Memory-Based and Disk-Based Algorithms for Very High Degree Permutation Groups

Gene Cooperman*
College of Computer Science
Northeastern University
Boston, MA 02115 / USA
gene@ccs.neu.edu

Eric Robinson
College of Computer Science
Northeastern University
Boston, MA 02115 / USA
tivadar@ccs.neu.edu

ABSTRACT

Group membership is a fundamental algorithm, upon which most other algorithms of computational group theory depend. Until now, group membership for permutation groups has been limited to ten million points or less. We extend the applicability of group membership algorithms to permutation groups acting on more than 100,000,000 points. As an example, we experimentally construct a group membership data structure for Thompson's group, acting on 143,127,000 points, in 36 minutes. More significantly, we require approximately 10 GB of RAM for the computation — even though a single permutation of Thompson's group already requires half a gigabyte of storage.

In addition, we propose a disk-based group membership algorithm with the promise of extending group membership to well over one billion (1,000,000,000) points. Such a disk-based algorithm has formerly been impossible, due in part to the lack of a practical disk-based algorithm for multiplying and taking inverses of such large permutations. Random access to disk is prohibitively expensive. We demonstrate the first practical disk-based implementation of the basic permutation operations. We also propose a disk-based architecture for group membership data structures.

Categories and Subject Descriptors

I.1.2 [**Symbolic and Algebraic Manipulation**]: Algorithms—*algebraic algorithms*

General Terms

Algorithms, Experimentation

Keywords

permutation groups, group membership, permutation multiplication, disk-based methods, Thompson's group

*This work was partially supported by the National Science Foundation under Grant CCR-0204113.

To appear in the International Symposium on Symbolic and Algebraic Computation, August, 2003, Philadelphia, Pennsylvania

1. INTRODUCTION

The theory of computational permutation group algorithms has become a mature field in recent years [2, 3, 4, 6, 10, 11, 12, 13, 29, 32, 35]. In the special case of small base groups, randomized Schreier-Sims has been successfully used. In addition, the nearly linear time group membership algorithm of Babai, Cooperman, Finkelstein and Seress [3] provides a variation with theoretically guaranteed bounds on the time. Implementations of these algorithms are also very fast in practice, as evidenced by their implementations in GAP [22] and MAGMA [5].

However, with the time requirements well under control, it is time to revisit the the space requirements for permutation group algorithms. For example, the aforementioned nearly linear time group membership algorithm requires storage for $\Omega(\log n)$ permutations. In the example of Thompson's group acting on $n = 143,127,000$ points, even one permutation requires more than half a gigabyte. Storing $\Omega(\log n)$ permutations of that size is clearly unacceptable.

Given only the generators of a group, finding the order of the group is an important problem. It is often the key to identifying the group. For example, most group membership algorithms also yield the order of the group.

Permutation groups play an important role *even* when the initial group is not given by a permutation representation. If a group is given by a presentation (defining equations) or given as a matrix group over a finite field, a common strategy is to find an equivalent permutation representation for the group, and to then find the order of that group. Unfortunately, that strategy is limited by the tendency for the equivalent permutation representation to have very high detree.

A major goal of this paper is to extend the applicability of permutation group algorithms such very high degree representations. One important application is to directly analyze some of the base cases in the matrix recognition project [30, 31] by converting matrix representations to permutation representations where the permutation degree is one billion or less.

A second important application is to allow the rich body of permutation group algorithms to be applied, where the permutation group degree was previously too large. Historically, this was important in the construction of permutation representations for the larger sporadic simple groups, in which case ad hoc techniques were required for verification of the construction. This paper shows the practicality of finding the order of such constructed permutation groups by a single, uniform set of heuristics.

In extending group membership to higher degrees, we modify the randomized Schreier-Sims group membership algorithm of Sims [35]. One variation is more space-efficient. Another variation adds additional pseudo-random group elements by applying the shallow Schreier trees of Cooperman and Finkelstein [13, 12]. Shallow Schreier trees were also the key to the nearly linear time algorithms of Babai, Cooperman, Finkelstein and Seress [3] for small base groups. A further enhancement uses a new heuristic, called *powerlevelling* (see Section 3.1.2).

A key to the success of the new algorithm is the use of a high quality random word generator. Several algorithms for random generation are available, including those of Babai [1], Celler et al. [7], and Cooperman [8]. Although the theoretical complexity guarantees are far from satisfactory, current implementations make random generation of group elements quite practical.

Babai's method was the first polynomial time method, operating in $O^{\sim}(n^6)$ time. The product replacement method of Celler et al. was shown by Pak [33] to run in at most $O^{\sim}(n^{10})$ time, although in implementations it is the most practical and most widely used of the three methods. The method of Celler et al. is particularly impressive in that after its initialization, it can produce a new, high quality pseudo-random element with only one group multiplication. Cooperman's method operates in $O^{\sim}(n^3)$ time. A variation of Cooperman's method is used here, for the sake of its ability to compactly represent pseudo-random group elements by a vector of 0/1 exponents with respect to a fixed straight line program.

Section 2 provides the background and a review of the Schreier-Sims group membership algorithm. Section 3 describes a space-efficient group membership algorithm for a permutation representation of Thompson's group, acting on 143,127,000 million points. Thompson's group is a sporadic simple group of order 90,745,943,887,872,000. Section 4 describes a proposed efficient disk-based version of the algorithm. Of particular interest is Section 4.1, which describes a practical algorithm for disk-based permutation multiplication and permutation inverse, along with timings.

