New Sequential and Parallel Algorithms for Generating High Dimension Hecke Algebras using the Condensation Technique

Gene Cooperman[1] and Michael Tselman[1]
gene@ccs.neu.edu and misha@ccs.neu.edu
College of Computer Science
Northeastern University
Boston, MA 02115 / U.S.A.

Abstract

Condensation is an important technique for finding irreducible representations of a group and other information. It maps a representation to a representation of a smaller algebra, the Hecke algebra or condensation algebra. A new algorithm is described, which allows one to take very large permutation representations and build a Hecke algebra of larger dimension than before. The technique allows one to make a space-time tradeoff with constant space-time product. Further, the technique parallelizes in a distributed memory environment (and hence also for shared memory), yielding a speedup linear in the number of processors. This has been used to obtain a condensed representation of dimension 5,693, for the finite simple group J_4 acting on a space of dimension 173,067,389. The computation was repeated using from 4 (lightly loaded) workstations up to 14 (heavily loaded) workstations with timings ranging from 6-1/2 days down to 2-1/2 days. Both homogeneous (SPARC) and heterogeneous (SPARC/Alpha) architectures were used.

1 Introduction

Condensation is an important technique first invented by Richard Parker and Thackray at Cambridge University to aid in finding new irreducible representations of a group and to analyze existing ones. The technique is described in Thackray's Ph.D. thesis. It has been applied in a number of settings [5, 9, 11, 12]. The condensation technique is used to "condense" a reducible representation of high dimension into a representation of a *Hecke algebra* or *condensation algebra* of smaller dimension. One then looks for desired information in the representation of the Hecke algebra, whose dimension is computationally more tractable, and then "lifts" the information back to the original representation.

A common application of condensation is in finding new irreducible matrix representations of a group. One begins with an action of a group G on a vector space V, and then forms the permutation representation on the orbit of a vector of V. This representation is always reducible for dimension greater than one. For small enough d, where d is the dimension of the permutation representation, there are other techniques, such as Parker's meataxe [7, 10] for finding irreducible representations. For large d, analysis of the Hecke algebra remains, as described in section 2.3, one of the few viable techniques for finding new irreducible representations. This produces a representation of a new, smaller subalgebra, which can be analyzed by the meataxe or other techniques.

Until now, the dimension of representations of the Hecke algebra was typically limited to at most 1,000 due to large storage requirements proportional to d. Reducing the dimension to that extent typically results in much loss of information. For example, there is a Hecke algebra acting on a module of dimension 1, but it is the trivial algebra. This paper describes how to store only a fraction of the d points through a space-time tradeoff with constant space-time product. Further, the technique parallelizes in a distributed memory environment with linear speedup in the number of processors. Using the new technique, one is now limited primarily by one's ability to compute with the matrices of a high dimensional Hecke algebra. In the case of J_4, we have produced a representation of a Hecke algebra of dimension 5693 from an initial permutation representation of dimension 173,067,389.

For G acting on the module V, we choose $f \in V$ such that G acts faithfully on f^G. As will be seen in section 2.2, an arbitrary subgroup $K < G$ yields a Hecke algebra acting on a module of dimension k, for k the number of K-orbits in f^G. It will further be seen that the representation of the image of $g \in G$ in the Hecke algebra is the matrix $(c_{ij}/|O_j|)$ for which O_i and O_j are the i^{th} and j^{th} K-orbits and c_{ij} is the number of points of O_i that are mapped to O_j under g. Intuitively, the image of g in the Hecke algebra achieves a smaller representation by smearing together the points in each K-orbit.

In computing c_{ij}, one must compute the image $x^g \in f^G$ for each $x \in f^G$, and then identify which K-orbit contains x^g. The problem is that one cannot afford to store all of the points of f^G. The solution is a space-time tradeoff:

> *Key Idea:* One chooses m and a pseudo-random "hash-like" function, h: $f^G \mapsto [0, \ldots, M]$ for $M \gg m$. (i.e. the function h should satisfy the simple uniform hashing assumption [4, pp. 224–227].) One then stores only those elements, x, of f^G for which $h(x) = 0 \bmod m$.

