Fast Multiplication of Large Permutations for Disk, Flash Memory and RAM

Vlad Slavici , Xin Dong*, Daniel Kunkle* and Gene Cooperman*
CCIS Department, Northeastern University, Boston, MA
{vslav,xindong,kunkle,gene}@ccs.neu.edu

ABSTRACT

Permutation multiplication (or permutation composition) is perhaps the simplest of all algorithms in computer science. Yet for large permutations, the standard algorithm is not the fastest for disk or for flash, and surprisingly, it is not even the fastest algorithm for RAM on recent multi-core CPUs. On a recent commodity eight-core machine we demonstrate a novel algorithm that is 50% faster than the traditional algorithm. For larger permutations on flash or disk, the novel algorithm is orders of magnitude faster. A disk-parallel algorithm is demonstrated that can multiply two permutations with 12.8 billion points using 16 parallel local disks of a cluster in under one hour. Such large permutations are important in computational group theory, where they arise as the result of the well-known Todd-Coxeter coset enumeration algorithm. The novel algorithm emphasizes several passes of streaming access to the data instead of the traditional single pass using random access to the data. Similar novel algorithms are presented for permutation inverse and permutation multiplication by an inverse, thus providing a complete library of the underlying permutation operations needed for computations with permutation groups.

Categories and Subject Descriptors: I.1.2 [Symbolic and Algebraic Manipulation]: Algebraic algorithms, Analysis of algorithms

General Terms: Algorithms, Experimentation, Performance

Keywords: permutation, permutation multiplication, permutation composition, permutation inverse, pseudo-random permutation

1. INTRODUCTION

Algorithms are introduced for efficiently executing the basic permutation operations for large permutations, permutations that range in size from 4 million points to permutations with billions of points.

*This work was partially supported by the National Science Foundation under Grant CNS-0916133.

ISSAC 2010, 25–28 July 2010, Munich, Germany.

The standard permutation algorithm is:
`for` $i \in \{0 \ldots N-1\}$ `Z[i] = Y[X[i]]`
for input permutation arrays `X[]` and `Y[]`, and output permutation array `Z[]`. All experiments are performed on random permutations. In this regime, almost every iteration incurs a cache miss.

The size of the permutation dictates the preferred architecture. At the high end of our regime (billions of points), the preferred architecture consists of parallel disks. Using parallel disks, we are able to efficiently multiply permutations with 12.8 billion points in under one hour using the 16 local disks of a 16-node cluster. (Table 4).

In the case of flash memory, it took under one hour to multiply two permutation with 2.5 billion points using a single machine with two solid state flash disks in a RAID configuration (see Table 2).

In the case of RAM, one has a choice of using a multi-threaded algorithm or multiple independent single-threaded processes. Both regimes of computation are useful. Where independent computations from a parameter sweep are performed, or where a parallelization of the higher algorithm is available, independent single-threaded processes are preferred. Where a single inherently sequential algorithm is the goal, the multi-threaded algorithm is preferred.

Experimental results show a 50% speedup in both cases. The novel algorithm has its primary advantage for permutations large enough that they overflow the CPU cache. In the case of a multi-threaded algorithm, we demonstrate the speedup on a recent eight-core commodity computer for permutations with 32 million points (see Table 8). In the case of single-threaded processes, we run eight competing processes simultaneously, and demonstrate the same 50% speedup over the traditional permutation algorithm. In this single-threaded case, the speedup is observed for permutations with as few as 4 million points (see Table 7).

Similar algorithms are also presented for permutation inverse and permutation multiplication by inverse. This completes the standard suite of permutation primitives required by packages that support permutation algorithms, such as GAP [6].

The importance of these new methods for computational group theory is immediately evident by considering a previous permutation computation of one of the authors. In 2003, a group membership permutation computation for Thompson's group was reported. Thompson's group acts on 143,127,000 points [4]. Those 143 million points from seven years earlier are well within the regime of interest discussed in this paper: between 4 million points and billions of points. That computation now fits on today's commodity comput-

ers, including the in-RAM technique of this paper, and would be expected to produce a result 50% faster.

In addition to permutations being given directly, permutations arise frequently as the output of a Todd-Coxeter coset enumeration algorithm. There are several excellent descriptions of this algorithm [1, 5, 13, 17]. In those cases, the first description of the group is as a finite presentation, and one employs coset enumeration to convert this into a more tractable permutation representation. The group can then be efficiently analyzed through such algorithms as Sims's original polynomial-time group membership and the rich library that has grown up around it. Examples of such large coset enumerations include parallel coset enumeration [2] used to find a permutation representation of Lyons's group on 8,835,156 points, sequential coset enumeration [7] used to find a different permutation representation of Lyons's group on 8,835,156 points, and a result [8] finding a permutation representation of Thompson's group on 143,127,000 points.