1.1 Previous Literature

The need for analysis of very high degree permutation groups has a long history. Almost ten years ago, Cooperman et al. [15, 14] produced a permutation representation of degree 9,606,125 for Lyons' group acting on a conjugacy class of subgroups of order three. The representation was found, using the matrix representation of Wilson [39] as a starting point. The representation was verified in a Monte Carlo manner by computing its order through ad hoc methods. Gollan then began his work on a revised existence proof of Lyons' group [23, 24, 25]. As one part of that work, he deterministically verified the order of the permutation representation through the "double coset trick", an independent rediscovery of an unpublished Verify algorithm of Sims.

Later, a coset enumeration of Lyons' group yielded a permutation action on 8,835,156 points, based on Sims' original unpublished presentation. The coset enumeration was executed in two different ways. It was carried out as a parallel enumeration by Cooperman and Havas [17] (described therein as part of the future work). It was also demonstrated as a sequential coset enumeration by Havas and Sims [27]. That presentation was verified as producing Lyons' group by Gollan and Havas [26].

Later work produced large permutation representations for Thompson's group acting on 143,127,000 and for Janko's group J_4 acting on 173,067,389 points. A permutation representation was implicit in the condensation computation for Thompson's group of Cooperman et al. [18, 21]. Weller [38] carried out a direct computation at approximately the same time. Havas et al. [28] produced a presentation for Thompson's group, and also a permutation representation thereof through coset enumeration. Weller [36, 37] did the same for Janko's group, using some of the hashing techniques of [14, 15] and the double coset trick of [23, 25]. That work was used in a revised existence proof for Janko's group [19].

Finally, the matrix recognition project [30, 31] expects to reduce certain matrix group recognition problems to the base case of the simple groups, which then require other methods for analysis. The methods of this paper provide a useful alternative in this setting, since they allow the well-developed computational methods for permutation groups to be applied. Note, for example, that any group with a representation in $\mathrm{GL}(30, 2)$ (the group of matrices of dimension 30 over the finite field with two elements) has a permutation representation on at most 2^{30}, or approximately one billion points.

The proposed disk-based algorithm grew out of work by Cooperman and Grinberg [16] in which a shared memory coset enumeration algorithm was found to be memory-bound. The result was a new faster algorithm for memory-based permutation multiplication by Cooperman and Ma [20]. That algorithm has been retargeted here to provide the missing link in a disk-based group membership algorithm.

2. BACKGROUND

A *group membership algorithm* takes as input a permutation, g, on n points, and a set of permutations, S, on n points, which generate a group $G = \langle S \rangle$. The membership algorithm decides if $g \in G$.

The original group membership algorithm by Sims [35] began a long period of new algorithmic research in permutation group algorithms. It works by divide-and-conquer.

2.1 Notation

Denote the points on which G acts by the integers $\Omega = \{1, 2, \ldots, n\}$. For $i \in \Omega$ and $g \in G$, let i^g denote the action of the permutation g on the point i. (Hence, $i^{gh} = (i^g)^h$ for $g, h \in G$.) Let e be the identity element of G. Let $H \leq G$ denote that H is a subgroup of G, and $H < G$ that H is a proper subgroup of G. Define the *point stabilizer subgroup*

$$G^{(i)} = \{g\colon\ g \in G, \quad \forall j < i, j^g = j\},$$

sometimes called "G move i" (and moving all the points larger than i). Note that this yields a *point stabilizer subgroup chain*

$$G = G^{(1)} \geq G^{(2)} \geq \cdots G^{(n)} = \{e\}$$

for e the identity.

Let $G/H = \{Hg\colon\ g \in G\}$ be the *set of cosets* of H in G (where $Hg = \{hg\colon\ h \in H\}$). Note that $i^{G^{(i+1)}g} = i^g$ for $g \in G$ (where $i^{G^{(i+1)}g} = (i^{G^{(i+1)}})^g$). So, for $h \in G^{(i)}g$, i^h is a signature of $G^{(i)}g$. In other words,

$$\forall h_1, h_2 \in G^{(i)}, \quad i^{h_1} = i^{h_2} \Leftrightarrow G^{(i)}h_1 = G^{(i)}h_2.$$

A *transversal of* $G^{(i)}/G^{(i+1)}$, $T^{(i)}$ is defined as a set of representatives of cosets of $G^{(i+1)}$ in $G^{(i)}$. So,

$$|T^{(i)}| = |G^{(i)}/G^{(i+1)}|.$$

Further, a transversal $T^{(i)}$ satisfies

$$\forall j \in \Omega, \quad i^g = j \Longleftrightarrow \exists t, T^{(i)} \cap G^{(i+1)}g = \{t\} \text{ with } i^g = i^t.$$

2.2 Review of Schreier-Sims Randomized Group Membership Algorithm

This section describes a variation of the Schreier-Sims algorithm that forms the basis for the algorithmic work of this paper. The goal of the algorithm is to construct transversals $T^{(i)}$ for all $i \geq 1$. Once $T^{(i)}$ is constructed, the group membership algorithm for $g \in G$ is solved, as seen below:

```
ALGORITHM A:
INPUT: permutation group G, transversals {T^(i)},
       permutation g on {1,2,...,n}
Let g_1 <- g
LOOP:
  For i = 1,..., n
  If T^(i) ∩ G^(i+1) g_i = ∅, then stop and return NOT_A_MEMBER
  Otherwise, let t_i ∈ T^(i) be the unique element
     such that T^(i) ∩ G^(i+1) g_i = {t_i}
  Set g_{i+1} <- g_i t_i^{-1}
  Note that g_{i+1} ∈ G^(i+1) and that g_i = g_{i+1} t_i
  If g_{i+1} ≠ e, then goto LOOP
Return g = g_1 <- t_i t_{i-1} ··· t_i
```

It also follows that any element $g \in G$ can be uniquely represented as

$$g = t_{n-1}t_{n-2} \cdots t_1 \text{ for } t_i \in T^{(i)}.$$

Hence, $|G| = |T^{(n-1)}|\,|T^{(n-2)}| \cdots |T^{(1)}|$. This solves the problem of computing the *group order*.