Since one need only store approximately $1/m$ of the points of f^G, the storage becomes manageable.

[1]Partially supported by NSF Grant CCR-9509783.

To appear,
Proceedings of the International Symposium on Symbolic and Algebraic Computation-96 (ISSAC-96),
August, 1996,
Zurich, Switzerland

One typically stores the points in a hash table, but the choice of hash table or other data structure is independent of the pseudo-random function, $h()$. Along with storing each point x, one must also store a pointer to the orbit in which it was found to lie. With this information, one can then identify the K-orbit containing an arbitrary image point, y^g. If $h(y^g) = 0 \bmod m$, then y^g will be found in the table. Otherwise, one does a search of other points in $(y^g)^K$. If $h(x) = 0 \bmod m$ for any point $x \in (y^g)^K$, the table is used to identify the K-orbit of x, which is also the K-orbit of y^g. On average, one makes such an identification after examining m points. Thus, we will typically save a factor of m in space, at the cost of an additional factor of m in time.

The algorithm we developed can be easily mapped on to the master-slave parallel environment. We implemented a master-slave version of the algorithm that can run on a network of workstations (SPARC, Alpha). We used the GCL/MPI software package which integrates a GCL version of COMMON LISP and an MPI (Message Passing Interface) communication library, and provides a top level master-slave framework. In a series of experiments the implementation demonstrated excellent scalability of the algorithm with the increase in the number of processors, while maintaining nearly optimal load balance.

As a result, for the J_4 group we computed a representation of a Hecke algebra of dimension 5693 in times ranging from 6-1/2 to 2-1/2 days using from 4 to 14 processors. The condensation was carried out using the subgroup $K = 2^{11} : 23$ for generating K-orbits.

2 The Condensation Technique

Although the condensation technique is described in some earlier papers, it is reviewed here to make the description self-contained.

2.1 Informal description

In typical applications of condensation, one begins with an algebra acting on a (possibly reducible) module of very high dimension. Then one finds an image algebra under a certain "natural", non-invertible map to be described later. Further, the image algebra (known as the *Hecke algebra*) acts on a module of much smaller dimension, and the map induces a map between the two modules. Under this induced map, each element of the image module is a linear combination of elements of the original module. For this reason, one can think of the "natural" map as acting as a type of "averaging" operator.

Intuitively, one has *condensed* the information of the larger representation into information of a smaller Hecke algebra. The smaller algebra is much more tractable from the viewpoint of computation, and one hopes to recover new information about the larger representation (such as finding irreducible submodules) from an analysis of the Hecke algebra.

2.2 Formal description

Consider the following permutation representation. Let the A be a group algebra FG for some field F and suppose that the group G acts by permutation on the set Ω. Let $V = F^\Omega$, the set of all functions from Ω into F. Then V can be made into a right A-module by defining the action of G on V and then extending by linearity to all of A. We define the action of $x \in G$ on V, according to the rule $(fx)(i) = f(i^{x^{-1}})$ for $x \in G, f \in V, i \in \Omega$. For each $i \in \Omega$, if one sets u_i to be the function which is 1 on i and 0 elsewhere, then $\{u_i : i \in \Omega\}$ is a basis for V and $u_i x = u_{i^{x^{-1}}}$.

One next defines $e \in A$, which will depend on a chosen subgroup $H \leq G$. Assume that the characteristic of F is coprime to $|H|$. Let $O_1, \ldots, O_\ell$ be the orbits of H acting on Ω. Let $e = \frac{1}{|H|}\sum_{h \in H} h$. Then e is an idempotent of A and $eh = e$ for all $h \in H$. So, $Ve \leq V$ is a right $eFGe$-subspace of dimension ℓ. Ve has a natural basis, $v_i = \sum_{j \in O_i} u_j$.

The element e is an "averaging" operator. If $v \in V$, then $ve(j)$ is the average value of $v(x)$ over all $x \in j^H$. Thus ve is constant on all orbits of H for all $v \in V$. Further, ege depends only on HgH and not on the choice of $g' \in HgH$.

