A Parallel Architecture for Disk-Based Computing over the Baby Monster and Other Large Finite Simple Groups

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

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

ABSTRACT

We outline a distributed, disk-based technique for computing over very large matrix groups. This technique is used to compute a permutation representation for the Baby Monster, a sporadic simple group that acts on 13,571,955,000 points. Its group order is approximately 4×10^{33}. This is a landmark because it is 100 times larger than any previous construction of a permutation representation. By using the computed on-disk data structures, computation over the Baby Monster is now feasible using the distributed disks of a cluster. Our work allows researchers to use either a matrix, a permutation, or a word representation for computing over the Baby Monster where previously only a matrix representation was available. The methodology is demonstrated by using as a signature the image of a vector that is stabilized by the maximal subgroup. The technique extends to finite simple groups and to other groups, through other signatures.

Categories and Subject Descriptors

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

General Terms

Algorithms, Experimentation

Keywords

permutation groups, matrix groups, group membership, disk-based methods, parallel computation, Baby Monster group

1 Introduction

The goal of providing a uniform computational methodology for working with the large simple groups has been a sought after target for a long time now. Some major achievements have been reached, such as the first constructions of permutation representations and strong generating sets for the Lyons group [15] and Janko's group J_4 [19], and now even the Thompson group [38, 39, 40, 20]. These accomplishments have been helped along by the sharing of information at websites such as Wilson's Atlas Web Page [41] which provides initial matrix representations for standard generators. This site also provides information for the sporadic simple groups, a natural ladder of challenge problems for group membership. Computing over the Baby Monster, the next to last in the ladder, had been all but infeasible. But with clusters that have access to terabytes of disk space and gigabytes of memory, a permutation representation for the Baby Monster has been discovered, and a computation of a strong generating set becomes feasible.

Computations such as these are more than just academic. They provide the ability to decompose large groups to answer many mathematical questions about them. They form the base case of the Matrix Recognition Project's [30, 31] recursive decomposition into normal groups. All of the sporadic simple groups can be identified based on tests on random elements [4]. Until now, however, individual group elements could not be identified uniquely, in terms of a word in the standard generators for example, for groups such as the Baby Monster. It is this form of identification that is required for the Matrix Recognition Project.

Previously permutation representations have been preferred to matrix representations for these large groups due to the the rich and mature body of permutation group algorithms [1, 2, 3, 7, 11, 12, 13, 14, 20, 29, 34, 36]. However, as the size of the groups worked with increases, a permutation representation becomes less attractive because of the large space requirement to store the permutations. In these cases, an effective solution utilizing a matrix representation is desirable.

In the Baby Monster, for example, a single permutation would require 65 GB using 5 bytes to represent a point. This is impractical with today's technology, though it may become feasible (making a direct permutation approach such as Cooperman's and Robinson's [20] feasible as well) once terabyte disks become available. A matrix representation of dimension 4370 over GF(2) is much more practical requiring only 2.3 MB per matrix.

Because of these factors, we follow the approach outlined by Butler [6] and extended by Murray and O'Brien [35]. We use a matrix representation but view the matrix group as a permutation group in which matrices act on vectors, rather than permutations on points. We implement a modification of the randomized Schreier-Sims algorithm that develops the point stabilizer chain of subgroups by finding Schreier gen-

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

ISSAC'06 July 9–12, 2006, Genova, Italy.

erators from random group elements.

Even though a matrix representation is used, a permutation representation can easily be derived. This is done by identifying the transversal of the first point stabilizer subgroup (the first *fundamental orbit*) with the points of the permutation domain. While this is technically possible for the Baby Monster, the size of the resulting permutations (130 GB) make this computation undesirable.

While computing a fundamental orbit, we generate a data structure for the corresponding *Schreier tree*. Schreier trees are defined formally in Section 2.2, but it suffices to think of them as small depth spanning trees for the fundamental orbit. Previously, in the work of Butler, Murray and O'Brien, all of the data for these trees was stored locally. Thus they could afford the traditional storage cost for a Schreier tree, a full vector as well as a *backpointer* (pointer to the parent in the tree).

Because of the size of the Baby Monster, the computation must be distributed. Traditionally, a 600 byte vector and *backpointer* would be passed across the network, now we only pass across a 12 byte compressed signature and a word in the generators. Because there are only 2 generators, a single step in the generator word costs only 1 bit, we assume a maximum length of 150 (a generous assumption), this implies a total of 20 bytes, and leads to an 83% reduction in the total network time, as well as a significant reduction in disk time (see Section 6 for more details).

