Parallel coset enumeration using threads

George Havas and Colin Ramsay
Centre for Discrete Mathematics and Computing,
Department of Computer Science and Electrical Engineering,
The University of Queensland, Queensland 4072, Australia.
Email: {havas,cram}@csee.uq.edu.au

Abstract

Coset enumeration is one of the basic tools for investigating finitely presented groups. Many enumerations require significant resources, in terms of CPU time or memory space. We develop a fully functional parallel coset enumeration procedure and we discuss some of the issues involved in such parallelisation using the POSIX threads library. Our results can equally well be applied to any master-slave parallel activity exhibiting a medium level of granularity.

1 Introduction

Coset enumeration is long established as a technique for the investigation of finitely presented groups. It takes a finitely presented group and a finitely presented subgroup as input, and attempts to find the index of the subgroup in the whole group. In principle, it will succeed whenever this index is finite. Coset enumeration was used well before the days of electronic computers, apparently first by Moore [11], and was popularised by Todd and Coxeter [15]. The first computer implementation was that of Haselgrove in 1953. This, along with other early implementations, is described by Leech [8]. Detailed accounts of the techniques used in coset enumeration can be found in [3, 5, 9, 12, 14].

Coset enumeration builds a table giving the action of each group generator and inverse on each coset. New cosets are defined to fill gaps in the coset table (traditionally called the Felsch strategy) or in the relator tables (traditionally known as the HLT strategy), and definitions are tested against the relators to find any coincidences (two distinct cosets are equal) or deductions (coset table entry is forced) consequent on them.

For many enumerations, Todd-Coxeter coset enumeration is straightforward but takes significant time. This is primarily due to the fact that to keep the number of cosets under control we fully explore the consequences of all definitions immediately, and most of the processing time is spent scanning each coset table entry at all essentially different positions in each relator. Much of this time is fruitless, in the sense that most such scans yield neither a deduction nor a coincidence. Previous work [4] has established that it is possible to process deductions in parallel, and our parallel enumerator, PACE, was developed specifically for those enumerations where the bottleneck is deduction processing.

PACE [13] was developed from the enumerator ACE [6], which itself is an enhanced version of the enumerator described in [5]. (A different approach to performing coset enumerations in parallel is discussed in [1].) Our aims were to provide a fully functional parallel coset enumerator including all modern capabilities. In particular, we wanted to allow a complete range of enumeration strategies, as provided in ACE, in distinction to the purely Felsch-type enumerator of [4]. We also wished to more carefully study the behaviour of parallel coset enumeration, taking the cost of issues such as synchronisation into account.

The obvious motivation for parallelising coset enumeration is to decrease the running time, and so produce results more quickly. Somewhat less obviously, it also allows a larger number of enumerations to be undertaken in a given time. The presentation pruning technique for removing redundant relators mentioned in Section 5 is an example of where this latter idea is useful. An alternative rationale for parallelisation is to note that in multiprocessor machines not only have the available processors to be shared between the competing processes, but the available memory must also be shared, as discussed in [17]. Many of the enumerations in which we are interested require large amounts of memory. Parallelising such enumerations allows us to reduce their time-memory cost, and their impact on other users of the system. Even if we cannot achieve a speed-up which is linear in the number of processors, a request to use 3/4 of the machine's memory and 20 of its CPUs overnight is more likely to be favourably received than a request to use 3/4 of the memory and 1 CPU for four days.

PACE is intended for shared-memory multiprocessor systems and, for maximum generality, we implemented the concurrency using the POSIX `pthread` library (see, for example, [2]). Parallelising ACE is straighforward, with a simple master-slave paradigm sufficing. The master is responsible for making all table entries and for processing deductions and coincidences. The master makes and stacks table entries, but does not otherwise process them. When sufficient table entries are stacked, they are divided amongst the available slaves, and the slaves perform the required scans. The scans which yield coincidences or deductions are stacked and passed back to the master. The master reruns these scans, processing the coincidences and deductions, and then returns to making table entries.

PACE operates in a multi-threaded SMP (symmetric multiprocessor) environment, so there is only a single copy all the enumerator's data structures. In particular, the coset table (which can be very large), is shared by the master and all the slaves. Only the master changes the coset table, with a slave's accesses to the table being read-only. Since deductions cause only localised table changes (unlike coincidences, which can change arbitrarily large portions of the table), we could allow the slaves to process them. However, we would still have to record them and pass them back to the master to detect and resolve conflicting deductions (i.e., coincidences). So this approach is unlikely to be faster, and is considerably more complicated.