In order to construct $T^{(i)}$ for all i, two problems must be solved.

- P1: Given a random element of $G^{(i)}$, find a random element of $G^{(i+1)}$.
- P2: Given either generators or random elements of $G^{(i)}$, construct a transversal, $T^{(i)}$.

The solution to Problem P1 follows by noting that any random element $g_i \in G^{(i)}$ has a unique representation $g_i = t_{n-1}t_{n-2} \cdots t_i$ for appropriate $t_j \in T^{(j)}$ for $j \geq i$. Further, the randomness of g_i implies that each t_j is as if chosen randomly from $T^{(j)}$. As in Algorithm A, one can determine from g_i the unique t_i and g_{i+1} such that $g_{i+1} = t_{n-1}t_{n-2} \cdots t_{i+1} = g_i t_i^{-1}$. The randomness of the t_j then imply that g_{i+1} is random in $G^{(i+1)}$.

The solution to Problem P2 follows from simple search algorithms, such as breadth-first search. Given sufficient random elements of $G^{(i)}$, they are guaranteed to generate $G^{(i)}$. Let $S^{(i)}$ be the generating set of $G^{(i)}$. Initialize a reachability set $R \subseteq \Omega$ to $R = \{(i, e)\}$.

```
ALGORITHM B:
INPUT: generating set S^(i) for G^(i)
While there exists g ∈ S^(i) and j ∈ R with j^g ∉ R do
  Add the pair (j^g, g) to R
```

The data structure described in Algorithm B is called a *Schreier tree.* Next, if $g \in G^{(i)}$, then one can find the unique t such that $G^{(i)}g \cap T^{(i)}$ as follows.

```
Initialize t <- e and initialize j <- i^g.
LOOP:
  If j = i, then stop and return t
  Otherwise, let (j,h) ∈ R satisfy i^t = j
  Set t <- ht and set j <- j^{g^{-1}}
  Goto LOOP
```

At termination, $i^t = i^g$ and hence we have constructed the element $t \in T^{(i)}$.

2.3 Shallow Schreier Trees

Shallow Schreier trees were developed by Cooperman and Finkelstein [13, 12]. The primary result on shallow Schreier tree follows.

THEOREM 2.1 ([12, THEOREM 3.5] (PARAPHRASED)). *For $\delta \geq 1$, let $d = \lceil 20\delta \log_2 |G^{(i)}/G^{(i+1)}| \rceil$. Then d random group elements suffice for a Monte Carlo algorithm to build a new Schreier tree for $G^{(i)}/G^{(i+1)}$ of depth d. The probability of error is less than $|G^{(i)}/G^{(i+1)}|^{-\delta}$.*

Intuitively, the theorem says that for a transversal of size T, $20 \log_2 T$ random elements suffice to build a Schreier tree. Further, the tree will have depth at most $20 \log_2 T$. The probability of failing to construct the tree within the stated bounds is less than $1/T$. The proof of the theorem uses a very conservative algorithm. We apply a more aggressive heuristic to reduce the constant 20.

2.4 Generation of Random Elements

A *random subproduct* on elements $(h_1, \ldots, h_k)$, is defined as $\overline{h_1 \cdots h_k} = h_1^{e_1} \cdots h_k^{e_k}$, where $e_i \in \{0, 1\}$ are chosen uniformly at random. Hence, each group element h_i appears in the random subproduct with probability exactly 1/2.

To generate random elements on a group G generated by $g_1, \ldots, g_k$, we recursively define g_i for $i > k$ by

$$g_i = \overline{g_1 \cdots g_{i-1}}$$

The chosen algorithm for random elements is purely heuristic, since the theoretical guarantees for currently known random generation algorithms are much too coarse. The algorithm is suggested by the more theoretical method of Cooperman [8]. Random subproducts were used earlier by Cooperman and Finkelstein [10, 11] in the context of a simple $O(n^4)$ randomized group membership algorithm for large base groups. Intuitively, if a distribution of random group elements covers a set that is close to a subgroup, then the result of Cooperman and Finkelstein shows that a random subproduct will produce a new random element that escapes from that set.

The motivation for our choice of heuristic is two-fold. Our first goal is to generate reasonably short words representing a random element of the permutation group. This saves space and time. In experiments, product replacement and Cooperman's method both satisfy the first goal.

The second goal is that the each random element be representable as a subproduct of a *fixed*, short word in the group generators. A word w is a *subword* of $g_1 \cdots g_k$ if $w = g_1^{e_1} \cdots g_k^{e_k}$ for some choice of $e_i \in \{0, 1\}$. (We define $g_i^1 = g_i$ and g_i^0 to be the identity.) This is especially desirable for the disk-based version of the algorithm (see Section 4).

When the group is given on two generators, the second goal is easy to achieve by many heuristics. One can choose a word $w_\ell = aba^{-1}b^{-1}aba^{-1}b^{-1}\cdots ab$ of length 4ℓ for generators a and b, and any word of length ℓ in those generators can be represented as a subword of w_ℓ. This second goal is harder to achieve when the group is given by more than two generators.

The stated heuristic is one of many that allow the group membership algorithm to perform in reasonable time. Further research is required to determine the best heuristic.