Given the permutation representation G acting on the $|\Omega|$-dimensional module V, we want to construct the representation for the algebra eAe acting on the smaller, ℓ-dimensional module Ve. We produce the new representation with respect to the basis $\{v_i\}$ for Ve above. Let us compute it for an element ege, such that $g \in G$. One can easily check that

$$v_i ege = \sum c_{ij} v_j, \text{ where } c_{ij} = |\{k \in O_i\colon\ k^g \in O_j\}|/|O_j|.$$

As one expects, c_{ij} depends only on HgH and not on the choice of $g' \in HgH$.

The goal (see below) is to find eAe-submodules of Ve that correspond to interesting A-submodules of V. Note that Ve always has the invariant subspace spanned by the single vector $v_1 + \cdots + v_\ell$. So Ve is never irreducible when $\ell > 1$.

2.3 Application

An important goal is to study the submodule structure of Ve under $eFGe$. One then finds a $w \in Ve$ such that $w^{eFGe} < Ve$. One hopes that $\overline{w}^{FG}$ is a proper subspace of V, and so one may have discovered a new, irreducible representation of G.

Note that if $\overline{U}$ is an invariant A-submodule, then $U = \overline{U}e$ is an invariant eAe-submodule. This follows since $\overline{U}A \subseteq \overline{U}$ implies $\overline{U}e \subseteq \overline{U}$ and so $(\overline{U}e)(eAe) = (\overline{U}e)Ae \subseteq \overline{U}Ae \subseteq \overline{U}e$.

2.4 Uncondense

An important requirement for the above is to find a pre-image of an element of Ve. Suppose we have a submodule of Ve generated by one element, and we wish to produce a corresponding submodule of the pre-image, V. Let $W = weAe \leq Ve$ be the submodule of interest. Since $v_i e = v_i$, $\overline{w} = \sum_i \sum_{j \in O_i} w_i u_j$ is a pre-image of $w = \sum_i w_i v_i$.

So, $\overline{W} = \overline{w}A \leq V$ is a pre-image of W under right multiplication by e.

2.5 Limits of applicability

Assume that G has a $n \times n$ matrix representation. Then the limits of such a computation on a typical workstation using standard linear algebra are about $|\Omega| \leq 10^7$ and $n \leq 10^4$ (for $n \times n$ matrices over $GF(2)$). It is easy to see how these limits come about. For $n = 10^4$, one requires about 12 megabytes per matrix and several hours for one matrix multiplication on a fast workstation. The limit of $|\Omega| \leq 10^7$ comes from the 40 Megabytes needed to store that many 4-byte words. The new technique allows us to raise $|\Omega|$ to 10^9 or 10^{10}. One is then faced with the difficult, linear algebra problem of extracting the smaller dimensional invariant subspace and corresponding representation from such a large initial vector space.

2.6 Background

The discussion of section 2.2 can be put in a more general context. Let A be a finite dimensional algebra. Let V be a right A-module. Then $A \hookrightarrow Hom_F(V)$.

Let $e \in A$, $e^2 = e$. This implies that $eAe \leq A$ is a subalgebra of A. Let Ve be a right eAe-module and note that $eAe \hookrightarrow Hom_F(Ve)$. The subalgebra, eAe, is called the *condensation algebra* of A, or the *Hecke algebra*.

One can then search for irreducible representations of eAe in Ve and hope to lift them to irreducible representations of A. There is a natural mapping (which is not one-one in general) from the lattice of A-submodules of V to the lattice of eAe-submodules of Ve. To see this, note that if W is an A-submodule of V, then $WeAe = (We)eAe \leq (Ve)eAe$ is a right eAe-submodule of Ve.

Still another application is to find decomposition matrices, as discussed in [9].