Our enumerators have a large number of parameters, with a wide choice of settings. The important ones for our examples are `ct`, `rt` and `no`. The parameters `ct` and `rt` indicate the relative weightings attached to defining new cosets using the Felsch strategy and the HLT strategy, respectively. The parameter `no` controls the number of the relators treated as subgroup generators, and scanned and closed again coset 1 (i.e., the subgroup) at the start of the enumeration (see [5]). For PACE, the n and p parameters control the concurrency (number of threads) and parallelism (average number of active threads), respectively (see Section 2).

In all the tests discussed, we used the native compiler (`cc`) of the system, and compiled the code with level 2 optimisation. Since maximum code portability was one of our aims we restricted ourselves, insofar as was possible, to ANSI C and the POSIX routines, and made no use of any system-specific C extensions or thread handling routines. The metrics we use in our tests are: T, the total number of cosets defined during an enumeration; M, the maximum number of cosets active at any time during an enumeration; the elapsed, or wall, time for an enumeration; the aggregated CPU time for all threads.

In Section 2 we discuss some of the factors which need to be considered when parallelising coset enumeration and when running performance tests, while in Section 3 we propose a model for the running time of our parallelisation. In Section 4 we use a small example to illustrate our discussion and model, while in Section 5 we discuss PACE's behaviour on a genuine enumeration. Finally, in Section 6, we discuss what conclusions we can draw from our work. Note that, although our discussion throughout is in terms of parallelising the Todd-Coxeter coset enumeration procedure, it could also be applied to any medium granularity master-slave process.

2 Synchronisation, scheduling and statistics

Although our parallelisation strategy is straightforward, it is still necessary to code the master-slave handovers using the POSIX synchronisation primitives to ensure correct behaviour. As we will see, given the granularity of our parallelisation and the cost of the synchronisation primitives, this code has a major effect on the performance of PACE. Given the variation between platforms, there does not seem to be any general best method, so a choice of methods is provided.

The easiest method is simply to create new threads as required at the master-slave handovers, and to wait until they have all terminated before resuming the master. This method is conceptually appealing and, given that the overheads of threads are much lower than those of processes, need not be uncompetative with the other methods. A second method is to use the traditional synchronisation primitive of semaphores, which allows control to passed back and forth between the master and the slaves. A third method is to use the `pthread` mutexes (mutual exclusion locks) to simulate a symmetric barrier, at which all threads must wait until all are ready to proceed. Note that in the second and third methods, inactive threads are blocked waiting for an event. So, in all three methods, the concurrency switches regularly between one (the master is active and the slaves blocked or not present) and n (the master is blocked and the slaves active). (In practice, we can use the master as one of the slaves.)

This concurrency variation can cause problems with some schedulers, which use the (average) concurrency to allocate an appropriate number of processors. Our final synchronisation methods keeps all threads continuously busy, and the concurrency fixed at n, by spin-locking on memory flags (protected by mutexes, for correctness). Although profligate, this method is potentially very fast, provided enough processors are available. However, its performance will degrade rapidly if not enough processors are allocated, unlike the first three methods, which degrade gracefully.

The POSIX `pthreads` standard does not mandate how threads are to be implemented, nor does it specify the mapping between threads and processes/processors. These may be done at the kernel or user level, or at both levels. This makes scheduling and priority control very system dependent, and means that considerable fine-tuning may be necessary for some applications. In our case we require that threads actually run in parallel on the specified number of processors. To ensure this on all systems, we set the scheduling scope to `SYSTEM`, instead of its default of `PROCESS`. Other than this, we use the defaults for all the thread control parameters.

Both ACE and PACE incorporate code to gather statistics and to output progress messages. Parallelisation can magnify the impact of such code on the running time, particularly if significant data collection is done in the non-parallel portion of the code. In general, if accurate testing or maximum speed-up is required, statistics and messaging code should be omitted. Results can also be distorted by the system's handling of memory. The first access to a location can cause a page fault and an allocation of physical memory to a virtual address. If a large coset table is being built as part of a series of test runs, then the first of the series can be measurably slower due to this overhead.