The algorithm of this paper is an extension of the one used to compute the Thompson Group[20]. However, because it is distributed and out-of-core, we must consider additional factors such as network bandwidth, disk speeds, synchronization, and both memory and disk size limitations. The nodes in the computation both work on a piece of the Schreier tree as well as store a piece of the hash.

The result of the computation of the first Schreier tree is a collection of *compressed signatures* with corresponding words in the generators distributed across many nodes. These nodes can later be polled in reasonable time for specific vectors in a discovery of the remaining Schreier trees. This enables group membership random generation, and many other permutation algorithms for efficiently computing in the full Baby Monster.

Sections 2 and 3 provide the background of the Schreier-Sims algorithm, along with definitions and notation. Section 4 provides an overview of the algorithm, while Section 5 provides further details. Section 6 estimates the running time of the Baby Monster computation within a factor of two. Such estimates are useful for predicting running times for other large groups. Section 7 discusses how the constructed permutation representation will be used in the future to apply the Randomized Schreier-Sims algorithm. Section 8 provides experimental results.

1.1 Related Work

Sims presented a specialized construction of the Baby Monster in 1980 [37]. In 1998, in a tour de force, a specialized construction of the Monster was produced by Linton, Parker, Walsh and Wilson [32]. This was later elaborated on by Holmes and Wilson [28]. However, in each of those cases, specialized techniques were needed due to the limitations of CPU, RAM and disk.

In 1994, Cooperman, Finkelstein, Tselman and York [15, 16] constructed a permutation representation of Lyons's group of degree 9,606,125 from matrix generators, and also produced a strong generating set. Their representation was of permutation degree 9,606,125 for Lyons's group acting on a conjugacy class of subgroups of order three.

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 [26]. That presentation was verified as producing Lyons's group by Gollan and Havas [25].

The next larger sporadic simple groups are Thompson's group acting on 143,127,000 points and Janko's group J_4 acting on 173,067,389 points. Cooperman, Hiss, Lux and Müller [18] and Cooperman and Tselman [21] carried out a condensation for Thompson's group, which implicitly yields a permutation representation. At approximately the same time, Weller [40] carried out a direct computation of Thompson's group. Furthermore, Havas et al. [27] produced a permutation representation for Thompson's group via a coset enumeration.

Weller [38, 39] also produced a permutation representation of Janko's J_4 group, using some of the hashing techniques of [15, 16] and the double coset trick of [23, 24]. That work was used in a revised existence proof for Janko's group [19].

In all cases not involving coset enumeration, the starting point was the matrix generators of Wilson's Atlas [41].

The randomized Schreier-Sims algorithm used in this paper depends on a source of random elements. For our purposes, the method of random subproducts of [8] works well.

Most recently, an implementation by Cooperman and Robinson [20] was able to compute over the Thompson Group, a sporadic simple group acting on $143,127,000$ points, relatively quickly (36 minutes). The resulting solution could both answer questions of membership and solve for the order of the group.

This computation was a landmark because in terms of the number of points on which a group acts, it was the first to solve for any sporadic simple group of this magnitude, previous computations had just managed groups of at most 10 million points. Whereas the computation of the Thompson group is a factor of ten over previously computed groups, the Baby Monster is a factor of one hundred over the Thompson group. Computing over the Baby Monster necessarily must consider a whole new range of potential limitations.

1.2 Comparison of Disk-Based and Memory-Based Approaches

Matrix-vector multiplication over small finite fields is limited by the bandwidth of RAM and not by the CPU speed under current technology. When the size of the orbit is larger than the aggregate RAM in the cluster, a well-known approach [33] is to conceptually partition the orbit and store only mimimal elements for each partition. In generating the Schreier tree, the full set of points of the partition is dynamically regenerated as needed, and each generator is still applied to each point of the partition.

We rejected this memory-based approach as being unacceptably slow. Storing all points of the orbit on disk saves us the cost of dynamically regenerating each partition repeatedly whenever it is the image of some point under some generator. Even though our disk-based algorithm is optimized to minimize memory bandwidth, we continue to find that memory bandwidth, and not disk speed, is the bottleneck (see Section 6.4).

