The Low-Index Subgroups Algorithm

Approaches to parallelisation in HPC-GAP

Michael Torpey

University of St Andrews

22nd August 2013

The question

Given a finitely presented group $G = \langle X|R\rangle$, what are its subgroups of index no more than N?

- $X =$ A set of generators, e.g. $\{a, b\}$.
- $R =$ A set of relators, e.g. $\{a^2, b^3, (ab)^5\}$ such that $a^2 = b^3 = (ab)^5 = 1$.
- $G = \langle a, b|a^2 = b^3 = (ab)^5 = 1\rangle \cong A_5$

The algorithm

- "Forced coincidence" approach
- Utilises Todd-Coxeter method for coset enumeration
- Expand coset table defining no more than n cosets, for some $n \geq N$.

Coset enumeration

Todd-Coxeter algorithm:

	a	a^{-1}	b	b^{-1}
$H = 1$				

Coset enumeration

Todd-Coxeter algorithm:

	a	a^{-1}	b	b^{-1}
$H = 1$	2			
$Ha = 2$		1		

- SET $1^a = 2$

Coset enumeration

Todd-Coxeter algorithm:

	a	a^{-1}	b	b^{-1}
$H = 1$	2	2		
$Ha = 2$	1	1		

- SET $1^a = 2$
- SCAN a^2 on coset 1:
  $1 \xrightarrow{a} 2 \xrightarrow{a} 1$

Coset enumeration

Todd-Coxeter algorithm:

	a	a^{-1}	b	b^{-1}
$H = 1$	2	2	3	
$Ha = 2$	1	1		
$Hb = 3$				1

- SET $1^a = 2$
- SCAN a^2 on coset 1:
  $1 \xrightarrow{a} 2 \xrightarrow{a} 1$
- SET $1^b = 3$

Coset enumeration

Todd-Coxeter algorithm:

	a	a^{-1}	b	b^{-1}
$H = 1$	2	2	3	4
$Ha = 2$	1	1		
$Hb = 3$				1
$Hb^{-1} = 4$			1	

- SET $1^a = 2$
- SCAN a^2 on coset 1:
  $1 \xrightarrow{a} 2 \xrightarrow{a} 1$
- SET $1^b = 3$
- SET $1^b = 4$

Coset enumeration

Todd-Coxeter algorithm:

	a	a^{-1}	b	b^{-1}
$H = 1$	2	2	3	4
$Ha = 2$	1	1		
$Hb = 3$			4	1
$Hb^{-1} = 4$			1	3

- SET $1^a = 2$
- SCAN a^2 on coset 1:
  $1 \xrightarrow{a} 2 \xrightarrow{a} 1$
- SET $1^b = 3$
- SET $1^b = 4$
- SCAN b^3 on coset 4:
  $4 \xrightarrow{b} 1 \xrightarrow{b} 3 \xrightarrow{b} 4$

Coincidences

Sometimes we may encounter a coincidence.
Example:

	a	a^{-1}
1	2	
2	3	1
3		2

- SCAN a^2 on coset 1
- $1 \xrightarrow{a} 2 \xrightarrow{a} 3$
- But we should have $1 \xrightarrow{a}\xrightarrow{a} 1$
- Hence 1 and 3 describe the same coset, and they can be combined

Coincidences

Sometimes we may encounter a coincidence.
Example:

	a	a^{-1}
1	2	2
2	~~3~~ 1	1
~~3~~		~~2~~

- SCAN a^2 on coset 1
- $1 \xrightarrow{a} 2 \xrightarrow{a} 3$
- But we should have $1 \xrightarrow{a}\xrightarrow{a} 1$
- Hence 1 and 3 describe the same coset, and they can be combined

Coincidences

Sometimes we may encounter a coincidence.
Example:

	a	a^{-1}
1	2	2
2	1	1

- SCAN a^2 on coset 1
- $1 \xrightarrow{a} 2 \xrightarrow{a} 3$
- But we should have $1 \xrightarrow{a}\xrightarrow{a} 1$
- Hence 1 and 3 describe the same coset, and they can be combined

Forcing a coincidence

- Eventually we cannot continue because either:
  - The coset table is complete, or
  - We have defined n cosets, the maximum number
- If the table is complete, we have a subgroup
- In any case, we now "force a coincidence"
- Take some pair of cosets i and j, and force $i = j$
- The resultant table now corresponds to a new subgroup with a new generator $\alpha_i \alpha_j^{-1}$ constructed from the coset representatives α_i and α_j
- Each choice of (i, j) is considered separately as a new branch in the search tree

Characteristics

We have a backtrack search that:

- is unpredictable in shape
- is unpredictable in size
- may return results before reaching a leaf
- can be split into independent branches

Parallelisation

Two approaches taken:

- Tasks (using `RunTask`, `WaitTask`...)
- Worker threads (`CreateThread`, `WaitThread`...)