3 Modelling the running time

The traditional model for master-slave parallel applications is to divide the processing time into a serial portion (Ser) and a parallel portion (Par), and to assume that the elapsed time (Wall) scales in the obvious manner with the number of CPUs (n). This yields the model:

$$\text{Wall} = \text{Ser} + \frac{\text{Par}}{n}\,.$$

This is the model used in [4]. It yields good results if the granularity of the parallelisation and the synchronisation times are such that the master-slave synchronisation overheads can be ignored. It also suffices if this overhead is independent of n or if it varies as n but can be parallelised; in these

cases, it simply increases Ser or Par. In PACE the parallelism is of medium granularity, and the synchronisation overhead cannot be ignored. In particular, the overhead varies with the number of threads, and part of it must be serialised in the master. We model this by assuming that each master-slave synchronisation takes Sync time over the course of the enumeration, and that there are $n-1$ synchronisations per master-slave cycle (since the master can also act as one of the n slaves). This yields the model:

$$\textsc{Wall} = \textsc{Ser} + (n-1)\textsc{Sync} + \frac{\textsc{Par}}{n}\,.$$

Note how Wall no longer tends to Ser as n is increased. Instead, it is now unbounded as n increases; there may or may not be a minimum for some intermediate value of n, depending on the relative sizes of Sync and Par. Note that the values of Ser, Sync and Par are determined by the given presentation, the chosen enumeration strategy, the platform being used, and the choice of synchronisation primitive. We have no direct control over their values, and predicting (or measuring) them accurately is difficult.

This model applies for a fixed number of master-slave cycles. However, in PACE the number of cycles is implicitly determined by a *parallelisation parameter* (p), which controls the number of table entries which are batched by the master before being processed by the slaves (that is, it sets the *work per cycle*). As p is increased, the number of master-slave cycles over the course of the enumeration drops proportionally. To model this, the second term in the above model should be divided by p. However, as p is increased we are deferring the processing of more and more data, for longer and longer. This causes the number of cosets T to change, altering Wall. (Usually, but not necessarily, T will increase.) If we assume that the amount of work performed is proportional to T, then we can model the effect of T's growth with p by multiplying the right hand side of our model by some function of p, say $g(p)$. This yields our final model:

$$\textsc{Wall} = g(p)\left(\textsc{Ser} + \frac{(n-1)\textsc{Sync}}{p} + \frac{\textsc{Par}}{n}\right).$$

There is no a priori way to decide on the form of $g(p)$. It varies widely, depending on the presentation and the enumeration stategy. Small values of p often yield only modest growth in T, while larger values can increase T dramatically (see Section 4). Now the derivation of our model is a trifle ad hoc, and we do not expect it to be particularly accurate quantitatively, especially for extreme values of n or p. However, extensive testing of PACE shows that it is qualitatively valid, and does capture the essential features of its behaviour. In particular, if one of n or p is fixed and the other varied from a low to a high value, then typically the running time first decreases and then increases again, displaying a definite minimum.

4 The group J

The authors of [4] used a group drawn from the paper by Vaughan-Lee [16] as their test case. However, their example is a little small for our purposes, and another group from [16] suits our purpose better. As part of his proof that 2-generator Engel-4 groups of exponent 5 are finite, Vaughan-Lee considers the group J as presented in Figure 1; note that upper-case letters indicate generator inverses. The subgroup $\langle a, a^b\rangle$ has index $5^6 = 15625$ in J, and a coset enumeration is used to establish this as part of the proof.

This presentation for J has 51 relators and, after expanding and reducing the Engel words, has a total length of 1349. An extensive investigation of how best to perform this enumeration is described in [10]. Best results are achieved with a high value of `ct`, a low value of `rt`, and a moderate value of `no`. For J, the default settings of PACE yield `ct` and `rt` values of 1000 and 2 respectively which, while not optimal, are suitable for our tests. The default for `no` is to include all the relators, so we modify this to 8, to include only the shortest eight relators.

