The HPCGAP Project:
Towards High Performance
Computational Algebra with GAP

Reimer Behrends, Alexander Konovalov, Steve Linton,
Frank Lübeck, Max Neunhöffer
Overview

• Why Parallelise?
• HPCGAP project
  • overall software model
  • use cases
  • a couple of technical highlights
    • dataspaces
    • deadlock prevention
  • skeletons -- our recommended parallel programming models
Every Year We Can Solve Bigger Problems

1990's Complexity Theory

All EXPTIME problems can be solved in linear time

Wait a linear number of years while Moore's law makes computers exponentially faster

• Not any more :-(
• Individual cores have almost stopped getting faster
  • Just more numerous
• Modern gamers PC: 4 core CPU, 240 core GPU.

• Supercomputers: $10^5$ cores today, $10^6$ planned, $10^8$ being explored.
• Bigger problems in five and ten years, will need paralellism
Popular Wisdom

If you have two processors, use them to solve two problems at once.

• Good advice for most people, until recently
• Less tenable in the world of 4-core laptops and 24-core desktops
• If you want to keep solving bigger problems for the next ten years you will have to engage with parallel programming
• We’re trying to make GAP a good platform for that
Parallel Programming is Hard

• A common trap (out of myriads):

  if not empty(queue) then
    take x from the queue and process it
  end if

• In a parallel program another thread may have emptied the queue between the test and the action
  • even if all the queue operations are serialised

• We need a way to allow computational mathematicians to design and implement parallel algorithms without having to worry about this kind of detail
  • and which will be acceptably efficient on the reasonable range of present and future hardware
HPCGAP

- EPSRC project (Infrastructures programme): St Andrews, EPCC, Aberdeen, Heriot-Watt
- 2009–2013
- **Goal** to support parallelism in GAP at all levels for users and programmers
  - Threaded GAP sessions on multicore machine
  - Distributed computing on clusters and supercomputers
  - Some ‘free’ speedups for non-experts on multicore
  - Easy ways for domain experts to describe and implement parallel algorithms
  - Allowing programmers to explore new parallel programming models
- Taking the opportunity to do a lot of necessary reengineering
# HPCGAP Model

<table>
<thead>
<tr>
<th>Level</th>
<th>Shared Memory</th>
<th>Distributed</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Low level</strong></td>
<td>Threads, Mutexes, Dataspaces</td>
<td>MPI binding, binary object marshalling</td>
</tr>
<tr>
<td>Primitives</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Mid level</strong></td>
<td>Channels, Single-assignment variables, Barriers, Knowledge bases</td>
<td></td>
</tr>
<tr>
<td>Common primitives</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>High level</strong></td>
<td>ParList, ParSum, ParReduce, DFS, Chain Reduction, First to Finish, .....</td>
<td></td>
</tr>
<tr>
<td>Skeletons</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Thread-safe Library</strong></td>
<td>Full GAP library -- some parts may have to be protected by “big locks”</td>
<td></td>
</tr>
<tr>
<td><strong>Parallelised Library</strong></td>
<td>Exact linear algebra, Base and strong generating set, Backtrack search</td>
<td></td>
</tr>
<tr>
<td>Key Bottleneck Routines</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Applications</strong></td>
<td>Compuations in E₈, Design search, homology,...</td>
<td></td>
</tr>
<tr>
<td>Exemplars</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Use(r) Cases

- **Naive user**
  - uses GAP just as before, but on multicore system
  - sees speed-up when he happens to hit a parallelised library routine
  - maybe backgrounds computations from the command-line

- **Sequential programmer**
  - existing GAP code will run unchanged in a single thread
  - if run in multiple threads is more likely to give a meaningful error message than unpredictable wrong results
Use(r) Cases 2

- **Skeleton user**
  - can write, run and debug parallel programs using our skeletons without too much pain
  - skeletons should cover a decent range of common algebraic computations

- **Parallel programmer**
  - developing specialised shared or distributed memory applications
  - using mid-level abstractions directly

