Journal of Symbolic Computation **33**, *pp. 563–585*

Scalable Parallel Coset Enumeration: Bulk Definition and the Memory Wall

Gene Cooperman* and Victor Grinberg*
College of Computer Science
Northeastern University
Boston, MA 02115
gene@ccs.neu.edu
victor@ccs.neu.edu

Abstract

Coset enumeration, like Gröbner bases, is a notoriously difficult algorithm to parallelize. We demonstrate a successful shared memory parallelization achieving a 7 times speedup on an Origin 2000 CC-NUMA computer using 16 CPUs. We take as a testbed, an enumeration of Lyons's group (8,835,156 cosets). This provides comparability with previous efforts in the literature (Cooperman and Havas [1997], Havas and Ramsay [2000]) for which the best previous speedup was a factor of 4. The new parallelization depends on two new heuristics, *clouds* and *shallow scan*. Clouds is an example of *bulk definition* of cosets, which forms the key to our more efficient parallelization. The parallelization is implemented using TOP-C. By taking advantage of TOP-C's option to compile for either shared or distributed memory, we also demonstrate the first efficient parallelization of a coset enumeration program using distributed memory.

Our faster results expose for the first time in the context of coset enumeration the "memory wall", i.e. the latency barrier of the RAM. We verify this memory wall by showing on an Origin 2000 that a memory-bound parallel program with 64 CPUs doing nothing but randomly accessing RAM achieves at best a speedup of only 3.75 over the single CPU version. Further, it is demonstrated that *even sequential versions* of coset enumerations programs are memory latency-bound in today's technology. The lessons of bulk definition and of the memory wall carry over to related algorithms such as Gröbner bases, Knuth-Bendix, and other symbolic algebra algorithms with intermediate swell.

*Partially supported by NSF Grant CCR-9732330.

1. Introduction

Two new heuristics are demonstrated for parallelizing large coset enumerations using shared memory, *clouds* and *parallel shallow scan*. (Shallow scan was originally called "prescan" in Cooperman and Grinberg [2001].) In practice, it is applied only when the table is mostly filled, and so we do not observe this behavior. We take Lyons's group as a challenge problem, with the challenge being to extract a large speedup beyond the factor of 4 that was previously achieved (Havas and Ramsay [2000]). We are able to demonstrate a factor of 7 speedup in these new results.

We further show that the bottleneck at that stage becomes the memory latency of the RAM. Surprisingly, we also demonstrate that even a sequential coset enumeration program may experience this memory wall when running primarily outside of cache. This is colloquially known as the "memory wall" (Wulf and McKee [1995])).

As a consequence of our investigation, we also find that the sequential version of the shallow scan heuristic has important benefits even for sequential enumerations, when applied to so-called "easy" enumerations. This sequential version reduces the computation time for the enumeration of Lyons's group by a factor of 3 from 4 hours down to 1.4 hours on a Pentium II Xeon 450 MHz Linux computer.

We also test our parallel algorithms on a variety of other groups. While not achieving the same scalability as with the Lyons group, we demonstrate the stability and robustness of the algorithms. The software is freely available at `ftp://ftp.ccs.neu.edu/pub/people/gene/cosets/`.

This paper contains several advances over what was peviously reported in Cooperman and Grinberg [2001]:

1. the discovery that RAM memory latency was the cause of the previously reported memory contention (Section 6);
2. parallel implementation of mixed Felsch-style and HLT-style scanning to cope with hard enumerations (Section 3.5);
3. a detailed description of the shallow scan heuristic;
4. testing on other groups (Section 5.4) besides Lyons's group (Sections 5.1 to 5.3); and
5. a demonstration as proof-of-principle of the ability to efficiently parallelize coset enumeration using distributed memory, although currently at the cost of replicating a copy of the coset table at each node (Section 5.5).

The first item deserves special note, since it is likely to affect other parallel (and possibly sequential) symbolic algebra programs in the future. We encountered the memory wall because coset enumeration is so memory-intensive, and because we found an algorithm to efficiently use more than 16 CPUs for parallel coset enumeration. The current trend is for clock speed to double every 18 months, while DRAM latency has not significantly improved in the last decade. Hence, the memory wall is becoming twice as severe every 18 months.

Our motivation for shared memory parallelization is two-fold. First, we anticipate more 4-way and 8-way SMP processors becoming available at the departmental level as their prices continue to fall. Second, we anticipate applying these techniques to problems using tens of gigabytes of RAM in the future. Such large problems could be executed sequentially using today's sequential coset enumerators, but only in a week or more. Few computation centers in a production environment would agree to dedicate tens of gigabytes of RAM over more than a week.

Our methodology is to introduce a heuristic for making many coset definitions at once. We call this a *bulk definition heuristic*. We then verify in tests for some sequential coset enumeration (perhaps on some smaller group) that use of the heuristic does not significantly increase the required CPU time. Under these conditions, we imagine perturbing an original coset enumeration in two stages.

1. Modify the sequential program so as to define cosets *only* using the bulk definition heuristic. (We have verified that this does not add significantly to the computation time.)
2. After each bulk definition, process the definitions in parallel by sending the next p new definitions to a slave for processing. The slaves identify in parallel for the master where the table must be updated, and the master then blocks all the slaves while updating the table.

Section 2 gives an overview of the difficulties of parallelization of coset enumeration. Section 3 describes the heuristics used to parallelize the software. Section 4 presents a diamond lemma, which justifies an approach by which a partial "test run" on a sequential processor allows us to predict the parallel performance. Section 5 presents our experimental results. Section 6 describes why today's memory architectures limit further parallel scaling of the performance. The shallow scan heuristic (Section 3.4) need not terminate, or it may return too large a table. Section 7 describes how to quickly remedy this in parallel.

1.1. Notation and terminology

There are several excellent descriptions of the Todd-Coxeter algorithm for coset enumeration and its variations (Cannon et al. [1973], Felsch [1961], Neubüser [1982], Todd and Coxeter [1936]). Here we content ourselves with setting some of the terminology needed for this article. Informally, coset enumeration is the process of finding all cosets of a subgroup H in G, given only defining equations (called *relations* or *relators*) in generators $S \subseteq G$, and a set of generators of H, expressed as words in the generators of G. (A relator is an element of the free group on S, while a relation is an equation in the generators S. Hence a relator R corresponds to the relation $R = 1$.) In a Felsch strategy, one usually includes all distinct cyclical permutations of the relator words in the set of relators, and also all distinct inverses of those words. Usually, S is chosen to be closed under inverses. The set of relations in the chosen generators of G determines a *presentation*.

In typical usage, no other representation of G is known, and the presentation is used to find a more concrete representation. A *coset table* is a table in which each row represents a coset c, and each column represents a generator g. A row-column entry (c, g) (which we call a *link*) is either blank (unknown) or filled in with a row number representing the coset cg. A blank entry is also called a *hole*. The *density* of a coset table is the number of non-blank entries of the table divided by the total number of entries of the table. (The number of entries is the product of the number of rows of the table and the number of generators.) Given an initially incomplete *coset table* one *defines a new coset* as a new row in the table, and adds a new link to this coset at a blank entry of a previous coset. One then *scans the link* through a relator loop. The term link is non-standard, and is used here for its connotation with the underlying Schreier coset graph.

A coset table defines a multiplication ϕ: $C \times S \to C \cup \{\text{blank}\}$ where C is the set of rows of the coset table, "blank" represents a blank entry, and S is the generating set. For W the set of words in S, extend the multiplication to ϕ: $C \times W \to C$, by defining $\phi(c, g_1g_2\cdots g_k) = \phi(\phi(c, g_1g_2\cdots g_{k-1}), g_k)$ and defining $\phi(c, g_1)$ as before. Then $\phi(c, g_1g_2\cdots g_k)$ is well-defined if and only if $\phi(c, g_1g_2\cdots gi_i)$ is not a blank for any $i < k$. For $e \in W$ the empty word, define $\phi(c, e) = c$.

A *relator loop* is defined by a table row c_i and a relator $g_1g_2\cdots g_k$. One speaks of *tracing the relator loop*, *tracing the relator from an initial coset* c_i, or *tracing the relator through the link* (c_i, g_1). This refers to computing $\phi(c_i, g_1g_2\cdots g_j)$ for a maximal $j \le k$ such that $\phi(c_i, g_1g_2\cdots g_j)$ is well-defined. Hence, one stops tracing a relator loop upon either reaching a blank in the table or reaching the end of the relator.

Two situations in tracing a relator loop $g_1g_2\cdots g_k$ at c_i are of special importance.

1. If one arrives at a blank in the table, then let $a \le 0$ be the largest index for which $c_a = \phi(c_i, g_1g_2\cdots g_a)$ is not blank. If $\phi(c_i, g_1)$ is blank, take $a = 0$. Then trace the relator loop $g_k^{-1}g_{k-1}^{-1}\cdots g_1^{-1}$ at c_i and let $b \ge 0$ be the smallest index for which $c_b = \phi(c_i, g_k^{-1}g_{k-1}^{-1}\cdots g_{b+1}^{-1})$ is not blank. Take $b = k$ if $\phi(c_i, g_k^{-1})$ is blank. If $b - a = 1$, then $g_{a+1} = g_b$ and one concludes that $c_ag_b = c_b$. One enters c_a in the table entry (c_b, g_b^{-1}), and one speaks of making a *deduction*. In a Felsch strategy, this deduction is then entered in a *deduction queue* and when the deduction is later processed from the queue, all relator loops through the link (c_a, g_b) are traced. (Recall that in a Felsch strategy the set of relators is closed under inverses, and so this is equivalent to tracing all relator loops through (c_b, g_b^{-1}).)

