Practical parallel coset enumeration

Gene Cooperman*and George Havas†
gene@ccs.neu.edu and havas@cs.uq.edu.au
College of Computer Science
Northeastern University
Boston, MA 02115, USA
and
School of Information Technology
The University of Queensland
Queensland 4072, Australia

Abstract

Coset enumeration is a most important procedure for investigating finitely presented groups. We present a practical parallel procedure for coset enumeration on shared memory processors. The shared memory architecture is particularly interesting because such parallel computation is both faster and *cheaper*. The lower cost comes when the program requires large amounts of memory, and additional CPU's allow us to lower the time that the expensive memory is being used.

Rather than report on a suite of test cases, we take a single, typical case, and analyze the performance factors in-depth. The parallelization is achieved through a master-slave architecture. This results in an interesting phenomenon, whereby the CPU time is divided into a sequential and a parallel portion, and the parallel part demonstrates a speedup that is linear in the number of processors. We describe an early version for which only 40% of the program was parallelized, and we describe how this was modified to achieve 90% parallelization while using 15 slave processors and a master. In the latter case, a sequential time of 158 seconds was reduced to 29 seconds using 15 slaves.

*Supported in part by NSF Grant CCR-9509783.
†Supported in part by the Australian Research Council.

1 Introduction

Coset enumeration programs implement systematic procedures for enumerating the cosets of a subgroup H of finite index in a group G, given a set of defining relations for G and words generating H. Coset enumeration is the basis of fundamental procedures for investigating finitely presented groups. Computer implementations are based on methods initially described by Todd and Coxeter [17]. A number of computer methods for coset enumeration have been described, including those of Cannon, Dimino, Havas and Watson [3] and of Havas [9]. The latter introduces some coset definition strategies which give substantial reductions in total cosets defined for some difficult enumerations.

Cannon *et al* [3] include comprehensive references to earlier implementations, while later descriptions may be found in Neubüser [13], Leech [12] and Sims [16]. Neubüser and Leech both provide useful introductions to coset enumeration while Sims, in a substantial chapter of his book, gives a formal account of coset enumeration in terms of automata and proves interesting results on coset enumeration behaviour.

The computational complexity of coset enumeration is not well understood. Even for a given strategy there is no computable bound, in terms of length of input and a hypothetical index, to the number of cosets which need to be defined in the coset enumeration process to obtain a complete table. (The existence of such a bound would violate unsolvability results for problems involving finitely presented groups.) Further, Sims [16, pp. 224-226] has proved that there is no polynomial bound in terms of maximum number of cosets for the number of coset tables which can be derived by simple coset table operations like those used in coset enumeration programs. This more practical result indicates that the running time of coset enumeration procedures in terms of space available may be very high.

We build a parallel architecture on TOP-C [5], which provides a lower master-slave layer that is loosely based on the STAR/MPI philosophy [4] (see section 3.1). The master-slave structure is convenient due to its ease of programming and debugging. The debugging is particularly easy because all messages are sequentialized through the master, allowing one to trace messages in order to more intuitively observe the operation of a parallel program.

Historically, TOP-C was preceded by STAR/MPI. STAR/MPI provides similar facilities for interactive languages. Implementations of STAR/MPI exist for Common LISP and for GAP (a general purpose language for group theory algorithms). STAR/MPI is based on distributed workstations linked by MPI (a well-known Message Passing Interface). TOP-C provides a C library for the same capabilities. Further, TOP-C can be used with threads on a shared memory architecture, or with MPI on a distributed memory architecture, without change to the end-user code.

We describe a practical parallel procedure for coset enumeration on shared

memory processors. It is built upon an implementation of the Felsch type procedure by Schönert, which is the basis of the coset enumerator in GAP [14]. The ensuing parallel enumerator is particularly well-suited to difficult enumerations. We make an in-depth case study of a single enumeration, using different parallelization techniques. The analysis of this single case is preferred to a large test suite, since if affords more opportunity for in-depth analysis of specific performance factors, as the algorithm is varied.