3 Algorithm and Data Structure

As discussed in section 2.2, one version of the condensation technique reduces to computing the orbit adjacency matrix associated with the condensation method. First we describe the notation and problem. Let a group G be generated by S. Let $K \leq G$ be generated by S'. Assume a permutation action and a faithful orbit, $\mathcal{O}$, of G. When we refer to K-orbits, we will have in mind only the K-orbits contained in $\mathcal{O}$. For purposes of computing the image of an element, $g \in G$, in the Hecke algebra, one must first construct a matrix such that the (i, j) entry is the number of points, $|\{x\colon\ x \in \mathcal{O}_i, x^g \in \mathcal{O}_j\}|$, where $\mathcal{O}_i$ and $\mathcal{O}_j$ are the i-th K-orbit and j-th K-orbit, respectively.

3.1 Data structure

One chooses a modulus, m, and creates a hash table with $c|\mathcal{O}|/m$ slots to hold elements of $\mathcal{O}$, where $c > 1$ is a small hash factor (typically 1.5 or 2). One need not know $|\mathcal{O}|$ in advance, since one can employ dynamic hash arrays, in which the size of the hash array is doubled and all elements are re-hashed whenever a certain load factor is reached.

Next a pseudo-random function $h()\colon\ \mathcal{O} \mapsto [0, \ldots, M]$ for $M \gg m$, is chosen satisfying the simple uniform hashing assumption [4, pp. 224–227]. The pseudo-random function may or may not be the same as the hash function for the hash table. Each element $v \in \mathcal{O}$ is stored in the hash table if and only if $h(v) = 0 \bmod m$. Along with v is stored an integer uniquely identifying in which K-orbit v lies. The modulus $m \geq 1$ is chosen as small as possible, so that the hash array of length $c|\mathcal{O}|/m$ fits into semiconductor memory (RAM).

Given an arbitrary element, $v \in \mathcal{O}$, one then wishes to discover the orbit number corresponding to v. The "key idea" of the introduction (finding a $x \in v^K$ for which $h(x) = 0 \bmod m$) must be modified if there is an orbit v^K such that for all $x \in v^K$, $h(x) \neq 0 \bmod m$.

In order to guarantee that each orbit has at least one representative point to be stored, one stores all canonical points, where "canonical" is defined as follows. Let m be a power of 2. For a given orbit, O, let $i = \max_{0 \leq j \leq \log_2 m}\{j\colon\ \exists x \in O, h(x) = 0 \bmod 2^j\}$. A point $x \in O$ is *canonical* in O if and only if $h(x) = 0 \bmod 2^i$.

Thus, one stores only the canonical elements of each orbit.

3.2 Algorithm

The algorithm consists of an appropriate integration of the following three routines:

`Find_orbit(`v`,`K`)`: Return the set of elements of the K-orbit, v^K. To compute the K-orbit this routine uses breadth-first search algorithm, starting with v and applying elements of the generating set S' until all elements are found.

`Hash_orbit(`v^K`)`: Return a set of canonical elements (as defined in section 3.1) of the orbit v^K.

`Find_image_orbits(`v^K`,`g`)`: $g \in G$. Return a list of $|v^K|$ elements, where for each element $x \in v^K$ the list contains a canonical element y in the orbit $(x^g)^K$. Essentially, this function returns a map of a given orbit into the other orbits under the group element g.

The algorithm needs to find all K-orbits and for each K-orbit find where a group element maps it to. This motivates the definition of two major tasks. The first (type I) task is, given a K-orbit representative, to compute the K-orbit and save its canonical elements in the hash table. Such an orbit is then tagged as "known". Functions `Find_orbit` and `Hash_orbit` are used to accomplish this task. The second (type II) task is, given a K-orbit representative and a group element g, to find where g maps each element of the K-orbit. This is implemented through a call to `Find_orbit` followed by a call to `Find_image_orbits`.

One maintains queues of type I tasks and of type II tasks. If a type I task produces a previously unknown orbit, a new type II task corresponding to that orbit is placed on the queue of tasks of type II. The information from a type II task is used to then hash the resulting image vectors and update the condensation matrix for those image vectors that hash to a previously known orbit. If one of the resulting vectors is part of a previously unknown orbit, then a new type I task will be placed on a queue of type I tasks.

If we are executing tasks of type II and the queue of type I tasks grows beyond some threshold and so requires too much space, one switches to executing tasks of type I (expanding orbits). When the queue of type I tasks is empty, one switches to executing tasks of type II.