FIGURE 1: The presentation for J

```
(a)^5, (b)^5, (ab)^5, (aab)^5, (abb)^5, (aabb)^5, (Abb)^5, (aB)^5, (aaB)^5,
(aaBB)^5, (aabaabb)^5, (aabaaB)^5, (aabaB)^5, (aabbaBB)^5, (aabbAAB)^5,
(aabAAb)^5, (aabAAbb)^5, (aabAB)^5, (aaBaaBB)^5, (abaab)^5, (abaaB)^5, (ababb)^5,
(abaB)^5, (abbaaBB)^5, (abbaBB)^5, (abbAbb)^5, (abbAB)^5, (abbABB)^5, (abAb)^5,
(abAbb)^5, (abAB)^5, (AbaaB)^5, (Ababb)^5, (AbbaaBB)^5, (AbbaB)^5, (AbbAAbb)^5,
(AbAbb)^5, (AAbaabb)^5, (AAbbaaB)^5, [a,b,b,b,b], [A,b,b,b,b], [a,B,B,B,B],
[A,B,B,B,B], [b,a,a,a,a], [B,a,a,a,a], [b,A,A,A,A], [B,A,A,A,A],
[a,ab,ab,ab,ab], [a,AB,AB,AB,AB], [b,ab,ab,ab,ab], [b,AB,AB,AB,AB]
```

To illustrate the effects of the synchronisation method, we used a SUN SPARCstation-20 system; the two processors were SPARCs, clocked at 60 MHz, and 384 MB of physical memory was available, while the OS was SunOS 5.5. Our tests used an executable with statistics and messaging turned off, and with the low value of $p = 16$ to enhance the effects of synchronisation. Overnight, when the system was not busy, we ran tests for $n = 1$ (no synchronisation overheads), $n = 2$ (system fully loaded), and $n = 3$ (system overloaded). The aggregated CPU times and elapsed times, in seconds, are recorded in Table 1.

All the figures for $n = 1$ are essentially the same, as we would expect, and give the actual processing time required to perform the enumeration. For the first three methods, the rise in CPU time with n reflects the impact of the synchronisation primitives. This rise depends strongly on the particular platform and its loading, as does the relative ranking of the three methods. With $n = 2$, we achieve a speed-up of around $\times 1.7$ (based on WALL times), while for the overloaded case of $n = 3$, we achieve a speed-up of around $\times 1.35$. For the mutex controlled spinning method, the jump in CPU time for $n = 2$ is much more pronounced, while the WALL time is roughly half the CPU time; as we would expect in both cases. The mutex results for $n = 3$ are *very* variable, and those given in Table 1 just illustrate the sort of degradation with this method when the system is overloaded.

TABLE 1: Synchronisation effects in J

	$n = 1$		$n = 2$		$n = 3$	
sync method	CPU	WALL	CPU	WALL	CPU	WALL
new threads	523.58	524.31	559.40	314.35	587.41	396.06
semaphores	523.89	524.63	546.06	305.42	565.66	379.96
barrier	524.79	525.21	551.13	303.70	568.83	378.45
mutex/spin	524.10	525.06	601.91	307.95	2154.55	12784.90

To test our model for the running time, we used a SUN Ultra-Enterprise system; the seven processors were ultraSPARCs, clocked at 250 MHz, and 3.5 GB of physical memory was available, while the OS was SunOS 5.6. The executable we used had statistics and messaging turned off, and synchronisation was via semaphores. The test runs were undertaken when the system usage was very low (after a reboot), and the number of processors requested was limited to five, to prevent processor availability interfering with the results. The elapsed times, in seconds, and the total number of cosets defined, T, for a range of p and n values are given in Table 2.

Note the presence of the behaviour discussed at the end of Section 3. So our model is valid, at least qualitatively, for enumerations in J. If we divide T by the running time for $n = 1$ (when there are no slave threads), then all the values obtained lie in the range 6774–7198. So our assumption that the amount of work performed is proportional to T is reasonable. This implies that, if $g(p)$ displays modest growth with p, we would expect the row and column minima to move to the right and down, respectively, and decrease as p and n are increased. We can observe this behaviour in Table 2, up to the point where there is a sudden large increase in T.

TABLE 2: Wall times and total cosets for J

	n					
p	1	2	3	4	5	T
1	133	177	212	275	366	935095
2	134	115	132	166	206	935104
4	134	99	98	112	155	935131
8	135	89	81	89	108	935761
16	135	83	71	71	82	936430
32	135	80	63	57	66	936829
64	135	78	60	52	56	937314
128	136	77	58	49	48	937916
256	136	77	58	47	45	939245
512	143	80	59	48	45	968672
1024	198	109	76	61	59	1425151
2048	199	109	77	61	57	1429317