2 Major considerations

2.1 Cost effectiveness: More CPU's make it cheaper

Even with the availability of much larger memories, space is still the most important factor in difficult coset enumerations. The determining factor on whether many hard coset enumerations can be completed or not is the space available for the coset table. In certain cases hundreds of megabytes of memory are required. Large amounts of memory are a scarce resource, and it is desirable to reduce the time during which it is needed. One computing environment which provides us with the capability of reducing the time for substantial memory use is the shared memory multiprocessor (SMP).

This was observed by Wood and Hill [19], who studied the situation analytically. In the course of the current research, the authors independently re-discovered such an analytical approach. Our calculation of the costs is presented below, with parameters chosen particularly to model the parallel coset enumeration described in this paper.

We can derive a rough formula that accounts for this phenomenon in which more CPU's are cheaper. Let us assume that a parallelized program has a constant overhead and then achieves a linear speedup in the number of processors. As we shall see, this is the case for the parallel coset enumerator. Let c be the number of CPU's, m be the amount of memory, and h be the constant overhead time. The time for the given calculation is then $t = h + k_1/c$ for an appropriate constant k_1. The cost of the architecture is then $k_2tc + k_3tm$, for appropriate constants k_2 and k_3. The cost of the architecture is minimized when

$$c = \sqrt{k_3 k_1 m/(k_2 h)}.$$

This implies that for every quadrupling of the memory needed for a calculation (assuming that the constant overhead remains fixed), we should double the number of CPU's to achieve the optimum cost.

2.2 Shared memory architectures

A shared memory architecture was preferred over a distributed memory architecture even though a distributed memory architecture might be more highly scalable. The reason for this is that the hardest coset enumerations

use hundreds of megabytes of memory in a coset table. Rather than duplicate the coset table on each processor, requiring enormous amounts of memory, it is much more economical to have all processors use a single copy of the coset table, stored in shared memory.

While the concept of a shared memory architecture is intuitive, perhaps the issues of memory contention are not sufficiently appreciated. Our testbed (see section 4.2) uses 8-way interleaved memory. Hence, we might expect memory contention in the case where we used 16 processors (one master and 15 slaves). This would exhibit itself in less than linear speedup for larger numbers of slaves. In practice, we observed no such memory contention, even with 16 processors.

This observation demonstrates a natural advantage of master-slave parallel algorithms in shared memory architectures. Each slave has a busy period (executing a task) and an idle period (waiting for the next task). Since the slaves are asynchronous, any temporary memory contention is usually caused by an accidental overlapping of busy periods for too many slaves. Such a condition will automatically be eliminated by a natural load balancing. The memory contention will cause some slaves to slow down until the slaves become out of sync again. In our experiments, as the number of slaves increased, the ratio of the average slave CPU time to the total elapsed time decreased. Hence, memory contention was apparently decreasing with the number of slaves.

2.3 Basics of coset enumeration

The primary components of coset enumeration procedures can be considered in terms of three distinct phases: definition/deduction making, relator tracing and coincidence processing. These phases are described in detail in various ways in [3, 9, 12, 13, 16], to which the reader is referred for further information. Comprehensive studies of aspects of coset enumeration performance are included in [3, 9, 16]. These show enormous differences in behaviour for various individual problems, depending on strategy details. In order to facilitate this study we restrict ourselves to just one coset enumeration strategy and one class of particularly difficult problems.

A preliminary and tentative description of parallel coset enumeration is given in [1] from a different perspective. Here we show how coset enumeration can be effectively parallelized on an SMP machine.

The coset enumeration procedure with simplest interactions between the three phases is the Felsch method, which is described in [3, 9, 12, 13, 16]. The fact that these interactions are straightforward makes it the most suitable candidate for parallelization.

We use Martin Schönert's C-language implementation of the Felsch method as our base code. This program is a predecessor of the GAP coset enumerator and provides a clean, modular foundation for our developments.