Secondarily, the disk based algorithm produces nearly optimal length transveral elements as words in the generators,

unlike the memory-based algorithm. This leaves us the option of saving all of our strong generators as relatively short words instead of computing the corresponding matrix. This saves memory space at the cost of time.

2 Background

The computation of the Baby Monster is possible largely due to the fact that it is a group with a small base, probably well under 10. We define base and other notation below, along with some common computational methods.

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" (fixing points less 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. The *b*ase of a group is the number of distinct proper subgroups of G in the chain above.

Let $G/H = \{Hg\colon\ g \in G\}$ be the *set of cosets* of H in G (where $Hg = \{hg\colon\ h \in H\}$). 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,

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

2.2 Review of Schreier-Sims Randomized Group Membership Algorithm

Computation of groups in this fashion is commonly done by computing what is known as a Schreier tree, or a tree in which the nodes represent vectors and the edges represent matrices. The tree is built in a typical breadth first style by starting out with the transversal at level l, then applying a random element in level l (generated as shown above) to the transversal. This is done repeatedly until no new points are seen. After this, the same method is used to generate level $l + 1$.

2.3 Generation of Random Elements

For more information about the generation of random elements of a group, see Cooperman and Robinson[20].

3 Notation

In describing the algorithm used to compute the first Schreier tree for the Baby Monster, the following terms will frequently be used.

Matrix, Generator The matrix generators for the Baby Monster. There are two of dimension 4370 in GF(2).

Initial Vector The vector which when used as the root node for the first Schreier tree leads to an orbit enumerating the cosets of the maximum subgroup.

Signature Image of the initial vector under repeated applications of the generators.

Bitstring, Word in Generators A binary string indicating which generators to apply to get from the Initial Vector to some desired signature.

Compressed Signature A signature compressed to 96 bits. Because $|FirstSchreierTree|^2 \ll 2^{96}$, every signature probabilistically has a unique compressed signature.

Hash Index A 40 bit value representing the hash of the compressed signature. The high bits of the hash represent the machine, and the low bits represent the hash entry on that machine.

Hash Array A bit array twice as large as the total number of signatures, 50% occupancy. The bit indicates whether the hash index is present, no corresponding hash value is stored. This is stored in RAM distributed across the nodes of the cluster.

Computation Queue A FIFO queue of entries containing a word in the generators and the corresponding signature. Each node stores its Computation Queue locally on disk.

Computation Block A fixed number (currently 25,000) of computation entries containing a word in the generators and the corresponding signature. This is stored in memory.

Check Message A message containing words in the generators and their corresponding compressed signature from a computation block.

Final, Collision Queue A queue of entries containing a word in the generators and the corresponding compressed signature that has been checked against the hash and either was a new entry (final) or resulted in a collision (collision). These are used to determine invalid collisions. Each node stores queues corresponding to its portion of the hash locally on its disk.

Final, Collision Block A fixed size (currently 10 MB) of final or collision entries containing a word in the generators and the corresponding compressed signature.

4 Distributed Algorithm for Search Space Discovery

For our discovery of the Baby Monster's first Schreier tree, we use a breadth-first search space discovery technique developed by us for this application, but widely applicable. This technique assumes a distributed computation where a unique hash may not be available. There are three phases.

Initialization. First, we must initialize our data (load our matrix generators) and obtain a hash space distributed across our nodes. We know this hash array will not be unique. In the case of the Baby Monster, it is only twice the length of the fundamental orbit and therefor will have many collisions. After this, the master node discovers a single computation block in our search space based off a root signature (initial vector). This initial block is then divided up between the nodes and added to their computation queues.

Phase One. During this phase, the computation queue is grown by applying the generators to the signatures at the head of the queue. The new signatures are then hashed (as a 40 bit value of the compressed signature) to determine uniqueness. Unique points are appended to the computation queue, added to the hash, and added to the final queue. Collisions are added to the collision queue. This phase finishes when the computation queue is emptied.

Phase Two. Here the final and collision blocks are sorted by their compressed signatures and duplicate values (values with the same compressed signature) are stripped from the collision queue. New values (invalid collisions) in the collision queue are sorted based on their word in the generators. This allows us to recompute their full signatures with minimal matrix-vector multiplications. Once the full signatures have been computed, they are added to a new computation queue locally.

