A Disk-Based Parallel Implementation for Direct Condensation of Large Permutation Modules

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

Jürgen Müller
Lehrstuhl D für Mathematik
RWTH Aachen
52062 Aachen / Germany
juergen.mueller@math.rwth-aachen.de

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

ABSTRACT

Through the use of a new disk-based method for enumerating very large orbits, condensation for orbits with tens of billions of elements can be performed. The algorithm is novel in that it offers efficient access to data using distributed disk-based data structures. This provides fast access to hundreds of gigabytes of data, which allows for computing without worrying about memory limitations.

The new algorithm is demonstrated on one of the longstanding open problems in the Modular Atlas Project [11]: the Brauer tree of the principal 17-block the sporadic simple Fischer group Fi_{23}. The tree is completed by computing three orbit counting matrices for the Fi_{23}-orbit of size $11,739,046,176$ acting on vectors of dimension 728 over $GF(2)$. The construction of these matrices requires 3-1/2 days on a cluster of 56 computers, and uses 8 GB of disk storage and 800 MB of memory per machine.

Categories and Subject Descriptors

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

General Terms

Algorithms, Experimentation

Keywords

permutation groups, matrix groups, disk-based computation, parallel computation, Brauer trees, condensation, sporadic Fischer group

1. INTRODUCTION

In recent years, in particular in the framework of the Modular Atlas project mentioned below, but also in other applications, the need of techniques to deal with very large permutation domains acted on by finite groups became apparent. Condensation, whose formalism is recalled in Section 2, is one of the workhorses allowing one to handle those, by sometimes dramatically decreasing the size of the objects to be managed explicitly, while retaining enough of their internal structure to remain useful.

In [7] this was done for the orbit of some vector under a linear action of the group in question, by first forming a permutation representation, which was then processed to find the associated condensed module. Such computations were limited to permutations with at most hundreds of millions of points, due to memory limitations. Those computations used a space-time trade-off in order to stay within the limits of aggregate RAM in a cluster. Under such a space-time trade-off, optimizing any computation with more than a billion points typically pushed the time requirements so far that the computation became impractical.

A more recent research direction [6, 22] looked at using distributed disk for large computations such as orbit enumeration. Here the disks of a cluster are accessed in a streaming manner for similar performance to a single memory module. Essentially this gives a computation access to terabytes of fast storage where previously only gigabytes were available. Many problems that were previously impossible have become feasible due to a large space-time tradeoff. Using one of these distributed disk-based techniques capable of generating orbits with tens of billions of vectors, it becomes advantageous to work with the vectors in the orbit directly rather than producing a permutation representation.

In this paper we present a distributed disk-based implementation of the direct condense technique. The details of the underlying sequential algorithm and the new distributed algorithm are given in Sections 3 and 4, respectively. Our implementation, which of course is of general purpose, has been successfully tested by way of the following example, which is detailed in Section 5: We consider the sporadic simple Fischer group Fi_{23}, which has a transitive permutation domain of size $11,739,046,176$. This is realized as a set of vectors of dimension 782 over $GF(2)$. Once this orbit has been enumerated, it is partitioned into the $6,486$ suborbits of a suitable condensation subgroup, and the associated orbit counting matrices for several elements of Fi_{23} are produced.

[*]This work was partially supported by the National Science Foundation under Grant ACIR-0342555.

1.1 Related Work

Enumeration and direct condensation of large orbits, and their application to modular representations and other aspects of finite groups, have raised some interest in recent years. The direct condense technique has been invented in [20], already including a notion of landmarks. It has been implemented as parallelized versions in [7, 12], and has been further developed using subgroup structures in [15, 17, 19].

As for concrete examples dealt with, in [5] the sporadic simple Thompson group was considered, where the computational architecture STAR/MPI (currently ParGCL) [3] was used, while in [18] the sporadic simple Lyons group was considered. In both cases, the aim was to complete certain Brauer trees. In [15, 16, 19, 22], the sporadic simple Baby Monster group was considered.

1.2 Acknowledgements

The authors thank Leo Hill for providing on the Northeastern University cluster the extensive disk resources needed for this computation. We also thank Daniel Kunkle of Northeastern University for helpful discussions concerning efficient disk-based data structures and algorithms.

2. BACKGROUND

2.1 Notation

In examining groups and their actions on a vector space, it becomes useful to have some shorthand notation: We use the expression v^G, where v is a vector and G is a group, to denote the G-*orbit* of v, or $v^G = \{v^g : g \in G\}$ where v^g is the action of g on v by vector-matrix multiplication. In addition, given a subgroup $K \leq G$, a G-orbit $O = v^G$ is a disjoint union of K-orbits $S = \{S_1, \ldots, S_k\}$, called the K-*suborbits* in O. In particular, we have $|S_1| + \ldots + |S_k| = |O|$. Finally, it will also be useful to have notation for applying a particular group element $g \in G$ to all elements in a set: if O is a set of vectors, $O^g = \{w^g : w \in O\}$.