1.1 Problem Description

In addition to the problem of permutation multiplication, two other standard permutation operations are typically supported by permutation subroutine packages: *permutation inverse* and *permutation multiplication by an inverse*. The last problem, $X^{-1}Y$, is often included as a primitive operation because there exists a more efficient implementation than composing inverse with permutation multiplication:

`for` $i \in \{0 \dots N-1\}$ `Z[X[i]] = Y[i]`

More formally, the problems are:

Let X and Y be two arrays with the same number of elements N, both indexed from 0 to $N-1$, such that:

$$0 \leq X[i] \leq N-1, \forall i \in \{0 \dots N-1\}$$

PROBLEM 1.1 (MULTIPLICATION). *Compute the values of another array, Z, with N elements, defined as follows:*

$$Z[i] = Y[X[i]], \forall i \in \{0 \dots N-1\}$$

PROBLEM 1.2 (INVERSE). *Compute X^{-1} such that:*

$$X[X^{-1}[i]] = X^{-1}[X[i]] = i, \forall i \in \{0 \dots N-1\}$$

PROBLEM 1.3 (MULTIPLY BY INVERSE). *Compute the result of multiplying a permutation by an inverse $X^{-1} \times Y$:*

$$Z[i] = Y[X^{-1}[i]], \forall i \in \{0 \dots N-1\}$$

1.2 Other Problems

While a full discussion is beyond the scope of this paper, we also note that the new algorithms presented for permutation multiplication also apply to object rearrangement:

```
Object Z[N], Y[N]
int X[N]
for i ∈ {0...N-1} Z[i] = Y[X[i]]
```

When the size of an object remains small compared to the size of a disk block, flash block, or cache line, then the algorithm can be used on disk, flash, or RAM, respectively. Further, the algorithm described here generalizes in an obvious way when Y is near to a permutation, but whose values may include duplicate entries from $\{0 \dots N-1\}$, while omitting other entries from $\{0 \dots N-1\}$.

Terminology.

In this paper we present three permutation multiplication algorithms for architectures with at least two levels of memory, in increasing order of performance: the *"external sort algorithm"*, the *"buckets algorithm"* and the *"implicit indices algorithm"*.

The terminology *"fast-memory/slow-memory"* refers to an algorithm which uses *slow-memory* as the slower, much larger lower-level memory (the one on which the permutation arrays are stored), and *fast-memory* as the faster, much smaller higher-level memory (which cannot hold the entire permutation arrays).

Organization of the Paper.

The rest of the paper is organized as follows: Section 2 presents related work, Sections 3 and 4 present our new fast algorithms, along with some theoretical considerations on their performance. Section 5 presents new fast algorithms for permutation inverse and multiplication by an inverse. Section 6 presents formulas for the optimal running time, under the assumption that the CPU cores are infinitely fast and that the single bus from CPU to RAM is the only bottleneck (or time to access flash memory or disk). Section 7 presents the experimental results, followed by the conclusion in Section 8.

Overview of the Algorithms.

Six algorithms are presented. Algorithms 1 and 2 are intended solely to explore the design space. Algorithms 1 and 2 are disk-based permutation multiplication algorithms using external sorting and a simple buckets technique, respectively. Algorithm 3 reviews an older method for permutation multiplication [3, 4], here called *implicit indices*. Algorithm 4 constitutes the central novelty of this work. It presents a multi-threaded parallel permutation multiplication algorithm. Tables 4 and 5, along with Section 3.2, present a generalization to parallel distributed disks. Algorithms 5 and 6 review older algorithms for permutation inverse and multiplication by inverse [3, 4], that are analogous to Algorithm 3. The generalization to the multi-threaded case (analogous to Algorithm 4) is omitted for lack of space, but experimental results are presented in Table 8.

Section 6 presents a new timing analysis applicable to Algorithms 3, 4, 5 and 6 and their parallel generalizations.

2. RELATED WORK

The current work builds upon [3]. In that work, the authors present a fast RAM-based permutation algorithm that worked well on the Pentium 4, due in part to the 128-byte cache line on that CPUs. Most later CPUs have 64-byte cache lines, and so that algorithm, which is reviewed in this paper as Algorithm 3, later achieved mixed results. Algorithm 3 was also used as a sequential disk-based algorithm in [4]. Related sequential algorithms for permutation inverse and permutation multiplication by inverse were also described in [3, 4].

For lower-level memory data, some of the main ideas of *disk-based computing* [14, 16] have been used successfully in recent years to solve or make progress on important problems in computational group theory [9, 10, 14, 15], where the size of the data is too large for one RAM subsystem or even the aggregate RAM of a cluster.

The memory gap and memory wall phenomena are very important for understanding the reasons behind the efficiency of our new algorithms and the limitations of both our new algorithms and the traditional algorithms. These phenomena are well-known in literature [3, 18]. All the algo-

rithms we describe, whether traditional or new, are memory-bound for certain parameters.

3. PERMUTATION MULTIPLICATION USING EXTERNAL MEMORY

New algorithms for large permutations are presented. For many problems in computational group theory, the size of a permutation is in the range of tens to hundreds of gigabytes.

The first case presented below deals with permutations that fit on a single disk, with a permutation occupying at least 10 GB of space, but not more than 50 GB. These same algorithms can be run on flash memory. Both disk and flash are types of external memory in wide use today. Table 2 presents experimental results obtained by running our implicit indices algorithm both on flash and on disk. In the following three subsections one can replace disk with flash and everything remains correct.