With respect to the coset table, we observe that definition/deduction making is a write only operation, relator tracing is a read-only operation and coincidence processing is read-write. Coincidence processing (which is an instance of the union-find algorithm) is itself very fast, but may cause very substantial alterations to the coset table, while the other phases make either no changes (relator tracing) or minor changes (definition/deduction making alters at most two coset table entries). Hence we choose to parallelize the two easy phases, but we perform the coincidence processing sequentially. (This is another reason for choosing the Felsch procedure as our base, since it causes relatively few coincidences when compared with other methods.) We allow parallel writing with reading, but we ensure that all writing is done by the master to avoid write conflicts.

One type of coset enumeration which can be particularly space and time-consuming arises in studying groups satisfying identical relations, see [10, 18] for example. In this kind of situation the enumerations can be difficult and often have many defining relations, since many instances of the identical relations are used.

Standard Felsch-style coset enumerations alternate between definition making and relator tracing. All definitions and deductions are traced at all relevant relators at appropriate times. The relator tracing may lead to both deduction making and coincidence processing. In difficult enumerations the bulk of the time is often spent in the relator tracing.

3 Parallelization

3.1 Master slave architecture

The basic concept of the master-slave architecture is that tasks are generated on the master, and assigned to a slave. The result is computed on the slave and returned to the master. Different tasks are executed asynchronously. A SPMD (Single Program Multiple Data) style of programming is encouraged, in which the execution of a task on a slave depends only on the global variables of that slave, and in which the user program maintains the same data in global variables on all processors.

The parallelization of Schönert's sequential code is accomplished using three layers of code. The lowest layer is a simple implementation of message passing using the SGI (see section 4.2) system calls for threads. This layer exports routines such as

```
set_num_procs()
par_call()
finalize()
send_command()
get_result()
get_last_tag()
get_last_source()
is_master()
myid()
```

This layer contains all CPU-dependent parts, and can be replaced by a corresponding file for other host architectures.

The middle layer is a new implementation of a master-slave architecture in C. It is similar to the corresponding layer in the implementations of STAR/MPI [4] in GAP and Common Lisp. It exports the routines and global variables

```
init_master_slave()
master_slave()
ms_barrier()
master_slave_stats()
master_slave_trace
trace_cmd_format
```

The third layer of code is a simple modification of Schönert's sequential coset enumerator. In particular, it required writing new functions, `get_task()` `do_task()`, and `get_task_result()`. These functions are parameters to `master_slave()`.

A full description of the master-slave functionality is given in [4]. For this paper, it suffices to understand that the call to `master_slave()` will provide a parallel analogue of the following sequential code:

```
{ void *task;
  while (task = get_task(), NOTASK != task)
    get_task_result(do_task(task), task);    }
```

However in the parallel version, each invocation of `do_task()` operates on a potentially different slave, and different iterations through the `while` loop operate as if the iterations were overlapped. The master executes `get_task()` and `get_task_result()` on an as-needed basis as slaves either return results, or demand new tasks to execute. For related systems based on distributed memory and remote procedure calls, rather than the tasks described here, see [7]. The work in [11] also provides an alternative parallel interface.

The entire code development, including gaining initial familiarity with the SGI architecture and writing code for all three layers, required about two weeks of part-time activity by one person. This was helped by using the previous code from the master-slave layer of STAR/MPI as a model.

The time does not include the additional, and larger time needed for the many experiments in parallelization using the debugged code. The lowest layer (message-passing for threads) consists of about 200 lines of code. The middle layer (master-slave) consists of about 450 lines of code. The highest layer required adding about 250 lines of new code in an existing sequential program with 1300 lines.

3.2 Parallelization of coset enumeration

The first step in our parallelization is to do relator tracing in parallel. For some difficult examples, relatively few relator traces lead to either deductions or coincidences, so we simply record any trace which is productive, and delay any consequent deduction making or coincidence processing till a separate sequential phase. This does not damage the effectiveness of the coset enumeration procedure. (In the sequential phase we have the overhead of having to repeat the relevant traces, but this is relatively insignificant.)

In our very first experiment, we tried to distribute relator tracing on a relator by relator basis. However there was insufficient work done on each slave to justify the master/slave overheads. Thus, we need an adequate amount of work to do on each slave.