3. MEMORY-BASED GROUP MEMBERSHIP

The algorithm proceeds as in Section 2.2. The most important and memory intensive portion of the algorithm is the computation of a transversal for $G^{(1)}/G^{(2)}$. First, the method by which the transversal for $G^{(1)}/G^{(2)}$ is computed and stored will be illustrated, and then it will be shown that this method can be easily extended to calculate transversals for $G^{(i)}/G^{(i+1)}$ for $i > 1$. Since for small base groups, the size of the transversal tends to decrease rapidly with increasing i, the remaining transversals are computed in a fraction of the space needed for $G^{(1)}/G^{(2)}$.

3.1 Building the First Schreier Tree

We tried two methods for computing the transversal for $G^{(1)}/G^{(2)}$. The first was a standard Schreier tree. The second method introduced additional random group elements to guarantee a shallow Schreier tree.

3.1.1 Method 1

A modified Schreier vector representation is used, but without backpointers. A separate bit vector for the 143,127,000 nodes records if a node has been seen before. The Schreier tree is explored in breadth-first order. New nodes in the Schreier tree are appended to the Schreier vector as a pair consisting of a string of bits representing a word in the generators, and the image of the point 1 under that word. Once the Schreier vector is complete, it is sorted based on image points. At the end, each vector location at index i stores a string of bits representing a word in the generators mapping 1 to i.

The storage to represent the word on original generators is not much more than the storage for a "backpointer" to the parent node in the Schreier tree. Storing the string directly tends to be more CPU efficient, by avoiding cache misses as one traces backpointers.

The Thompson group had a maximum depth of 72 and an average depth of 59 for the transversal elements discovered and the mapping was complete (a mapping was discovered from the orbit to all of the other locations).

3.1.2 Method 2

The second method works harder to ensure an acceptably shallow Schreier tree. There is a tension between using many pseudo-random group elements as generators to reduce the Schreier tree depth and few generators so as to minimize storage requirements.

This method does use backpointers to indicate the parent node of a given node in the Schreier tree. Initially, a subtree of the Schreier tree for $G^{(1)}/G^{(2)}$ is built to a specified depth (depth 20 in the case of Thompson's group). At this depth, we reached 138,000 points using the two original generators.

Random elements are then used to extend the initial subtree. Since our random elements are always subwords of a fixed word, we save storage by storing them as bit representations indicating the exponents relative to the fixed word. For Thompson's group, 35 bits suffice. Our random word is produced as g_k for $k = 9$, based on Section 2.4. The average length of g_k as a word in the original generators is 15, and words longer than length 30 are rejected.

Each successive pass applies a new random element to each of the 138,000 nodes of the initial subtree. Furthermore, for each new node discovered by application of that random group element, the tree is extended by a subtree rooted at the newly discovered node. The subtree extension is built using only only the original group generators. A good depth for the subtree was found to be 20.

This method guarantees that all transversal elements are expressible as words in at most one random element. This is done because the random group elements are much more expensive to apply than group generators. This process continues until the number of nodes gained per pass through the tree is reduced to some specified percentage of the permutation degree. A good percentage for Thompson's group was found to be 0.5%.

At this point, a computational method, which we call *powerlevelling*, begins. A *powerlevel* is the application of all of the original generators to every node in the tree. If any of these applications discovers a new node, the tree is extended by a subtree rooted at the new node. For Thompson's group, a depth of 30 for this subtree was used. Powerlevelling is then recursively applied to all nodes of the new subtree. If at any point during the powerlevelling process there are no new nodes gained by passing through the tree, the Schreier tree is complete.

The depth of the Schreier tree was bounded above by 101: applications of original generators to depth 20; applications of approximately 15 random elements to each node of the subtree; application of original generators for an additional incremental depth of 20; and a powerlevelling phase incrementally increasing the depth by 30+30=60. Since a transversal element, expressed as a word, could include a random element, and since a random element was expressed as a word of length up to 30, the transversal element was expressible as a word of length at most 131.

3.2 Finding a Small Suborbit

A *suborbit* is an orbit of the point stabilizer subgroup. The transversal of $G^{(2)}/G^{(3)}$ is a suborbit. A small transversal greatly reduces the computation time for computing the transversal of $G^{(2)}/G^{(3)}$. The initial suborbit on the point 2 was of length 35 million, which made the computation time too long. Notwithstanding this consideration, if the transversal of $G^{(2)}/G^{(3)}$ is too small, then $|G^{(3)}/G^{(4)}|$ becomes comparable to $|G^{(2)}/G^{(3)}|$, and the random words used to compute the transversal for $|G^{(2)}/G^{(3)}|$ become very long.

For balance, we use a heuristic that searched for a suborbit of size approximately 0.6% of the full permutation degree or smaller. The heuristic uses the incremental rate R at which new elements of the transversal are found in order to decide when to stop exploring the current suborbit, and choose a suborbit with a different initial random point. If

f is the number of points found and τ the desired suborbit size, then the heuristic rejects a suborbit if $0.5f + fR > \tau$. Since initial random points are more often found in larger suborbits, the search is biased toward large suborbits satisfying $0.5f + fR \leq \tau$. The suborbit found is somewhat smaller (of size 179,712), but suitable.

3.3 Building the Remaining Schreier Trees

The remaining two levels were discovered by applying random elements to build a tree (see Section 2.3). The quality of the random element was found to be very important. For a Schreier tree of depth d, random elements from words significantly longer than d were needed.