Sequential implementation

Recursion

```
DescendantSubgroups := function(...)
    subgps := [];
    CosetEnumeration(...);
    if IsComplete(table) then
        Add(subgps, thisSubgroup);
    fi;
    for each pair of cosets (i,j) do
        Append(subgps,
               DescendantSubgroups(<table with i=j>, ...)
              );
    od;
    return subgps;
end;
```

Using Tasks

The "shotgun" approach

```
DescendantSubgroups := function(...)
    subgps := [];
    tasks := [];
    CosetEnumeration(...);
    if IsComplete(table) then
        Add(subgps, thisSubgroup);
    fi;
    for each pair of cosets (i,j) do
        Add(tasks, RunTask(DescendantSubgroups, <args>) );
    od;
    for task in tasks do
        Append(subgps, TaskResult(task) );
    od;
    return subgps;
end;
```

Using Tasks

Speedup

- Effective up to 4 cores
- Little speedup beyond 4 cores
- Enormous time for large problems – overheads

Using Worker Threads

Objects

- `workQueue` – Channel of jobs to be done
- `numJobs` – Number of jobs still incomplete
- `resultsChan` – Channel used to store results
- `finish` – Semaphore indicating that all work is complete
- `Work` – Function executed by each new thread
- `ExecuteJob` – New name for `DescendantSubgroups`

Using Worker Threads

Top-level function

```
LowIndexSubgroups(G, maxIndex, numWorkers)
    ...
    <Create workQueue, resultsChan, numJobs, and finish>

    workers := List([1..numWorkers],
                    i->CreateThread(Work, <args>)
                   );
    ExecuteJob(..., workQueue, resultsChan, numJobs);

    WaitSemaphore(finish);
    SendChannel(workQueue, fail);
    Perform(workers, WaitThread);

    <Extract all the results from resultsChan>
    ...
end;
```

Using Worker Threads

Work function

```
Work := function(workQueue, resultsChan, ...)
    while true do
        j := ReceiveChannel(workQueue);
        if j = fail then
            SendChannel(workQueue,fail);
            break;
        fi;
        ExecuteJob(j.table, j.label, ...);
        atomic numJobs do
            numJobs := numJobs - 1;
            if numJobs = 0 then
                SignalSemaphore(finish);
            fi;
        od;
    od;
end;
```

Using Worker Threads

ExecuteJob function

```
ExecuteJob := function(...)
    CosetEnumeration(...);
    if IsComplete(table) then
        SendChannel(resultsChan, thisSubgroup);
    fi;
    for each pair of cosets (i,j) do
        newJob := rec(table := table,
                      label := b,
                      reps := reps,
                      gens := Concatenation(gens,[newGen])
                     );
        SendChannel(workQueue, newJob);
        atomic numJobs do
            numJobs := numJobs + 1;
        od;
    od;
end;
```

Using Worker Threads

Speedup

- Effective up to 4 cores
- Little speedup beyond 4 cores
- Huge number of jobs created – all threads attempting to read from workQueue very often, resulting in a bottleneck
- If only workers could explore subtrees themselves, so long as all cores are busy...

“Minimal” Job Sharing

- If every thread has work to do, a thread processes a complete job depth-first with no communication
- If there is no work left on the queue, a thread must branch
- Avoids either heavy communication on a single channel, or long-idle workers
- New parameter in `ExecuteJob` – `depthFirst`

In the `Work` function:

```
atomic readonly numJobs do
    depthFirst := numJobs > numWorkers;
od;
```

- Still have workers idle, waiting for another thread to branch

Improvements

- Decide whether to branch *inside* depth-first search
- Always keep a “buffer” of items on the queue, to reduce idle workers – means more breadth-first
- Attempt to predict size of subtree and “branch intelligently”

Other approaches:

- Retrospective job sharing
- Random depth-first search