2.2 Condensation

Condensation, or more precisely *fixed point condensation*, was invented in [23] to aid in finding new irreducible representations of a group and to analyze existing ones. Its theoretical underpinnings as particular Schur functors are described in [9]. Since its invention it has been used in a number of settings. The so-called *direct condense techniques* related to the present work has already been mentioned in Section 1.1. For more details we refer the reader, for example, to the overviews in [14, 15].

Fundamentally, the goal is to *condense* a permutation representation on a large number of points, or a matrix representation of a high dimension, into a manageable matrix representation. The condensed representation typically has a much smaller dimension than the original one, which allows one to reasonably compute with condensed matrices, and to extract information about the original representation. Computing with the original representation would have been infeasible.

2.3 Fixed point condensation

Formally we consider the group algebra FG of the group G over the field F of characteristic p. Letting $K \leq G$ subgroup having order $|K|$ prime to p, a so-called *condensation subgroup*, there is the idempotent $e = |K|^{-1} \cdot \sum_{g \in K} g \in FG$. Then to any (right) FG-module M we associate the *condensed module* Me, which is a module of the so-called *Hecke algebra* $eFGe$. In practice, Me is the subset of M consisting of the elements left fixed by K, from which comes the name *fixed point condensation*.

2.4 Permutation modules

If FO is the permutation module associated to the finite G-set O, then the condensed module FOe is described as follows: Letting $S_1, \ldots, S_k$ be the K-suborbits in O, we let $S_i^+ := \sum_{w \in S_i} w \in FO$ be the associated orbit sums. Then $\{S_i^+, \ldots, S_k^+\}$ is an F-basis of FOe, and for $c \in G$ the action of the *condensed* element ece on FOe is given as $S_i^+ \cdot ece = \sum_j C_{ij}(c) \cdot |S_j|^{-1} \cdot S_j^+$, where

$$C_{ij}(c) = |\{w \in S_i \colon w^c \in S_j\}|.$$

Hence condensing $c \in G$ essentially boils down to computing the *orbit counting matrix* $C(c) = [C_{ij}(g)]$ of dimension k.

In the *direct condensation technique*, as it is used here, we do not write down permutations to describe the G-action on O, but instead use a linear G-action on a vector space V to give an implicit description of O as follows: We specify a subgroup $H \leq G$ and a vector $v \in V$ such that $H = \mathrm{Stab}_G(v)$. Thus the orbit $O = v^G$ is equivalent as a G-set to the set of cosets of H in G.

3. SEQUENTIAL ALGORITHM AND PREDICTED TIME

Here the sequential condensation algorithm is presented along with an analysis predicting the running time for Fi_{23}. The running time is based on the architecture described in Section 4.5.

3.1 Sequential Algorithm

Condensation, as described in Section 2.2, can be broken up into three phases. These phases are shown below:

3.1.1 Orbit Enumeration

Once a suitable vector $v \in V$ has been determined, $O = v^G$ must be computed. This can be done in a straightforward manner using a breadth-first search algorithm as shown in Algorithm 1. Let $gens_G = \{g_1, \ldots, g_m\}$ be the generators of G.

Algorithm 1: *discoverOrbit*

```
Input: $v, gens_G$
Output: $O$
let $open \leftarrow$ a queue with only $v$ in it;
set $O \leftarrow$ an empty set;
while open is not empty do
    dequeue $h$ from open;
    for $g \in gens_G$ do
        $t = h^g$;
        if $t$ has not been seen then
            add $t$ to $O$;
            enqueue $t$ on open;
```

3.1.2 Suborbit Partitioning

Once the elements of O have been enumerated and a condensation subgroup K has been selected, O can be partitioned into K-suborbits S. This is done by performing a breadth-first search using the generators $gens_K = \{k_1, \ldots, k_n\}$ of K over the elements in O until all the elements have been seen. This is shown in Algorithm 2.

Algorithm 2: *partitionOrbit*

```
Input: $O, gens_K$
Output: $S$
set $S \leftarrow \emptyset$;
while $O$ is not empty do
    select any $o \in O$;
    let $s \leftarrow discoverOrbit(o, gens_K)$;
    remove the elements in $s$ from $O$;
    add $s$ to $S$;
```

3.1.3 Orbit Counting

Once O has been partitioned into K-suborbits S and a condensation element $c \in G$ has been selected, the orbit counting matrix $C(c)$ must be computed. This is done by counting, for each suborbit, how many elements in that suborbit map to each other suborbit, when c is applied to them. This is shown in Algorithm 3.

Algorithm 3: *orbitCount*

```
Input: $S, c$
Output: $C(c)$
set $C(c) \leftarrow$ a $|S| \times |S|$ all-zeros matrix;
for $i \in \{1, \ldots, |S|\}$ do
    for $s \in S_i$ do
        $j = x$ s.t. $s^g \in S_x$;
        $C_{i,j}(c) = C_{i,j}(c) + 1$;
```

3.2 Optimizations

To store O in full for Fi_{23} would require storing approximately 11.7×10^9 100-byte vectors. This would require a total storage of approximately 1 terabyte. Here two methods for reducing that space are presented. These methods add to the computation time required by the algorithms presented in this section. In addition, however, another method is presented that reduces the time for vector-matrix multiplications.