3.4 Experimental Results

Figure 1 shows the breakdown of CPU times for Thompson's group. Computation of the second transversal requires 8 random group elements, while the third transversal requires 5 random group elements. The computation used the first method (Section 3.1.1) to compute the Schreier tree of the first transversal. The 15 minutes divides into 7 minutes to find all transversal elements and then 8 minutes to sort the Schreier vector based on image points. The second method was also tried, and required 20 minutes. For comparison, the entire computation for Lyons' group acting on 9,606,125 points completed in less than one minute.

Preprocessing	3 minutes
Transversal 1 (143,127,000 points)	15 minutes
Finding small suborbit (28 tries)	11 minutes
Transversal 2 (179,712 points)	5 minutes
Transversal 3 (3,528 points)	2 minutes
Total	36 minutes

Figure 1: Summary of Computation Times

The computation was run on a 1.3 GHz IBM pSeries 690 Turbo computer. We used IBM's VisualAge C++ Professional compiler (version 5), generating a 64-bit executable. The computer had 32 GB of RAM, shared among 32 Power4 CPUs. Although it is part of a parallel, shared memory computer, all tests were done only sequentially. While it would have been easy to parallelize our implementation, the rapid speed made it unnecessary.

4. DISK-BASED GROUP MEMBERSHIP

In this section, we extend the previous algorithm to work on a disk. We assume a base of constant size (i.e., there are only $O(1)$ levels with non-trivial transversals $|T^{(i)}| > 1$). In the case of disk access, random access is particularly expensive. Hence, we require that the algorithm be modified sufficiently to use only $O(1)$ space and $O^{\sim}(1)$ sequential disk accesses. (By $O^{\sim}(1)$, we mean $O(log^c n)$ for some constant c.) In this model, an access to one byte on disk costs as much as an access to a megabyte *as long as the megabyte is accessed sequentially.*

The criterion of considering a short and long sequential access as of equal cost is motivated by the desire to use disks with fast data transfer rates. RAID-0 and RAID-3 disks use striping to achieve data transfer rates that are typically faster than 50 MB/s. Assuming a disk access time of 10 ms for such disks, one could have read 500 KB during the time that one waits for a single disk access.

An additional scenario in which high disk data transfer rates can be achieved is a cluster of workstations with a fast local area network. In such a scenario, one can treat the fast network and local disks of the cluster as if they are part of a single striped disk subsystem. In this case, one uses the cluster for the sake of parallel disks, rather than for the sake of parallel CPUs.

Next, two new algorithmic ideas are required to retarget our previous algorithm to a disk-based computation. These are a more efficient disk-based algorithm for permutation multiplication and inverse, and a more efficient disk-based version of Algorithm B.

4.1 Disk-Based Permutation Multiplication and Inverse

A fast cache-aware algorithm for permutation multiplication and inverse in main memory was introduced in a research note by Cooperman and Ma [20]. It gains its speed by using sequential access to data in place of the more natural random access.

Here, we apply the same ideas toward disk-based permutation operations, where the advantages of sequential access are even larger. Briefly, assume a permutation representation `X[]`, where `X[i]=j` means that the permutation `X` maps the point i to the point j. We wish to multiply permutations `X` and `Y`.

```
for i = 1 to n, do Z[i]=Y[X[i]]
```

Note that if `Y` is a random permutation of the points $1, \ldots, n$ and if `Y` is much larger than memory, then each iteration of the loop is very likely to result in a random access to disk.

We replace such random access to disk by sequential access as follows. Let m be the number of array entries that can be held in memory. Assume $m < n$. There are three phases. Assume a temporary array `D`, partitioned into blocks `B[i]` with each block of size $b < m$. Define $h(\cdot)$ by $h(a) = \lfloor a/b \rfloor$.

1. For i from 1 to n, copy `X[i]` to the next free slot in `B[h(X[i])]`.
2. For each block `B[j]`, $j \leq n/b$, load into memory the segment of the `Y` array, {Y[(j-1)b], Y[(j-1)b+1], ..., Y[jb-1]}. Then replace each element `a`∈`B[j]` by `Y[a]`.
3. We now invert the operations of Part 1. For each block `B[j]`, initialize a pointer to the beginning of the block. For i from 1 to n, copy the next occupied slot of `B[h(X[i])]` to `Z[i]`.

By maintaining buffers for `X`, `Z` and each block `B[j]`, one can efficiently manage I/O to these arrays on disk.

Similar ideas apply to permutation inverse and multiplication by permutation inverse:

```
for i = 1 to n, do Z[X[i]] = i;      (z = x^-1)
for i = 1 to n, do Z[X[i]] = Y[i];   (z = x^-1 y)
```

As before, one uses the blocks `B[j]`. The algorithm proceeds in only two phases. For permutation inverses, the pair `(i, X[i])` is copied to `B[h(X[i])]` in the first phase. Then in the second phase, for j from 1 to n/b, the block `B[j]` is brought into memory, and for each pair (a, b) ∈`B[j]`, one sets `Z[b]`← `a`. A well-known alternative for fast inverses follows from using external sorting [34, Chapter 11] to sort `(i, X[i])` on `X[i]`.

For multiplication by permutation inverse, one follows the two phase algorithm as described for permutation inverses. However, one employs the pair `(Y[i], X[i])` instead of the pair `(i, X[i])`. Hence, for the pair (a, b) ∈`B[j]`, the second phase computation, `Z[b]`← `a`, becomes `Z[X[i]] = Y[i]`.

Experimental times for permutation multiplication are shown below. The corresponding times for a disk-based algorithm based on the traditional permutation algorithm (with random access) is not shown. Such random access would likely require days to complete in all cases.