2. If one reaches the end of the relator and the final table row $c_f = \phi(c_i, g_1g_2\cdots g_k)$ is distinct from the initial coset c_i, then one identifies the two table rows

c_i and c_f as representing the same coset, and one speaks of a *coincidence*. Most enumeration strategies immediately process the coincidence by removing one table row (for specificity, we remove c_f), and recursively processing coincidences between $\phi(c_i, g)$ and $\phi(c_f, g)$ for all cases where those two entries are distinct and both non-blank. Furthermore, in a Felsch strategy, if $\phi(c_f, g)$ is non-blank and $\phi(c_i, g)$ is blank for some $g \in S$, then one adds to the *deduction queue* a deduction $c_i g = \phi(c_f, g)$.

1.2. Challenge problem: parallel coset enumeration of Lyons's group

Amrhein et al. [1996] proposed an efficient parallel Gröbner basis algorithm as a challenge problem. To the extent that coset enumeration is a related algorithm, it should be considered a related challenge problem. For specificity, we take greater parallelization of Lyons's group to be the challenge problem, since this group has previously been studied in the literature (Cooperman and Havas [1997], Havas and Ramsay [2000]).

Let g be an arbitrary generator of a group G. A coset enumeration has two phases:

1. definition of a new coset, c_2, as $c_2 = gc_1$; and
2. modification of a table of cosets by:
   (a) processing all newly discovered coincidences of cosets in the table; and
   (b) adding newly discovered deductions, $c_2' = g'c_1'$, for existing cosets c_1' and c_2' in the table, and G-generator g'.

1.3. Previous work in literature

Coset enumeration and its efficient implementation has a long history as an important problem. (See Bandler [1956], Beetham [1984], Beetham and Campbell [1976], Campbell [1975], Cannon et al. [1973], Coxeter and Moser [1957], Felsch [1961], G.Havas and Lian [1991], Leech [1984], Neubüser [1982].) Efficient parallelization of coset enumeration is a difficult problem and has a shorter history (Akl et al. [1991], Cooperman and Havas [1997], Havas and Ramsay [2000]). Akl et al. [1991] presented the first parallel enumeration using a different algorithm than the ones considered in this paper.

Cooperman and Havas [1997] used search parallelism to demonstrate a medium grain parallelism on shared memory computers using TOP-C (Cooperman [1996, 1999], G.Cooperman [2001]) and POSIX threads. In that paper, they presented as future work, the application of their methodology to a parallel enumeration of Lyons's group. That enumeration was carried out soon after the paper was written, achieving a speedup of about a factor of two using four processors. The presentation for Lyons's group used by Cooperman and Havas was justified in Havas and Sims [1999].

The next improvement was by Havas and Ramsay [2000], who used a similar methodology to parallelize Lyons's group at the level of 16 processors, achieving a speedup of approximately 4. That paper sutdies the degradation due to delays in applying updates (see Problem 1 of Section 2) versus the synchronization overhead.

2. Difficulties of search parallelism in coset enumeration

In order to provide enough work for each slave, the master is forced to make many definitions at once, along with associated *links* to previous cosets in the table. One then passes off these new links to the next slave requesting work. The slaves scan the links in parallel and return a list of relator loops for the master to rescan. The master updates the coset table. In our own experiments, we send out 100 or 1,000 links per slave, and since we scale up to 31 slaves, this implies 3,100 or even 31,000 outstanding links, before the master receives the results and can update the table. In all implementations of coset enumeration to date (including ours), the master blocks all slaves when the the master updates the coset table.

This type of parallel algorithm is called *search parallelism* by Amrhein et al. [1996] in the context of Gröbner bases. An excellent discussion of search parallelism and of the literature for parallel Gröbner basis implementations can be found in that reference.

Typical parallelizations suffer from at least three sources of degradation. The discussion assumes a single large table is kept in shared memory and accessed by all processes. We present the degradations in the order of the most serious one first.

Problem 1 The delay in applying updates to the table after a definition has been made rises proportionally with the number of slave processes. If there are n slaves and each slave works on 100 links per task, then up to $100\,n$ links may still be unprocessed when the master makes the next definition. While the evidence is ambiguous, we believe that this effect is already present with as few as 4 to 8 processors.

Problem 2 In most parallelizations to date, the master blocks all slaves when updating the table, and this portion of the computation is essentially sequential, with only the master process being active. This is done so that the slaves do not read erroneous information during a time that the table may be in an inconsistent state. In our situation (and in Havas and Ramsay [2000]), this sequential time spent by the master was measured at less than 5%, but this effect can become important when we scale from 16 to 32 processors. Our own heuristics for bulk definition probably introduce an additional overhead of 5% due to sequential management by the master. This makes it an important effect already for 16 processors.

Problem 3 As the number of processors continues to increase, memory contention limits the scalability. One often sees many slave processors working on the same region of the table, causing cache delays. We observe this last situation when we scale from 32 to 64 processors.

An obvious solution to the first problem is to lower the task size of a slave. However, this solution soon leads to its own problems due to synchronization overhead. The paper of Havas and Ramsay [2000] contains an excellent study of this phenomenon. This paper proposes a solution of a very different nature. As discussed later, by using bulk definition, we can continue to use relatively large task sizes.

The second problem of sequential computation during updates of the table, is one for which an easy solution is available. The table can be partitioned into small "pages", and the master can update a single page, while the slaves continue to access all other pages. A new version of TOP-C is being developed with support for pages, and this solution will be applied to the problem of master updates when the new version of TOP-C is ready. Our experiments indicate that this effect becomes important only after a speedup of a factor of 10 or 20 has been achieved.

The third problem of memory contention has not previously been observed for coset enumeration, because previous enumerations had achieved a speedup of only 4 times. Our measurements show that the memory-bound programs randomly accessing main RAM (not cache) achieve a speedup of only 3.6 on an Origin 2000. This subtle problem is analyzed in detail in Section 6. It is quite serious. On an Origin 2000 with 64 processors, we observe essentially no difference in elapsed time between 32 and 64 processors, as the increased memory contention outweighs any advantage to using more processors.

3. Method of parallelization

The original enumerator was written solely with the idea of exploring heuristics for sequential coset enumeration. The program was later parallelized with TOP-C (Cooperman [1996], G.Cooperman [2001], Task Oriented Parallel C/C++). TOP-C allows us to make relatively few modifications to the sequential program (resulting in about 250 lines of additional application code in a sequential application with 3,200 lines of code). TOP-C also allows us to link the same application code with our choice of a TOP-C library for sequential, shared memory parallel, or distributed memory parallel computation, in order to test in a variety of environments. The TOP-C methodology has also been applied to interactive languages. In particular, we point out that the TOP-C parallel routines can be called from GAP (Groups, Algorithms and Programming), where the routines have been implemented as the ParGAP package (Cooperman [2001]).

Our computation reserves 1 processor as a manager or "master" for the computation. Our computation is organized so that at any given moment, either the master is active or else all of the other "slave" threads are active, but the master and a slave are never active at the same time. We test its parallelism by scaling up from 1 to 32 processors.

We observe very little delay due to the use of thread synchronization primitives. To estimate this, we varied our input task size from 1,000 deductions to be scanned by a slave thread to 10,000 deductions. Even though this implies 10 times as many master-slave thread synchronizations, we estimate the increase in elapsed time of the computation to be less than 5 minutes. This holds as we vary the number of processors from 1 to 32.

3.1. Base software for coset enumeration

Our base code is a sequential coset enumerator of approximately 5,000 lines of C++ code. It is similar to ACE, except that it is smaller and has fewer options than ACE. This smaller program is useful for making simple modifications to explore new heuristics. We compare our time with that of ACE for Lyons's group and found that when ACE was run in either its default mode (Fel:0, standard Felsch), or in Felsch mode (Fel:1, Felsch with gap-filling for gaps of length 1), the CPU times on Linux running a 450 MHz Pentium Xeon were within 10%. Both ACE and our own enumerator completed in approximately 4 hours. Among the heuristics added to this sequential enumerator, are two that are important for parallel computation. We call them *clouds* and *shallow scan*.

3.2. Scanning order

A Todd-Coxeter enumerator needs to perform three basic tasks, or steps. Those are coset creation, relator scanning and recursive coincidence processing. Of the three tasks the scanning is typically by far more time consuming. For example, in a Felsch enumerator every relator needs to be scanned in every possible position through every link entered into the table, unless the link is destroyed by a collapse before it's scanned. For the presentation that we have used for our benchmarking the total length of relators scanned for each new link is, on the average, about 4500.

Since scanning is by far the most time-consuming part of the enumerator, it's the part that is most necessary to parallelize. Hence, this enumerator parallelizes only scanning, and not coset creation, entering a deduction in the coset table, or coincidence processing.

We follow the traditional strategy for parallelization of scanning laid down in Cooperman and Havas [1997]. The master thread takes up to `INPUT_SIZE` deductions from the deduction queue and passes them to a slave thread. Given a deduction $cg = c'$, the slave threads trace all relators through the link (c, g). Each slave scans in parallel. If a trace yields a deduction or coincidence, then the slave enters a triple (c, g, r) in the slave thread's private output queue, where r is the

r-th relator. If for a given pair (c_1, g_1) there are more than `EXPANSION_FACTOR` triples (c_1, g_1, r) for varying r, then all triples of the form (c_1, g_1, r) are replaced by a single triple $(c_1, g_1, -1)$ on the output queue. When a slave finishes scanning, the slave passes its output queue back to the master. The master then re-traces only the given relators (c_1, g_1, r) from the output queue. Given a triple $(c_1, g_1, -1)$, the master traces all relators through the link (c_1, g_1). Our default values are `INPUT_SIZE=1000` (a task size of 1,000) and `EXPANSION_FACTOR=50`.