5 The Lyons simple group

Although suitable as a small test case, J is not representative of the enumerations for which PACE is intended. One of the first real enumerations to which PACE was applied was the Lyons simple group, Ly. Ly has order 51765179004000000, and a presentation for it is described in [7]. From this, a presentation on the generator set $\{a, b, c, d, e, z\}$, suitable for coset enumeration, can be produced. This presentation has 53 relators, of total length 887, and an index 8835156 coset enumeration of the subgroup $\langle a, b, c, d, e\rangle$ (that is, $G_2(5)$) is part of the work described in [7]. For our experiments with this presentation of Ly, we used an SGI Origin 2000 machine; the 64 CPUs were MIPS R10000 chips, clocked at 195 MHz, and 16 GB of physical memory was available, while the OS was IRIX64 6.5. Using Felsch-type methods with ACE, the enumeration completed in 13 hr 41 min, with a maximum coset count equal to the index and a total coset count of 11849288. As e is the only involution among the generators, the completed coset table requires 371 MB of memory.

Ly is mentioned as a possible target for parallelisation in [4], and [7] reports that this has been done and an approximately linear speed-up achieved. Prior to testing PACE with the enumeration of $G_2(5)$ in Ly, we pruned the presentation. As noted in [7], it is easy to eliminate either e or d from the presentation, but this greatly increases its length. An alternative approach is to note that some of the relators which contain z (that is, which are not responsible for defining $G_2(5)$) are redundant. A simple greedy algorithm can be used to eliminate relators which are not formally required and whose removal does not render the coset enumeration substantially more difficult.

FIGURE 2: The reduced presentation for Ly

```
(e)^2, (Da)^2, (az)^2, (d)^4, BDbd, zzAA, (c)^5, (b)^5, AABaab, ddAAAA, AAcaacc,
ececBBC, (ab)^4, ZbaBzaabaB, BBcbceceBCC, ebcBebCBBcB, ebebccBBcbC, ACabCBAcabcB,
(ABab)^3, ZBcbzABACaba, ZbcBzBAbCBab, CACaCaCAccaCA, ABcbabCBCBccb, ZAcazBCBaCAbb,
ZczABcAbcBaba, BCbcACaCBcbAca, (eabbaBA)^2, eBcbeabcBcACBB, ZbbaBBzabbABBA,
BCbAcaBcbCACac, CACacaCAcACaCACa, CAcacbcBCACacbCB, CaCAcacAcaCACacA,
cacbcBACCaCbCBAc, CBCbcBcbCCACACaca, BAcabbcBcbeCebACa, ACacAcaebcBCBcBCe,
eDCDbABCDACaCAcabb, BCbbCBBcbbACacAcaB, eaaebcBACaCAcbABBA, BACaCAcacbCACacAca,
AbcBabCBCbCBBcbAcaC, BBabbzBBabbzBBabbzaa, (aabaBab)^3, eAcaeACaeAcabCBaBcbaaa,
deBAceAeACACacAcabcBcbaBBA, dCACacAcadACaCAcacdCACacAcA, (dCACacAca)^5,
ZDZACDbABCDbABCBaBBaBzdzabAbbAbbCacbcBdcBBdcbc
```

One such reduction yields the presentation given in Figure 2, which has 49 relators and total length 692. Using ACE, this enumeration takes 8 hr 4 min. The maximum number of cosets is unchanged, while the total increases slightly to 11922259. Since the running time for a large enumeration such as this is dominated by the time to scan each coset table entry at each essentially different position in the relators, and since the coset table is initially empty and fills up as the enumeration progresses, then we would expect that the running time is quadratic in the total length of the presentation.

Our results support this heuristic argument, with $(692/887)^2 = 0.61$ and $484/821 = 0.59$. So even a modest reduction in presentation length can be significant.

Before running PACE with more than one thread, two test runs were done to assess the potential of Ly for parallelisation. The first of these used a test mode which is essentially the same as ACE, except that deductions are handled via a function, instead of as inline code. This mode also gathers detailed statistics regarding deduction handling. A batch size of 1 (that is, $p = 1$) was used, so the enumeration was Felsch-type. In this run, the maximum and total counts were unchanged, but the running time increased by about 5 min compared to ACE. A total of 56099439 deductions were stacked, and 56038295 survived to be processed (the remainder referenced cosets which had become redundant). These surviving deductions generated 6162825103 scans, which yielded 38178487 deductions and 1775940 primary coincidences. So only 0.65 % of the scans are actually necessary. Furthermore, all but approximately 5 min of the running time was spent processing deductions.

