phi(p_i^{e_i})$ (Section A.3).
7: Compute L_i for $i = h, h+1, \ldots, t$ (Algorithm 4).
8: Compute $V_i(s_i)$ for $s_i = 0, 1, \ldots, d_i L_i - 1$ for $i = 1, 2, \ldots, t$ (Section 3.3.2).
9: **for** each divisor $v \neq q$ of q **do**
10: Compute $L_i(v)$ for $i = h, h+1, \ldots, t$ (Algorithm 4).
11: Temporarily scale $V_i(s_i)$ with $p_i^{e_i - f_i}$ for $i = 1, 2, \ldots, t$
12: Generate leaders for subproblem v using summation with partial sums (Section 3.3.3).
13: **end for**



(a) $m \times n = 19 \times 19$ (b) $m \times n = 16 \times 25$

Fig. 3. Visualization of subproblems for a square and a rectangular transposition problem. The subproblems (divisors) are numbered in ascending order.

there are 24 subproblems if we count $v = q$. The offset zero is technically not a part of any subproblem, but for consistency we associate it with $v = q$ since $\gcd(0, q) = q$. Figure 3(a) visualizes the subproblem membership of each of the 361 integers $(0 : q - 1)$. The integers are laid out as the elements of a matrix in CM order. A light (dark) shade represents a subproblem with a small (large) divisor v. The rectangular 16×25 example in Figure 3(b) is included for reference.

Note that converting a square matrix from CM or RM format to a blocked format involves at least one rectangular transposition except for rare cases such as $M = n_b$. One of the notable exceptions is the conversion between CCRB and RRRB for a square matrix with square blocks. It leads to a fused operation with square transposition both as the outer and inner problem.

5. ARBITRARY MATRIX DIMENSIONS

In this section, we assume arbitrary matrix dimensions, i.e.,

$$m = Mm_b + r_m \quad \text{and} \quad n = Nn_b + c_n, \quad \text{where } 0 \le r_m < m_b,\ 0 \le c_n < n_b.$$

The integers r_m and c_n are the number of trailing rows and columns, respectively.

The standard blocked formats require that the matrix sizes m and n are divisible by the block sizes m_b and n_b, i.e., $r_m = c_n = 0$. Padding, i.e., deliberately storing the matrix as a leading submatrix of a larger matrix to satisfy this constraint, is an effective way to bypass the problem of arbitrary matrix dimensions. However, a legacy library can not rely on padding since it would require changing existing software.

A first generalization is to maintain the notion that a blocked format lays out the blocks, including the trailing blocks, in either CM or RM format. It is a reasonable approach, but it makes conversion and transposition more expensive than necessary. For instance, our in-place conversion algorithms can not be applied directly to matrices stored using this scheme.

Below we describe another generalization to which our in-place conversion algorithms apply directly. Partition A into four submatrices,

$$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}$$

with A_{22} of size $r_m \times c_n$. There are MN blocks in A_{11} of size $m_b \times n_b$; M blocks in A_{12} of size $m_b \times c_n$; N blocks in A_{21} of size $r_m \times n_b$; and one block in A_{22} of size $r_m \times c_n$. When A is stored in CM or RM format, these submatrices are interleaved in memory. However, a simple procedure that we call *packing*, which is essentially what Dow [1995] refers to as a *cut*), rearranges the matrix in a (semi-)in-place fashion while reading and writing the matrix only once. Assuming that A is initially stored in CM format, the packing procedure rearranges the submatrices so that they are stored contiguously in the order A_{11}, A_{12}, A_{21}, followed by A_{22}. Each submatrix is stored in CM format. The key idea is that each submatrix has a *homogeneous* block size, i.e., all blocks have the same size, and can therefore be stored in any storage format independently of the other submatrices.

A suitable generalization of CRRB, for instance, is obtained by converting each of the four submatrices of A from CM to CRRB format. The block sizes of the four submatrices are different, but each submatrix has a homogeneous block size. Our conversion algorithms therefore apply to all of the submatrices.

With this generalization, it is simple and cheap to convert between pairs of generalized blocked formats since the conversion reduces to conversion of each submatrix. For certain conversions, such as from CCRB to RRRB, the roles of the submatrices A_{12} and A_{21} need to be reversed.

The next section describes the packing procedure in more detail, see also Gustavson [2008].