This algorithm may fail to discover all orbits if $\langle g \cup S' \rangle \neq G$. This can be repaired by using other generators, $g' \in S$, to find additional orbits.

3.3 Space requirements

An analysis of the space requirements is particularly important for a calculation of this size, since that often determines the feasibility of the calculation. In order to describe concrete bounds, we assume G is represented as a matrix group. We assume that there is a vector v_0 for which ${v_0}^G$ is a regular orbit of G.

Space requirements for the hash table are bounded above by

$$c|v|(\frac{|\Omega|}{m} + Cn) \text{ bytes}$$

where $c > 1$ is the hash factor, $|v|$ is the number of bytes needed to store a vector, $\Omega = v_0^G$ and so $|\Omega|$ is the size of the regular orbit, m is the chosen modulus, n is the number of K-orbits, and C is a constant that is approximately 1.45.

The bound is surprisingly tight, as shown by the calculation in section 3.4.

Space requirements for the condensation matrix are estimated as $n \times n \times \lceil log_2 l \rceil$ bits, where l is the largest matrix entry. Obviously, $l \leq |K|$, where K is the subgroup used to generate K-orbits. One also temporarily needs approximately $|v||v^K|$ bytes storage for the current K-orbit, v^K. Additional storage is required by the two queues described in the section 3.2 and is proportional to the size of several K-orbits.

If the field F of the group algebra is large or infinite (for example, the rationals), then a careful encoding of the condensation matrix can save further space. The factor $\lceil log_2 l \rceil$, where l is the largest matrix entry, can be reduced by the following heuristic. If "most" matrix entries are known to be less than $l' < l$, then one uses a matrix data structure that has only $\lceil \log_2 l' \rceil$ bits per entry. When a matrix entry reaches l' or more during the course of the algorithm, one stores exactly l' for that matrix entry, and this serves as an indicator to look for the "correct" matrix entry in a second data structure. The second data structure can be a hash vector with the matrix indices as keys, or one can choose any other compact data structure for storing sparse matrices.

3.4 Average number of canonical points per orbit

The space bounds are dominated by the necessary size of the hash array. Accordingly, it is worthwhile to find more precise estimates of the number of canonical points per orbit, where the canonical points are defined as in section 3.1.

Let $m = 2^\ell$ be the chosen modulus. We now find the expected number of entries that will be stored in the hash table for an orbit, $\mathcal{O}$, of length k, according the the algorithm, above.

Assume that the pseudo-random function h is a true random function from elements of $\mathcal{O}$ into the integers. Let $Q_j = |\{x \in \mathcal{O}\colon\ h(x) = 0 \bmod 2^j\}|$. In particular, define $Q_0 = k$.

Note that $E(Q_\ell \mid Q_\ell > 0)\Pr(Q_\ell > 0) = E(Q_\ell \mid Q_\ell > 0)\Pr(Q_\ell > 0) + E(Q_\ell \mid Q_\ell = 0)\Pr(Q_\ell = 0) = E(Q_\ell)$.

Note that the event $Q_i > 0 \wedge Q_{i+1} = 0$ is the same as the difference of the event $Q_{i+1} = 0$ and the event $Q_i = 0$. Then the expected number of entries is