Table 1: The Phase One Managers

Manager	Purpose	Block Passed	Receiving Manager
Read	Reads blocks from disk	Head Computation	Computation Manager
Computation	Performs matrix-vector multiplications	Tail Computation	Check Manager
Check	Strips duplicates from blocks	Tail Computation	Write Manager
		Check Message	Hash Manager (Network)
Hash	Detects duplicates in block	Check Message	Check Manager (Network)
Write	Writes blocks to disk	Tail Computation	None (Disk)

After this, Phase One and Two are repeated ad infinitum until no new values are discovered in Phase Two. At this point, the search space (Schreier tree) has been fully discovered.

4.1 Phase One Overview

In order to overlap network, disk, and computation time, the tasks of Phase One are pipelined. The details of Phase One are best described according to the responsibilities of the five threads that manage the phases of this pipeline: the read manager, the computation manager, the check manager, the hash manager, and the write manager. Table 1 4.2 contains a list of all the managers and who they pass information to. Following is a description of those managers.

Read Manager. Because the computation queue is stored in computation blocks on disk, This thread must load up a new computation block from the head of the queue on disk and hold it until it is ready to be processed.

Computation Manager. Recall that the Schreier tree is generated by breadth-first search. This thread accepts a head computation block from the *Read Manager* and then computes the children signatures for that block by applying the two generators. These new signatures form tail computation blocks. Each of these blocks are held until they are ready to be checked. Once all the tail blocks have been discovered the head computation block may be freed.

Check Manager. Since the hash table is distributed, it is this thread's job to send out hash check requests to all of the nodes. It starts by scanning the computation block. For each entry, it appends the compressed signature and word in the generators for that entry to a message whose destination is the machine responsible for the hash of that compressed signature. After this, it sends all of the messages out and waits for replies. These replies tell it which signatures to remove from the computation block. After removing these signatures, it holds the block until it is ready to be written.

Hash Manager. This thread waits for an incoming hash check request and checks each of the entries in it against its hash table. If there is a collision, it marks the entry with a deletion flag and adds it to its local collision block. Otherwise, it updates the hash table and adds the entry to its local final block. Once it finishes with the check request, it sends back a response that contains the deletion flags. It may also store the final and collision blocks to disk if they have reached their size limits.

Write Manager. This thread writes computation blocks to the tail of the computation queue and then frees the space used by these blocks.

4.2 Phase Two Overview

Phase two goes through three distinct passes. Each of these passes generates the data for the next pass. The passes are described here.

Formatting the Blocks and Sorting. Initially the final and collision blocks generated by the current pass contain entries of different lengths in an unsorted order. It is the job of this pass to standardize the entry sizes and sort them based on their compressed signatures.

To do this, the program must keep track of the largest word in the generator for each block. The values in the final and collision blocks are then read from disk and converted into new blocks each with a standardized size. Quicksort is then called on the blocks to sort based on their compressed signatures. This is done one at a time for each new block discovered in phase one. Once a block is completed, it is written back to disk.

Removing Duplicates. The process of removing duplicates is made easy because the compressed signatures are now in sorted order in each block. A single pass through all of the final blocks and the collision blocks generated by this pass can use a priority queue for each to strip all duplicates from the collision blocks, leaving only new elements that were the result of invalid hash collisions. These elements are added to new final blocks and once a complete final block has been discovered, it is sorted by the word in the generators (leftmost in the tree first) and written to disk.

Rebuilding the Signatures. Now, with the new final blocks discovered in the previous step, we can rebuild their full signatures using the words in the generators. Because our new blocks are now in sorted order, we can reconstruct the tree using as few matrix-vector multiplications as possible. Intermediate vectors along the path are stored and used for later computations where applicable. Because of sorting, we can use another priority queue to store only a vector for every level in the tree, and we also never repeat a matrix-vector multiplication on a single node.

As the signatures are rebuilt, they are added back into new computation blocks and put into the computation queue. As stated above, once this is finished phase one is restarted. The computation finishes when no new points are discovered in phase two.

5 Details of the Algorithm

We are looking for a permutation representation of a finite simple group, given a matrix representation. We will construct a permutation domain in one of two ways: either as the set of images of an appropriately chosen initial vector; or as words in the matrix generators. Both representations are compact. In both cases, the image of an appropriately chosen vector serves as a *signature* for the element of the permutation domain. The first scenario is somewhat more efficient, but is not applicable to all finite simple groups. The second scenario is fully general.