5.1 Packing and Unpacking

Given is n_{seg} segments with n_{con} consecutive elements each placed in memory with a stride of $n_{\text{con}} + n_{\text{sep}}$. The first segment starts at offset zero. In other words, there

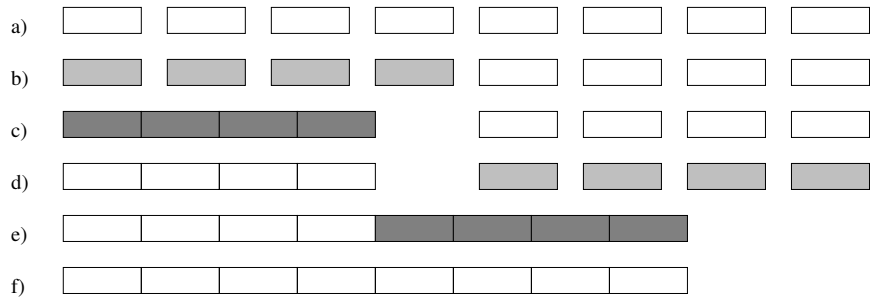


Fig. 4. Parallel scheme for the PACK procedure. The $(n_{\text{seg}} - 1)n_{\text{sep}}$ intermediate elements are initially copied to workspace and at the end they are copied back. a) 8 segments initially scattered in memory, b) 4 threads copy one segment each into cache-resident workspace, c) 4 threads synchronize and write their copied segments back, d) the next group of 4 segments is copied, e) and written back after synchronization. The final arrangement is illustrated in f).

are n_{sep} elements between two adjacent segments that are not part of any segment. Figure 4(a) illustrates the case $n_{\text{seg}} = 8$.

The PACK procedure rearranges the segments so that they are stored contiguously starting at offset zero. It maintains the order of the elements that belong to segments as well as the order of the intermediate elements. The UNPACK procedure is the inverse of PACK.

A semi-in-place implementation of PACK first copies the $(n_{\text{seg}} - 1)n_{\text{sep}}$ intermediate elements into workspace. The segments are then moved to their new locations one after the other. Finally, the copied elements are written back directly following the packed segments.

A shared-memory parallelization of PACK using p threads is illustrated by the example $p = 4$ and $n_{\text{seg}} = 8$ in Figure 4. A barrier synchronization ensures that the segments have been completely copied before they are written back.

6. COMPUTATIONAL EXPERIMENTS

In-place conversion is necessary in cases where out-of-place conversion is not possible due to a lack of available memory. Even when out-of-place conversion is an option, it might turn out that in-place conversion is faster, e.g., due to its smaller memory footprint.

Out-of-place transformations can be slower for various reasons. One example is on systems with a write-back cache in combination with a write-allocate policy. On such systems, a write miss triggers a read of the cache line, even if the entire cache line is going to be overwritten. This is precisely the scenario in out-of-place transformations. In contrast, an in-place transformation reads a cache line which it later overwrites, hopefully while it is still in the cache. Thus, both the read and the write are essential operations.

The potential difference between in-place and out-of-place is captured by the following benchmarks:

—**copy** (t_{copy}): copies a vector ($w \leftarrow v$).

—**scale** (t_{scale}): scales a vector in-place ($v \leftarrow \alpha v$).

Table I. Characteristics of the HPC2N systems Akka and Sarek.

Name	*Akka*	*Sarek*
Processor	Dual Intel Xeon QC L5420	Dual AMD Opteron 248
Frequency	2.5 GHz	2.2 GHz
Compiler	PathScale 3.2	PathScale 3.2
Switches	`-O3 -mp`	`-O3 -mp`
t_{copy}	3.02 ns (2530 MB/s)	6.52 ns (1170 MB/s)
t_{scale}	1.62 ns (4710 MB/s)	4.54 ns (1680 MB/s)
$t_{\text{copy}}/t_{\text{scale}}$	1.86	1.44

Both benchmarks are perfectly parallelizable and use only stride-1 accesses, so they are idealized out-of-place (copy) and in-place (scale) transformations. Measurements need to be carried out on vectors that do not fit in cache in order to reflect the main memory bandwidth. After dividing the running time with the length of the vectors, the quantities t_{copy} and t_{scale} are obtained in the unit seconds per element. We compare the benchmarks against the normalized running time of in-place conversion. Ideally, the results should be close to t_{scale}.

6.1 Experimental Setup

In all our experiments, we selected the best time after executing each problem on $p = 1, 2, \ldots, n_c$ threads, where n_c is the number of cores on one node of the machine. To reduce the impact of system noise on the measurements, we executed each problem setup three times and selected the best. Table I lists some information about the two machines at the High Performance Computing Center North (HPC2N) facility in Umeå, Sweden that we ran our tests on.

Note that we would expect the ratio $t_{\text{copy}}/t_{\text{scale}}$ to be close to 1.5, since the out-of-place transformation has potentially 50% more memory traffic.