3.2.1 Fast Vector-Matrix Multiplication

The generators for Fi_{23} are of dimension 782 over GF(2). Under optimal conditions, the memory subsystem on an individual node is capable of reading or writing 2.12 GB/s. Given that we have access to 64 bit operations (specifically XOR), the time to perform a vector-matrix multiplication in Fi_{23} is 3.6×10^{-5}s. This time is dominated by the time to access memory.

However, to speed up the time for vector-matrix multiplications, we can use a technique called *greasing*. Greasing, which was invented by Richard Parker, 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 [8] and MAGMA [1]. 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.

3.2.2 Compressed Values

Rather than using full 100 byte vector values, 12 bytes can be used to represent each element in O and guarantee that it is distinct from every other element in O with a high probability. The representation size must be on the order of $lg(|O|^2) = 2 \times lg(|O|)$ in order to guarantee a high probability of uniqueness.

Storing this representation along with the path in the generators to the value from v requires on average 30 bytes of storage per element. In order to use these values for vector-matrix multiplications, however, the path to the value from v must be applied to get the full value. While the use of compressed values reduces the space of the computation, it results in additional vector-matrix multiplications.

Orbit enumeration requires storing full values only for the frontier (or open queue). Once the generators have been applied to a value, it can be stored in its compressed form with no additional computation time. Suborbit partitioning requires rebuilding a single value for each suborbit. This time is trivial in comparison to the time to generate the suborbit itself. Finally, orbit counting requires rebuilding only one value from each suborbit as well. After this, the full suborbit can be produced as it was in the suborbit partitioning phase and c can be applied to all values in that suborbit. Once again, the time to reproduce a single value is trivial in comparison to the time to generate the suborbit itself.

3.2.3 Landmark Discovery

As the size of the orbit O grows, it becomes increasingly difficult to store the elements of O seen during the enumeration phase. While the enumeration phase can use streaming disk in a breadth-first search, this cannot be done easily for the suborbit partitioning or the orbit counting phases, which must randomly look up elements. Therefore, an approach known as *landmark discovery* [4, 7] is commonly used to allow O to fit in memory.

In landmark discovery, a subset of the orbit elements are declared to be *landmarks* and retained in memory. The non-landmarks are discarded. This leads to storing only $1/L$ elements, where $1/L$ is the *landmark ratio*. Though this reduces the total storage, it requires additional work during the suborbit partitioning and orbit counting phases.

In the suborbit partitioning phase, if there exists at least one landmark in each suborbit then a breadth-first search from that landmark over $gens_K$ can be used to produce the full suborbit. Because of this, the landmark ratio is typically selected so that it is large enough to guarantee with high probability that at least one landmark will be seen in each suborbit. Any missing suborbit can still be detected and added during the orbit counting phase.

In addition, in the orbit counting phase, if the elements in $s^c : s \in S_i$ are not landmarks, then a breadth-first search from the non-landmark elements over $gens_K$ must be performed until a landmark is found.

3.3 Predicted Time

Given the use of landmark discovery using greasing for vector-matrix multiplication, the runtime of a single condensation can be predicted. Those times are presented in this section.

3.3.1 Predicted Vector-Matrix Multiplication Time

We found that by using greasing with a band size of 8, a reasonable speedup using only a small amount of memory was obtained:

Mem Space	$782 \times \lceil 782/8 \rceil \times 2^8$b	20MB
CPU Time	$\lceil 782/64 \rceil \times \lceil 782/8 \rceil$ops	4.2×10^{-7}s
Mem Time	$\lceil 782/8 \rceil \times \lceil 782/8 \rceil$B	4.5×10^{-6}s

By using greasing and about 40 megabytes of extra space per machine, a single vector-matrix multiplication can be sped up by a factor of eight as compared to using the standard method. Experimentally, we obtained a greasing time closer to 2.0×10^{-5}s, and it is this time we will use for the predicted time of algorithm.

3.3.2 Predicted Sequential Time using Landmarks

Breadth-first search requires time proportional to the number of elements in the search and the branching factor. This implies a total of $|O| \times |gens_G|$ vector-matrix multiplications for the orbit enumeration phase, and $|O| \times |gens_K|$ vector-matrix multiplications for the suborbit partitioning phase. The use of landmarks increases the number of vector-matrix multiplications in the orbit counting phase from $|O|$ to $L \times |O|$. Given the time for vector-matrix multiplication by greasing and the fact that both G and K have two generators, this implies a total time of 25.8 CPU days on a single machine.

3.3.3 Predicted Parallel Time with Linear Speedup

The predicted running time for each phase of the computation is shown below. These times assume computation on a cluster of 56 nodes with nearly linear parallel speedup. These times compare closely to the experimental times for the parallel disk-based algorithm found in Section 4.5. The experimental time for orbit enumeration is notably larger than the predicted time. The predicted time does not take into account the time for duplicate detection in large disk-based breadth-first searches. More details on this can be found in [22].