3.1 Local Disk and Flash

The traditional implementation for permutation multiplication would be:

```
for (i = 0; i < N; i++) Z[i] = Y[X[i]];
```

Using this implementation would be impractical. For large enough pseudo-random permutations, most array accesses are to random locations on disk. Thus a memory page would be swapped in from disk at almost every array element access. On most current systems a memory page is on the order of 4 KB. If the element size is 8 bytes, then for each 8 bytes the traditional algorithm accesses the system would actually transfer 4 KB of data, which results in a 4 KB/8 bytes = 512 times ratio of transferred to useful data. This was indeed observed for naive permutation multiplication running in virtual memory (see Table 3).

A few important notions are defined before discussing the details of the three new algorithms for external memory.

Definition 1. System and Algorithm Parameters

The values in each permutation array X, Y and Z can be represented on β bytes.

$Hlms$ = the size of the higher-level memory component, in number of elements of β bytes.

Any arrays used in the algorithms can be divided into blocks of length $Bl = (Hlms/2)$ number of elements. Two blocks must simultaneously fit in $Hlms$.

$Nb = N/Bl$ is the total number of blocks in an array.

3.1.1 Using External Sort

The disk-based permutation multiplication method using external sorting is described in Algorithm 1.

Using the concept of buckets that fit in RAM, one can significantly improve the performance of the algorithm. RAM buckets are an alternative to external sorting which trades the $n \log n$ running time of sorting for random access within RAM. RAM buckets have significantly sped up computations that previously used external sorting [11].

3.1.2 Using RAM Buckets

The RAM buckets method is described in Algorithm 2. The RAM bucket size has to be chosen such that two RAM buckets simultaneously fit in RAM. Considering that both the index i and the value $X[i]$ are represented using the same number of bytes, one needs $2 \times N/Hlms$ buckets (here $Hlms$ is the size of RAM).

Algorithm 1 Permutation Multiplication Using External Sort

Input: Permutation arrays X and Y, of size N
Output: Z, s.t. $Z[i] = Y[X[i]], \forall i \in \{0 \dots N-1\}$

Phase 1: Scan X and, for each index i, save the pair $(i, X[i])$ to an array D on disk.

Phase 2: Externally sort all pairs $(i, X[i])$ in array D increasingly by $X[i]$. Now $\forall j \in \{0 \dots N-1\} \exists i \in \{0 \dots N-1\}$ such that $D[j] = (i, X[i])$ and $X[i] = j$.

Phase 3: Scan both array Y and the pairs $(i, X[i])$ in the array D at the same time. $\forall j \in \{0 \dots N-1\}$ we have $D[j] = (i, X[i])$, such that $X[i] = j$. Save the pair $(i, Y[j])$ to an array D' on disk.

Phase 4: Externally sort the array D' increasingly by the index i in pairs $(i, Y[j])$. Now the D' array contains pairs $(i, Y[X[i]])$ in increasing order of i. For each index i, copy $Y[X[i]]$ to the i index in the Z array.

Algorithm 2 Permutation Multiplication Using RAM buckets

Input: Permutation arrays X and Y, of size N
Output: Z, s.t. $Z[i] = Y[X[i]], \forall i \in \{0 \dots N-1\}$

1: All arrays are split into Nb equally sized buckets, each containing $Bl = N/Nb$ elements. The bucket size can be at most one-half the size of RAM. Bucket i of array A is denoted A_i. Bucket b contains indices in the range $[b * Bl, (b+1) * Bl)$.
// **Phase 1: bucketize**
2: Scan array X and, for each index i, save the pair $(i, X[i])$ in the bucket $D_{X[i]/Bl}$.
// **Phase 2: permute buckets**
3: **for** each bucket b **do**
4: Load buckets D_b and Y_b into RAM.
5: **for** each index i in this bucket **do**
6: Let $D_b[i] = (j, X[j])$.
7: Save the pair $(j, Y_b[X[j]])$ to bucket $D'_{j/Bl}$.
// **Phase 3: combine buckets**
8: **for** each bucket b **do**
9: Load buckets D'_b and Z_b into RAM.
10: **for** each index i in this bucket **do**
11: Let $D'_b[i] = (j, Y[X[j]])$.
12: Set $Z_b[j] = Y[X[j]]$.

Algorithm 2 presents a few important improvements over Algorithm 1. Note that in phase 2 of Algorithm 2, there is no need to save the index in the buckets of array Y, since it is implicit in the ordering. Thus a bucket of array Y occupies twice as little space as a bucket of pairs $(i, X[i])$. In phase 3, Z is also divided into $2 \times N/Hlms - 1$ buckets, and all indices from the j-th bucket of D' correspond to positions in the j-th bucket of Z. Algorithm 2 completely eliminates sorting and, in practice, shows a 4 times (or more) speedup over the External Sort-based algorithm if the computation is disk-bound (see Table 5, the 1 node case).

Both algorithms 1 and 2 need to save the index of each value of the X permutation, thus resulting in disk arrays as large as twice the size of the initial arrays. The implicit indices RAM/disk algorithm (Algorithm 3) avoids saving the indices to disk arrays.