6.2 Experiments

The computations involved in finding cycle leaders are not for free. In Section 6.2.1, we examine the overheads of the three representative cycle leader algorithms BITTABLE (essentially the approach of Berman [1958]), BRENNER (essentially the approach of Brenner [1973]), and GKKLEADERS (Algorithm 5) during in-place transposition. See Section 7 for a brief explanation of BITTABLE and BRENNER.

We examine the conversion between pairs of formats in Section 6.2.2.

6.2.1 *Cycle-Following Algorithms: Performance and Overhead.* We measured the time spent searching for cycle leaders while executing ADJACENTSWAP (Algorithm 3) with $n_{\text{ind}} = 1$ and $L = 64$ on 50 problems with $m \times n$ chosen randomly from $(2 : 500)$. All threads are cooperatively performing a single in-place matrix transposition ($n_{\text{ind}} = 1$). Choosing the problem size at random increases the chance to detect problematic cases, since the numbers m or n and $q = mn - 1$ are critical to the cycle structure.

Figure 5 summarizes the overhead, measured on Akka, calculated as a percentage of the total execution time. The overhead for both BRENNER and BITTABLE is around 10%, whereas the overhead of GKKLEADERS is a fraction of one percent for all but small matrices. Thus, the overhead of our a priori method is negligible and substantially lower than the two search-based algorithms. Note that the overhead of the search-based algorithms is around 10% because of the large chunk size (64

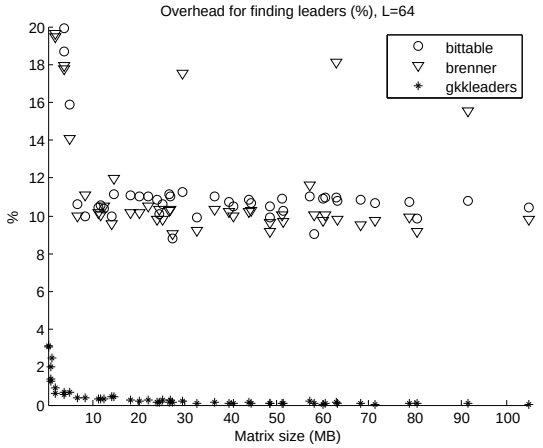


Fig. 5. Overhead of finding cycle leaders on Akka for 50 instances of AdjacentSwap with $n_{\text{ind}} = 1$ and $L = 64$.

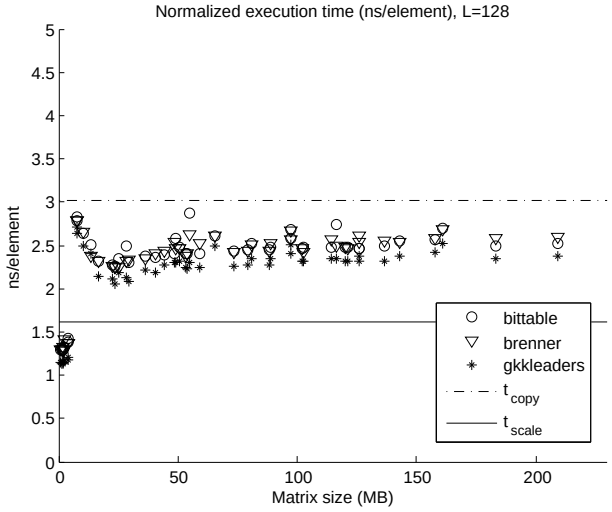


Fig. 6. Normalized execution time (nanoseconds per element) on Akka for 50 instances of AdjacentSwap with $n_{\text{ind}} = 1$ and $L = 128$.

elements). For smaller chunk sizes the ratios would increase and vice versa.

Figures 6 and 7 illustrate the normalized execution time on the 50 randomly chosen problems. This time, the block size is $L = 128$ and the same problems and

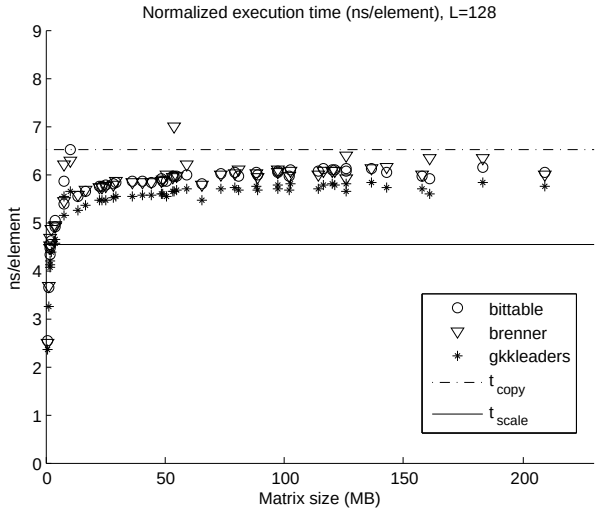


Fig. 7. Normalized execution time (nanoseconds per element) on Sarek for 50 instances of AdjacentSwap with $n_{\text{ind}} = 1$ and $L = 128$.

tests are run on both Akka and Sarek. For comparison with the practical peak, the values of t_{copy} and t_{scale} are included in the graphs. Note that the normalized execution time is often below the t_{copy} benchmark result. In those cases, in-place conversion is faster than out-of-place conversion.