The effect of this algorithm is as if the slave threads examined the deduction queue of the master, and told the master to trace only those relators r through links (c, g) such that (c, g, r) would lead to a coincidence or deduction. The master can ignore all other tracing, and so the master spends most of its time waiting for new results from the slaves.

In addition, two other parameters control entry from *sequential mode* into *parallel mode* and exit from parallel mode back into sequential mode. The master will process the deduction queue alone and in sequential mode unless there are at least `START_PAR_SIZE` deductions on the deduction queue. In the latter case, the computation switches to parallel mode with the master thread passing tasks of up to `INPUT_SIZE` to each slave, as described above. If in parallel mode, fewer than `STOP_PAR_SIZE` deductions remain on the master's deduction queue, then the computation switches back into sequential mode. Parallel mode is implemented by a single call to `TOPC_master_slave(GenTaskInput, DoTask, CheckTaskResult, Update)` where the callback functions `GenTaskInput`, `DoTask`, `CheckTaskResult` and `Update` implement the parallel algorithm described above. The default values of `START_PAR_SIZE` and `STOP_PAR_SIZE` were 500 and 150, respectively.

Finally, there is the issue of granularity. The master thread is not allowed to modify the coset table unless all slave threads are inactive. We follow the default TOP-C strategy dictated by the reader-writer problem with writer preference. However, we also experimented with finer task granularity by adding a parameter `LOCK_RELEASE_FREQUENCY`. A slave releases its implicit read lock on the coset table `LOCK_RELEASE_FREQUENCY` times during each task. However, this parameter was found to make little difference, and we stopped using it by default.

3.3. Cloud heuristic for bulk coset definition

A simple bulk coset creation strategy, successfully used in some previous parallel coset enumerators (Cooperman and Havas [1997], Havas and Ramsay [2000]), is to create n table rows, linking them to the first n gaps in the coset table. With n equal to 1 this strategy creates coset representatives in exactly Felsch order. At higher values of n it approximates Felsch coset creation order, but some reordering occurs because of a delay in processing collapses. Still, coset representatives are created in a breadth-first manner.

We propose a different coset creation strategy that creates new coset representatives as a connected structure. This connected structure, which we call a *cloud* is precomputed and reused every time we need to define new table rows. This cloud is generated by partially enumerating the full group while ignoring the subgroup generators. This auxiliary enumeration stops after reaching a preset number of coset table rows.

In order to *apply the cloud*, the cosets of the cloud are renumbered and copied to the end of the coset table. The coset table then represents two connected components of a graph: the original table T, and the cloud C. One then chooses a *cloud application point*. a coset from the original table. This is identificed with the origin coset of the cloud, and the collapse routine is invoked. The collapse routine causes the higher numbered cosets of the cloud component to collapse onto cosets of the original component. The cloud cosets that are elmiinated by the colapse routine are called *transient cosets*. The collapse routine requires little CPU time.

After the transient cosets are eliminated by the collapse routine, the enumerator adds to the deduction queue the following links. If $c \in C$ collapses onto $c' \in T$, then the link (c', g) is added. If $c \in C$ does not collapse and if $cg \in C$ collapses onto $k' \in T$, then the link (k', g^{-1}) is added. One can show that these added deductions suffice to guarantee termination of the coset enumeration. Typically this process creates more than `PAR_START_SIZE` deductions on the queue, and the computation switches into parallel mode.

In our experiments, we achieve reasonable performance when about half of the cloud cosets collapsed during each cloud application. Part of the efficiency of cloud-based coset definition comes from the fact that since the relators have already been scanned through the links of the cloud. Cloud application involves only the scanning of the new links created during the initial collapse. A good choice of the cloud application point is essential for performance of the enumerator.

- If too many cosets of the cloud collapse, the number of new cosets defined is small, and so the CPU time increases.
- If too few cosets of the cloud collapse, the enumerator eventually constructing a sparse structure to the coset table, causing the memory usage to grow significantly. This results in "intermediate swell".

We found that a good heuristic for choosing a cloud application point is to perform a random walk on the coset table, starting with the origin, and stopping when a gap is encountered on the path of the random walk. Then we apply the cloud at the coset reached by the random walk two, three or four steps earlier. The timing is relatively insensitive to the actual choice, and we standardize on two steps backwards. For our benchmark Lyons enumeration this heuristic allowed the enumeration to complete typically with only a small overshoot in the number of cosets in the intermediate stages of the enumeration, typically less then the size of the complete set of cosets, thus limiting memory use to the memory required to store the full table.

Intuitively, this can be understood by thinking of the set of enumerated cosets in a breadth-first Felsch-type scan as if the enumerated cosets formed a ball. We also think of the cloud as if it were a ball. If the cloud were applied with its center exactly at the edge of the ball, the intersection of the cloud with the set of enumerated cosets would be much less than half the size of the cloud. This tends to yield a sparse data structure when the cloud is instantiated in the coset table. By taking two steps backward and then applying the cloud, we increase the amount of overlap between the cloud and the enumerated cosets.

Two additional enhancements of the cloud heuristic were employed.

1. If our target was a cloud of size 1,000,000, then we experimented with constructing a cloud of some larger factor (for example a cloud of size 2,000,000) and truncating it by keeping only the first 1,000,000 rows of the coset table for the cloud. The result was a cloud of only slightly higher density (an additional 1% or less), and the results were not significantly different.
2. In applying a cloud, we specified both a cloud size and a *Felsch interval* (with default value 4) between cloud applications. A Felsch interval of size n meant that we would define a coset, scan all relators and process any deductions for n cosets exactly according to the Felsch strategy. This Felsch interval did yield significantly improved results, and we postulate that it was useful for filling random holes close to the origin coset, which were important for the efficiency of coset enumeration.

3.4. Shallow scan heuristic

Next, we discuss the shallow scan heuristic. This heuristic is designed especially to overcome a performance penalty that occurs primarily in the latter stages of an "easy" enumeration such as Lyons's group. In the latter stages, significant time is spent finding and filling the last few blanks in the coset table. Because the blanks are sparse in the coset table, the clouds heuristic becomes quite inefficient at this stage. This is why the shallow scan heuristic, even in its sequential version, is important for parallel coset enumeration with clouds.

The shallow scan heuristic is not needed for "hard" enumerations. It neither accelerates nor degrades the performance significantly in the cases of "hard" enumerations that we tested. This is because in "hard" enumerations, the time spent filling the last few holes tends to be dominated by the time to process a large collapse after the intermediate swell, and the processing of the large collapse already parallelizes well.

Define the distance of a coset from the identity coset as the shortest word in the group generators that maps the given coset to the identity coset. A Felsch-type coset enumeration algorithm tends to define cosets in a breadth-first manner. Thus, we can talk about the algorithm as having explored level n if it has reached all cosets at distance n from the trivial coset. In the case of Lyons's group (and most others), one sees that the largest level (as measured by the number of cosets at that level) is followed by levels of rapidly decreasing size. In the case of Lyons's group, the maximum level is level 11, and it has approximately 5 million cosets, followed by a level 12 with only about 1 million cosets. Yet we observe that the enumeration of level 12 of 1 million cosets tends to be almost as time-consuming as level 11 with 5 million cosets. The number of cosets at each level is:

0: 1; 1: 1; 2: 4; 3: 18; 4: 89; 5: 568; 6: 3,262; 7: 19,214; 8: 92,016;
9: 429,248; 10: 2,075,443; 11: 5,070,691; 12: 1,144,434; 13: 167

To counter this, we note that a Felsch-style algorithm will at some point have defined most cosets in level 11 and almost none of the cosets in level 12. In this situation, there is a heuristic for rapidly defining a large majority of the cosets in level 12. One chooses a subset of the group generators, S', where our experience suggests that the size of S' be at least 3. One iterates through the coset table looking for a table row (or coset) c, such that cg is still undefined in the coset table for $g \in S$. One then defines a new coset $c' = cg$.

We find empirically that when most cosets at level 11 have been defined, this heuristic defines table representatives for most of the cosets of level 12. To see this intuitively, make the heuristic assumption (which is not strictly true, but allows one way of understanding the success of this heuristic) that cg is randomly at an adjacent or identical level to that of c, weighted according to the number of cosets at each level. Let c' be a coset at level 12 with no current table representative. Let g be an arbitrary generator in our subset S'. Our heuristic assumption says that $c = c'g^{-1}$ is more likely to occur at level 11 than at level 12 or level 13. Since most cosets at level 11 have been defined, c most likely has a table representative. Hence, $c' = cg$ will most likely be discovered by the shallow scan heuristic. So, if c' is at level 12, then $c = c'g^{-1}$ is more likely to occur at level 11 (with 5 million cosets) than at level 12 or level 13 (with only 1 million cosets). Hence, most cosets c' at level 12 will be discovered by filling in holes at level 11.

It is also clear that if S' is the singleton set, then we will seldom define two table representatives of the same coset by this heuristic. That would imply that $c_1 g$ and $c_2 g$ are identified with each other. This can only happen if c_1 and c_2 are table representatives of the same coset. Since we observe that the number of active entries in the table always remains below the total number of cosets, and since the cosets through level 11 represent a large majority, there must be relatively few redundant representatives of cosets in the table. So it must be a rare situation that c_1 and c_2 correspond. As the size of S' grows, we may create more redundant table representatives at level 12, but we empirically observe good performance for $|S'| = 3$.