Our second test run was a 'parallel' run with only one thread (that is, $n = 1$); since the master also acts as one of the slaves, there are no thread management overheads. This allows the effect of deferring the processing of the necessary deductions to be measured. The maximum and total counts were again unchanged, but the running time increased further, to 8 hr 18 min. A total of 56100949 deductions were stacked, and the slave stacked 166598619 scans as necessary. Of these, 166555503 survived and were rerun by the master, generating 38179085 deductions and 1775438 primary coincidences. Note how the number of scans rerun by the master is more than four times those necessary. This implies that the deferral of processing the deductions and primary coincidences causes them to be detected and stacked multiple times. Despite this, the number of scans done by the master, which represent non-parallelisable work, is only 2.7 % of those of the first test.

Table 3: Cosets and times for Ly

p	M	T	Wall	CPU	C/W
16	8835170	12325413	40578.38	601895.32	14.83
			54644.80	781914.67	14.31
32	8835184	12530641	22919.57	339870.83	14.83
			24420.89	363262.50	14.88
64	8835211	12824890	18783.72	279760.92	14.89
			22920.13	305294.59	13.32
128	8835253	13175662	11515.14	168930.21	14.67
			12419.61	184755.18	14.88
256	8835375	13518860	9332.24	139714.90	14.97
			13257.53	188774.28	14.23
512	8835423	13828854	8269.79	125297.98	15.15
			10399.50	154770.21	14.88
1024	8835774	14091169	7892.05	120709.02	15.30
			8707.73	130594.01	15.00
2048	8836557	14298984	7859.57	121675.08	15.48
			8769.50	130901.05	14.93
4096	8839151	14476620	7481.54	114611.88	15.32
			8209.20	123945.94	15.09
8192	8842913	14645976	7537.96	118803.84	15.76
			7863.96	119312.80	15.17

When running PACE with more than one thread we chose to illustrate the behaviour observed on a real system, instead of the more usual, but highly artificial, situation of a very lightly loaded system. The data in Table 3 was gathered from a series of runs with $n = 16$ and with synchronisation via mutex spinning. Recall that p is the deduction batching factor, while M and T are the maximum and total number of cosets defined. Note how M increases slightly above the index, due to the deferral of deduction processing, but that the increase is always less than p, as we would expect. The value of T increases more strongly with p, but does not exhibit any serious blow-out. Wall and CPU are the elapsed time and aggregated CPU time, in seconds, while the last column gives

the ratio of CPU to WALL. The C/W figure could be regarded as a quality measure for the run, since it indicates its *average* parallelism, as opposed to its concurrency (fixed at 16).

Several runs for each value of p were made, and Table 3 records the details of the best and worst WALL times seen. We made no attempt to undertake all our runs when the machine was lightly loaded. In fact, given the not insignificant time and memory requirements of an enumeration in Ly, this would not have been possible without making special arrangements. Over the course of our runs, the machine loading varied unpredictably, from essentially zero (just after a reboot) to overloaded (in terms of CPUs or memory, or both). Although users of the Origin 2000 are encouraged to run their jobs via the batch system, we chose to run our tests interactively. The batch system scheduler suspends and resumes jobs, and swaps them in and out of memory, as the system loadings vary, and this can grossly distort a run's timings. Running interactively does, however, have the effect of favouring our tests over the batch jobs. As the system was, in general, more heavily loaded with batch jobs than with interactive jobs, our test results are therefore a little biased (i.e., we got more parallelism than we deserved!).

Our choice of n and synchronisation method is aggressive, in the sense that the runs are potentially very fast, but the performance will degrade rapidly as the system load increases. If the parallelism is close to the concurrency, the slaves spend minimal time spinning, so CPU time is low and WALL approximates CPU/16. If we are allocated less CPUs as the system becomes busier, then the threads spend longer spinning waiting for processors, so the CPU time increases. This increases WALL, with the blowout being exacerbated by the increased CPU time being spread over a smaller number of processors. Although blurred by other factors, this sensitivity of WALL to the parallelism is evident in Table 3. Note also the slight tendency for C/W to increase with p, due to the changing balance between PAR and SER and the smaller number of master-slave cycles.

It might seem unfair to assess PACE in this manner but in practice we have to run our programmes on machines with variable and unpredictable loading, and we would like to obtain the programme's results as quickly as possible. Given the sensitivity of WALL to the parallelism, and the usual timing variations encountered in multiuser systems, the difference between our best and worst results can be quite large. Consequently, it is difficult to find the best value of p, and to verify our model, without performing a large number of tests.