6.2.2 *Data Format Conversion: Performance.* The time required to perform a data format conversion depends on many parameters such as matrix size, block size, formats involved, etc. We selected a matrix with size $m = n = 9984$ and partitioned it into blocks of size $m_b = n_b = 64$. Thus, the blocked matrix is $M \times N$ with $M = N = 156$. Since the block size divides the matrix size, there is no need for packing and unpacking. Conversion proceeds along the edges of the graph in Figure 1 and hence some conversions involve several stages. In these experiments, we have not used fused conversions between CCRB and RRRB or CRRB and RCRB. Thus, conversion between CM and RM is essentially a four-stage process. The transpositions that arise during the conversions are 156×64 or 64×156 with $L = 64$; 156×156 with $L = 4096$; or 64×64 with $L = 1$.

We have applied GKKleaders to all resulting in-place transpositions, including those that are square, except for $L = 1$ when we perform the element-wise transposition out-of-place in cache.

The normalized execution times (nanoseconds per element) observed on Akka are as follows.

From \ To	CM	CCRB	CRRB	RCRB	RRRB	RM
CM		2.2	4.8	4.1	6.6	8.9
CCRB	2.3		2.5	1.8	4.4	6.7
CRRB	4.8	2.6		4.4	1.9	4.1
RCRB	4.1	1.8	4.4		2.6	4.8
RRRB	6.7	4.4	1.9	2.6		2.3
RM	8.9	6.6	4.1	4.8	2.3	

The average (normalized) time per stage is shown below. Recall from Table I that $t_{\text{copy}} = 3.02$ and $t_{\text{scale}} = 1.62$ on Akka.

From \ To	CM	CCRB	CRRB	RCRB	RRRB	RM
CM		2.2	2.4	2.1	2.2	2.2
CCRB	2.3		2.5	1.8	2.2	2.2
CRRB	2.4	2.6		2.2	1.9	2.1
RCRB	2.1	1.8	2.2		2.6	2.4
RRRB	2.2	2.2	1.9	2.6		2.3
RM	2.2	2.2	2.1	2.4	2.3	

The results show that the performance is good and without anomalies.

On Sarek, the corresponding results for pairwise conversion are listed below.

From \ To	CM	CCRB	CRRB	RCRB	RRRB	RM
CM		5.8	11.3	10.9	16.3	22.2
CCRB	5.8		5.5	5.0	10.5	16.3
CRRB	11.3	5.5		10.5	5.0	10.9
RCRB	10.9	5.0	10.5		5.5	11.3
RRRB	16.3	10.5	5.0	5.5		5.8
RM	22.2	16.3	10.9	11.3	5.8	

Similarly, the average (normalized) time per stage shows good performance for all pairs. Recall that $t_{\text{copy}} = 6.52$ and $t_{\text{scale}} = 4.54$ on Sarek.

From \ To	CM	CCRB	CRRB	RCRB	RRRB	RM
CM		5.8	5.7	5.5	5.4	5.5
CCRB	5.8		5.5	5.0	5.2	5.4
CRRB	5.7	5.5		5.2	5.0	5.5
RCRB	5.5	5.0	5.2		5.5	5.7
RRRB	5.4	5.2	5.0	5.5		5.8
RM	5.5	5.4	5.5	5.7	5.8	

7. RELATED WORK ON IN-PLACE TRANSPOSITION

There is a vast literature on in-place matrix transposition and the more general topic of in-place permutation. There are also many published semi-in-place and out-of-place algorithms. Here, we only mention the references which have had the most influence on our work.

By using a bit-table one can tag which cycles of the transposition permutation that have been shifted. A new cycle is easily determined by finding an unmarked element in the bit-table. This is essentially the method proposed by Berman [1958]. It requires $\mathcal{O}(mn)$ extra bits of storage unless the bit-table is embedded in the data.