3.1.3 With Implicit Indices

Algorithm 3 Permutation Multiplication using implicit indices

Input: Permutation arrays X and Y, of size N
Output: Z, s.t. $Z[i] = Y[X[i]], \forall i \in \{0 \dots N-1\}$

1: All arrays are split into Nb equally sized buckets, each containing $Bl = N/Nb$ elements. The bucket size can be at most one-half the size of RAM. Bucket i of array A is denoted A_i. Bucket b contains indices in the range $[b * Bl, (b+1) * Bl)$.
// **Phase 1: bucketize**
2: Traverse the X array and distribute each value $X[i]$ into bucket $D_{X[i]/Bl}$ on disk.
// **Phase 2: permute buckets**
3: **for** each bucket b **do**
4: Load buckets D_b and Y_b into RAM.
5: **for** each index i in this bucket **do**
6: Set $D_b[i] = Y_b[D_b[i]]$.
// **Phase 3: combine buckets**
7: For each value $X[i]$, let j be the next value in bucket $D_{X[i]/Bl}$. Note that $j = Y[X[i]]$. Set $Z[i] = j$ and remove that value from bucket $D_{X[i]/Bl}$.

The correctness of Algorithm 3 can be proved by following the three phases for a generic index $i \in \{0 \dots N-1\}$: in phase 1 value $X[i]$ is distributed into bucket $j = X[i]/Bl$ at position k of array D, so that $D[k] = X[i]$. In phase 2, $D[k] = Y[D[k]]$, which can be written $D[k] = Y[X[i]]$. In phase 3, $Z[i] = D[k]$, which can be written $Z[i] = Y[X[i]]$.

The implicit indices version runs about twice as fast as the buckets version (see Table 4). The implicit indices RAM/ disk algorithm performs the following steps: a sequential read of the X array and a sequential write of the D (temporary) array in phase 1 (*2 sequential accesses*); a sequential read of the D array, a sequential read of the Y array and a sequential write of the D array (*3 sequential accesses*); and a sequential read of the X array, a sequential read of the D array and a sequential write of the Z array (*3 sequential accesses*). In total, there are 8 sequential accesses

It is interesting to compare the running time of the implicit indices algorithm and the running time of a permutation multiplication algorithm that we implemented in Roomy [12], which uses Algorithm 2. Roomy is a general framework for disk-based and parallel disk-based computing which provides a high-level API for manipulating large amounts of data. The disk-based implicit indices algorithm is generally twice as fast as the Roomy implementation.

3.2 Many Disks

Here we describe how the three disk-based algorithms for permutation multiplication, presented in Section 3.1, can be used with the many disks in a cluster of computers.

Serial permutation multiplication using external sort is described in Algorithm 1. To parallelize it, all arrays are first split into sub-arrays, each of which is placed on the disk of a single compute node in the cluster. All operations on those arrays are performed in parallel. In cases where one node generates data that references a sub-array on another node, that data is first sent over the network, then saved to disk. In our implementation, there is a separate thread of execution on each node that handles the writing of this remote data to the local disk. Finally, there is a synchronization point after each phase, to insure that all nodes are done with one phase before beginning the next.

Permutation multiplication using buckets (Algorithm 2) is made parallel in the same way. The arrays are already split into sub-arrays (buckets), and the same methods are used for data distribution, parallel processing, and synchronization.

There is one additional modification necessary to parallelize permutation multiplication using implicit indices (Algorithm 3). Because the algorithm depends on the specific ordering of elements in each bucket, the buckets can not be written to in parallel. This is solved in the same way that Algorithm 4 extends Algorithm 3: each bucket is further split into sub-buckets, so that each node has its own sub-bucket to write to. Unlike the multi-threaded RAM case, the parallel disk case does not need an extra phase to compute the sizes of the sub-buckets, since the buckets are represented with files, which are dynamically sized.

4. PERMUTATION MULTIPLICATION IN RAM

The traditional permutation multiplication algorithm for cache/RAM can be trivially-parallelized. Each thread processes a contiguous region of the `X[]` permutation array. Although this incurs frequent cache misses, it tends to scale linearly on current commodity computers *until one goes beyond four cores*. This is because the single bus to RAM becomes saturated by the pressure of the several cores. In Table 8 of Section 7, one sees this happening approximately with 3 threads for permutation multiplication and for 4 threads for inverse and multiplication by inverse.

Algorithm 3 of Section 3 presented a single-threaded disk-based algorithm to overcome the many page faults. The same algorithm can be implemented for cache/RAM to minimize cache misses. That algorithm's cache/RAM version is preferred for permutation algorithms that can be parallelized at a higher level and then call a single-thread permutation multiplication algorithm. Here, we consider a multi-threaded version for the case when the higher level algorithm does not parallelize well. The corresponding results at the level of eight cores are presented in Table 7 in Section 7. As described in the extrapolation in Section 7.3, both the new single-threaded and the new multi-threaded algorithms are expected to have an even greater advantage at the 16-core and higher level in the future.

Algorithm 4 provides the multi-threaded version for multiplication using cache/RAM. Intuitively, it operates by splitting the buckets of Algorithm 3 into sub-buckets. Within a given bucket, each thread "owns" a contiguous region (a sub-bucket) for which it has responsibility. Algorithm 4 requires one extra phase (Phase 1) in order to determine in advance the size of the sub-bucket to allocate for each thread.

Some alternative designs were also explored. A brief summary of the alternatives considered is presented along with our reasons for rejecting them.

- Using pthread private data via "__thread" (problem: uses too much memory).
- Using pthread locks to synchronize memory access (problem: synchronization delays).
- Using an atomic add operation to a single global counter (problem: internally, it still uses a lock).