$$\begin{aligned}
&E(Q_\ell \mid Q_\ell > 0)\mathrm{Pr}(Q_\ell > 0) \\
&\quad + \sum_{i=0}^{\ell-1} E(Q_i \mid Q_i > 0 \wedge Q_{i+1} = 0)\mathrm{Pr}(\mathrm{Q_i} > 0 \wedge \mathrm{Q_{i+1}} = 0) \\
&= E(Q_\ell) + \sum_{i=0}^{\ell-1} E(Q_i \mid Q_{i+1} = 0)\mathrm{Pr}(\mathrm{Q_{i+1}} = 0) \\
&\quad - E(Q_i \mid Q_i = 0)\mathrm{Pr}(\mathrm{Q_i} = 0) \\
&= E(Q_\ell) + \sum_{i=0}^{\ell-1} E(Q_i \mid Q_{i+1} = 0)\mathrm{Pr}(\mathrm{Q_{i+1}} = 0) \\
&= k/2^\ell + \sum_{i=0}^{\ell-1} \frac{k/2^{i+1}}{1 - 1/2^{i+1}}(1 - 1/2^{i+1})^k \\
&= k/2^\ell + \sum_{i=0}^{\ell-1} \frac{1}{1 - 1/2^{i+1}} k/2^{i+1}[(1 - 1/2^{i+1})^{2^{i+1}}]^{k/2^{i+1}} \\
&\leq k/2^\ell + \sum_{i=0}^{\ell-1} 2k/2^{i+1}\exp(-k/2^{i+1}) \\
&\to k/2^\ell + (2/\ln 2)\int_{i=0}^{\ell-1} (\ln 2)k/2^{i+1}\exp(-k/2^{i+1})\,di \\
&= k/2^\ell + [(2/\ln 2)\exp(-k/2^{i+1})]_{i=0}^{i=\ell-1} \\
&\leq k/2^\ell + (2/\ln 2)\exp(-k/2^\ell) \\
&= k/m + (2/\ln 2)\exp(-k/m) \\
&\leq k/m + 2/\ln 2
\end{aligned}$$

The number $2/\ln 2$ is about 2.885. This analysis took 2 as an upper bound for $1/(1 - 1/2^{i+1}) = 1 + 1/(2^{i+1} - 1)$. A finer analysis would analyze the summation separately for the terms 1 and $1/(2^{i+1} - 1)$. The analysis is omitted for reasons of space, but it would show an upper bound closer to $k/m + 1/\ln 2 \approx k/m + 1.45$, and a a numerical calculation shows this to be a reasonably tight upper bound for a wide range of k.

3.5 Time requirements

The time is dominated by `Find_image_orbits`. A single call to `Find_image_orbits` requires approximately $|v^K|mt$ time where t is the time for a single application of a group element in G to an element $v \in \Omega$. However, if we take advantage of the common special case where G has a matrix representation and v is a vector, then the time can be considerably shortened.

The optimization described here has not yet been included in the software. We hope to have a future version with faster timings based on this idea. One pre-computes some large number of distinct matrices in the subgroup K. When `Find_image_orbits`(V^K, g) invokes a search for a canonical element $y \in (x^g)^K$ for $x \in v^K$, one needs to find r so that $h((x^g)^r) = 0 \bmod m$ for some $r \in K$. One can choose the pseudo-random function, h, in its action on $w \in v_0^G$ so that $h(w)$ is uniquely determined by a fixed subset of the coordinates of w. Thus, given the vector x^g and the matrix r, one need compute only the fixed subset of the coordinates of $(x^g)^r$ and not all of $(x^g)^r$. In this way, the time is reduced to $|v^K|(t + mt')$ where t' is the time to compute the fixed subset of coordinates for the result of a matrix-vector multiplication.

4 Parallelization

The algorithm in section 3.2 can be easily parallelized using a master-slave architecture. This architecture is especially well suited for search and enumeration problems, where the master coordinates the search and the slaves asynchronously execute heavy duty computations in the search space. A master-slave architecture requires splitting the problem into a set of tasks that can be executed in parallel. The master is responsible for generating tasks and collecting results from slaves. Slaves simply execute tasks issued by the master.

The algorithm maps onto this architecture well as it has already been defined in terms of tasks (of type I and II). This task decomposition dictates the following data layout. The master maintains the main hash table as well as the two queues mentioned above. Each slave contains the data for computing in a group and hashing. This includes the generating sets S and S', the vector f, the modulus m, the hashing functions, etc.

The results of a type I task are used to update the hash array and add a new orbit and orbit representative to the queue of orbits. The results of a type II task are added to the queue of vectors for potential representatives of new orbits. If the queue grows too large, the master switches to generating type I tasks. If the queue becomes empty, the master switches to generating type II tasks. As events proceed asynchronously the master has to check if the potentially new orbit representative still does not belong to any known orbit, before the master issues a task of type I.