One can go one step further and remove the bit-table completely. Thus, one needs a method to distinguish between a new cycle and a previous cycle, which was the original reason for the bit-table. By designating the smallest element in a cycle as the leader of that cycle, it is possible to determine if a candidate is a leader or not by traversing its cycle. If a smaller element is found, then the candidate is rejected and otherwise it is accepted. This algorithm is originally due to J. C. Gower. It was presented by Windley [1959] and later improved and implemented by Brebner and Laflin [1970]. One of the improvements proposed by Brebner and Laflin [1970] is to exploit dual cycles, i.e., if s is a cycle leader, then $q - s$ is a cycle leader for the dual cycle. A cycle and its dual sometimes coincide. However, this can be detected and handled efficiently if a cycle and its dual are shifted simultaneously [Brebner and Laflin 1970; Brenner 1973; Cate and Twigg 1977].

The computational complexity of Gower's algorithm when applied to general permutations is $\mathcal{O}(q^2)$ in the worst case. If both the permutation and its inverse are known, then searching for a smaller element in both directions simultaneously reduces this upper bound to $\mathcal{O}(q \log q)$ [Knuth 1971; 1998; Fich et al. 1995; Gustavson and Swirszcz 2007]. We are not aware of any other complexity results related to the transposition permutation.

Windley [1959] described, apart from Gower's algorithm, also an algorithm of his own. It is in-place but not based on cycle leaders. Windley's algorithm was later implemented by Boothroyd [1967].

Pall and Seiden [1960] analyze the transposition permutation using Abelian group theory and they give a pen-and-paper algorithm for a priori determination of cycle leaders and lengths. An important observation they make is that the cycle leader problem naturally partitions into one subproblem for each divisor of $q = mn - 1$.

Brenner [1973] uses the result of Pall and Seiden [1960] to produce an effective method which reduces the number of rejected candidates of Gower-type algorithms.

Gustavson and Swirszcz [2007] improved on Brenner's algorithm in several ways. They search for smaller elements in both directions simultaneously, which makes their algorithm $\mathcal{O}(q \log q)$ in the worst case. They also point out that computing $kn \mod q$ in 32-bit arithmetic leads to a possible catastrophic overflow. They go on to prove that the equivalent 32-bit expression $kn - q \left\lfloor \frac{k}{m} \right\rfloor$ is correct in wrap-around arithmetic, even though both multiplications might overflow.

Dow [1995] presents algorithms and techniques suitable for vector computers and machines wtih caches. He uses cutting and padding to reduce modestly rectangular transposition problems to square in-place transposition problems. He also gives a few multi-stage transposition algorithms, similar to the ones presented earlier by Alltop [1975].

For more information on the large topic of multi-stage matrix transposition for out-of-core or cache-based systems, see for instance Huang et al. [1993], Johnson [1995], Gustavson [2008], and Karlsson [2009].

8. CONCLUSIONS

In-place storage format conversion allows software libraries that include matrix computations to use separate internal and external data layouts even when the matrix can not be copied.

Conversion between standard and blocked storage formats can be built entirely upon in-place matrix transposition in a parallel and cache-efficient way. General matrix dimensions are best handled by partitioning the matrix into four submatrices so that each submatrix has a homogeneous block size. The submatrices can then be stored separately and independently in any storage format.

A priori determination of cycle leaders is necessary for effective load balancing in a parallel environment. Our new GKKLEADERS algorithm has a very small overhead when applied to storage format conversion and is superior to previously known cycle-following algorithms.

Computational experiments suggest that in-place conversion is fast in general and that a one-stage in-place conversion is sometimes even faster than out-of-place conversion.

APPENDIX

A. SUBALGORITHMS OF GKKLEADERS

This appendix discusses implementation issues related to three of the key subalgorithms in GKKLEADERS (Algorithm 5).

A.1 Prime Factorization

GKKLEADERS and its subalgorithms make use of the prime factorization of q as well as of $p-1$ for all primes p in the factorization of q. To find the prime factorization of $\phi(p^e) = p^{e-1}(p-1)$ we use the factorization of $p-1$ and append p^{e-1}. Since q is relatively small, e.g., less than $2^{31}-1$, we may use a trial division algorithm in conjunction with a precomputed table containing all primes less than $\sqrt{2^{31}-1}$. This is practical since the table has only 4792 entries. Even if q is a large prime, no more than this number of trials is required before the trial division algorithm detects that q is indeed a prime.

Storing a table of primes becomes impractical at some point. With a chunk size of $L = 64$, the matrix must have at least $64 \cdot 2^{31} = 2^{37}$ entries before q exceeds 2^{31}. Since this is equivalent to one terabyte of double precision matrix data, a large q rarely occurs.