Phase	Time
Orbit Enumeration	2 hours
Suborbit Partitioning	2 hours
Orbit Counting	7 hours
Total	11 hours

4. DISTRIBUTED DISK-BASED ALGORITHM

We present our distributed disk-based algorithms for the computation of orbit counting matrices for $G = Fi_{23}$. In the language of Section 2, V has dimension 782 over $GF(2)$. Choosing a suitable subgroup $H = O_8^+(2)\colon 2 < G$ there indeed is a $v \in V$ such that $\mathrm{Stab}_G(v) = H$. Letting $O = v^G$ this leads to $|O| = 11,739,046,176$. Moreover, we choose $K = S_6(2)\colon 2 < H$, which leads to $k = 6,486$ suborbits in O. These choices are justified in Section 5.

The cluster we are using for this computation has 56 nodes, each with 4 gigabytes of local memory and 10 gigabytes of local disk. Due to the orbit size, $|O| = 11,739,046,176$, O is too large to store in memory across the cluster and must use distributed disk. Here a disk-based solution to this problem is presented in terms of the three phases of condensation discussed in Section 3.1.

4.1 Terminology

Before examining the algorithm itself, some common terminology must be considered.

Owner of a Vector Given the compressed signature w_c of a vector w, a subset of the bits of that compressed signature are used to determine a unique node, $\mathcal{N}(w_c)$ in the computation that is responsible for storing that compressed signature. For a set of compressed signatures, O, $\mathcal{P}_i(O) = \{w_c \in O : \mathcal{N}(w_c) = i\}$ is the set of compressed signatures belonging to node i in O.

Owner of a suborbit Given a set of compressed signatures W representing the values in a suborbit, a canonical ordering for those compressed signatures is chosen. The smallest $\mathcal{C}(W) \in W$ is the canonical member of W. The owner of W, $\mathcal{N}(W) = \mathcal{N}(\mathcal{C}(W))$ is the node owning the canonical member of W. It is responsible for storing the information for that orbit. For a set of suborbits S, $\mathcal{P}_i(S) = \{s \in S : \mathcal{N}(s) = i\}$ is the set of suborbits belonging to node i in S.

4.2 Orbit Enumeration

We follow the general approach of [22] for orbit enumeration to produce $O = v^G$. This approach uses a distributed hash array while performing a breadth first search. Any empty hash slot indicates a value has not been seen previously. If the hash slot is not empty, either the value is a duplicate or there has been a hash collision. In this case, the value is dropped from the frontier and placed in a collision queue. Values in the collision queue are later checked to determine where hash collisions occurred by using external sort and a streaming scan through the values. This allows disk-based duplicate detection to take place using streaming access only. After hash collisions have been detected, these values are added to the frontier.

Given the use of compressed values, the amount of space required by the entire search is only 6.4 gigabytes per node. This fits easily on distributed disk. The hash used for this computation required only 2 bits per entry, or 53 megabytes per node. This allows for a hash that fits easily into distributed memory. This hash is organized in such a way that for all values $w \in O : w$ hashes to node i, $\mathcal{N}(w_c) = i$, or every value hashes to the node that owns it. This allows the messages that check the distributed hash to double as the messages that store values in the orbit on the node that owns them.

While the use compressed values reduces the amount of space required enough to fit the search on disk, it also increases the number of vector-matrix multiplications required. Those values discovered whose hash slots are empty are added immediately to the frontier and are never stored in their compressed form in the collision queue. However, those elements that have hash collisions must later be added back into the frontier. In order to do this, their full values must be computed. Fortunately, it is only a small percentage, around 22.5%, of the values for which this must be done. In addition, many of the calculations can be batched so that value-generator pairs are not computed multiple times for values that have similar paths.

4.3 Suborbit Partitioning

Suborbit partitioning to form the K-suborbits can be viewed in terms of the actions of the nodes owning the data in question on that data. This data includes the initial orbit, O, the suborbits in S, and the landmarks in those suborbits, $\mathcal{L}$.

4.3.1 Use of Landmarks

Since O is distributed and disk-based, it is not possible to quickly remove values from it as they are encountered. For this reason, instead of removing values from O, a list of previously encountered values is maintained. Rather than storing all values, only landmark values are recorded in order to allow this list to fit in distributed RAM across the cluster. A landmark ratio of $L = 7$ was selected, requiring 360 megabytes of landmark storage of compressed signatures, by the node that owns them. It would have been possible to store all the values in their compressed form in memory, using 2.5 gigabytes of memory. However, because the cluster is shared, using a smaller percentage of the total memory per node was preferable.

Landmarks had to be selected carefully. First, the portion of the compressed signature that determined the owner of a vector had to be distinct from the portion deciding whether or not that vector was a a landmark. Without this, all landmarks would be owned by a subset of the nodes of the computation.

Also, it was known prior to the computation that v was a fixed point under K, and therefore would be in a suborbit by itself. Our landmark selection was made in such a way that the v was always considered a landmark. Other missing suborbits would be discovered during the orbit counting phase, although this did not occur in our computation.

4.3.2 Processing the Orbit Values

Each node n processes a piece of O corresponding to $\mathcal{P}_n(O)$. This is done in a manner similar to Algorithm 2. Now, however, rather than removing values in O, a list of known landmarks, $\mathcal{L}_n \in O$, owned by node n is stored. Algorithm 4 shows how this is done.