In order to speed up the processing phase, we then apply only those of the essential rotated relations that begin with g^{-1} and end with a missing link in the table next to the newly defined coset c'. Furthermore, we recursively continue to use only such relations beginning with g^{-1} and ending with a missing link, and ignore all other relations.

Strictly speaking, ignoring the remaining links threatens to lose our guarantee of correctness, in that we may fail to process a necessary coincidence. However, the shallow scan heuristic is only used at the stage of the enumeration when the coset table has been mostly filled, and the full set of relators is highly redundant. The shallow scan heuristic allows one to select non-redundant relators with high efficiency.

One can show that if one fails to apply some relators to a finite number of links and the coset enumeration still terminates, the resulting table will contain an integer multiple of the correct number of cosets. Further, if one views the generators as permutations of the table rows, then the set of table representatives of a give coset will in fact be a block of imprimitivity. In practice we don't observe the loss of consistency leading to an incorrect result of enumeration. We do discuss a way to check correctness of the coset table in Section 7.

In the case of Lyons, the shallow scan brought down the sequential time for the last phase from 5 hours 14 minutes down to 41 minutes using the shallow scan heuristic. (See Section 5.1 for details.)

A more aggressive self-tuning version of this shallow scan heuristic has been used in sequential computations to reduce the computation time for Lyons's group by a factor of 3 (from 4 hours down to 1.4 hours). This will be described further in a separate paper, which also describes additional heuristics intended exclusively for sequential coset enumeration.

3.5. Parallelization of a mixed Felsch/HLT strategy

We define the *Felsch horizon* of an enumeration as the set of rows of a coset table for which some table entries have not yet been filled in. The distance of a table row to the origin is the length of the shortest path from that row to the origin row, where a step in that path consists of moving from the current row to a new row, where the new row is the value of any table entry in the current row. A pure Felsch strategy tends to yield a Felsch horizon in which the distance to the origin of any two rows on the Felsch horizon differ by at most one. *Mixed Felsch and HLT strategies* have been used heuristically for two purposes:

1. an occasional HLT scan may succeed in discovering a coincidence far beyond the horizon of a Felsch scan.
2. an HLT scan will reshape the Felsch horizon.

In order to cope with harder enumerations we implemented a simple mixed Felsch/HLT/clouds strategy, based on the mixed strategy used by ACE (Havas and Ramsay [1999]).

The mixed strategy of ACE uses parameters named `CT` and `RT`. After defining every `CT` cosets Felsch style, the enumerator switches to HLT tracing, and traces relators at `RT` cosets. Without a priori knowledge of how the given presentation would behave, one would typically use `CT` = `PRESENTATION_LENGTH` and `RT` = 1. This setting results in the enumerator defining roughly the same number of cosets in Felsch and HLT modes.

In addition to supporting clouds, our mixed strategy also includes minor changes targeting it for more efficient parallelization:

1. The strategy has three phases instead of just two: cloud application, Felsch-style coset definition, and HLT-style coset definition.
2. Felsch-style and HLT-style coset definition are performed (on the master) between every two cloud applications. When using clouds, we typically set the Felsch interval (`CT`) to a very low value, such as 1 or even 0.

   For HLT tracing, every relator is traced at the number of rows given by `RT`. In our enumerator, the default is set according to the formula `RT = CLOUD_SIZE/PRESENTATION_LENGTH`. This keeps the proportion of cosets defined during HLT tracing roughly the same as with a reasonable setting not using clouds, i.e. roughly one half of the cosets are defined during cloud application, and one half during HLT tracing.
3. HLT-style tracing may result in the discovery of deductions and coincidences. These are processed immediately, adding the deductions to the deduction queue. The scanning of these queued deductions, however, is not performed immediately, but rather delayed until just before the next cloud application. At that time the deduction queue contains deductions resulting from the last cloud application, Felsch coset definition, and HLT tracing. The whole queue is scanned in parallel.

   Since most of the processed deductions come from cloud application and HLT-style relator tracing, delaying their scanning has a significantly smaller negative impact than that seen with previous Felsch-based parallelizations.

4. Bulk definition coset enumerations need not change the order of coset definition

The theorem of this section provides a type of "Diamond Lemma" that apparently is new in the literature. Its conclusion is needed to justify our parallel approach. It shows that the parallel enumeration described in this paper, after each bulk definition and after associated coincidences and deductions, will produce a partial coset table that is identical to that produced by a corresponding sequential enumerator with bulk definition. Further, the partial coset tables are independent of the number of parallel processors used. Therefore, for a fixed size cloud (and we choose one with 1,000,000 cosets in the case of Lyons), we do not see the "intermediate swell" in the number of cosets of the coset table as the parallelism increases. All previous enumerators exhibit that intermediate swell.

To process a coincidence the enumerator performs a collapse and to process a deduction it inserts a new link into the coset table. The order of processing of coincidences and deductions is defined by string $s \in (c_{a,b}|l_{a,b,g})*$, where each $c_{a,b}$ defines a collapse by specifying a pair of coinciding table rows a and b, and each $l_{a,b,g}$ defines a new link by specifying table rows a and b and generator g such that $\overline{a}^g = \overline{b}$, where $\overline{a}$ and $\overline{b}$ are cosets represented by a and b respectively.

We call the state of the coset table *consistent* if the table contains no deductions or coincidences. Define a *relator loop at* c_1 *leading to a coincidence* (c_1, c_2) *in a table* T as a relator that begins at table row c_1, and after being traced to the end, arrives at table row c_2. Hence $c_1 = c_2$ in the group. Define a *relator loop at* c_1 *leading to a deduction* (c_1, c_2) *in a table* T as a relator that begins at table row c_1, and after being traced through the last generator but one, arrives at a row c_2 for which the table has no value in the column corresponding to g, the last generator in the relator. Hence, $c_2 g = c_1$.

Theorem 4.1 (bulk definition theorem): *Suppose that for a given initial state T of the coset table, coincidences and deductions are processed by performing collapses and entering links into the table as appropriate, until the table is consistent. Suppose that, when collapsing coinciding table entries, the enumerator always retains the lowest numbered table row representing the coset. Then the resulting state of the table does not depend on the order in which deductions and coincidences are processed.*

Let $s = h_1 \ldots h_n$ be the sequence of collapses and link insertions dictated by the contents of the coset table that leads from state T to $T' = s(T)$. There is a natural correspondence between rows of T and T', where every row of T that collapses under s corresponds to the row in T' to which it collapses. This allows us to view s as a map from the rows of T to the rows of T', and also as a map from links in T to links in T'. Since neither collapses nor link insertions ever remove links from the table, the latter map is well-defined. We use the same notation for both step sequences and for maps induced by those step sequences.

We compare two sequences $s_1 : T \to T_1$ and $s_2 : T \to T_2$ to show that the table states T_1 and T_2 that they lead to are identical. We consider step h_q of $s_2 = s_2' h_q s_2''$, and proceed by induction on q. Let $X : s_2'(T) \to T_1$ be the map from the rows and links of the table at the end of s_2' to the rows and links of T_1, defined by $X(a) = s_1(s_2'^{-1}(a))$.

Assume the induction hypothesis that X is well-defined, i.e. all pre-images (in T) of a row in $s_2'(T)$ map under s_1 to the same row in T_1. Consider the map $X' : h_q(s_2'(T)) \to T_1$, defined by $X'(a) = s_1(s_2'^{-1}(h_q^{-1}(a)))$. We will show that X' is well-defined.

There must be a relator loop $\rho \in s_2'(T)$ that leads to the discovery of the deduction or coincidence responsible for step h_q. Since T_1 is a consistent state of the coset table, the loop $X(\rho)$ must be closed. We consider two cases, depending on the type of step h_q.

1. If h_q is a collapse, then the collapsed table rows correspond to the same row in T_1. So X' is well-defined.
2. If h_q is a link insertion, then X' is trivially well-defined for rows. Since $X(\rho)$ is a closed loop, a link corresponding to the link inserted by h_q is present in T_1. So X' is well-defined for links as well.

Next, we examine the base case. Since $s_2 = s_2' h_q s_2''$ is a well-defined map, the pre-conditions of the inductions are met for $q = 1$, so the map $Y : T_2 \to T_1$ defined by $Y(a) = s_1(s_2^{-1}(a))$ is well defined. Similarly for Y^{-1}. Since both Y and Y^{-1} are well-defined, the map Y must be bijective. Also, since collapses in both s_1 and s_2 select the surviving row by the same method (the lowest numbered row), the table states T_1 and T_2 are identical. ∎

5. Experimental Results

The following computations were executed at Boston University on their Origin 2000 processors running Irix 6.5. The gcc-2.95 compiler was used because the native SGI compiler did not support our use of templates and STL. Each processor was a 195MHz R10000 SGI processor (64-bit MIPS R10000 architecture) with 4 MB of cache and 128 MB of RAM per processor. The memory was organized in banks of 256 MB, with two processors sharing each bank.

For verification of the robustness of the results, certain of the tests were re-run at the University of Queensland facility, which has a similar facility, except that they have 64 nodes on a single machine with 256 MB of RAM per processor.

At Boston University, our jobs typically ran on unloaded machines, while at the University of Queensland, there was typically a continuous load with a load factor (utilization) varying from 45 to 55, spread among 64 CPU's. Timings were reproducible between the two machines. Our jobs used approximately 1 GB of RAM, each, and we did not observe excessive paging.