To increase the work per task, we then allocated individual definitions and deductions to slaves for tracing at all relators. (This is why long presentations provide good cases for parallelization.) This keeps slaves busy while there is a long enough work queue often enough. In standard Felsch methods this situation arises after a coincidence cascade, when long deduction queues are built up. In our case study that comprises about half the time.

When the work queue was small, many slaves were waiting idle. To improve on this, we added a parameter, `PAR_THRESHOLD`, for which 20 was found to be a good value. When a new work queue is created, the program operates in sequential mode (to avoid communication overhead). Only if the work queue grows in length to more than `PAR_THRESHOLD`, does the program switch over to the parallel, master-slave program. This is advantageous when the work queue is of length only one or two, which was often the case.

In order to increase the percentage of time when there is a long work queue, we alter the standard Felsch method, which defines one coset when there is no other work to do. Instead we define a number of cosets, namely `PAR_THRESHOLD`, in order to generate a long enough work queue to initiate parallel processing. This also does not damage the effectiveness of the coset enumeration procedure, but it does lead to local perturbations in coset definition order and minor variations in maximum and total coset statistics. This increases the parallelizable component of our coset enumeration procedure from about 40% to as much as 90% for our case study example.

4 A case study

4.1 Problem description

Our example is provided by a group we call T which plays an important role in Vaughan-Lee's proof [18] that Engel-4 groups of exponent 5 are locally finite. The group T is $G/\zeta(G)$ in [18, Lemma 11]. T is generated by a, b, c and satisfies the relations:
$a^5 = b^5 = c^5 = 1$;
$(a^r b^s c^t)^5 = 1$ for $r, s, t = \pm 1, \pm 2$;
$[a, b, a] = [a, b, b] = 1$;
$[a, c, a] = [a, c, c] = 1$;
$[b, c, c, c] = [b, c, c, b] = [b, c, b, c] = [b, c, b, b] = 1$;
$[c, a, b, a] = [c, a, b, b] = [b, a, c, a] = [b, a, c, c] = 1$;
$[b, a, c, [b, a]] = [c, a, b, [c, a]] = 1$.

We choose this example because it is difficult enough to provide a good test and it has enough defining relations to make it well-suited for our purposes. For the record, the subgroup $\langle ab, bc \rangle$ has index 3125 and requires a maximum of 25691 and total of 26694 cosets in a standard sequential Felsch enumeration.

4.2 Parallel facilities and experimental methodology

The work was done on Boston University's SGI Power Challenger Array. That computer consists of 18 R8000 CPU's, each running at 90MHz. The CPU's share 2 gigabytes of 8 way interleaved memory.

It was not possible to run our tests at a time when no one else was on the computer. Hence, our numbers had a small variability due to external loads. Nevertheless, the numbers were reproducible to within 10% during times of light loading. We ran each case several times, so as to distinguish outlier cases (due to momentary heavy loads on the machine) from the general body of results.

There was also an issue of which numbers to report as relevant. We took statistics for the total elapsed time, the CPU time on the master, the CPU time on a slave (actually the total CPU time of all slaves divided by the number of slaves), and load factors indicating how often the master and slaves were kept busy.

In reporting overall statistics, we chose to consider the total elapsed time and the average CPU time of a slave as the two figures of importance. The slave CPU time is also a measure of the elapsed time that the slave was active, as described below. One would prefer to measure the elapsed time that the slave is actively working on tasks, but the measurement would introduce too much overhead. Instead, we assume that the slave CPU time spent on a task is almost 100% of the elapsed time during that task. So, the slave CPU time is identified as the time for the "parallel portion" of the program. The

difference of the total elapsed time and average slave CPU time was then labelled the time for the "sequential portion" of the program.

Note that one cannot point to a time when all slaves proceed in parallel (i.e. are busy at the same time). The decomposition into sequential and parallel portions works because each slave is busy for approximately the same amount of time, although the busy periods are not synchronous across slaves.

A slave on a lightly loaded machine would use almost all of the CPU time while it had a task, and it would use almost none of the CPU time while it was waiting for the next task. Hence, the CPU time on the slave was also considered to be a good measure of the elapsed time for the busy periods of the slave, i.e. the sum of the elapsed times during each task execution.