A.2 Multiplicative Order

The multiplicative order of n modulo p^e is the smallest integer ℓ such that $n^\ell \equiv 1 \pmod{p^e}$. Testing whether a given integer x is a *multiple* of ℓ is cheap. Simply compute $c = n^x \bmod p^e$ and if $c = 1$ then $\ell \mid x$. Once the prime factorization of $\phi(p^e)$ is known, Algorithm 6 computes ℓ cheaply, by taking out one factor at a time from $x = \phi(p^e)$. Since Algorithm 6 is repeatedly computing powers of n modulo p^e, the fast exponentiation should be arranged so that a table of n^{2^i} for $i = 0, 1, \ldots, \lfloor \log_2 e \rfloor$ is precomputed and reused.

A.3 Primitive Root

Finding a primitive root g of an odd prime p and all powers of p is one of the more computationally intensive parts of GKKLEADERS. The algorithm that we use has much in common with Algorithm 6 in the sense that it tests whether the multiplicative order of a candidate primitive root is $\phi(p) = p-1$. Since we are

Algorithm 6 MULTIPLICATIVEORDER(n, p, e)

Input: Positive integer n, a prime p, and a positive exponent e.
Purpose: Computes the multiplicative order ℓ of n modulo p^e.

```
1: Set x ← φ(p^e)
2: for each prime r in the decomposition of φ(p^e) do
3:    Compute c ← n^{x/r} mod p^e using fast exponentiation
4:    while c = 1 and r | x do
5:       x ← x/r
6:       Compute c ← n^{x/r} mod p^e using fast exponentiation
7:    end while
8: end for
9: Exit with ℓ ← x
```

satisfied with any primitive root, we may as well calculate the least primitive root which is typically very small. A small primitive root also leads to faster modular multiplication via conditional subtraction.

A primitive root g of p is a primitive root for all powers of p unless $g^{p-1} \equiv 1 \pmod{p^2}$, in which case $g+p$ is such a primitive root. It is an extremely rare event that the least primitive root of p fails the test. The basic algorithm is given in Algorithm 7. There are several ways in which the basic algorithm can be improved.

Algorithm 7 PRIMITIVEROOT(p, e)

Input: An odd prime p and a positive exponent e.
Purpose: Computes a primitive root g of p^f for $f = 1, 2, \ldots, e$.

```
1: for g = 2, 3, ... do
2:    for each prime r in the decomposition of φ(p) do
3:       Compute c ← g^{φ(p)/r} mod p using fast exponentiation
4:       if c = 1 then
5:          Reject g and continue with the next candidate
6:       end if
7:    end for
8:    if g^{φ(p)} ≡ 1 (mod p^2) then
9:       g ← g + p
10:   end if
11:   Exit with g as a primitive root
12: end for
```

For instance, there is no need to test a candidate g which is a power of some other integer since such a number can not be a primitive root. Furthermore, some primitive roots occur much more frequently than others, so trying the candidates in ascending order is probably not optimal.

B. PROOFS

B.1 Proof of Theorem 3.1

PROOF. We first verify that $\mathcal{S}$ has the correct size by observing that

$$\ell|\mathcal{S}| = \ell \prod_{i=h}^{t} d_i L_i = \prod_{i=h}^{t} N_i = \phi(q).$$

Dividing both sides by ℓ shows that $|\mathcal{S}| = \phi(q)/\ell$, as required of a leader set.

The idea behind the rest of the proof is to show that for each leader $a = (a_h, \ldots, a_t) \in \mathcal{S}$, no other element $b = (b_h, \ldots, b_t)$ in the cycle of a is in $\mathcal{S}$. Some index components of a and b will then satisfy $a_i \neq b_i$. Pick the smallest such i. Since b is in the cycle of a we get $a_j + n_j x \equiv b_j \pmod{N_j}$ for all j. Exploiting that $a_j = b_j$ for $j < i$ leads to

$$\begin{aligned} x &\equiv 0 \pmod{K_h}, \\ &\vdots \\ x &\equiv 0 \pmod{K_{i-1}}, \\ a_i + n_i x &\equiv b_i \pmod{N_i}. \end{aligned} \tag{8}$$

Congruences h through $i-1$ are solved simultaneously by

$$x = \operatorname{lcm}(K_h, K_{h+1}, \ldots, K_{i-1})\tilde{x}, \quad \tilde{x} \in \mathbb{Z}.$$

Substitute x into the i-th congruence above and get

$$a_i + n_i F_i \tilde{x} \equiv b_i \pmod{N_i}$$

where $F_i = \operatorname{lcm}(K_h, K_{h+1}, \ldots, K_{i-1})$. This last congruence is solvable for $\tilde{x}$ if and only if

$$\gcd(n_i F_i, N_i) \mid (b_i - a_i).$$

Let $\tilde{n}_i = n_i/d_i$. Then

$$\gcd(n_i F_i, N_i) = d_i \cdot \gcd(\tilde{n}_i F_i, K_i) = d_i \cdot \gcd(F_i, K_i) = d_i L_i.$$