We list the presentation in Table 1. The subgroup generators are a, b, c, d and e. The presentation is based on the work of Sims [1973], and is the one used by Cooperman and Havas [1997]. The result of Havas and Ramsay [2000] is based on a slightly improved presentation for purposes of computer enumeration, to be found in Havas and Sims [1999].

A cloud of size 1,000,000 was used for the cloud heuristic. We took a fixed random seed for all runs, so as to make the runs reproducible. We ran our threads using the SGI-specific scope `BOUND_NP` (bound threads). While our aggregate CPU times were very stable during repeated runs, we found that the elapsed times could differ by as much as 3%.

5.1. Lyons: overall results

The parallel computation on one slave executed in two distinct phases. The total time was 9.4 hours = 473 min. (clouds phase) + 41 min. (shallow scan phase). With 32 processors, this broke down into 23 min. (clouds, par.) + 50 min. (clouds, seq.) + 41 min. (shallow scan). The reason for assigning 50 minutes of the cloud phase as an inherently sequential portion in our implementation is explained in Section 5.3. The shallow scan phase has not yet been parallelized. (See Section 3.4 for a discussion of correctness of the shallow scan phase.)

We first ran our program in straight Felsch mode. It completed in 11 hours and

Lyons's group; Subgroup generators: none
ee, $b^{-1}d^{-1}bd$, $ffa^{-1}a^{-1}$, $bbbbb$, $ccccc$, $a^{-1}a^{-1}b^{-1}aab$, $dda^{-1}a^{-1}a^{-1}a^{-1}$, $d^{-1}ada^{-1}a^{-1}a^{-1}$, $f^{-1}afaaa$, $a^{-1}a^{-1}caacc$, $ececb^{-1}b^{-1}c^{-1}$, $aaaaaaaa$, $abababab$, $f^{-1}bab^{-1}faabab^{-1}$, $ebebccb^{-1}b^{-1}cbc^{-1}$, $ebcb^{-1}ebc^{-1}b^{-1}b^{-1}cb^{-1}$, $a^{-1}c^{-1}abc^{-1}b^{-1}a^{-1}cabcb^{-1}$, $a^{-1}b^{-1}aba^{-1}b^{-1}aba^{-1}b^{-1}ab$, $b^{-1}b^{-1}cbccbbc^{-1}b^{-1}c^{-1}c^{-1}$, $f^{-1}b^{-1}cbfa^{-1}b^{-1}a^{-1}c^{-1}aba$, $f^{-1}bcb^{-1}fb^{-1}a^{-1}bc^{-1}b^{-1}ab$, $a^{-1}b^{-1}cbabc^{-1}b^{-1}c^{-1}b^{-1}ccb$, $f^{-1}cfa^{-1}b^{-1}ca^{-1}bcb^{-1}aba$, $f^{-1}a^{-1}cafb^{-1}c^{-1}b^{-1}ac^{-1}a^{-1}bb$, $eb^{-1}cbeabcb^{-1}ca^{-1}c^{-1}b^{-1}b^{-1}$, $b^{-1}c^{-1}ba^{-1}cab^{-1}cbc^{-1}a^{-1}c^{-1}ac$, $f^{-1}bbab^{-1}b^{-1}fabba^{-1}b^{-1}b^{-1}a^{-1}$, $c^{-1}ac^{-1}a^{-1}caca^{-1}cac^{-1}a^{-1}c^{-1}aca^{-1}$, $c^{-1}a^{-1}c^{-1}ac^{-1}a^{-1}caca^{-1}c^{-1}aca^{-1}ca$, $b^{-1}a^{-1}cabbcb^{-1}cbcb^{-1}b^{-1}c^{-1}ba^{-1}c^{-1}a$, $b^{-1}a^{-1}c^{-1}ac^{-1}a^{-1}cacbc^{-1}a^{-1}c^{-1}aca^{-1}ca$, $ec^{-1}a^{-1}c^{-1}aca^{-1}caebcb^{-1}c^{-1}b^{-1}cbc^{-1}$, $ed^{-1}c^{-1}d^{-1}ba^{-1}b^{-1}c^{-1}d^{-1}a^{-1}c^{-1}ac^{-1}a^{-1}cabb$, $eaaebcb^{-1}a^{-1}c^{-1}ac^{-1}a^{-1}cba^{-1}b^{-1}b^{-1}a^{-1}$, $eabbab^{-1}aebcb^{-1}a^{-1}c^{-1}ac^{-1}a^{-1}ca^{-1}$, $a^{-1}bcb^{-1}abc^{-1}b^{-1}c^{-1}bc^{-1}b^{-1}b^{-1}cba^{-1}cac^{-1}$, $b^{-1}c^{-1}bbc^{-1}b^{-1}b^{-1}cbbcb^{-1}c^{-1}a^{-1}c^{-1}aca^{-1}ca$, $b^{-1}b^{-1}abbfb^{-1}b^{-1}abbfb^{-1}b^{-1}abbfaa$, $aabab^{-1}abaabab^{-1}abaabab^{-1}ab$, $f^{-1}c^{-1}a^{-1}c^{-1}aca^{-1}cafcbbc^{-1}a^{-1}b^{-1}c^{-1}bab^{-1}b^{-1}$, $bc^{-1}b^{-1}a^{-1}c^{-1}ac^{-1}a^{-1}cacbcb^{-1}c^{-1}a^{-1}c^{-1}aca^{-1}ca$, $bc^{-1}b^{-1}cbcb^{-1}a^{-1}c^{-1}ac^{-1}a^{-1}caca^{-1}c^{-1}ac^{-1}a^{-1}ca$, $ea^{-1}caea^{-1}c^{-1}aea^{-1}cabc^{-1}b^{-1}ab^{-1}cbaaa$, $b^{-1}c^{-1}ba^{-1}c^{-1}ac^{-1}a^{-1}cacb^{-1}cbc^{-1}a^{-1}c^{-1}aca^{-1}ca$, $c^{-1}b^{-1}c^{-1}bcb^{-1}cbc^{-1}a^{-1}c^{-1}aca^{-1}cac^{-1}a^{-1}c^{-1}aca^{-1}ca$, $a^{-1}c^{-1}a^{-1}c^{-1}aca^{-1}cac^{-1}ac^{-1}a^{-1}caca^{-1}c^{-1}ac^{-1}a^{-1}cac$, $de^{-1}b^{-1}a^{-1}ce^{-1}a^{-1}e^{-1}a^{-1}c^{-1}a^{-1}c^{-1}aca^{-1}cabcb^{-1}cbab^{-1}b^{-1}a^{-1}$, $dc^{-1}a^{-1}c^{-1}aca^{-1}cada^{-1}c^{-1}ac^{-1}a^{-1}cacdc^{-1}a^{-1}c^{-1}aca^{-1}ca^{-1}$, $c^{-1}a^{-1}c^{-1}aca^{-1}cac^{-1}a^{-1}c^{-1}aca^{-1}cac^{-1}a^{-1}c^{-1}aca^{-1}cac^{-1}a^{-1}c^{-1}aca^{-1}cac^{-1}a^{-1}c^{-1}aca^{-1}ca$, $fbdfbba^{-1}b^{-1}a^{-1}bbc^{-1}aca^{-1}c^{-1}dc^{-1}bba^{-1}cbf^{-1}d^{-1}b^{-1}f^{-1}c^{-1}a^{-1}cbc^{-1}d^{-1}ca^{-1}cacb^{-1}a^{-1}b^{-1}a^{-1}b$, $dc^{-1}a^{-1}c^{-1}aca^{-1}cadc^{-1}a^{-1}c^{-1}aca^{-1}cadc^{-1}a^{-1}c^{-1}aca^{-1}cadc^{-1}a^{-1}c^{-1}aca^{-1}cadc^{-1}a^{-1}c^{-1}aca^{-1}ca$, $f^{-1}d^{-1}f^{-1}a^{-1}c^{-1}d^{-1}ba^{-1}b^{-1}c^{-1}d^{-1}ba^{-1}b^{-1}c^{-1}b^{-1}ab^{-1}b^{-1}ab^{-1}fdfaba^{-1}bba^{-1}bbc^{-1}acbcb^{-1}dcb^{-1}b^{-1}$ $dcbc$, $c^{-1}d^{-1}a^{-1}c^{-1}a^{-1}bc^{-1}d^{-1}fa^{-1}db^{-1}c^{-1}a^{-1}bc^{-1}d^{-1}bc^{-1}b^{-1}a^{-1}c^{-1}ab^{-1}abab^{-1}fa^{-1}c^{-1}d^{-1}ac^{-1}b^{-1}ac^{-1}$ $d^{-1}a^{-1}c^{-1}acb^{-1}b^{-1}c^{-1}b^{-1}a^{-1}b^{-1}b^{-1}f$, $fbab^{-1}a^{-1}b^{-1}b^{-1}a^{-1}b^{-1}cbbcbdcb^{-1}b^{-1}dcbcfab^{-1}b^{-1}a^{-1}cbc^{-1}ba^{-1}c^{-1}c^{-1}dcaca^{-1}cb^{-1}d^{-1}f^{-1}c^{-1}b^{-1}c^{-1}$ $d^{-1}bbc^{-1}d^{-1}b^{-1}c^{-1}b^{-1}b^{-1}c^{-1}babbaba^{-1}b^{-1}f^{-1}d^{-1}acbca^{-1}c^{-1}d^{-1}ab^{-1}a^{-1}ca^{-1}b^{-1}b^{-1}a^{-1}ba^{-1}$

Table 1: Lyons's Group Presentation

53 minutes, with 42,640 CPU seconds. 12,196,727 total cosets were defined. This time can be compared to the result of Havas and Ramsay [2000], in which they executed in 13 hours with the same presentation. They then developed a shorter presentation, which yielded a smaller sequential time of 8 hours and 4 minutes for their shorter presentation.