Algorithm 4 Multi-threaded cache/RAM Permutation Multiplication using Implicit Indices

Input: Permutation arrays X and Y, of size N, the number of cache buckets Nb, the number of threads T.
Output: Z, s.t. $Z[i] = Y[X[i]], \forall i \in \{0 \dots N-1\}$

1: All arrays are split into Nb equally sized buckets, each containing $Bl = N/Nb$ elements. The bucket size can be at most one-half the size of cache. Bucket i of array A is denoted A_i. Bucket b contains indices in the range $b \times Bl$ to $(b+1) \times Bl$.
2: Each thread t, $0 \le t \le T-1$, will handle indices in the range $t \times N/T$ to $(t+1) \times N/T - 1$.
// **Phase 1: create sub-buckets**
3: Create a temporary array D, split into $T \times Nb$ sub-buckets. $D_{b,t}$ is the sub-bucket corresponding to bucket b and thread t. The bucket D_b is the concatenation of all sub-buckets $D_{b,t}$. The size of a sub-bucket is first determined by an additional scan of X.
// **Phase 2: bucketize**
4: Each thread scans the portion of X that it is responsible for, and saves each $X[i]$ to sub-bucket $D_{t,X[i]/bl}$.
// **Phase 3: permute buckets**
5: Each thread locally permutes each bucket b that it is responsible for, setting $D_b[i] = Y_b[D_b[i]]$.
// **Phase 4: combine buckets**
6: Each thread computes the final values $Z[i]$ that it is responsible for. For each such index i, let j be the next value in sub-bucket $D_{t,X[i]/Bl}$ that has not been removed (Note that $j = Y[X[i]]$). Set $Z[i] = j$ and remove that value from sub-bucket $D_{t,X[i]/Bl}$.

- Exploiting L1 cache via a two-level algorithm, similar to two-level external sort (problem: delays due to extra passes).

Section 7.3 presents experimental results for the cache/RAM multi-threaded implicit indices algorithm.

5. PERMUTATION INVERSE. MULTIPLICATION BY AN INVERSE

While Algorithms 5 and 6 are not new [3, 4], their multi-threaded generalizations analogous to Algorithm 4 are novel. Experimental results for running permutation inverse and multiplication by an inverse, as well as theoretical estimates for these runs, can be found in Table 8.

Permutation Inverse.

The traditional algorithm for permutation inverse is:

```
for (i = 0; i < N; i++) Y[X[i]] = i;
```

The bottleneck is still the random access (this time write access) to the Y array.

Permutation Multiplication by an Inverse.

For the multiply-by-inverse, the traditional algorithm is:

```
for (i = 0; i < N; i++) Z[X[i]] = Y[i];
```

At the end of the loop $Z[i] = Y[X^{-1}[i]], \forall i \in \{0 \dots N-1\}$.

Algorithm 5 Permutation Inverse Using Implicit Indices

Input: Permutation array X, of size N
Output: $Y[X[i]] = i, \forall i \in \{0 \dots N-1\}$
Phase 1: Scan array X and distribute each value $X[j]$ in array D at block number $k = X[j]/Bl$. At the same time write value j at the same index in block k of D' as $X[j]$ was written at in block k of D.
Phase 2: Scan the D' and D arrays sequentially at the same time and, for each index j, write $Y[D[j]] = D'[j]$.

Algorithm 6 Permutation Multiplication by an Inverse Using Implicit Indices

Input: Permutation arrays X and Y, of size N
Output: $Z[i] = Y[X^{-1}[i]], \forall i \in \{0 \dots N-1\}$
Phase 1: Scan array X and distribute each value $X[j]$ in its corresponding block of array D. At the same time write value $Y[j]$ at the same index in D' as $X[j]$ was written at in D.
Phase 2: Scan the D' and D arrays sequentially and, for each index j, write $Y[D[j]] = D'[j]$.

6. PERFORMANCE ANALYSIS

The analysis presented here can be used to estimate the running time for the implicit indices algorithms, when using any 2-level memory hierarchy, including cache/RAM, RAM/flash, RAM/disk. The implicit indices algorithms include Algorithm 3, its generalization to Algorithm 4, Algorithms 5 and 6, and their parallel generalizations.

Definition 2. System and Algorithm parameters (Analysis)
Hrl = higher-level read memory latency (*seconds*)
Lrl = lower-level read memory latency (*seconds*)
Lwl = lower-level write memory latency (*seconds*)
Lwb = lower-level write memory bandwidth (*bytes/second*)
Lrb = lower-level read memory bandwidth (*bytes/second*)
Es = array-element size (*bytes*)
N = array length (*bytes*)
Nb = number of blocks per array
Bs = bucket size (*bytes*)
$Bl = N/Nb$ (block length) (*bytes*)

We refer to permutation multiplication as PM, to permutation inverse as PI, and to permutation multiplication by an inverse as PMI. The next three formulas estimate the running time when memory is the bottleneck for PM, PI, and PMI, respectively. Note that in the case of cache/RAM N/Lrb must be added to each formula, due to the extra pass.