Thus, we used the average slave CPU time as an indication of the total elapsed time that the typical slave was active. Intuitively, this would seem to be a good measure of the portion of the program that operates in parallel, and the statistics bore this out. The average slave CPU time was almost exactly inversely proportional to the number of slaves.

4.3 Degree of Parallelization

Given the time, P, for the parallel portion and the time, S, for the sequential portion of the computation, as described above, one wants a figure of merit for the degree of parallelization. For N the number of slaves, we use a degree of parallelization,

$$\frac{N \cdot P}{S + N \cdot P}.$$

The quantity $N \cdot P$ should be (and is, in our case) approximately constant reflecting the total amount of parallelizable work to be done. The remaining quantity, S, reflects a period of time when most slave processors are idle, waiting for the master to provide additional tasks. Hence, the degree of parallelization is reflected by the ratio above.

The formula can be justified in terms of Brent's scheduling principle [2], which was originally derived for the PRAM model of parallelism. The principle says that an upper bound for the elapsed time for a parallel computation with p processors is $w/p + t$, where w is the total amount of work to be done (the total elapsed time required by a single processor), and t can be considered either as the total number of synchronization operations required or the elapsed time in the limit as infinitely many processors are applied.

Since in our case, $N \cdot P$ is constant as N varies, we identify $N \cdot P$ with the work w, which must also be constant. If we also identify S, the sequential portion, with t, then the formula reduces to saying: An upper bound on the elapsed time using N processors is $w/p + t = P + S$, which is indeed the case for us.

13. Z. Lin, Ali Saberi, "Semi-global exponential stabilization of linear discrete time systems subject to input saturation via linear feedbacks", Systems and Control Letters, 24(1995) 125-132.
14. J. S. Shamma, "Optimization of the l^{∞}- Induced Norm Under Full State Feedback", *IEEE Trans. on A.C.*, vol. 41, no. 4, april 1996.

Chapter 2. $\mathcal{L}^2$-Disturbance Attenuation for Linear Systems with Bounded Controls: An ARE-Based Approach

R. Suárez[1], J. Alvarez-Ramírez[1], M. Sznaier[2], C. Ibarra-Valdez[1]

[1] División de Ciencias Básicas e Ingeniería, Universidad Autónoma Metropolitana-Iztapalapa, Apdo. Postal 55-534, 09340 México D.F, México, Tel. 724-46-58, Fax 7-24-46-53. email:rsua@xanum.uam.mx

[2] Department of Electrical Engineering, The Pennsylvania State University. University Park, PA 16802. email:msznaier@frodo.ee.psu.edu

1. Introduction

Feedback stabilization of systems subject to bounded inputs has been a long–standing problem in control theory. In the case where the plant itself is linear, most approaches fall within one of the two following categories: i) Saturating a linear controller designed ignoring the control bounds and analyzing the properties of the resulting closed–loop system to establish whether or not it satisfies the performance requirements; and ii) Designing new types of nonlinear controllers, trying to account for the saturation constraint in the design process.

For linear controls with saturation, sufficient conditions for global or semiglobal stabilization, and the qualitative characterization of the region of attraction of the origin, has been studied from different viewpoints (see for instance [1, 3, 7, 14, 15, 25]). While this technique provides a simple, intuitively attractive way of handling saturation constraints, it has been shown that it might fail to find a stabilizing controller, even when one does exist. For instance, the nth-order integrator ($n \geq 3$) is an example of a linear system which can be globally stabilized using a continuous bounded function [24], but can not be globally stabilized by means of a saturated linear feedback [7]. Various approaches to the problem of feedback stabilization of linear systems by means of nonlinear controllers, different from a linear saturated controller, have been considered. Sussmann et al. [28] generalized a result due to Teel [32], prove that controllable linear systems with no open-loop eigenvalues with positive real part (**NPRPE** systems) can be globally stabilized by means of linear combinations and nested compositions of saturations of linear feedbacks. Lyapunov rather than finite-time stability can be obtained using two design techniques originally intended to obtain continuous approximations to time optimal control laws [26]. Komarov [12] proposed an approach based upon an inner estimation of the attainability sets by means of ellipsoids, leading to a feedback control law obtained by solving a matrix differential equation. Although, Komarov's control is a global stabilizer for the nth-order

and alternative computer architectures. The results to date make it possible to consider enumeration of 10 million cosets in the Lyons-Sims group. Such enumerations will use a very large amount of memory, and thus a shared memory architecture will be required as much for cost effectiveness as for speed.