Algorithm 4: *ppartitionOrbit*

```
Input: $\mathcal{P}_n(O), gens_K$
for each compressed signature $w_c \in \mathcal{P}_n(O)$ do
    if $isLandmark(w_c)$ and $w_c \notin \mathcal{L}_n$ then
        $w = buildValue(v, path(w_c))$;
        $s = discoverOrbit(w, gens_K)$;
        $s = compress(s)$;
        $s = stripNonlandmarks(s)$;
        $s = sortCanonical(s)$;
        $sendSuborbit(s, path(w_c))$;
```

Each local landmark from $\mathcal{P}_n(O)$ is compared with a list of landmarks sent by other nodes, $\mathcal{L}_n$. For each local landmark that has not been encountered previously, the suborbit for that landmark needs to be built. Before this can be done, the compressed value needs to be expanded into its full value by following the path associated with it from v in $gens_G$. Once the suborbit has been computed locally, the values in it are compressed, non-landmarks are stripped, and it is sorted in canonical order. This places the canonical element for s, $\mathcal{C}(s)$, first. The suborbit along with the path to reach that suborbit are then sent to the node $\mathcal{N}(s)$.

4.3.3 Processing the Suborbits

When a node $\mathcal{N}(s)$ receives a suborbit s it owns, it must process that suborbit. This is shown in Algorithm 5.

Algorithm 5: *ppartitionSuborbit*

```
Input: $s, path$
if $s \notin \mathcal{P}_n(S)$ then
    Get an original number $id \in \{1, \ldots, k\}$;
    Store $\{path, \mathcal{C}(s), id\}$ in $\mathcal{P}_n(S)$;
    $s = sortOwner(s)$;
    $sendLandmarks(s)$;
```

The suborbit is first checked to see if it is a duplicate by scanning through $\mathcal{P}_n(S)$, the suborbits in S owned by node $n = \mathcal{N}(s)$, and looking at the canonical elements. If it has not been seen, it is processed. It first gets a unique $id \in \{1, \ldots, k\}$. This is obtained by requesting an id from a unique master node, who keeps track of what ids have been seen before. After this, the information for the suborbit is stored locally in $\mathcal{P}_n(S)$ and the values in the suborbit are sorted according to their owners. For each node n, the values $\mathcal{P}_n(s)$ owned by the node n, along with the suborbit id, are then sent to the owner n.

4.3.4 Processing the Landmarks

When a node n receives a set of landmarks it owns, it must store those landmarks in $\mathcal{L}_n$. This is shown in Algorithm 6.

Algorithm 6: *ppartitionLandmarks*

```
Input: $l, id$
$l = sortCanonical(l)$;
Add $l$ to $\mathcal{L}_n(id)$;
```

Each node n stores $\mathcal{L}_n$, an array of size k. Each entry i in that array, $\mathcal{L}_n(i)$, corresponds to the set of known landmarks in suborbit i owned by node n. These values are sorted canonically to allow for quick lookup.

4.3.5 Nearly Linear Speedup for Parallel Implementation

The parallel algorithm provides a nearly linear speedup compared to the sequential algorithm. Since the only time the same suborbit is generated multiple times is when multiple nodes are producing the same suborbit simultaneously, this means at worst a slowdown factor of $n = 56$. However, since k is relatively large in comparison to n, on average, each suborbit is typically generated only once. Since the suborbits are processed in parallel, this provides nearly linear speedup.

Each suborbit is computed only once on average. This implies each landmark $l \in S_i$ is sent only twice, once to reach its suborbit's owner, $\mathcal{N}(S_i)$, and a second time to reach its owner, $\mathcal{N}(l)$. The bandwidth of the network is sufficient so that the bottleneck of the computation is still the CPU-intensive vector-matrix multiplication and not the sending of the data. The latency of the network is not a factor because landmarks are sent out in large groups to the nodes that own them. At most $k \times 56 = 363,261$ messages will be passed across the network. Because of the algorithm design, duplication checks for individual landmarks are local to the nodes that own those landmark and do not incur a communication penalty.

Finally, the time for sort, binary search, and hash lookup in the suborbit's breadth first search are relatively small when compared to the vector-matrix multiplication time. This implies a run time dominated by the time to perform vector-matrix multiplications in the breadth-first search, just as in the sequential algorithm.

4.4 Orbit Counting

Orbit counting can also be viewed in terms of the actions of the nodes owning the data in question on that data. In this case, the data is the set of suborbits, S, and the landmarks of the neighboring values. For some condensation element c, each node n holds a piece of the resulting orbit counting matrix $C(c)$ corresponding to the set of rows $\{i : \mathcal{N}(S_i) = n\}$. The resulting data is combined once the computation finishes.

4.4.1 Initialization

In order to speed up the process of determining which suborbits a large set of landmarks are in, the way in which $\mathcal{L}_i$ for each node i is stored is changed. On the node n, rather than storing $\mathcal{L}_n = \{\mathcal{L}_n(1), \ldots, \mathcal{L}_n(k)\}$, $\mathcal{L}_n$ is stored as a single block in which each entry corresponds to the compressed signature of a landmark along with the id of the suborbit of that landmark. These values are then sorted based on the canonical ordering of their compressed signatures.