5.1 Initial Vector: Case of Action of Matrices on Vectors

Many matrix representations of simple groups, including those for which we present experimental results in the case of the Baby Monster, Janko's group (J_4) and Harada-Norton (HN) (see Section 8), there is an initial vector v such that elements of the permutation domain can be represented as vector images vg, for g a matrix element g in the group. The vector image, vg is the *signature* of the element of the permutation domain.

We illustrate the case for the Baby Monster. For the Baby Monster, the algorithm initially requires the two group generators and a vector that is fixed by the largest maximal subgroup of the Baby Monster (which is isomorphic to $2.^2E_6(2) : 2$. It is well known that the orbit of such a vector will form the smallest possible domain for a permutation representation. Given any generators for the Baby Monster in the desired representation (dimension 4370 over GF(2)), Wilson's Atlas Web Page [41] produces an efficient algorithm to find standard generators (a, b). It also provides generators (x, y) for the largest maximal subgroup in terms of the standard generator. One then takes the intersection of the fixed spaces of x and y. Any non-trivial vector in that intersection will do.

5.2 Initial Vector: Case of Conjugate Matrix Action

Since not all matrices have signatures based on an initial vector under the matrix action, we employ an alternative action for other matrix representations of finite simple groups. This method was developed by Cooperman et al. [15, 16], and the original papers describe details, and additional optimizations not discussed here.

We begin by considering the conjugate action of the group on subgroups of prime order. We can construct a "small" permutation representation, if not always the smallest degree representation.

Next, in order to make such a construction effective, we only use matrix-vector computations, instead of matrix-matrix computations. To do so, we express each element of the permutation domain as a word w in the generators of the group. For a fixed subgroup H of the chosen conjugacy class, the conjugate subgroup $H^w = w^{-1}Hw$ is a "point" in the permutation domain. The matrices in the group act on these permutation "points" by conjugation, yielding a new word of length one more than the original word.

Although a "point" may be represented by more than one word, both words would have the same signature. This is accomplished by choosing at random an initial vector v of the underlying vector space. It is shown [15, Lemma 3.1] that with high probability, the randomly chosen initial vector v has the property that a group element g in the group is uniquely determined by the image vector vg. (The probability of having chosen a "bad" v is less than $|G|/q^{n-m}$ for a group $G \leq GL(n, q)$ with m the maximum dimension of a fixed point subspace of any non-identity element of G.

Assuming a properly chosen v, each "point" H^w is uniquely determined by the set of image vectors $vw^{-1}Hw$. Since H is of prime order p (and usually small), it suffices to store the lexicographically smallest of the p image vectors $vw^{-1}Hw$ in a hash table.

Although this method was not implemented for the parallel architecture, it was the basis of a sequential implementation that constructed a permutation representation of Lyons's group for the first time [15, 16]. The permutation degree of Lyons's group in its smallest conjugate action is 9,606,125.

5.3 Hash Table

A memory resident perfect hash is impossible for the Baby Monster, This is because the compressed signatures alone would require over 150 gigabytes of storage, or over 5 gigabytes of storage per machine in a 32 node cluster.

Other methods do exist for determining the uniqueness of a point based on a smaller amount of data[9]. These methods rely on applying generators to the target to reach some well known signature. This essentially trades computation time to reduce computation space. Since we want to reduce the number of matrix-vector multiplications to make our method as fast as possible, an approach such as this is infeasible.

Instead of trying to determine without error whether a compressed signature is a duplicate, we will only determine whether that compressed signature is unique. Elements that result in hash collisions will be queued up in a collision queue to be checked later in the computation.

Using this approach, we can set a hash table comprised of single bits to be twice the size of the expected number of elements. This implies 28×10^9 bits, or approximately 4×10^9 bytes. This leads to a hash table that uses only 128 MB per processor.

From this, we can predict the number of elements in the collision queue during the first (and largest) pass through phase one to be the number of valid collisions plus the number of invalid collisions. For the Baby Monster this implies $14 \times 10^9 + 14 \times 10^9/2/2$, 18 billion element total, or 600 million elements per node. Assuming a max tree depth of 150, this implies the collision queue uses 350 GB total, or just over 1 GB per machine.

5.4 Collision Queue

Recall that the purpose of the collision queue is to look at items that hash to the same value and determine whether they are actual duplicates or just hash collisions. The method by which it does this is describe in the Phase Two section of the Overview 4.2. Also, the space used by the collision queue is approximately 1 GB per processor. This implies that we can easily store the entire collision queue on disk.