The first equality comes from factoring out d_i. The second equality follows from $\tilde{n}_i$ and K_i being coprime and the GCD property $\gcd(ac, b) = \gcd(a, b)$ if $\gcd(c, b) = 1$. The last equality is applying the definition of L_i. So, we get

$$d_i L_i \mid (b_i - a_i).$$

Since $b_i \neq a_i$ and $a_i \in \{0, 1, \ldots, d_i L_i - 1\}$, we have $b_i \notin \{0, 1, \ldots, d_i L_i - 1\}$. Therefore, $(b_h, b_{h+1}, \ldots, b_t) \notin \mathcal{S}$ and so $\mathcal{S}$ is a leader set. □

B.2 Proof of Lemma 3.3

Proof. We give a proof only for $\hat{n}_1 = 1+4k$ since the proof for $\hat{n}_1 = 2^{e_1} - (1+4k)$ is similar. Let $y = 2^x \tilde{y}$ where $\tilde{y}$ is positive and odd. Thus, $d_1 = \gcd(y, N_1) = 2^x$ since N_1 is a power of two. We proceed by showing that $2^x \mid k$ and $2^{x+1} \nmid k$. Write

$$n = 1 + 4k = 5^{n_1} + 2^{e_1} s = (1+4)^{n_1} + 2^{e_1} s = \sum_{i=0}^{n_1} \binom{y}{i} 2^{2i} + 2^{e_1} s$$

where s is some integer. Solve for k to obtain

$$k = y + \sum_{i=2}^{n_1} \frac{y!}{i!(y-i)!} 2^{2i-2} + 2^{e_1 - 2} s. \tag{9}$$

Note that $i!$ does not have more than $i-1$ twos in its prime factorization. Thus, after cancelling the twos in 2^{2i-2} with the twos in $i!$, we are left with (at least) 2^{i-1}. Therefore, each term in the sum is divisible by 2^{x+1}. Note that $2^{e_1 - 2}$ is divisible

by 2^{x+1} since $y < 2^{e_1-2}$. It follows that k is divisible by 2^x. Furthermore, $2^{x+1} \nmid k$ since all but one term in the right hand side of (9) is divisible by 2^{x+1}. □

B.3 Proof of Lemma 3.4

PROOF. Suppose p_i is odd. Factor the index n_i into $n_i = p_i^x s$ where s is an integer and $p_i \nmid s$. It follows that

$$K_i(f_i) = \frac{N_i(f_i)}{d_i(f_i)} = \frac{N_i(f_i)}{\gcd(n_i, N_i(f_i))} = \frac{p_i - 1}{\gcd(s, p_i - 1)} p_i^{\max\{0, f_i - x - 1\}}.$$

The first and the second equalities use the definitions of K_i and d_i in (2). The third equality uses the definition of N_i in Section 3.1.2. If $K_i(f_i + 1) < p_i$, then the exponent of p_i is 0 both for $K_i(f_i)$ and $K_i(f_i + 1)$ and hence $K_i(f_i) = K_i(f_i + 1)$, as claimed. If $K_i(f_i + 1) \geq p_i$, then the exponent of p_i for $K_i(f_i)$ is one less than for $K_i(f_i + 1)$ and hence $K_i(f_i) = K_i(f_i + 1)/p_i$, which concludes the proof. □

REFERENCES

ALLTOP, W. O. 1975. A computer algorithm for transposing nonsquare matrices. *IEEE Trans. Comput. 24,* 10, 1038–1040.

BADER, M. AND HEINECKE, A. 2008. Parallel matrix multiplication based on space-filling curves on shared memory multicore platforms. In *Proceedings of the 2008 Computing Frontiers Conf. and co-located Workshops: MAW'08 and WRFT'08.* Ischia, 385–392.

BADER, M. AND ZENGER, C. 2006. Cache oblivious matrix multiplication using an element ordering based on a Peano curve. *Linear Algebra and its Applications 417,* 2–3 (Sept.), 301–313.

BERMAN, M. F. 1958. A method for transposing a matrix. *Journal of the ACM 5,* 4, 383–384.

BOOTHROYD, J. 1967. Algorithm 302: Transpose vector stored array. *Commun. ACM 10,* 5, 292–293.

BREBNER, M. A. AND LAFLIN, S. 1970. Algorithm 380: In-situ transposition of a rectangular matrix. *Commun. ACM 13,* 5, 324–326.

BRENNER, N. 1973. Algorithm 467: Matrix transposition in place. *Commun. ACM 16,* 11, 692–694.

CATE, E. G. AND TWIGG, D. W. 1977. Algorithm 513: Analysis of in-situ transposition. *ACM Trans. Math. Softw. 3,* 1, 104–110.