FORMULA 6.1 (PM TOTAL ESTIMATED TIME).

$$N \times \left(\frac{Lwl + Lrl}{Bs} + \frac{3}{Lwb} + \frac{5}{Lrb} + \frac{Hrl}{Es} \right)$$

FORMULA 6.2 (PI TOTAL ESTIMATED TIME).

$$3N \times \left(\frac{1}{Lrb} + \frac{1}{Lwb} + \frac{2 \times Lwl}{Bs} + \frac{Hrl}{Es} \right)$$

FORMULA 6.3 (PMI TOTAL ESTIMATED TIME).

$$N \times \left(\frac{4}{Lrb} + \frac{3}{Lwb} + \frac{2 \times Lwl}{Bs} + \frac{Hrl}{Es} \right)$$

7. EXPERIMENTAL RESULTS

7.1 Local disk and flash

Tests were ran on an AMD Phenom 9550 Quad-Core at 2.2 GHz with 4 GB of RAM, running Fedora Linux with kernel version 2.6.29. The machine has both a disk drive (Seagate Barracuda 7200.10 250GB) and 2 RAID-ed flash SSD drives (2 × INTEL SSD SSDSA2MH080G1GC, 80 GB each).

Table 1 contains the measured system parameters of this machine. Table 1 also contains the measured system parameters for one of the disks of the cluster that was used to run the "parallel RAM/parallel disk" algorithms. The parallel disk bandwidth assumes that network bandwidth is not a limiting factor. Table 4 shows this to be the case for permutation arrays of size up to 25 GB.

Table 1: Measured system parameters for external memory.

	Disk	Flash	Cluster disk
Read BW (MB/s)	85	200	51
Write BW (MB/s)	82	26	51
Latency (ms)	10	14	39
Latency RAM (ns)	233	211	169

Table 2 shows a comparison between the new RAM/disk algorithm and the new RAM/flash algorithm, both based on implicit indices. The estimates from the formulas of Section 6 are also presented, to confirm that the algorithm is limited by the bandwidth of disk and flash.

Table 2: Running times of our new RAM/disk and RAM/flash algorithms and comparison with estimated running times. Element size is 8 bytes. Bucket size is 2 MB, block size is 1 GB.

Nr. elts. (billions)	Running Time (*seconds*)					
	Using Disk					
	PM		PI		PMI	
	real	est	real	est	real	est
1.25 (10 GB)	1609	1388	1002	1149	1253	1269
2.5 (20 GB)	3205	2776	2259	2298	2736	2538
	Using flash					
	PM		PI		PMI	
	real	est	real	est	real	est
1.25 (10 GB)	1584	1849	1212	1747	1348	1798
2.5 (20 GB)	2807	3698	2604	3494	2711	3596

Table 3 details our findings about the traditional permutation multiplication algorithm ran in virtual memory on the same machine. The experimental results confirmed our expectations: when the working set is at least twice the size of available RAM, using the traditional algorithm in virtual memory is infeasible. We also implemented a buffered traditional algorithm and ran parallel versions of both the simple traditional and buffered traditional algorithm. While the parallel buffered traditional algorithm clearly outperforms the parallel simple traditional one, the first is still infeasible when the working set overflows RAM by a significant percentage.

Table 3: Comparison of the traditional algorithm and the buffered traditional algorithm with disk-based and flash-based external memory. Element size: 4 bytes. RAM size is 4 GB. Arrays X, Y and Z are the work set.

Nr. elts (millions)	traditional algorithm time (seconds)			
	sequential		parallel	
	disk	flash	disk	flash
750 (3.0 GB)	3476	1198	1802	489
825 (3.5 GB)	$> 4hrs$	$> 4hrs$	$> 4hrs$	$> 4hrs$
	Buffered algorithm time (seconds)			
	sequential		parallel	
	disk	flash	disk	flash
750 (3.0 GB)	150	130	142	115
825 (3.5 GB)	$> 4hrs$	11762	$> 4hrs$	3561

7.2 Many disks

These experiments were run on a cluster of computers, each with two dual-core 2.0 GHz Intel Xeon 5130 CPUs and 16 GB of RAM, a locally attached 500 GB disk, running Linux kernel version 2.6.9. The network used a Dell PowerConnect 3348 Fast Ethernet switch. Only one process was used per node, to avoid competition for the single disk.

Tables 4 and 5 give a comparison of the three disk-based permutation algorithms presented in Section 3.1, based on: external sorting; RAM buckets; and implicit indices.

Table 4: Comparison of three parallel-disk permutation multiplication algorithms for increasing permutation size, using 16 nodes of a cluster. Elements are 8 bytes each. A "∗" indicates that the estimated time is not accurate, because the network became a bottleneck.

Nr. elts. (billions)	Algorithm Time (seconds)			
	Sort	Bucket	Implicit Indices	
			real	estimated
0.8 (6 GB)	538	105	77	70
1.6 (12 GB)	1151	202	100	139
3.2 (24 GB)	3440	490	270	279
6.4 (48 GB)	7484	2364	1571	∗
12.8 (95 GB)	15697	6838	3228	∗

Table 5: Comparison of three parallel-disk permutation multiplication algorithms for increasing parallelism, using from 1 to 16 nodes of a cluster. Elements are 8 bytes each. Permutations have 1.6 billions elements each (12 GB).