Computer	n (data size, bytes)	b (block size, bytes)	Time (s)
Pentium II/RAID-5	16 MB	1 MB	17
Pentium II/RAID-5	32 MB	1 MB	38
Pentium II/RAID-5	64 MB	1 MB	108
Pentium II/RAID-5	128 MB	1 MB	450
Pentium II/RAID-5	64 MB	1 MB	108
Pentium II/RAID-5	64 MB	2 MB	118
Pentium II/RAID-5	64 MB	4 MB	82
Pentium II/RAID-5	64 MB	2 MB	118
SunBlade 100	64 MB	2 MB	409.0
800 MHz AMD	64 MB	2 MB	501.4

Figure 2: Fast Multiplication of two disk-based random permutations
(disk usage: $4n$; main memory usage: b)

The 800 Mhz AMD is running a Linux 2.4 kernel with 128 MB RAM. The SunBlade 100 is running at 500 MHz with 512 MB. The Pentium II with RAID-5 is running at 350 MHz with a Linux 2.4 kernel and 256 MB RAM. All times were repeatable to within 10%. The measured disk data transfer rates were 2.25 MB/s (SunBlade, with NFS filesystem over fast Ethernet), 1.95 MB/s (AMD, local EIDE disk), and 6.5 MB/s (Pentium II/RAID-5).

Figure 2 shows that the times are roughly proportional to the data transfer rates, and are largely independent of main memory, at least as long as two blocks and all buffers can fit in main memory at the same time. The experiments were run under those conditions.

The experiment with the Pentium II and 128 MB was slow because the working set of the memory was restricted to 1 MB. This meant that the many buffers corresponding to blocks `B[j]` were excessively small for efficient disk I/O. The system limits of the Linux operating system prevented us from manipulating multiple files of several gigabytes in order to demonstrate permutation multiplication on one billion points. Extrapolating linearly from the example of 64 MB (16 million points), a permutation multiplication of one billion points requires $64 \times 108 = 6912$ s, or *two hours*. By using striped disks, this time for multiplying permutations of one billion points could be reduced by at least a factor of ten.

4.2 Outline of Disk-Based Schreier-Sims Group Membership

The key to the Schreier-Sims algorithm is to build a Schreier tree. As noted for Problem P1 of Section 2.2, if g is a random element of $G^{(i)}$ and a transversal of $G^{(i)}/G^{(i+1)}$ is known, then one can efficiently compute a corresponding random element of $G^{(i+1)}$. Since there are only a constant number of non-trivial transversals (constant base size), the length of the random word produced in the solution of Problem P1 will be $O^{\sim}(1)$. Since our algorithm requires only $O(\log n)$ random group elements per transversal, it will coset $O^{\sim}(1)$ to produce the random group elements over the life of the algorithm.

Hence, it suffices to show how to efficiently build a single Schreier tree. As seen by Theorem 2.1, for a transversal of length n, $O(\log n)$ random group elements suffice to build the tree. The heuristics of Section 3 can often be used to reduce that time.

In addition to permutation operations on disk, we will require one other algorithmic subroutine. We require an efficient disk-based sorting algorithm. External sorting [34, Chapter 11] is a well known algorithm for this purpose. One can often accomplish it at the cost of only two passes of disk I/O.

We assume that we have previously stored both the initial generators and their inverses. The inverses can be computed efficiently as in Section 4.1.

We next describe how to find the nodes of a Schreier tree corresponding to $T^{(j)}$ for $G^{(j)}/G^{(j+1)}$. Let R be the reachable set of nodes as described in Algorithm B of Section 2.2. We wish to apply a word $w = g_1 g_2 \cdots g_k$ to all of the nodes in $R = (r_1, \ldots, r_\ell)$. We assume the nodes of R are stored on disk in the same ordering as the points Ω on which the group acts.

1. We compute $((r_1, r_1^{g_1}), \ldots, (r_\ell, r_\ell^{g_1}))$, which must be stored on disk.
2. We then sort this into a new ordering according to increasing $r_i^{g_1}$. Now $\{(r_i, r_i^{g_1}) \colon\ 1 \le i \le \ell\}$ is ordered montonically increasing according to the second position of each pair.
3. Next, apply g_2. We can follow one pass over $\{(r_i, r_i^{g_1}) \colon\ 1 \le i \le \ell\}$ and over g_2, to create $\{(r_i, r_i^{g_1 g_2}) \colon\ 1 \le i \le \ell\}$, which is stored on disk.
4. We then sort this into a new ordering according to increasing $r_i^{g_1 g_2}$. Now $\{(r_i, r_i^{g_1 g_2}) \colon\ 1 \le i \le \ell\}$ is ordered montonically increasing according to the second position of each pair.
5. Repeat steps 3 and 4 for successive g_i, until $\{(r_i, r_i^{w}) \colon\ 1 \le i \le \ell\}$ has been computed and is ordered monotonically increasing according to the second position.
6. Scan both $\{(r_i, r_i^{w}) \colon\ 1 \le i \le \ell\}$ the array R in order and for those nodes, and for those elements $r_i^w \notin R$, enter them into R along with their parent node, r_i.

4.2.0.1 *Optimizations.*

Several optimizations are possible. After the first transversal, the size of the later transversals will typically be much smaller. The reachable set may be small enough to fit within main memory. Also, if the reachable set is very small, it will be more efficient to access disk in a random access manner, similarly to what one does if all data structures fit in RAM.

Next, we consider how to efficiently trace multiple words at the same time. We may estimate in advance how many words for random group elements will be needed. In that case, we can take advantage of the fact that the random words are all subwords of the same fixed word $\overline{w}$. Hence, we produce a set of random words, W, based on the fixed word $\overline{w} = g_1 \cdots g_k$. In particular, for $v \in W$, let $v = g_1^{e_1} \cdots g_k^{e_k}$. For $\overline{w} = g_1 \cdots g_k$, we proceed as described earlier. Let the word $v'_s = g_1^{e_1} \cdots g_{s-1}^{e_{s-1}}$ for exponents $\{e_1, \ldots, e_{s-1}\} \subseteq \{0, 1\}$ corresponds to a prefix of v.