4.4.2 Processing the Suborbits

Given some condensation element c, the subset $\mathcal{P}_n(S)$, corresponding to the set of suborbits owned by node n, are processed according to Algorithm 7.

Algorithm 7: *pcondenseSuborbits*

```
Input: $\mathcal{P}_n(S), c$
let $C(c) \leftarrow$ a distributed $k \times k$ all-zeros matrix for
$s_o \in \mathcal{P}_n(S)$ do
    $w = buildValue(v, path(s_o))$;
    $i = id(s_o)$;
    $s = discoverOrbit(w, gens_K)$;
    $s = s^c$;
    $s = findClosestLandmarks(s, gens_K)$;
    $s = sortOwner(s)$;
    $sendLandmarks(s)$;
    for $m \in nodes$ do
        $C_{i,\cdot}(c) = C_{i,\cdot}(c) + receiveCounts(m)$;
```

Each node processes each suborbit it owns. It rebuilds that suborbit and then applies the condensation element c to each element in the suborbit. After that, it must perform a breadth-first search for each element in s in $gens_K$ to find the closest landmark (stored as a compressed value). The compressed values are sorted according to the nodes that own them and then are sent to those nodes. The result returned is the number of landmarks in each suborbit owned by that node. These results are added up to form $C_{i,\cdot}(c)$.

4.4.3 Processing the Landmarks

When a node n receives a set of landmarks it owns, it must compute how many of those landmarks are in each suborbit. This is shown in Algorithm 8.

First the result res is initialized to an all zeros vector of size k. The landmarks received are sorted in canonical order. By sorting, a single pass through $\mathcal{L}_n$ is sufficient to find the ids of all landmarks in s. When an $i = id(l \in s)$ is encountered, res_i is incremented.

Algorithm 8: *pcondenseLandmarks*

```
Input: $s$
$res$ = an all zeros row vector of size $k$;
$sortCanonical(s)$;
$l$ = start of $\mathcal{L}_n$;
for $s \in S$ do
    while $\mathcal{L}_n(l) < s$ do increment $l$;
    $i = id(\mathcal{L}_n(l))$;
    $res_i = res_i + 1$;
$sendCounts(res)$;
```

4.4.4 Nearly Linear Speedup for Parallel Implementation

Here also there is a nearly linear speedup for orbit counting when compared to the sequential algorithm. Each suborbit is processed only once, as in the sequential algorithm. Each value in a suborbit, upon projection by c, is processed only once to find the closest landmark in the generators of $gens_K$. This implies that the number of vector-matrix multiplications in the parallel algorithm is exactly the same as the number in the serial algorithm.

Each suborbit is computed only once. Here, though, a value for each compressed signature in the suborbit must be passed across the network. Let the amount of data sent across the network in the suborbit partitioning phase be D, orbit counting requires the sending of $L \times D/2$ data. However, the same increase in the number of vector-matrix multiplications must also be performed, meaning ratio of time spent in the network and in performing vector-matrix multiplications is identical. As before, the bottleneck lies with the vector-matrix multiplications.

Finally, $\mathcal{L}$ must be scanned k times to locate the ids for landmarks. While this does add some time to the computation, it is still not significant when compared to the vector-matrix multiplications.

4.5 Experimental Results

We used a cluster of 56 computers in the computation of the orbit counting matrices. Each computer was an Intel dual-processor Xeon running at 3.20 GHz, running Red Hat Linux 3.2.3 under Rocks. The time for each portion of the algorithm, as well as the total storage requirements, is presented here.

Phase	Time	Memory	Disk
Orbit Enumeration	18 hours	500 MB	8 GB
Suborbit Partitioning	4 hours	800 MB	300 MB
Orbit Counting	20 hours	800 MB	300 MB
Total	42 hours	800 MB	8 GB

Given the predicted time on a single machine with a landmark ratio $L = 7$ of 25.8 CPU days, this would imply 0.46 CPU days on 56 computers, as shown in Section 3.3.3. This is within a factor of four of the predicted time. The factor would be only two if not for the naive predicted time for the orbit enumeration phase from Section 3.3.2.

5. THE BRAUER TREE

We show how the computed orbit counting matrices are used to determine the missing labels of the vertices of the Brauer tree of the principal 17-block of the sporadic simple Fischer group Fi_{23}. We were particularly interested in this example for the following reason:

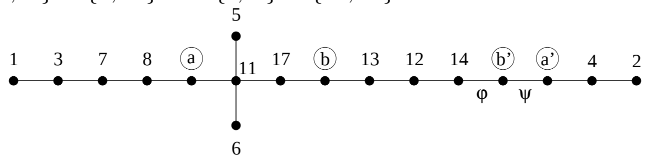
Table 1: The principal 17-block of Fi_{23}.

i	χ	$\chi(1)$	$\chi(e)$	1_H^G
1	1	1	1	1
2	3	3588	1	1
3	6	30888	8	3
4	13	789360	2	1
5	15	837200	0	0
6	16	837200	0	0
7	24	5533110	27	4
8	60	97976320	58	3
9	62	153014400	44	1
10	63	153014400	44	1
11	76	264536064	35	0
12	77	264536064	140	0
13	79	287721720	147	1
14	92	476702577	185	0
15	94	504627200	167	1
16	95	504627200	167	1
17	98	559458900	128	0

5.1 The Modular Atlas project

The aim of the Modular Atlas project [11, 24, 25], which was initiated two decades ago and is still running, is to determine the p-modular decomposition matrices and the Brauer character tables of the groups listed in the Atlas [2]. As far as the blocks of cyclic defect are concerned, which encompass the case where p divides the order of the group under consideration but p^2 does not, the decomposition problem can be rephrased as the problem of determining the associated Brauer trees. For the sporadic groups and their automorphism groups, a comprehensive collection of Brauer trees has been computed in [10], but quite a few questions still remain open.