7 Acknowledgements

We acknowledge the provision of time by Boston University's MARINER project on the SGI Power Challenger Array, where this work was done. We also gratefully acknowledge use of Martin Schönert's coset enumeration program.

References

[1] S. Akl, G. Labontè, M. Leeder and K. Qiu, "On doing Todd-Coxeter coset enumeration in parallel", *Discrete Applied Math.* 34: 27–35 (1991).

[2] R.P. Brent, "The Parallel Evaluation of General Arithmetic Expressions", *J. of ACM* 21: 201–208 (1974).

[3] John J. Cannon, Lucien A. Dimino, George Havas and Jane M. Watson, "Implementation and analysis of the Todd-Coxeter algorithm", *Math. Comput.* 27: 463–490 (1973).

[4] G. Cooperman, "STAR/MPI: Binding a Parallel Library to Interactive Symbolic Algebra Systems", *Proc. International Symposium on Symbolic and Algebraic Computation (ISSAC '95)*, ACM Press, 1995, pp. 126–132.

[5] G. Cooperman, "TOP-C: A Task-Oriented Parallel C Interface", 5^{th} *International Symposium on High Performance Distributed Computing* (HPDC-5), IEEE Press, 1996, pp. 141–150.

[6] G. Cooperman, L. Finkelstein, B. York and M. Tselman, "Constructing Permutation Representations for Large Matrix Groups", *Proc. International Symposium on Symbolic and Algebraic Computation (ISSAC '94)*, ACM Press, 1994, pp. 134–138.

[7] A. Diaz, E. Kaltofen, K. Schmitz and T. Valente, "A System for Distributed Symbolic Computation", *ISSAC'91 (Proceedings of the 1991 International Symposium on Symbolic and Algebraic Computation)*, ACM Press, 1991, pp. 323–332.

[8] H.W. Gollan, *A new existence proof for Ly, the sporadic simple group of R. Lyons*, Preprint 30, Institut für Experimentelle Mathematik, Universität GH Essen, 1995.

[9] George Havas, "Coset enumeration strategies", *ISSAC'91 (Proceedings of the 1991 International Symposium on Symbolic and Algebraic Computation)*, ACM Press, 1991, pp. 191-199.

[10] George Havas and M.F. Newman, "Applications of computers to questions like those of Burnside", *Burnside Groups*, Lecture Notes in Math. 806, Springer, 1980, pp. 211–230.

[11] N. Kajler, "CAS/PI: a Portable and Extensible Interface for Computer Algebra Systems", *Proc. of Internat. Symp. on Symbolic and Algebraic Computation (ISSAC-92)*, ACM Press, 1992, pp. 376–386.

[12] John Leech, "Coset enumeration", *Computational Group Theory*, Academic Press, 1984, pp. 3–18.

[13] J. Neubüser, "An elementary introduction to coset-table methods in computational group theory", *Groups — St Andrews 1981*, London Math. Soc. Lecture Note Ser. 71, Cambridge University Press, 1984, pp. 1–45.

[14] M. Schönert *et al.*, GAP – *Groups, Algorithms and Programming.* Lehrstuhl D für Mathematik, RWTH, Aachen, 1996.

[15] C.C. Sims, "The Existence and Uniqueness of Lyons Group", *Finite Groups '72*, North-Holland, 1973, pp. 138–141.

[16] C.C. Sims, *Computation with finitely presented groups*, Cambridge University Press, 1994.

[17] 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).

[18] Michael Vaughan-Lee, "Engel-4 groups of exponent 5", *Proc. London Math. Soc.* (to appear).

[19] D.A. Wood and M.D. Hill, "Cost-effective parallel computing", *IEEE Computer* 28: 69–72 (1995).