Once the excess data has been removed from the collision queue, the new structure will have approximately 25% of the full tree. In addition, by sorting and not repeating multiplications, we amortize the cost of rebuilding the tree. With 30 nodes, typically, this implies doing only 25% of the vector-matrix multiplcations we would normally have to do. This represents a major improvement over methods that find a common node, because we only need to perform additional matrix-vector multiplications in approximately 12.5% of the cases and we only do 25% of these, meaning our cost is 3% the depth of the tree per element.

6 Theoretical Computation Time

We can accurately estimate the running time of the computation based purely on the architectural parameters and the parameters of the Baby Monster group representation. We will assume a 30 node cluster of 2 GHz computers with 512 megabytes of DDR-266 RAM (1 gigabyte of sequential data per second). In addition we will assume 200 gigabytes of disk space per node. (Note that even at today's prices of less than one dollar per gigabyte, the cost of disks on all nodes is still under $6,000.) We also assume a conservatively estimated transfer rate of approximately 10 megabyte

per second. Finally, we assume the computers are connected by Fast Ethernet (100 MHz).

We assume that the depth of the first Schreier tree is no greater than 150, and that the average depth is no greater than 75. We know from computation that the depth of the first tree in the Lyons group was 37 [15] and the maximum depth of the first tree in the Thompson group was 72 [20]. Our current test run for the Baby Monster has finished the full first pass discovering 9 billion of the 14 billion points, and the maximum depth in the run is 144, with an average depth approximately 75. Later passes extend the tree from random nodes. We do not expect the average or maximum depth to grow significantly.

6.1 Matrix-Vector Multiplication

Recall that the generators for the Baby Monster are of dimension 4370 over GF(2). We assume for the purposes of these computations that we have access to a 64 bit "exclusive or" operation. Most I/O to RAM follows a streaming access pattern. Our cluster uses 266 MHz DDR RAM and the 8 byte Pentium bus. Under optimal conditions, this allows us to read or write to RAM at 2.12 GB/s.

However, to speed up the time of matrix-vector multiplications, we can use a technique called greasing. Greasing precomputes multiplication tables by combining bands of rows for a matrix in order to speed up subsequent multiplications. This technique is also used in GAP [22] and MAGMA [5]. Since we only use two matrices (the generators), this method can be used to speed up the computation using a reasonably small amount of space. We found using a band size of 8 gave a reasonable speedup for a reasonable amount of memory:

Mem Space	$4370 \times \lceil 4370/8 \rceil \times 2^8$bits	76MB
CPU Time	$\lceil 4370/64 \rceil \times \lceil 4370/8 \rceil$ops	2×10^{-5}s
Mem Time	$\lceil 4370/8 \rceil \times \lceil 4370/8 \rceil$B	1.4×10^{-4}s

We can see that by using greasing and about 150 megabytes of extra space, we are able to perform a matrix-vector multiplication eight times faster than a naive method, and four times faster than a method that skipped rows multiplied by zero. Experimentally, we obtained greasing times closer to 3×10^{-4}, and it is these times we will use for the rest of our calculations.

6.2 Minimum Computation Time

In the computation of the first Schreier tree, we can assume that we need at least one matrix-vector multiplication for each element discovered, as well as an additional matrix-vector multiplication for dead ends (each node must connect to some other node in the tree, bounding the maximum number of matrix-vector multiplications performed). This implies a minimum of $3 \times 10^{-4} \times 14 \times 10^9 \times 2 = 84 \times 10^5$ seconds or 84 computer days. Spread over 32 processors, this implies at least 3 days of computation.

Traditional methods to reduce hash size operate by performing extra matrix-vector multiplications at each node discovered. Our method only performs on the order of 2 additional matrix-vector multiplications per signature, for a total of 9 days. Other methods typically do not achieve such a low bound.

6.3 Time for Program Components

Time for Read and Write Managers. The Read and Write Managers must make disk accesses to the full signatures. In addition, they must load up the words in the generators for these signatures. This requires reading/writing $14 \times 10^9 \times (100B + 550B) = 9TB$. Given our disk rate this implies 9.1 days total, or 0.25 days per machine. This implies a half of a day both reading and writing.

Time for Computation Manager. We know that the Computation Manager will perform no more than the minimum number of matrix-vector multiplications, as it performs a depth first search and eliminates duplicate nodes along the way. This implies that the processor time spent in the computation manager is approximately 3 days.