5 Implementation

The algorithm has been implemented using GCL, a dialect of COMMON LISP. The matrix-vector multiplication routine was written in C. The parallel implementation uses the GCL/MPI software package described in [3]. This package can be obtained from
`ftp://ftp.ccs.neu.edu/pub/people/gene/starmpi/`.
The implementation consists of approximately 2250 lines of LISP code out of which 1300 lines are previously developed matrix/vector processing routines. The matrix/vector routines were implemented by storing for each matrix a lookup table of all linear combinations of each set of four adjacent rows over $GF(2)$. This achieves a computation time for a matrix-vector multiplication in $GL(112, 2)$ of 37 microseconds on a SPARC-5. Such a lookup table was first used by Arlazarov et. al. [2] (see also [1]), and was popularized by Parker [10] in his software for the meataxe algorithm.

The implementation has been tested on clusters of SPARC-5 and Alpha 3000/300 workstations both in homogeneous and in mixed architecture settings. Varying the number of slaves on partial runs of the program showed almost linear scalability of the algorithm. Experiments also showed an excellent load factor on all slaves. Each slave process was run on a separate workstation, and was using $\approx$90% CPU time on an unloaded workstation and $\approx$ 30% on a workstation shared with 2 other CPU intensive processes.

6 Saving the result

The condensation matrix produced by the algorithm may take a large amount of memory. In the J_4 case a condensation matrix was computed over the integers. So, the space reserved for the condensation matrix was $5693 \times 5693 \times 16$ bits, or about 64 Mbytes of RAM. This brings up a problem of storing several condensation matrices on the disk or transferring them on the net. We developed a simple compression scheme for our particular matrices, but the scheme can be applicable to other cases too.

The compression scheme is based on the empirical observation that most entries are less than 31 in value (requiring only 5 bits). We use three files to store the matrix entries in a specified order (such as row major order). The first file contains the low 4 bits, the second contains the fifth bit, and the third contains the full value for those matrix entries requiring more than 5 bits. If a matrix entry is less than 31, it is stored directly using 5 bits. If a matrix entry is 31 or larger, the first two files are used to store a value of 31, signifying that the actual matrix entry "overflowed" the 5 bits. By scanning the first two files, one can find the index in the matrix of all overflow entries. The third file is used to store all overflow entries as 16-bit quantities, in the same order in which the entries were found in the first two files. This encoding brings the disk storage requirement for J_4 down from approximately 64 Mbytes per matrix to approximately $(5/16)64 = 20$ Mbytes per matrix.

7 Results for J_4

Condensation was carried out on a permutation representation of J_4 of dimension 173,067,389, using the subgroup $K = 2^{11} : 23$ for generating K-orbits. The resulting Hecke algebra representation acted on a module of dimension 5693. In all experiments the same 25 MHz SPARC-10 was used as a master because it had 256 Megabytes of memory (RAM), most of which was required for the data structures.

The first complete run of the program was performed on a cluster of 25 MHz SPARC-2 workstations. Eight workstations were used for running 8 slave processes. These workstations were mostly taken from the student laboratory early in the quarter, and were lightly loaded by other users. It took approximately 6-1/2 days to complete the computation.

In the next experiment we used four 75 MHz SPARC-5's and one 25 MHz SPARC-10 as slaves with the same master. All machines were lightly loaded by other users. The computation was completed in 3 days (76 hours). As the processing power used is roughly equivalent to 13 SPARC-2 workstations we see an approximately linear speedup over the previous experiment.

We also ran the program in the mixed architecture environment using the same SPARC-10 workstation for the master, but using 14 Alpha processors for slaves. We used an older Alpha 3000/300x model with 175 MHz processors. Unlike the SPARC workstations in our previous runs, the Alpha workstations were heavily loaded due to two other long term CPU-intensive jobs. So our program was not able to use more than 1/3 of the resources on most machines, with occasional use of 1/2 of the resources on some. On this basis, the computation took 2-1/2 days.