At step s, for each element $v \in W$, we maintain an ordered set $\{(r_i, r_i^{v'_s})\colon\ 1 \le i \le \ell\}$. If $e_s = 1$, then v'_s participates in step s and we will compute $v'_{s+1} = v'_s g_s$. In that case, we will compute $\{(r_i, r_i^{v'_s g+1})\colon\ 1 \le i \le \ell\}$, and maintain it as an ordered set. We execute step s by scanning R simultaneously with each ordered set $\{(r_i, r_i^{v'_s})\colon\ 1 \le i \le \ell\}$ that is participating in step s.

4.3 Extensions

The permutation operations and external sorting were both required as algorithmic disk-based subroutines. Both algorithms are limited as the size of the data grows compared to the size of main memory. This is because both algorithms require holding buffers for many streams of data simultaneously in memory. For example, in the case of permutation multiplication, these are the data blocks `B[j]`. The size of a data block `B[j]` is limited to approximately half of the size of main memory. The number of buffers is proportional to the quotient of the size of data by the size of a buffer for a single data block `B[j]`. As the size of the data grows, not all of the buffers can simultaneously fit in main memory.

In both cases, the algorithms are extended by taking a block `B[j]` as being larger than main memory. This keeps the number of buffers at a manageable level. In the case of external sort, this trick is well-known. In the case of the permutation multiplication algorithm of Section 4.1, at step 2 one is required to permute data from a segment of the `Y` array and store it in a `B[j]` buffer. This is effectively an example of a permutation multiplication problem, and one applies the disk-based permutation multiplication recursively to solve it. Similar ideas are used for permutation inverse and multiplication by the permutation inverse.

Many of the other algorithms of computational permutation group theory can also be extended to disk-based algorithms using the methodology outlined in this section. In particular, this include verification of group membership.

5. COMPACT REPRESENTATION OF SCHREIER TREES

An important consideration is the space storage for the representation of the Schreier trees. A natural representation requires a pointer from each node to its parent, along with a label indicating which group element maps the parent to the child. A pointer typically requires four bytes, and upon breaking the 4 GB barrier, it requires eight bytes.

The method of Cooperman and Finkelstein for Schreier coset graphs [9] shows how to encode both the pointer and the label in a data structure that takes *two bits* or less of storage per node. That method depends on having a perfect hash encoding of the cosets. Such a perfect hash encoding exists, since for $h \in G^{(i)}g$, i^h is a signature of $G^{(i)}g$, as noted in Section 2.1.

A future implementation will use this more compact storage method. This re-design of the Schreier tree data structure is likely to reduce the overall storage requirements of the group membership application for Thompson's group to well under 4 GB. Because of this intended re-design, we did not take as much care in minimizing the storage requirements of this preliminary version.

6. CONCLUSION

Computational permutation group algorithms and heuristics for very high degree permutation groups are a missing link in the ability to analyze the large group representations being produced today. Such large permutation representations arise both from presentations of groups and from matrix representations. In this paper, we have raised the limits for what permutation degrees are practical by one or two powers of ten.

For the future, we expect disk-based methods to raise the limits still further. Applying such disk-based methods will depend on the availability of disks with high data transfer rates. Such technology is available both through disk striping (e.g. RAID-0 and RAID-3) and through the use of clusters with many local disks and a high bandwidth local area network. Implementing the disk-based algorithm closer to this "bleeding edge" of technology will require further work.

For today, the memory-based algorithm presented here is eminently practical. Although the storage requirements for Thompson's group for this preliminary implementation were approximately 10 GB, the methods of Section 5 will likely reduce the memory requirements to under 4 GB.

7. ACKNOWLEDGEMENTS

We are grateful to Michael Weller for providing us with the large permutation representations that formed the basis of these experiments, and for describing his own experiences with such large computations. The generators for Thompson's group are standard generators in the sense of Wilson, and were generated by Weller from one of Wilson's matrix representations [39]. We thank Xiaoqin Ma and Viet Ha Nguyen for discussions about issues of random and sequential access in RAM. We also thank the Mariner Project at Boston University for providing the experimental facilities.

8. REFERENCES

[1] L. Babai. Local expansion of vertex-transitive graphs and random generation in finite groups. In *Theory of Computing*, pages 164–174, New York, 1991. (Los Angeles, 1991), Association for Computing Machinery.

[2] L. Babai, G. Cooperman, L. Finkelstein, E. M. Luks, and A. Seress. Fast Monte Carlo algorithms for permutation groups. *J. Comp. Syst. Sci.*, 50:296–308, 1995.

[3] L. Babai, G. Cooperman, L. Finkelstein, and A. Seress. Nearly linear time algorithms for permutation groups with a small base. In *Proc. of International Symposium on Symbolic and Algebraic Computation ISSAC '91*, pages 200–209. (Bonn), ACM Press, 1991.

[4] L. Babai, E. M. Luks, and A. Seress. Fast management of permutation groups I. *SIAM J. Computing*, 26:1310–1342, 1997.

[5] W. Bosma, J. Cannon, and C. Playoust. The MAGMA algebra system i: The user language. *J. Symbolic Comput.*, 24:235–265, 1997.

[6] G. Butler and J. J. Cannon. Computing in permutation and matrix groups I: Normal closure,

commutator subgroups, series. *Math. Comp.*, 39:663–670, 1982.