Time for Check Manager. The Check Manager can make a single pass through the computation block when removing invalid computation entries, since it will also be writing these blocks to disk, it is assume that the single pass through memory is significantly less than the time it takes to write that memory to disk, and is therefore negligible.

Also, the Check Manager must do message passing. For each computation entry recorded (including duplicates) it must pass a message of size at most 20 bytes (derived from the compressed signature and the maximum word length) and receive a message of the same size. This implies a total of 560×10^9 bytes. Given a network speed of 100 megabits per second, this would require a total of one half network day. This time is dominated by the time spent in memory and on disk. Note that this does not hold true as more nodes are added unless the aggregate network bandwidth also grows.

Time for Hash Manager. The hash manager accesses one bit of memory for each signature that it is given. Once again, we can assume that this time is dominated by the rest of the computation.

In addition, it must also write a 20 byte entry to disk for each signature that it is given. Even though this is twice the number of actual entries written by the Write Manager, the size is 30 times smaller. Therefore we can expect the check manager's disk access to be the dominating time, and this piece to require under 0.1 disk days.

Time for Formatting the Blocks and Sorting. We only need to sort each block once. This requires a single read and write to disk for each final and collision block. Given the disk time computed for the Hash Manager, this is a total of under 0.1 days. We assume the time for quicksort is negligible compared to the disk access.

Time for Removing Duplicates. We only need to view each element in the Collision Queue once for removing it as a duplicate (half the items reading and writing implies under 0.1 days). The Final Queue, however, requires us only to read, but we must check all of the values at each pass. The base time to read half of the total items is 30 minutes. And given that we have a hash twice the size of the Schreier tree itself, we can expect to perform $log_2(14 * 10^9) = 34$ passes. This implies a total time of 17 hours. Once again, we assume the time for quicksort is negligible.

Time for Rebuilding the Signatures. We perform approximately 2 additional matrix-vector multiplications for each element to determine whether it is a duplicate. Given the base time of 3 days to compute all elements, this should mean that rebuilding signatures takes 6 days.

6.4 Combined Times

The full time including all components of the process is shown here.

Manager	Disk Time	CPU/RAM Time
Read/Write	0.5 days	0 days
Computation	0 days	3 days
Check	0 days	0 days
Hash	< 1 day	< 1 day
Formatting	< 1 day	< 1 day
Removing	< 1 day	< 1 day
Rebuilding	< 1 day	6 days
Total	*2 days*	*10 days*

In addition, the algorithm will also spend a minimal amount of network time checking computed blocks. This implies the total time spent by our algorithm is 10 days assuming we overlap disk and processor time.

These estimates indicate that the initial phase one computation should complete in 3 days. Experimentally, phase one finished in 4.56 days.

7 Future Work: Computation of Remaining Levels

We expect to construct the first Schreier tree, with 1.4×10^{10} points, in approximately a week and a half. Once computed, this Schreier tree can be used to easily discover the remaining trees using the methods outlined by Cooperman and Robinson [20]. We expect a further half day to make a single pass through the final blocks to construct a sufficient number of random elements of the point stabilizer subgroup. With these random elements, we can use earlier methods to compute the remaining Schreier Trees (which now are of a reasonable size).

8 Experimental Results

As stated previously, the experimental time for a matrix-vector multiply was 3×10^{-4} seconds. Some additional experimental times are shown below.

8.1 Times for Other Groups

In addition to running our algorithm on the Baby Monster, we have also run it on other groups such as Harada-Norton and J_4. For Harada-Norton, where the initial Schreier tree has 1,140,000 nodes, we use a representation of dimension 760 over GF(2). Our computation finishes within 15 minutes. The times for this group are too small to be of use for predicting times for the Baby Monster.

J_4 is a group whose first Schreier tree has 173,067,389 nodes. For it we use a representation of dimension 112 over GF(2). We ran a computation of the full Schreier tree in a non-dedicated cluster of SunBlade 1500 workstations. Each machine had a 1 gigahertz processor, 40 GB of local disk, and 1 GB of RAM. The network was Gigabit Ethernet.

The initial first phase finished in 50 minutes. After this, the initial second phase required only 2 minutes to compute. This drastic difference in time is most likely due to the lack of network communication in the second phase, coupled with the fact that when the matrix-vector multiplications are being performed, our program is not running other managers unlike during phase one. The first phase of the second iteration required only 15 minutes and the corresponding second phase time was negligible. In total, J_4 ran for just over an hour and a half and used only 16 MB of memory per node.