We were also surprised that while the Alpha processors have a clock rate more than twice as fast as the SPARC-5 processors, benchmarking the essential procedures on both architectures showed a smaller advantage for the Alpha workstations (less than a factor of 1.5). We attribute this to the nature of our application, which favored the SPARC processor. First, we do not use floating point or 64-bit integer arithmetic, where the Alphas would have a significant advantage. Second, our computation is highly non-local in memory and the cache misses hurt the 175 MHz Alpha CPU much more than they hurt the 75 MHz SPARC CPU.

We believe that a mixed architecture (SPARC master and Alpha slaves) setup may also suffer in performance because of the byte conversion and extra malloc operations done by MPI in situations when a different byte ordering is used by different architectures.

It is important to note that while the algorithm was loading all slaves up to the maximum, the load on a master (running on a 25 MHz SPARC-10) was within 30-60% in both SPARC experiments and around 80-90% in the Alpha experiment. This suggests that the SPARC-2 master processor was nearing saturation, and a faster SPARC-5 or Alpha would be required in order to add still more slaves with the concurrent linear speedup.

8 Acknowledgements

The authors wish to thank Stephen Linton who first brought the condensation method to their attention with his note on the subject [8] (later expanded on by Larry Finkelstein).

The authors also thank Larry Finkelstein, Stephen Linton and Klaus Lux for generous discussions on the condensation technique. They also thank Klaus Lux and Jürgen Müller for their detailed comments on this manuscript.

The authors wish to thank Robert Wilson for first suggesting that the condensation technique be applied to J_4, and for providing the necessary group and subgroup generators. He was also very generous in describing his own computational experiences with the condensation technique.

Finally, the authors wish to thank Clement Lam and Concordia University for providing guest accounts and information with which to do initial testing on their DEC Alpha workstations. The information was especially useful in porting from the SPARC to the DEC Alpha architecture.

References

[1] Aho, A., Hopcroft, J., and Ullman, J. (1974). "The Design and Analysis of Computer Algorithms", Addison-Wesley, p. 245.

[2] Arlazarov, V.L., Dinic, E.A., Kronrod, M.A. and Faradzev, I.A., (1970). "On Economical Construction of the Transitive Closure of a Directed Graph", Dokl. Nauk SSSR **194**, pp. 487–488. English translation in Soviet Math. Dokl. **11**:5, pp. 1209–1210.

[3] G. Cooperman, "STAR/MPI: Binding a Parallel Library to Interactive Symbolic Algebra Systems", *Proc. of International Symposium on Symbolic and Algebraic Computation (ISSAC '95)*, ACM Press, pp. 126–132.

[4] T. Cormen, C. Leiserson and R. Rivest, *Introduction to Algorithms*, pp. 219–243, MIT Press and McGraw-Hill, 1990.

[5] H.W.Gollan and G.O. Michler "Construction of a 45694-dimensional simple module of Lyons' sporadic group over $GF(2)$", *Linear Algebra and Its Applications*, preprint.

[6] W. Gropp, E. Lusk and A. Skjellum, *Using MPI*, MIT Press, 1994.

[7] D.F. Holt and S. Rees, "Testing Modules for Irreducibility", J. Austral. Math. Soc. Ser. A **57**, pp. 1–16, 1994,

[8] S. Linton (revised by L. Finkelstein), untitled manuscript describing Parker's condensation technique.

[9] K. Lux, J. Müller, M. Ringe, "Peakword Condensation and Submodule Lattices: An Application of the Meat-Axe", *J. Symb. Comp.* **17** (1994), pp. 529–544.

[10] R. Parker, "The computer calculation of modular characters. (The Meat-Axe)", in M. Atkinson (ed.), *Computational Group Theory*, Academic Press, London, 267-74, 1984.

[11] A.J.E. Ryba, "Computer Condensation of Modular Representations", *J. Symb. Comp.* **9** (1990), pp. 591–600.

[12] M. Wiegelmann, *Fixpunktkondensation von Tensorproduktmoduln*, Diploma thesis, Dept. of Math., RWTH Aachen, 1994.

[13] Message Passing Interface Forum (author),"MPI: A Message-Passing Interface Standard", International Journal of Supercomputing Applications **8**, Number 3/4, 1994.