We then added the shallow scan heuristic to our algorithm, achieving a sequential time of 7 hours and 30 minutes, with 26,819 CPU seconds. 12,405,647 cosets were defined.

We then added the clouds heuristic to our sequential algorithm. This extended the time to 9 hours and 24 minutes due to the poor sequential performance of the clouds heuristic. 17,908,206 cosets were defined.

We then ran the parallel version (1 slave) in a similar time (with slight variations as we repeated the experiment). 17,908,206 cosets were defined.

Our fastest time was with 32 processors (31 slaves). When we used sequential shallow scan, this ran in 1 hour and 54 minutes using 56,884 CPU seconds. 17,908,206 cosets were defined. Note that the the total number of cosets defined did not change when using 32 processors, thereby confirming Theorem 4.1 with respect to our bulk coset definition algorithm.

Our time with 16 processors was only a little larger than for 32 processors.

When using the shorter presentation of Havas and Ramsay, we were able to reduce the time to about 1 hour and 21 minutes with 35,395 CPU seconds. In comparison, Havas and Ramsay report a time of 2 hours and 18 minutes using approximately 119,000 CPU seconds. Their much higher CPU time is explained because they used spin locks, which resulted in a form of busy waiting for their slaves. They chose not to let their slave threads block because the scheduler on their system would kill any jobs not having a high enough load factor.

Due to our cloud heuristic for bulk coset definition, the maximum number of active cosets always occurred at the end of the computation, and is equal to the total number of cosets defined. This is unlike previous parallelizations, whose computations contained "intermediate swell".

5.2. Lyons: methodology for timing measurements

The *aggregate CPU time* is the system plus user time reported by the operating system. (The system time was less than 1% of the user time in all cases.) The average load factor (sometimes called a utilization ratio) is the aggregate CPU time divided by the elapsed time.

The average load factor is reported as an independent statistic of interest, since it indicates the degree of parallelism that may be available in any bulk coset definition algorithm for coset enumeration. The average load is always less than the number of slave threads since the master is mostly idle, and so it is of interest to look at the trend when the load factor is normalized by the number of slaves. The elapsed time should be taken as the figure of merit and *not* the average load time.

The times in the table are all truncated at 7.86 million active cosets because this is the point in the experiments where the shallow scan heuristic first took effect. We do not include the time due to shallow scan in the tables below, since it is currently a sequential computation, The time for shallow scan, even in its current sequential form, appears to be small enough that it does not affect our overall conclusions. The portion of the computation beyond 7.86 million active cosets is always 41 minutes. A future version will implement a parallel version of shallow scan.

5.3. Lyons: scaling with the number of processors

We first ran a series of tests to find out how our program scaled with the number of processors. The first two tables show the results for slave tasks consisting of 100 and 1,000 links, respectively.

We modeled our times based on a model in which Amdahl's Law is assumed to hold when the number of slaves is small (1, 3, and 7 slaves in this case). We stop at 7 slaves because this is the last point for which we see a good fit with a linear regression model. Over this range, we apply Amdahl's Law states that $t_n \equiv \mathrm{ELAPSED\_TIME} = \mathrm{SEQ} + \mathrm{PAR/n}$ for n slaves. This yields an estimate of 50 minutes of sequential time per computation.

slaves+master cpu's	elapsed time time (h:m)	Aggregate CPU sec	average load
1 cpu	8:43	30,022	0.95
3+1 cpu's	3:15	29,345	2.5
7+1 cpu's	2:08	33,432	4.3
15+1 cpu's	1:33	38,341	6.9
31+1 cpu's	1:13	54,430	12.4
63+1 cpu's	1:15	108,569	29.9

Table 2: Timings (task size: 100; clouds phase only, 7.86×10^6 cosets)

slaves+master cpu's	elapsed time time (h:m)	Aggregate CPU sec	average load
1 cpu	8:43	30,022	0.95
3+1	3:17	29,171	2.5
7+1	2:05	33,442	4.5
15+1	1:30	38,644	7.2
31+1	1:18	59,615	12.73

Table 3: Timings (task size: 1,000; clouds phase only, 7.86×10^6 cosets)

We then fix the sequential time as $t_1 = 50$ minutes and use this to extrapolate how the parallel portion of the CPU time degrades from linear for larger numbers of slaves. Under this assumption, Table 4 shows the linear parallelism beginning to degrade at 15 and 31 slaves. The numbers not in parentheses are the elapsed times of the parallel portion of the program (in minutes). The numbers in parentheses are nt_n/t_1 for n slaves. To the extent that the ratio rises above 1.0, this represents a degradation from linear speedup. We attribute this divergence from Amdahl's Law to memory contention.

task size	1 slave	3 slaves	7 slaves	15 slaves	31 slaves
	(elaps. time × # slaves: normalized)				
100	473	145	78	43	23
100	(1.0)	(0.9)	(1.15)	(1.4)	(1.5)
1,000	473	148	75	40	23
1,000	(1.0)	(0.9)	(1.1)	(1.3)	(1.5)

Table 4: Line 1: Parallel portion of elapsed time (min.); Line 2: Total work (normalized to 1.0 for 1 slave); parallel portion of clouds phase, only

Our results were highly reproducible. For example, when we ran our tests with 1 cpu and with 15 slave threads (with a task input size of 1,000) at the University of Queensland, we found the following numbers, which compare closely with the corresponding runs at Boston University, as seen in Table 5.

slaves+master cpu's	elapsed time (h:m)	Aggregate CPU sec	average load
1 (U. of Q.)	9:44	34,400	0.98
1 (B.U.)	9:24	32,562	0.96
15+1 (U. of Q.)	1:35	38,054	6.68
15+1 (B.U.)	1:30	38,644	7.16

Table 5: Timings at U. of Queensland (task size: 1,000; clouds phase only, 7.86×10^6 cosets)

The processor utilization ratio (load factor) was surprisingly steady throughout the computation. We take the 32 processor run as an example. It tended around a ratio of 12 (for cpu time consumed divided by real elapsed time), holding steady until reaching the point where the enumerated cosets tended to collapse upon each other (approximately at 7.5 million enumerated cosets). After that, there was a gradual increase in the utilization ratio from 12 to 16.

5.4. Other groups: "hard" enumerations

Table 7 represents the results of testing our algorithms on several presentations. We selected the presentations that have sufficiently large CPU and memory requirements to warrant parallelization. Some of the enumerations in our sample would not run with straight Felsch strategy at all, and required a mixed strategy.

We purposely ran the enumerations with default settings, rather than optimal settings, in order to simulate the typical usage pattern of a coset enumerator, when no a priori information about the optimal enumeration strategy for a particular presentation is available. See Section 3.5 for the mixed strategy default.

One of the five enumerations that we chose for this series of test, `3onan`, turned out to be quite unstable. This enumeration runs with a large collapse at the end, and the total number of cosets defined before the collapse significantly depends on various parameter settings, and, more importantly, on the seed of the random number generator used to select the cloud attachment point. Therefore we don't include the results for that enumeration in the table.

We used the shallow scan heuristic on all our test runs, triggering at 90% density. However, shallow scan makes no difference for enumerations `fi24` and `J`, which run with a nearly constant coset table density until a large collapse at the end. The enumeration of `t` quickly reaches 90% density. However, for this enumeration, the clouds heuristic runs efficiently up to 98% density. To make the results more representative, we set the shallow scan trigger point for `t` at 98%.

For each presentation the measurements include a sequential run without clouds (combined Felsch-HLT strategy), as well as a sequential run with clouds, and a combined clouds-HLT strategy.

Note that in all cases a certain amount of speedup is achieved by simply replacing the Felsch component of a mixed Felsch/HLT heuristic by clouds. We