CHAN, E., VAN DE GEIJN, R., QUINTANA-ORTÍ, E. S., QUINTANA-ORTÍ, G., AND VAN ZEE, F. G. 2008. Programming algorithms-by-blocks for matrix computations on multithreaded architectures. Tech. Rep. TR-08-04, Department of Computer Sciences, University of Texas as Austin, USA. Jan.

DONGARRA, J. AND KURZAK, J. 2009. QR factorization for the CELL Broadband Engine. *Scientific Programming 17,* 1–2, 31–42.

DOW, M. 1995. Transposing a matrix on a vector computer. *Parallel Computing 21,* 12, 1997–2005.

ELMROTH, E., GUSTAVSON, F., JONSSON, I., AND KÅGSTRÖM, B. 2004. Recursive blocked algorithms and hybrid data structures for dense matrix library software. *SIAM Review 46,* 1, 3–45.

FICH, F. E., MUNRO, J. I., AND POBLETE, P. V. 1995. Permuting in place. *SIAM J. Comput. 24,* 2, 266–278.

FRASER, D. 1976. Array permutation by index-digit permutation. *Journal of the ACM 23,* 298–309.

GALLIVAN, K., JALBY, W., MEIER, U., AND SAMEH, A. 1988. The impact of hierarchical memory systems on linear algebra algorithm design. *International Journal of Supercomputer Applications 2,* 1, 12–48.

Gustavson, F. 2000. New generalized matrix data structures lead to a variety of high-performance algorithms. In *Proceedings of the IFIP TC2/WG2.5 Working Conference on the Architecture of Scientific Software*. Kluwer, B.V., Deventer, Netherlands, 211–234.

Gustavson, F. 2008. The relevance of new data structure approaches for dense linear algebra in the new multicore/manycore environments. Tech. Rep. RC24599, IBM Research. (Also submitted to PARA'08).

Gustavson, F., Henriksson, A., Jonsson, I., Kågström, B., and Ling, P. 1998. Recursive blocked data formats and BLAS's for dense linear algebra algorithms. In *Applied Parallel Computing. Large Scale Scientific and Industrial Problems, 4th International Workshop, PARA '98*. Lecture Notes in Computer Science, vol. 1541. 195–206.

Gustavson, F., Karlsson, L., and Kågström, B. 2009. Distributed SBP Cholesky factorization algorithms with near-optimal scheduling. *ACM Transactions on Mathematical Software 36,* 2, 11:1–11:25.

Gustavson, F. and Swirszcz, T. 2007. In-place transposition of rectangular matrices. In *Applied Parallel Computing. State of the Art in Scientific Computing, PARA 2006*, B. Kågström et al., Eds. Lecture Notes in Computer Science, vol. 4699. Springer, 560–569.

Hong, B., Park, N., and Prasanna, V. K. 2003. Tiling, block data layout, and memory hierarchy performance. *IEEE Trans. Parall. Distrib. Syst. 14,* 7, 640–654.

Huang, C. H., Johnson, J. R., Johnson, R. W., Kaushik, S. D., and Sadayappan, P. 1993. Efficient transposition algorithms for large matrices. In *Proceedings of Supercomputing '93*. 656–665.

IBM. 1986. IBM engineering and scientific subroutine library. ESSL Guide and Reference, SA22-7272-00.

Johnson, J. R. 1995. Matrix transposition. Department of Mathematics and Computer Science, Drexel University, Philadelphia, PA 19104. Manuscript.

Karlsson, L. 2009. Blocked in-place transposition with application to storage format conversion. Tech. Rep. UMINF 09.01. ISSN 0348-0542, Department of Computing Science, Umeå University, Umeå, Sweden. Jan.

Knuth, D. 1998. *The Art of Computer Programming, Volumes 1 and 2, 3rd Edition*. Addison-Wesley.

Knuth, D. E. 1971. Mathematical analysis of algorithms. In *Proceedings of IFIP Congress*. North-Holland, 19–27.

Lam, M. D., Rothberg, E. E., and Wolf, M. E. 1991. The cache performance and optimizations of blocked algorithms. *ACM SIGARCH Computer Architecture News 19,* 2, 63–74.

Niven, I., Zuckerman, H. S., and Montgomery, H. L. 1991. *An Introduction to the Theory of Numbers, 5th Edition*. Wiley.

OpenMP.org. http://openmp.org/wp/. Accessed December 3rd 2009.

Pall, G. and Seiden, E. 1960. A problem in Abelian groups, with application to the transposition of a matrix on an electronic computer. *Mathematics of Computation 14,* 70, 189–192.

Windley, P. F. 1959. Transposing matrices in a digital computer. *The Computer Journal 2,* 1, 47–48.