6 Conclusions

We have developed a fully functional parallel coset enumeration procedure. Our implementation is reasonably portable, based on ANSI C and POSIX threads. It gives appropriate speed-up, so that difficult enumerations can be completed significantly faster than serially.

Our results demonstrate that for master-slave parallelisations of medium granularity the cost of the synchronisation primitives cannot be ignored, even for small numbers of processors. Simply increasing the concurrency and expecting (the parallel proportion of) the running time to decrease is naive, and is often counterproductive. It is possible to achieve a significant speed-up, but this can be well below that implied by the number of processors employed, and it is critically dependent upon the control parameters used.

Our model of PACE's running time and our empirical experience suggest that, all other things being equal, there is an n/p combination which yields minimum running time. However, all other things are usually *not* equal. Variations in system loading interact strongly with the enumeration parameters and the POSIX primitives used, so the repeatability of results is low. And, of course, each presentation must be considered individually. So it is questionable how relevant it is to know how best to do a particular enumeration. It is satisfying to have this information but, in the absence of the ability to use this data to accurately predict how best to undertake other enumerations on other platforms, its value is moot. However, it does provide some guidance on how best to proceed when presented with a new enumeration.

Acknowledgements: The work of both authors was supported by the ARC. We are grateful to the University of Queensland's High Performance Computing Unit for access to the SGI Origin 2000, and to the Department of Mathematics at the University of Auckland for access to the SUN Ultra-Enterprise.

References

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

[2] David R. Butenhof. *Programming with POSIX Threads*. Addison-Wesley, 1997.

[3] John J. Cannon, Lucien A. Dimino, George Havas, and Jane M. Watson. Implementation and analysis of the Todd-Coxeter algorithm. *Mathematics of Computation*, 27(123):463–490, July 1973.

[4] Gene Cooperman and George Havas. Practical parallel coset enumeration. In G. Cooperman, G. Michler, and H. Vinck, editors, *Workshop on High Performance Computing and Gigabit Local Area Networks*, Lecture Notes in Control and Information Sciences, 226, pages 15–27. Springer-Verlag, 1997.

[5] George Havas. Coset enumeration strategies. In Stephen M. Watt, editor, *ISSAC'91 (Proceedings of the 1991 International Symposium on Symbolic and Algebraic Computation)*, pages 191–199. ACM Press, 1991.

[6] George Havas and Colin Ramsay. Coset enumeration: ACE version 3, 1999.
Available as `http://www.csee.uq.edu.au/~havas/ace3.tar.gz`.

[7] George Havas and Charles C. Sims. A presentation for the Lyons simple group. In P. Dräxler, G.O. Michler, and C.M. Ringel, editors, *Computational Methods for Representations of Groups and Algebras*, Progress in Mathematics, 173, pages 241–249. Birkhäuser, 1999.

[8] J. Leech. Coset enumeration on digital computers. *Proceedings of the Cambridge Philosophical Society*, 59:257–267, 1963.

[9] John Leech. Coset enumeration. In Michael D. Atkinson, editor, *Computational Group Theory*, pages 3–18. Academic Press, 1984.

[10] Tom Litt. Coset enumeration. MSc thesis, Mathematical Institute, University of Oxford, 1999.

[11] E.H. Moore. Concerning the abstract groups of order $k!$ and $\frac{1}{2}k!$ holohedrically isomorphic with the symmetric and the alternating substitution-groups on k letters. *Proceedings of the London Mathematical Society* (1), 28:357–366, 1897.

[12] J. Neubüser. An elementary introduction to coset-table methods in computational group theory. In *Groups – St. Andrews 1981*, London Mathematical Society Lecture Note Series 71, pages 1–45. Cambridge University Press, 1984.

[13] Colin Ramsay. PACE 1.000: Parallel ACE. Technical Report #17, Centre for Discrete Mathematics and Computing, The University of Queensland, 2000.

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

[15] J.A. Todd and H.S.M. Coxeter. A practical method for enumerating cosets of finite abstract groups. *Proceedings of the Edinburgh Mathematical Society*, 5:26–34, 1936.

[16] Michael Vaughan-Lee. Engel-4 groups of exponent 5. *Proceedings of the London Mathematical Society* (3), 74:306–334, 1997.

[17] David A. Wood and Mark D. Hill. Cost-effective parallel computing. *IEEE Computing*, 28(2):69–72, 1995.