`J`; Subgroup generators: a, $b^{-1}ab$
a^5, b^5, $(ab)^5$, $(ab^{-1})^5$, $(aab)^5$, $(abb)^5$, $(a^{-1}bb)^5$, $(aab^{-1})^5$, $(aabb)^5$, $(aab^{-1}b^{-1})^5$, $(abab^{-1})^5$, $(aba^{-1}b)^5$, $(aba^{-1}b^{-1})^5$, $(aabab^{-1})^5$, $(aaba^{-1}b^{-1})^5$, $(abaab)^5$, $(abaab^{-1})^5$, $(ababb)^5$, $(abba^{-1}b^{-1})^5$, $(aba^{-1}bb)^5$, $(a^{-1}baab^{-1})^5$, $(a^{-1}babb)^5$, $(a^{-1}bbab^{-1})^5$, $(a^{-1}ba^{-1}bb)^5$, $(aabaab^{-1})^5$, $(aaba^{-1}a^{-1}b)^5$, $(abbab^{-1}b^{-1})^5$, $(abba^{-1}bb)^5$, $(abba^{-1}b^{-1}b^{-1})^5$, $a^{-1}baba^{-1}b^{-1}aba^{-1}bab^{-1}a^{-1}b^{-1}ab^{-1}a^{-1}baba^{-1}b^{-1}ab^{-1}a^{-1}bab^{-1}a^{-1}b^{-1}ab$, $aba^{-1}bab^{-1}a^{-1}baba^{-1}b^{-1}ab^{-1}a^{-1}b^{-1}aba^{-1}bab^{-1}a^{-1}b^{-1}aba^{-1}b^{-1}ab^{-1}a^{-1}b$, $a^{-1}b^{-1}ab^{-1}a^{-1}bab^{-1}a^{-1}b^{-1}aba^{-1}baba^{-1}b^{-1}ab^{-1}a^{-1}baba^{-1}b^{-1}aba^{-1}bab^{-1}$, $ab^{-1}a^{-1}b^{-1}aba^{-1}b^{-1}ab^{-1}a^{-1}baba^{-1}bab^{-1}a^{-1}b^{-1}aba^{-1}bab^{-1}a^{-1}baba^{-1}b^{-1}$, $b^{-1}abab^{-1}a^{-1}bab^{-1}aba^{-1}b^{-1}a^{-1}ba^{-1}b^{-1}abab^{-1}a^{-1}ba^{-1}b^{-1}aba^{-1}b^{-1}a^{-1}ba$, $bab^{-1}aba^{-1}b^{-1}abab^{-1}a^{-1}ba^{-1}b^{-1}a^{-1}bab^{-1}aba^{-1}b^{-1}a^{-1}bab^{-1}a^{-1}ba^{-1}b^{-1}a$, $b^{-1}a^{-1}ba^{-1}b^{-1}aba^{-1}b^{-1}a^{-1}bab^{-1}abab^{-1}a^{-1}ba^{-1}b^{-1}abab^{-1}a^{-1}bab^{-1}aba^{-1}$, $ba^{-1}b^{-1}a^{-1}bab^{-1}a^{-1}ba^{-1}b^{-1}abab^{-1}aba^{-1}b^{-1}a^{-1}bab^{-1}aba^{-1}b^{-1}abab^{-1}a^{-1}$, $baaba^{-1}b^{-1}abbab^{-1}a^{-1}a^{-1}b^{-1}b^{-1}a^{-1}baaba^{-1}b^{-1}b^{-1}a^{-1}bab^{-1}a^{-1}a^{-1}b^{-1}ab$, $a^{-1}b^{-1}b^{-1}a^{-1}bab^{-1}a^{-1}a^{-1}b^{-1}abbaaba^{-1}b^{-1}b^{-1}a^{-1}baaba^{-1}b^{-1}abbab^{-1}a^{-1}$, $abbab^{-1}a^{-1}baaba^{-1}b^{-1}b^{-1}a^{-1}a^{-1}b^{-1}abbab^{-1}a^{-1}a^{-1}b^{-1}aba^{-1}b^{-1}b^{-1}a^{-1}ba$, $b^{-1}a^{-1}a^{-1}b^{-1}aba^{-1}b^{-1}b^{-1}a^{-1}baabbab^{-1}a^{-1}a^{-1}b^{-1}abbab^{-1}a^{-1}baaba^{-1}b^{-1}$, $(aabaabb)^5$, $(aabbab^{-1}b^{-1})^5$, $(aabba^{-1}a^{-1}b^{-1})^5$, $(aaba^{-1}a^{-1}bb)^5$, $(aab^{-1}aab^{-1}b^{-1})^5$, $(abbaab^{-1}b^{-1})^5$, $(a^{-1}bbaab^{-1}b^{-1})^5$, $(a^{-1}bba^{-1}a^{-1}bb)^5$, $(a^{-1}a^{-1}baabb)^5$, $(a^{-1}a^{-1}bbaab^{-1})^5$
`g(4,10)`; Subgroup generators: none
a^2, b^{10}, $ab^{-1}ab(abab-)^3ab^2ab^{-2}$
`t`; Subgroup generators: y
$x^2; y^3; (xy)^{13}; [x,y]^5; [x,yxy]^4; (xyxyxyxyxy^{-1})^6$
`fi24`; Subgroup generators: a, b, c, d, e, f, g, h, i, k, l, $(fedcbafghijfkl)^{17}$
a^2, b^2, c^2, d^2, e^2, f^2, g^2, h^2, i^2, j^2, k^2, l^2, $(ab)^3$, $acac$, $adad$, $aeae$, $afaf$, $agag$, $ahah$, $aiai$, $akak$, $alal$, $(bc)^3$, $bdbd$, $bebe$, $bfbf$, $bgbg$, $bhbh$, $bibi$, $bkbk$, $blbl$, $(cd)3$, $cece$, $cfcf$, $cgcg$, $chch$, $cici$, $ckck$, $clcl$, $(de)^3$, $dfdf$, $dgdg$, $dhdh$, $didi$, $dkdk$, $dldl$, $(ef)3$, $egeg$, $eheh$, $eiei$, $ekek$, $elel$, $(fg)3$, $fhfh$, $fifi$, $fkfkfk$, $flfl$, $ghghgh$, $gigi$, $gkgk$, $glgl$, $hihihi$, $hkhk$, $hlhl$, $ikik$, $ilil$, $klklkl$, $a(cdefghk)^9$, $ajaj$, $bjbj$, $cjcj$, $djdj$, $ejej$, $fjfj$, $gjgj$, $hjhj$, $ijijij$, $jkjk$, $jljl$
`3onan`; Subgroup generators: a, b, e, f, g, $dcegfdefdfecgd$
a^2, b^2, c^2, d^2, e^2, f^2, g^2, $(ab)^3$, $a = cfcf$, $a = cgcg$, $(bc)^5$, $bdbd$, $bfbf$, $bgbg$, $(cd)^3$, $cece$, $(de)^5$, $(df)^3$, $(dg)^3$, $e = fgfg$, $(abc)^5$, $(bcd)^5$, $a = (cde)^5$, $[c,(dfg)^5]$

Table 6: Presentations used

attribute this speedup to the fact that the structure, created in the coset table by cloud application, initially (before any scanning of deductions) has higher density than the structure created by Felsch-style coset definition. Therefore the amount of work needed to finish filling the blanks in this structure is smaller. With the Lyons enumeration this effect is insignificant, since the cloud is quite sparse.

5.5. Parallelization in distributed memory

Because the parallelization of the coset enumerator was based on TOP-C, we were able to build a distributed memory parallel version of the enumerator with little additional effort. This early version of a distributed enumerator is intended as a proof of principle. It runs at the cost of replicating the global memory, including the memory for the coset table, on each slave node. Hence, the current

Pres./run mode	#cosets	max. rows	total defined rows	transient rows	cpu time	wall clock time
J	15,625					
sequential		1,435,508	1,481,363	*N/A*	366.36	368
seq./clouds		3,377,062	3,581,841	56,316	180.98	183
clouds/4 slaves					181.40	115
clouds/8 slaves					187.67	108
clouds/16 slaves					198.63	103
clouds/32 slaves					214.56	100
g(4,10)	9,302,480					
sequential		9,302,480	18,420,848	*N/A*	1,024.29	1,037
seq./clouds		9,302,480	23,941,794	24,840,779	942.14	957
clouds/4 slaves					885.91	641
clouds/8 slaves					981.58	616
clouds/16 slaves					1232.16	612
clouds/32 slaves					1527.02	629
fi24	306,936					
sequential		5,978,264	6157996	*N/A*	1,050.24	1,068
seq./clouds		2,024,704	2,417,636	7,886,428	632.10	644
clouds/4 slaves					671.99	440
clouds/8 slaves					742.48	418
clouds/16 slaves					909.80	410
clouds/32 slaves					1294.54	407
t	5,990,400					
sequential		5,990,400	6,481,720	*N/A*	788.01	796
seq./clouds		6,796,187	8,147,860	11,752,724	651.16	664
clouds/4 slaves					594.07	338
clouds/8 slaves					742.68	360
clouds/16 slaves					924.12	293
clouds/32 slaves					1460.50	292
3onan	368280					
sequential		4,504,685	5,924,724	*N/A*	460.36	466
seq./clouds		unstable, depends on random number generator seed				

Table 7: Tests on other groups

design of the distributed memory enumerator is not practical for very large enumerations, since it is already difficult to find one computer with many gigabytes of RAM, and it would be extremely difficult to replicate that much RAM on each node of a distributed architecture.

The total development time to add distributed memory capability to our shared memory parallelization was about 200 lines of additional code and 8 man-hours, including testing. This is in addition to the 250 lines of code written to convert our original sequential enumerator into a shared memory parallel enumerator. The original enumerator had about 3,200 lines of code.

TOP-C provides three libraries with which the code can be re-linked by a compile-time option. Hence, the same source code serves as a sequential enumerator, a shared memory parallel enumerator, and a distributed memory parallel enumerator.

When in parallel mode, the master process distributes work to the slaves from its own deduction queue, and the slaves scan relators and report back to the master on which relators lead to a coincidence or deduction. In the parallel mode, only the master processes deductions from its queue, while the slaves do not touch their queue. Upon exiting from parallel mode the deduction queues of the slave processes will no longer reflect the deduction queue of the master. Hence, one must synchronize the queues by copying the deduction queue of the master to each of the slaves. This requirement is the basis for most of the additional 200 lines of code for the TOP-C distributed memory version.

The distributed memory parallelization is demonstrated for Fi_{24}/Fi_{23}. It was difficult to choose a good example, since the methodology requires large clouds, which are most effective on large enumerations, while the limited RAM available on each slave node (256 MHz) is not sufficient for large enumerations. The presentation for Fi_{24}/Fi_{23} is given in Table 6.

We ran with no cloud, and a ct/rt ratio of 10/1. Hence, after every 10 coset table definitions, we entered one relator in the relator table (or equivalently, executed one HLT lookahead pass). With four slaves, the enumeration completed in 93 seconds, while a sequential enumeration completed in 118 seconds. Because there was insufficient work for the slave, the enumerator executed in parallel mode only for the last 25 seconds.