- **Skeleton developer**
  - building efficient skeleton implementations for a variety of environments
  - tuning new mid and high level abstractions for performance.
Dataspaces -- The Issues

- **Goal:** As much sequential code as possible should be “automatically” thread-safe
  - minimal surprises for the programmer
  - surprises that do happen should be understandable and repeatable errors, not unrepeatable wrong answers

- **Problem:** GAP code accesses and modifies many global data structures “behind the scenes”
  - storing computed data for re-use
  - caching method selections and types
  - creating universal structures on demand
  any of this can create race conditions, or deadlocks
Dataspaces -- The Idea

**Key observation:** Most access to shared data fits a few patterns
- read-only access
- adding information to objects
- using parallel constructs like channels

**Idea:**
- read only and “well-behaved” objects are global
- newly created mutable objects are only accessible to the creating thread, unless explicitly shared
- shared mutable objects can only be accessed with an appropriate lock

The nastiest direct data races are eliminated, but existing GAP code should largely work within a thread.
Dataspaces -- More Precisely

- Every GAP object lies in just one *dataspace*

- These are of three kinds:
  - thread-local -- one of these associated with each thread
  - global -- one of these
  - shared -- possibly many of these, typically each with just a few objects

- By default, a thread may access:
  - any object in its thread-local dataspace, to read or write
  - any object in the global dataspace via the operations provided by the kernel for it

- Access to objects in shared dataspaces is controlled by read and write locks
  - each dataspace can have one writer or any number of readers

- To move an object between two dataspaces you need write access to both of them. No object can ever leave the global dataspace.
Dataspaces -- The Big Picture

Thread-local Dataspace
[1,2,3]
rec(a := 3, b := (1,6))

Thread-local Dataspace
[1,2,3]
rec(a := 3, b := (1,6))

Thread-local Dataspace
[1,2,3]
rec(a := 3, b := (1,6))

Global Dataspace
1765489256789421067

Thread
Access via atomic kernel ops
Free access

S10

Access when lock held

Objects for which the kernel permits only atomic operations
Mainly immutable
All such objects are here

Newly created non-global objects go here

Migration

Shared Dataspaces

[7,11]
Deadlock Prevention

- Deadlock occurs when two threads each hold a lock the other is waiting for.
- Can avoid it if you have a partial order on locks and only wait for locks > all the locks you hold.
- HPCGAP constructs the order “on-the-fly”, adding to it whenever a thread gets two locks.
- Subsequent attempts to take those locks in the other order will give immediate errors.
Skeletons

• We don’t expect most parallel GAP programmers to work directly with threads, or even with channels

• Instead we will provide skeletons: higher-level parallel functions which you flesh out with pieces of sequential code to make parallel programs

• Skeletons may need some “tuning”, especially on larger machines.
Some of My Favourite Skeletons

- **ParList**
  - **Inputs:** A list of objects; a function of one argument \textit{with no side-effects}
  - **Outputs:** The list of results of applying the function to the objects
  - **Note:** No guarantee of what order the function executions will happen in (or where).
  - Captures ``trivial" or "embarrassing" parallelism -- lots of completely independent tasks
  - Still challenges with regularity, scaling, robustness.

- **DFS:** Find all the nodes of a (directed-)graph given by seed nodes and a function for finding (out-) neighbours of a node.
  - Variations of this handle orbit and orbit-stabilizer computations.
  - Hard part is to recognise where you have been

- **Duplicate Elimination**
  Given two lists of keys and data, find, and deal specially with, all the keys present in both lists, return a list of the results and the non-duplicates
  - This handles addition of sparse vectors \textit{inter alia}
  - **Chain Reduction**
    Given an ordered list of objects, reduce each object against each earlier object in some way.
    - Gaussian elimination is one application
  - **Pair Completion**
    Makes new objects from \textit{pairs} of old ones, reduces new objects \text{wrt} old ones and possibly v.v.
    - Knuth-Bendix, Buchberger's algorithm
  - **First to finish**
    Try a number of algorithms to solve a problem -- report the first solution
    - hard part seems to be to stop the other ones