Nr. nodes	Algorithm Time (seconds)		
	Sort	Bucket	Implicit Indices
1	28952	7069	5576
2	13555	3627	2861
4	6197	677	354
8	2227	336	167
16	1185	202	100

Table 4 shows the results of using 16 nodes of a cluster, with permutation sizes ranging from 800 million elements (6 GB) to 12.8 billion elements (95 GB). In general, the three algorithms scale roughly linearly with permutation size. The most notable exception is a 5-fold increase in the running times of the bucket and implicit indices algorithms when moving from 24 GB to 48 GB permutations. We believe that this is due to network traffic on an older Fast Ethernet switch. Until that point, the bottleneck was likely disk bandwidth. The sorting based algorithm does not see a similar effect because its time is dominated by the in-RAM sorting process, not inter-node communications.

Table 5 shows the results of using between 1 and 16 nodes of the cluster, with permutations having 1.6 billion elements (12 GB). Again, the time for each algorithm scales roughly linearly with the number of nodes. The non-linear scaling when moving from 2 to 4 nodes is likely due to the bottleneck moving between disk and the network.

In general, the bucket algorithm takes about 1.5 to 2 times longer than the implicit indices algorithm, with the largest differences occurring with larger permutations and more parallelism. The implicit indices algorithm is more efficient because of the smaller amount of data that must be saved to disk. The sorting based algorithm takes roughly 5 to 10 times longer than the implicit indices algorithm, largely due to the time needed to sort data in RAM.

7.3 RAM

For cache/RAM, the performance of permutation multiplication, inverse and multiplication by an inverse was demonstrated on a recent 8-core commodity machine: two Quad-core Intel Xeon E5410 CPUs running at 2.33 GHz, with a total of 24 MB L2 cache — 12 MB L2 cache per socket and 16 GB of RAM made up of four memory modules. Table 6 lists the system parameters measured on this system.

Table 7 concerns the case of independent permutation computations running in parallel, with one computation per core. We believe that the traditional algorithm is close to saturating the bandwidth from CPU to RAM, both in the case of 8 threads and 8 processes. Table 8 provides confirming evidence of bandwidth saturation in comparing 4 threads versus 8 threads. As described in Section 6, the new algorithm is more bandwidth-efficient. We see that benefit for 8 processes but not for 8 threads. We speculate that is due to cache poisoning as the threads compete for the same cache.

Table 6: Measured system parameters for cache/ RAM. Latency for cache is negligible.

Read bandwidth	5859 MB/s
Write bandwidth	3850 MB/s
Latency of 1 random access	302 ns

Table 7: Comparison of traditional and new algorithms, using thread or process-based parallelism. Permutations have 4 million 4-byte elements each.

	Eight Threads	**Eight Processes**
Traditional	0.042 s	0.048 s
New	0.054 s	0.026 s

In Table 8 one can find running times for Algorithm 4 and the multi-threaded generalizations of Algorithms 5 and 6, as well as theoretical estimates of these running times based on the formulas in Section 6.

The new permutation multiplication algorithm is faster by about 50% than the traditional algorithm for permutations of 32 million elements or more, when using 8 threads. Our new algorithm is also faster than performing 8 multi-threaded traditional permutation multiplications in a row by at least a factor of 1.6. In contrast, when using only one thread (with seven cores idle), the time represents a mixture of RAM bandwidth and CPU power. Hence, the traditional and new algorithms have similar performance.

Extrapolation on memory bandwidth results.

In the near future, commodity machines will continue to gain additional CPU cores at a rate based on Moore's Law. But the number of memory modules on the motherboard is likely to remain fixed (while the density of each memory module continues to rise). Hence the memory bandwidth is unlikely to grow significantly.

Table 8 shows the times for the traditional algorithm already approaching an asymptotic value for the transition from 4 threads to 8 threads. Furthermore, the timings for 8 threads is close to the timing for the theoretically optimal case for bandwidth limited computation.

The new algorithm shows a significant improvement in time in the transition from 4 threads to 8 threads. In the case of permutation multiplication, the timing for 8 threads approaches that of the theoretically optimal memory bandwidth limited case. On the other hand, the algorithms for permutation inverse and permutation multiplication by an inverse show the potential for additional improvements in timings as more cores become available. This is seen by comparing the numbers for 8 threads and the optimal case.

8. CONCLUSIONS

New algorithms were presented for multiplication of large permutations for disk and flash (Section 3), for the aggregate disks of a cluster (Section 3.2) and a multi-threaded algorithm for RAM (Section 4). These algorithms make permutation multiplication a practical operation for large permutations that do not fit in RAM. Further, the multi-threaded cache/RAM implicit indices algorithm clearly outperforms the trivially-parallel traditional algorithm when using multiple threads on machines with many cores.

Acknowledgments.

We gratefully acknowledge CERN for making available an 8 core machine for testing.

9. REFERENCES

[1] J. J. Cannon, L. A. Dimino, G. Havas, and J. M. Watson. Implementation and analysis of the Todd-Coxeter algorithm. *Math. Comp.*, 27:463–490, 1973.

[2] G. Cooperman and G. Havas. Practical parallel coset enumeration. In *Proc. of Workshop on High Performance Computation and Gigabit Local Area Networks*, volume 226 of *Lecture Notes in Control and Information Sciences*, pages 15–27. Springer Verlag, 1997.