6. Memory latency limits and the trivial parallel program

We analyze the memory contention problem described in Problem 3 of Section 2 and show the Lyons coset enumeration to be limited by saturation of the memory subsystem. To test the memory subsystem, we have written a trivial parallel thread program. Each thread repeatedly reads random entries from a global integer array. The array is 40 MB, so that most accesses are outside of cache. Our goal is to emulate the access to the coset table in the coset enumeration process.

Table 8 compares the memory bandwidth of two parallel architectures to some sequential architectures, while Table 9 demonstrates that the memory bandwidth grows, but reaches saturation with eight or fewer threads.

Finally, Table 10 shows that the elapsed time (wall clock time) to enumerate Lyons's group is determined primarily the memory bandwidth of the Lyons's enumeration (the number of accesses to the coset table per second). Hence, the Lyons's enumeration is largely *independent* of the CPU speed for the given number of threads.

In a separate experiment (whose results are omitted for lack of space), we showed that the Lyons's enumeration saturates the available bandwidth with only 7 slave threads for the Origin 2000, and with only 2 slave threads for the IBM SP-2. This explains why we failed to achieve more than 7 times speedup on the Origin 2000, and why separate experiments failed to achieve more than 2 times speedup on the IBM SP-2.

Computer	Memory accesses/second
Pentium I, 166 MHz, EDO DRAM	2.8×10^6
Pentium II, 350 MHz, PC-100 RAM	3.35×10^6
Pentium II, Xeon, 450 MHz, PC-100 RAM	3.7×10^6
AMD Athlon, 700 MHz, PC-133	2.8×10^6
Origin 2000, 195 MHz MIPS R1000	1.7×10^6
IBM SP-2, Nighthawk-2, 375 MHz RS6000 Power3	2.6×10^6

Table 8: Measured memory bandwidth (single thread, 40 MB, random access)

Computer	# threads	Accesses/s	Accesses/s/thread
Origin 2000	1	1.6×10^6	1.6×10^6
	2	2.1×10^6	1.05×10^6
	4	3.7×10^6	9.2×10^5
	8	5.7×10^6	7.2×10^5
	16	5.8×10^6	3.6×10^5
	64	6.0×10^6	9.4×10^4
IBM SP-2	1	2.6×10^6	2.6×10^6
	2	3.4×10^6	1.7×10^6
	4	2.6×10^6	6.5×10^5
	8	5.2×10^6	6.5×10^5
	16	5.2×10^6	5.3×10^5
Pentium II	1	3.7×10^6	3.7×10^6
(4 CPUs)	2	6.4×10^6	3.2×10^6
Linux 2.2.13	3	8.8×10^6	2.9×10^6
	4	10.4×10^6	2.6×10^6

Table 9: Measured memory bandwidth (multiple threads, 40 MB, random access)

Computer	# threads (master +slaves)	Accesses/s	Elapsed time (Wall clock min.)	Elaps. time $\times$ Acc./s (normalized, Origin 2000 = 1.00)
Origin 2000	1+31	5.8×10^6	73	1.00
IBM SP-2	1+15	5.2×10^6	82	1.01
Pentium II Xeon	1+4	10.4×10^6	49	1.20

Table 10: Bandwidth (accesses/s) vs. Elapsed (wall clock) time for Lyons's group (clouds phase only, 7.86×10^6 cosets), normalized (Origin 2000 = 1.00))

7. Future work: checking correctness of shallow scan

We previously noted that the shallow scan heuristic does not guarantee termination nor correctness. Shallow scan is only needed for "easy" enumerations. Nevertheless, correctness can be checked by running a routine similar to HLT lookahead at the end. The routine would rescan every relator (not rotation or inverse, just the original relators), at every coset in the table. If the table is correct, the routine does not need to modify the table. Hence, it can be efficiently implemented as trivial parallelism. Similarly, the routine could be invoked in the middle of an enumeration to again guarantee termination.

8. Conclusion

We have demonstrated techniques such as bulk definition, the diamond lemma (allowing us to predict parallel performance based on a sequential run), and a shallow scan heuristic, all of which are helpful in speeding up parallel performance. These techniques allowed us to reach a parallel regime where memory latency dominates. Surprisingly, the timing of coset enumeration for this regime correlates closely with memory latency and is independent of CPU speed.

This memory wall is expected to affect many symbolic algebra programs that exhibit intermediate swell, since intermediate swell is typically accompanied by memory accesses outside of cache, and highly random access patterns. Symbolic algebra is especially vulnerable because it tends to use integer operations rather than floating point (thus losing the opportunity to hide the memory latency with longer CPU operations). The RAM latency should be distinguished from memory bandwidth, which is helpful only for sequential access to RAM.

As the gap between CPU clock cycle and RAM latency continues to grow, the situation will only become worse, to the point where even a sequential coset enumeration will be memory-limited. We also demonstrated the first distributed memory implementation of coset enumeration, and such distributed memory techniques may be important in the future for overcoming the memory wall.

9. Acknowledgments

We gratefully acknowledge the support for these computations by both the Scientific Computing and Visualization (SCV) group at Boston University and the High Performance Computation Unit at the University of Queensland. The University of Queensland provided generous amounts of development time. We are also indebted to Michael Dugan of the Boston University SCV group for technical advice. We are grateful for discussions with Edmund Robertson, George Havas, Steve Linton and Leonard Soicher, who contributed both interesting groups on which to test our results, and useful intuition to help improve our program. Finally, we thank the referees for valuable comments.

References

S. Akl, G. Labonté, M. Leeder, and K. Qiu. On doing Todd-Coxeter coset enumeration in parallel. *Discrete Applied Math.*, 34:27–35, 1991.

B. Amrhein, O. Gloor, and W. Küchlin. A case study of multi-threaded Gröbner basis completion. In *ISSAC 96 (Proc. 1996 International Symposium on Symbolic and Algebraic Computation)*. ACM Press, 1996.

P.A. Bandler. A method for enumerating the cosets of an abstract group on a digital computer. Master's thesis, Manchester University, 1956.

M.J. Beetham. Space saving in coset enumeration. In *Computational Group Theory*, pages 19–25, London, 1984. (Durham, 1982), Academic Press.

M.J. Beetham and C.M. Campbell. A note on the Todd-Coxeter coset enumeration algorithm. *Proc. Edinburgh Math. Soc.*, 20:73–79, 1976.

C.M. Campbell. *Applications of the Todd-Coxeter Coset Enumeration Algorithm.* PhD thesis, University of St Andrews, 1975.

J.J. Cannon, Lucian A. Dimino, George Havas, and Jane M. Watson. Implementation and analysis of the Todd-Coxeter algorithm. *Math. Comp.*, 27:463–490, 1973.

G. Cooperman. TOP-C: A task-oriented parallel C interface. In *HPDC-5 (5^{th} International Symposium on High Performance Distributed Computing)*, pages 141–150. IEEE Press, 1996.

G. Cooperman. TOP-C: Task-Oriented Parallel C for distributed and shared memory. In *Workshop on Wide Area Networks and High Performance Computing*, volume 226 of *Lecture notes in control and information sciences*, pages 109–118, 1999.

G. Cooperman. Parallel GAP: Mature interactive parallel computing. In *Groups and Computation III*, pages 123–138. DeGruyter Publishers, 2001.

G. Cooperman and V. Grinberg. Scalable parallel coset enumeration using bulk definition. In *ISSAC 01 (Proc. 2001 International Symposium on Symbolic and Algebraic Computation)*, pages 77–84. ACM Press, 2001.

G. Cooperman and G. Havas. Practical parallel coset enumeration. In G. Cooperman, G. Michler, and H. Vinck, editors, *Proc. of Workshop on High Performance Computation and Gigabit Local Area Networks*, volume 226 of *Lecture notes in control and information sciences*, pages 15–27. Springer-Verlag, 1997.

H.S.M. Coxeter and W.O.J. Moser. *Generators and Relations for Discrete Groups*, volume 14 of *Ergeb. Math. Grenzgeb.* Springer-Verlag, Berlin, Göttingen, Heidelberg, 4th edition, 1957.

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

G.Cooperman. TOP-C (Task Oriented Parallel C/C++) home page. software available at URL: http://www.ccs.neu.edu/home/gene/topc.html, College of Computer Science, Northeastern University, Boston, 2001.

G.Havas and J-X Lian. New implementations of coset enumeration. Technical report, University of Queensland, Key Centre for Software Technology, Department of Computer Science, 1991.

G. Havas and C. Ramsay. Coset enumeration: ACE version 3. Url, University of Queensland, http://www.csee.uq.edu.au/˜havas/ace3.tar.gz, 1999.

G. Havas and C. Ramsay. Parallel coset enumeration using threads. In *Proc. 2000 Asian Symposium on Computer Mathematics*, volume 38 of *Lecture Notes Series on Computing*, pages 29–38. World Scientific Publ., 2000.

G. Havas and C.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. Birkhäuser, 1999.

J. Leech. Coset enumeration. In *Computational Group Theory*, pages 3–18. Academic Press, 1984.

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

C. Ramsay. PACE 1.000: Parallel ACE. Technical report 17, University of Queensland, Centre for Discrete Mathematics and Computing, 2000. software available at http://archive.csee.uq.edu.au/˜cram/ce.html.

C.C. Sims. The existence and uniqueness of Lyons' group. In *Finite Groups '72*, pages 138–141. North Holland, 1973.

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

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