In particular, for Fi_{23} the shape of the Brauer tree of its principal 17-block, and the labeling of its vertices, up to four possible cases, have been determined in [10]. Table 1 provides the numbering of the irreducible ordinary characters in the principal 17-block, where their Atlas [2] numbers and their degrees are given in the second and third columns, respectively. The associated Brauer tree is as follows, where $\{a, a'\} = \{9, 10\}$ and $\{b, b'\} = \{15, 16\}$.



The task now is to determine which of these four cases actually occurs. We apply an analysis similar to that used in [5, 18].

5.2 The Orbit

Standard generators $gens_G = \{g_1, g_2\}$ of $G = Fi_{23}$, in the sense of [26], are given in [24]. The standard generators are given both in terms of the smallest faithful permutation representation on 31,671 points, and in terms of the smallest faithful matrix representation in characteristic 2, i.e. in dimension 782 over $GF(2)$.

We now look for a subgroup $H < G$ such that the simple modules S_φ and S_ψ, affording the Brauer characters φ and ψ as indicated above, are modular constituents of the permutation module 1_H^G. We choose a subgroup

$$H = O_8^+(2)\colon 2 < S_8(2) < G,$$

where both $H = O_8^+(2)\colon 2 < S_8(2)$ and $S_8(2) < G$ are maximal subgroups. Using the facilities to compute with class functions and to determine fusions of conjugacy classes available in GAP [8], we find the multiplicities of the ordinary irreducible characters in the permutation character 1_H^G as given in the fifth column of Table 1. These imply that S_φ and S_ψ are modular constituents of 1_H^G.

To apply a direct condensation technique, the G-set underlying 1_H^G must be realized as a set of vectors in a suitable linear representation of G. Actually, it turns out that in the representation space V of dimension 782 over $GF(2)$ there is a (unique) vector v such that $\mathrm{Stab}_G(v) = H$. This yields a manageable orbit $O = v^G \subseteq V$.

5.3 The Condensation Subgroup

In general, given an FG-module M with Brauer character φ, which is extended arbitrarily to a class function $\tilde{\varphi}$ on G, we have $\dim_F(Me) = \langle \varphi|_K, 1_K\rangle = \langle \tilde{\varphi}, 1_K^G\rangle_G$, where $\langle \cdot, \cdot \rangle_G$ denotes the scalar product for class functions. As each Brauer character can be written as a linear combination of ordinary characters, these scalar products can be determined from ordinary characters. If the block under consideration is described by a Brauer tree, these linear combinations can directly be read off from the tree.

Here, we choose the condensation subgroup

$$K = S_6(2)\colon 2 < H < G,$$

a maximal subgroup of H. Using the facilities to compute with class functions available in GAP [8], we determine the dimensions $\chi(e) = \langle \chi, 1_K^G\rangle_G$ of the condensed modules of the ordinary irreducible characters in the principal block, as given in the fourth column of Table 1. In particular, these dimensions imply that $\varphi(e) = 124$ and $\psi(e) = 43$. Hence the condensed modules $S_\varphi e$ and $S_\psi e$ are constituents of $(1_H^G)e$, having the indicated dimensions. Similarly, we find $k = \dim((1_H^G)e) = \langle 1_H^G, 1_K^G\rangle_G = 6,486$. Thus the condensed module has manageable dimension to be analyzed explicitly using MeatAxe [21] techniques.

5.4 Determining the Brauer Tree

We specify $c = g_2$, the second standard generator of G, and compute the traces $\mathrm{Tr}_{S_\varphi e}(ece)$ and $\mathrm{Tr}_{S_\psi e}(ece)$ for the possible cases $[a, b] \in \{[9, 15], [10, 15], [9, 16], [10, 16]\}$, using the formula

$$\mathrm{Tr}_{Me}(ece) = |K|^{-1} \cdot \sum_{g \in K} \mathrm{Tr}_M(cg),$$

where the right hand side can be determined from the Brauer character of M by p-modular reduction, provided we know the cardinalities of the intersections of the coset Kc with the various conjugacy classes of G.

To find those, we have to run through all $|K| = 2,903,040$ elements of Kc, and to determine to which conjugacy class of G it belongs. Conjugacy testing is done using the permutation representation on 31,671 points, the facilities dealing with permutation groups available in GAP [8], and some specially tailored programs using cycle structures and class multiplication coefficients. This needs about 100 CPU hours on a single machine to be completed. (Parallelizing this as well would of course be possible, but we have not pursued this further.)