[7] F. Celler, C. R. Leedham-Green, S. H. Murray, A. C. Niemeyer, and E. O'Brien. Generating random elements of a finite group. *Comm. Algebra*, 23:4931–4948, 1995.

[8] G. Cooperman. Towards a practical, theoretically sound algorithm for random generation in finite groups. `arXiv:math.PR/0205203`, `http://arxiv.org/abs/math.PR/0205203`.

[9] G. Cooperman and L. Finkelstein. New methods for using cayley graphs in interconnection networks. *Discrete Applied Mathematics*, 37/38:95–118, 1992. (special issue on Interconnection Networks).

[10] G. Cooperman and L. Finkelstein. Randomized algorithms for permutation groups. *Centrum Wissenschaft Institut Quarterly (CWI)*, pages 107–125, June 1992.

[11] G. Cooperman and L. Finkelstein. Combinatorial tools for computational group theory. In *Groups and Computation*, volume 11 of *Amer. Math. Soc. DIMACS Series*, pages 53–86. (DIMACS, 1991), 1993.

[12] G. Cooperman and L. Finkelstein. A random base change algorithm for permutation groups. *J. Symbolic Comput.*, 17:513–528, 1994.

[13] G. Cooperman, L. Finkelstein, and N. Sarawagi. A random base change algorithm for permutation groups. In *Proc. of International Symposium on Symbolic and Algebraic Computation ISSAC '90*, pages 161–168, Tokyo, Japan, 1990.

[14] G. Cooperman, L. Finkelstein, M. Tselman, and B. York. Constructing permutation representations for matrix groups. *J. Symbolic Comput.*, 1997.

[15] G. Cooperman, L. Finkelstein, B. York, and M. Tselman. Constructing permutation representations for large matrix groups. In *Proceedings of International Symposium on Symbolic and Algebraic Computation ISSAC '94*, pages 134–138, New York, 1994. (Oxford), ACM Press.

[16] G. Cooperman and V. Grinberg. Scalable parallel coset enumeration: Bulk definition and the memory wall. *J. Symbolic Comput.*, 33:563–585, 2002.

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

[18] G. Cooperman, G. Hiss, K. Lux, and J. Müller. The Brauer tree of the principal 19-block of the sporadic simple thompson group. *J. Experimental Math.*, 6:293–300, 1997.

[19] G. Cooperman, W. Lempken, G. Michler, and M. Weller. A new existence proof of Janko's simple group J4. In *Computational Methods for Representations of Groups and Algebras*, volume 173 of *Progress in Mathematics*, pages 161–175, 1999.

[20] 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.

[21] G. Cooperman and M. Tselman. New sequential and parallel algorithms for generating high dimension Hecke algebras using the condensation technique. In *Proc. of International Symposium on Symbolic and Algebraic Computation (ISSAC '96)*, pages 155–160. ACM Press, 1996.

[22] The GAP Group. *GAP — Groups, Algorithms, and Programming, Version 4.3*, 2002. `http://www.gap-system.org`.

[23] H. Gollan. A new existence proof for Ly, the sporadic simple group of R. Lyons. *Preprint 30*, 1995.

[24] H. Gollan. A contribution to the revision project of the sporadic groups: Lyons' simple group Ly. *Vorlesungen aus dem FB Mathematik*, 26, 1997.

[25] H. Gollan. A new existence proof for Ly, the sporadic simple group of R. Lyons. *J. Symbolic Comput.*, 31:203–209, 2001.

[26] H. Gollan and G. Havas. On Sims' presentation for Lyons' simple group. In *Computational Methods for Representations of Groups and Algebras*, volume 173 of *Progress in Mathematics*, pages 235–240, 1999.

[27] 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.

[28] G. Havas, L. Soicher, and R. Wilson. A presentation for the Thompson sporadic simple group. In *Groups and Computation III*, pages 193–200, New York, 2001. (Ohio, 1999), de Gruyter.

[29] W. M. Kantor. Sylow's theorem in polynomial time. *J. Comp. Syst. Sci.*, 30:359–394, 1985.

[30] C. Leedham-Green. The computational matrix group project. In *Groups and Computation III*, pages 229–248, New York, 2001. (Ohio, 1999), de Gruyter.

[31] C. Leedham-Green, E. O'Brien, and C. Praeger. Recognising matrix groups. In J. Grabmeier, E. Kaltofen, and V. Weispfenning, editors, *Computer Algebra Handbook*, pages 474–475, 2003.

[32] E. M. Luks. Computing the composition factors of a permutation group in polynomial time. *Combinatorica*, 7:87–99, 1987.

[33] I. Pak. The product replacement algorithm is polynomial. In *Proc. 41^{st} IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 476–485. IEEE Press, 2000.

[34] R. Ramakrishnan and J. Gehrke. *Database Management Systems*. McGrawHill, second edition, 2000.

[35] C. C. Sims. Computation with permutation groups. In *Proc. Second Symp. on Symbolic and Algebraic Manipulation*. ACM Press, 1971.

[36] M. Weller. Construction of large permutation representations for matrix groups. In W. J. E. Krause, editor, *High Performance Computing in Science and Engineering '98*, pages 430–. Springer, 1999.

[37] M. Weller. Construction of large permutation representations for matrix groups ii. *Applicable Algebra in Engineering, Communication and Computing*, 11:463–488, 2001.

[38] M. Weller. Computer aided existence proof of Thompson's sporadic simple group. manuscript, 2003.

[39] R. Wilson. Atlas of finite group representations. `http://www.mat.bham.ac.uk/atlas`.