8.2 Times for the Baby Monster

At publication time, we have computed 9 billion of the 14 billion signatures in just 5.08 days (completing an initial first phase in 4.56 days and second phase in 0.52 days). This implies that the computation time for the full group should be about 11 days. Due to systems programming issues, the full computation is still pending.

Remark. *In order to keep our method general to all groups, we have not performed a simple optimization. Since the generators for the Baby Monster are of order 2 and order 3. We can reduce the total number of matrix-vector computations by not checking those computations that lead to a previously seen vector.*

It is important to note that while discovery of new signatures may slow down later in the computation, our times are based on the total number of nodes discovered (including duplicates).

8.3 Experimental Setup.

We used a cluster of 30 nodes: 1.5 GHz Pentium 4 CPU, 512 MB RAM, 1.2 TB of local disk space per node (Only 200 GB per local node was used.) The computation was done under Redhat Linux 7.2 using the g++ 3.3 compiler for C++. The MPI implementation used was MPINU (the MPI subset provided with TOP-C [10]).

9 Acknowledgement

We gratefully acknowledge the conversations of Jürgen Müller on computing in the Baby Monster. We also gratefully acknowledge Jiri Schindler and EMC, in general, for providing a dedicated cluster with which to carry out our computation. In addition, we greatly appreciate the many detailed discussions with Jiri Schindler on systems issues. We also thank the referees for helpful comments.

10 Conclusion

Computing over the Baby Monster is not a matter of discovering a unique and highly specialized algorithm for the group, but rather one that emphasizes striking the right balance in terms of resources and time. We are required to use the processor, disk, and memory efficiently and to balance our load in a distributed computation for this group. By optimizing our architecture for the difficult case of the Baby Monster, other smaller groups can be addressed in smaller time by the same, uniform architecture.

11 References

[1] 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.

[2] 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.

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

[4] L. Babai and A. Shalev. Recognizing simplicity of black-box groups and the frequency of p-singular elements in affine groups. In *Groups and Computation III*, Ohio State Univ. Math. Res. Inst. Publ., Berlin, 2000. (Ohio, 1999), de Gruyter.

[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. The Schreier algorithm for matrix groups. In *SYMSAC '76, Proc. ACM Sympos. symbolic and algebraic computation*, pages 167–170, New York, 1976. (New York, 1976), Association for Computing Machinery.

[7] G. Butler and J. J. Cannon. Computing in permutation and matrix groups I: Normal closure, commutator subgroups, series. *Math. Comp.*, 39:663–670, 1982.

[8] 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.

[9] Cooperman, Finkelstein, and Sarawagi. Applications of cayley graphs. In *AAECC: Applied Algebra, Algebraic Algorithms and Error-Correcting Codes, International Conference*. LNCS, Springer-Verlag, 1990.

[10] G. Cooperman. Top-c: A task-oriented parallel c interface. In 5^{th} *International Symposium on High Performance Distributed Computing (HPDC-5)*, pages 141–150. IEEE Press, 1996. software at `http://www.ccs.neu.edu/home/gene/topc.html`.

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

[12] 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.

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

[14] 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.

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

[16] 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.

[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 E. Robinson. Memory-based and disk-based algorithms for very high degree permutation groups. In *Proc. of International Symposium on Symbolic and Algebraic Computation (ISSAC '03)*, pages 66–73. ACM Press, 2004.

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

[25] 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.

[26] 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.

[27] 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.

[28] P. E. Holmes and R. A. Wilson. A new computer construction of the Monster using 2-local subgroups. *J. London Math. Soc.*, 67:349–364, 2003.

[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] S. A. Linton, R. A. Parker, P. G. Walsh, and R. A. Wilson. Computer construction of the Monster. *J. Group Theory*, 1:307–337, 1998.

[33] F. Lübeck and M. Neunhöffer. Enumerating large orbits and direct condensation. *Experiment. Math*, 10:197–206, 2001.

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

[35] S. H. Murray and E. O'Brien. Selecting base points for the Schreier-Sims algorithm for matrix groups. *J. Symbolic Comput.*, 19:577–584, 1995.

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

[37] C. C. Sims. How to construct a baby monster. In M. Collins, editor, *Finite simple groups II*, pages 339–345. (Durham 1978), Academic Press, 1980.

[38] 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.

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

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

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