Table 8: Running times (seconds) of our new implicit indices permutation multiplication for cache/RAM. As explained, we need a machine with at least 8 cores working on the new algorithm in parallel for the CPU to be a less significant factor. Element size is 4 bytes. A bucket here is a cache line, the block size is variable between runs. The values in the column labeled "Optimal" are derived from the equations in Section 6 using values based on Table 6.

Nr. elem. (millions)	Running Time (seconds)									
	1 thread		2 threads		4 threads		8 threads		Optimal	
	Permutation Multiplication									
	new	trad.	new	trad.	new	trad.	new	trad.	new	trad.
32 (128 MB)	0.81	0.81	0.70	0.51	0.43	0.45	0.29	0.42	0.25	0.39
64 (256 MB)	1.98	1.67	1.47	1.27	0.85	0.95	0.62	0.88	0.50	0.77
128 (512 MB)	4.68	4.09	2.98	2.52	1.72	2.10	1.16	1.81	1.01	1.54
	Permutation Inverse									
	new	trad.	new	trad.	new	trad.	new	trad.	new	trad.
32 (128 MB)	1.03	1.83	0.86	1.04	0.54	0.63	0.34	0.59	0.18	0.53
64 (256 MB)	2.39	3.70	1.75	2.06	0.96	1.31	0.66	1.21	0.37	1.06
128 (512 MB)	5.33	7.48	3.52	4.15	2.00	2.65	1.35	2.51	0.75	2.11
	Permutation Multiplication by an Inverse									
	new	trad.	new	trad.	new	trad.	new	trad.	new	trad.
32 (128 MB)	1.06	1.84	0.87	1.10	0.53	0.66	0.35	0.60	0.20	0.55
64 (256 MB)	3.72	2.46	1.77	2.24	0.99	1.33	0.70	1.23	0.41	1.10
128 (512 MB)	5.57	7.52	3.62	4.22	2.08	2.72	1.43	2.55	0.83	2.19

[3] G. Cooperman and X. Ma. Overcoming the memory wall in symbolic algebra: A faster permutation algorithm (formally reviewed communication). *SIGSAM Bulletin*, 36:1–4, Dec. 2002.

[4] G. Cooperman and E. Robinson. Memory-based and disk-based algorithms for very high degree permutation groups. In *ISSAC*, pages 66–73, 2003.

[5] H. Felsch. Programmierung der Restklassenabzählung einer Gruppe nach Untergruppen. *Numerische Mathematik*, 3:250–256, 1961.

[6] GAP Group. *GAP – Groups, Algorithms, and Programming, Version 4.2*. `(http://www.gap-system.org)`, 2000.

[7] G. Havas and C. Sims. A presentation for the Lyons simple group. In *Computational Methods for Representations of Groups and Algebras*, volume 173 of *Progress in Mathematics*, pages 241–249, 1999.

[8] G. Havas, L. Soicher, and R. Wilson. A presentation for the Thompson sporadic simple group. In *Groups and Computation III, Computational Methods for Representations of Groups and Algebras*, volume 8 of *Ohio State University Mathematical Research Institute Publications*, pages 193–200. de Gruyter, 2001.

[9] D. Kunkle and G. Cooperman. Twenty-six moves suffice for Rubik's cube. In *Proc. of International Symposium on Symbolic and Algebraic Computation (ISSAC '07)*, pages 235–242. ACM Press, 2007.

[10] D. Kunkle and G. Cooperman. Solving Rubik's cube: Disk is the new RAM. *ACM Commmunications*, 51:31–33, 2008.

[11] D. Kunkle and G. Cooperman. Harnessing parallel disks to solve Rubik's cube. *Journal of Symbolic Computation*, 44:872–890, 2009.

[12] D. Kunkle and G. Cooperman. Roomy. URL: http://sourceforge.net/apps/trac/roomy/wiki, 2009.

[13] J. Neubüser. An elementary introduction to coset table methods in computational group theory. In C. Campbell and E. Robertson, editors, *Groups – St Andrews 1981*, volume 71 of *London Math. Soc. Lecture Note Ser.*, pages 1–45, Cambridge, 1982. Cambridge University Press.

[14] E. Robinson and G. Cooperman. A parallel architecture for disk-based computing over the baby monster and other large finite simple groups. In *Proc. of International Symposium on Symbolic and Algebraic Computation (ISSAC '06)*, pages 298–305. ACM Press, 2006.

[15] E. Robinson, G. Cooperman, and J. Müller. A disk-based parallel implementation for direct condensation of large permutation modules. In *Proc. of International Symposium on Symbolic and Algebraic Computation (ISSAC '07)*, pages 315–322. ACM Press, 2007.

[16] E. Robinson, D. Kunkle, and G. Cooperman. A comparative analysis of parallel disk-based methods for enumerating implicit graphs. In *Parallel Symbolic Computation (PASCO '07)*, pages 78–87. ACM Press, 2007.

[17] J. Todd and H. Coxeter. A practical method for enumerating cosets of a finite abstract group. *Proc. Edinburgh Math. Soc., II. Ser. 5*, 5:26–34, 1936.

[18] W. Wulf and S. McKee. Hitting the memory wall: Implications of the obvious. *ACM Computer Architecture News*, 23(1):20–24, 1995.