We do not reproduce the full class distribution here, but just note the following: The element c to be condensed must be chosen such that the four cases can be distinguished by looking at the above mentioned traces. Since the cases yield Brauer characters which only differ on elements of order divisible by 13, this essentially boils down to a condition on the intersections of Kc with the conjugacy classes containing such elements. Here is the result for $c = g_2$:

$13A$	43044	$13B$	43526
$26A$	111166	$26B$	111782
$39A$	67678	$39B$	66560

Now this yields the following, where the entries are understood to be in $GF(17)$:

$[a, b]$	$[9, 15]$	$[10, 15]$	$[9, 16]$	$[10, 16]$
$\mathrm{Tr}_{S_\varphi e}(ece)$	10	3	14	7
$\mathrm{Tr}_{S_\psi e}(ece)$	7	14	7	14

5.5 Conclusion

We use the subalgebra of the Hecke algebra generated by $\{eg_1e, eg_2e, eg_1g_2e\}$, where $gens_G = \{g_1, g_2\}$. By the technique described in Section 4, we determine the associated orbit counting matrices, which essentially describe their action on the condensed permutation module $(1_H^G)e$.

Using MeatAxe [21] techniques, in particular those to determine submodule structures [13], it turns out that this subalgebra already is sufficiently large to pick the constituents $S_\varphi e$ and $S_\psi e$ of $(1_H^G)e$. By inspection, it is found that $\mathrm{Tr}_{S_\varphi e}(ece) = 7$ and $\mathrm{Tr}_{S_\psi e}(ece) = 14$ for $c = g_2$, implying $a = 10$ and $b = 16$, and we are done.

6. REFERENCES

[1] Wieb Bosma, John Cannon, and Catherine Playoust. The MAGMA algebra system i: The user language. *J. Symbolic Comput.*, 24:235–265, 1997.

[2] J.H. Conway, R.T. Curtis, S.P. Norton, R.A. Parker, and R.A. Wilson. *Atlas of finite groups.* Clarendon Press, Oxford, 1985.

[3] G. Cooperman. STAR/MPI: Binding a parallel library to interactive symbolic algebra systems. In *Proc. of International Symposium on Symbolic and Algebraic Computation (ISSAC '95)*, volume 249 of *Lecture Notes in Control and Information Sciences*, pages 126–132. ACM Press, 1995. software at URL: `http://www.ccs.neu.edu/home/gene/software.html\#starmpi` and `http://www.ccs.neu.edu/home/gene/pargcl.html`.

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

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

[6] 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, 2003.

[7] 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.

[8] The GAP Group. *GAP — Groups, Algorithms, and Programming, Version 4.4*, 2006. `http://www.gap-system.org`.

[9] J. Green. *Polynomial Representations of GL_n.* Lecture Notes in Mathematics 830. Springer-Verlag, 1980.

[10] G. Hiss and K. Lux. *Brauer Trees of Sporadic Groups.* Oxford Univ. Press, Oxford, 1989.

[11] C. Jansen, K. Lux, R. Parker, and R. Wilson. *An Atlas of Brauer Characters*, volume 11 of *London Math. Soc. Monographs, (N. S.).* Clarendon Press, Oxford, 1995.

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

[13] K. Lux, J. Müller, and M. Ringe. Peakword condensation and submodule lattices: An application of the MeatAxe. *J. Symb. Comp.*, 17:529–544, 1994.

[14] J. Müller. Computational representation theory: remarks on condensation. Lecture Notes, 2003. `http://www.math.rwth-aachen.de/~Juergen.Mueller/`.

[15] J. Müller. On endomorphism rings and character tables. Habilitationsschrift, RWTH Aachen, 2003.

[16] J. Müller. On the action of the sporadic simple baby monster group on the cosets of $2^{1+22}.Co_2$. Preprint, 2006.

[17] J. Müller, M. Neunhöffer, and F. Noeske. GAP-4 package orb, 2006. `http://www.math.rwth-aachen.de/~Max.Neunhoeffer/Computer/Software/Gap/orb.html`.

[18] J. Müller, M. Neunhöffer, F. Röhr, and R. Wilson. Completing the Brauer trees for the sporadic simple Lyons group. *LMS J. Comput. Math.*, 5:18–33, 2002.

[19] J. Müller, M. Neunhöffer, and R. Wilson. Enumerating big orbits and an application: B acting on the cosets of Fi_{23}. Preprint, to appear in J. Algebra, 2006. `http://www.math.rwth-aachen.de/~Juergen.Mueller/`.

[20] R. Parker and R. Wilson. Unpublished, 1995.

[21] M. Ringe. *The C-MeatAxe, Version 2.4, Manual.* RWTH Aachen, 2000.

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

[23] J. Thackray. *Modular representations of some finite groups.* PhD thesis, Univ. of Cambridge, 1981.

[24] R. Wilson. Atlas of finite group representations. `http://brauer.maths.qmul.ac.uk/Atlas/v3/`.

[25] R. Wilson. The modular atlas homepage. `http://www.math.rwth-aachen.de/homes/MOC/`.

[26] R. Wilson. Standard generators for sporadic simple groups. *J. Algebra*, 184:505–515, 1996.