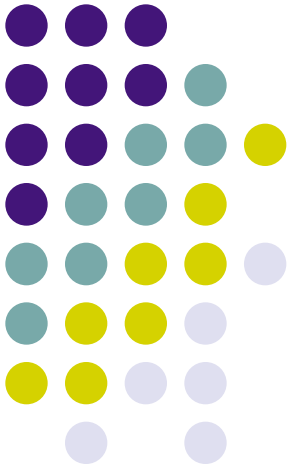
Parallelising the computational algebra system GAP

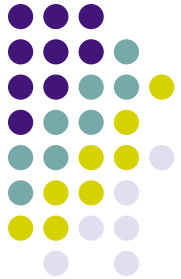
Reimer Behrends
Alexander Konovalov
Stephen A. Linton
School of Computer Science
University of St. Andrews

Frank Lübeck
LDFM
RWTH Aachen

Max Neunhöffer
School of Mathematics and Statistics
University of St. Andrews



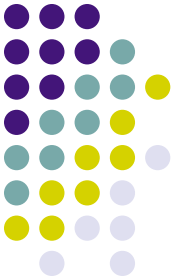
The GAP system



- System for Computational Discrete Algebra
- Interpreted language
- Problem: Parallelising the GAP language (for shared memory systems) as part of the HPC-GAP project
  - Thousands of established users
  - Hundreds of thousands of lines of GAP code in the standard distribution

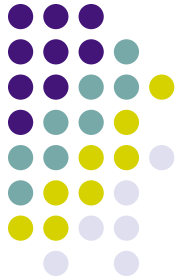
  $\Rightarrow$ Limits on language redesign

GAP Users



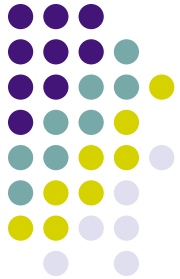
- How many of their resources (time, effort, money) can they invest in parallelism?
- Spectrum ranges from “little” to “a lot”.
- Pure domain experts (mathematicians)
- Domain experts who need performance
- Parallelism experts

Correctness and performance



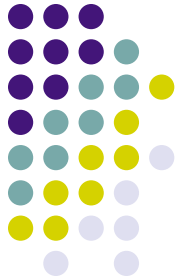
- Difficult to reconcile in parallel programming (non-determinism, race conditions, deadlocks)
- But: Both are important
- Make correctness easy even when programming for performance
  - Safe default behavior
  - Flag concurrency errors aggressively

Data spaces



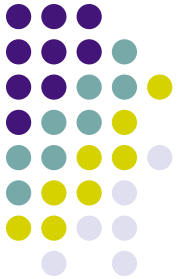
- Each GAP object belongs to exactly one data space
- Thread-local data spaces: One per thread.
  - All objects originate in a thread-local data space
  - May migrate to other data spaces later
  - Present the illusion of a mostly sequential environment
  - Ease of use for parallelism non-experts
  - Limited concurrency potential

Data spaces (continued)



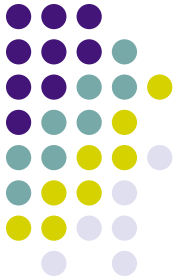
- Shared data spaces
  - Have an associated R/W lock
  - Need to be locked and unlocked explicitly before access
- One public data space
  - Contains only atomic objects
  - Can be accessed without explicit locking

Object migration



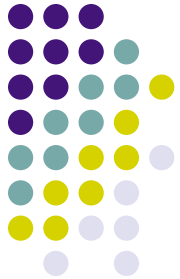
- Threads interact by migrating objects
- Migrating an object changes its data space membership
- Does not copy its contents
- Cheap operation (update a C pointer)
- Variants:
  - Thread-local -> Thread-local: Message passing
  - Thread-local -> Shared, Public: Sharing
  - Shared -> TL: Privatization

Parallelism for non-experts



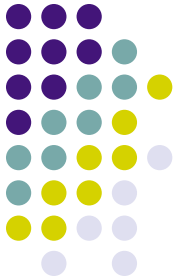
- Domain experts see a sequential world.
  - Their program is one thread
  - Indistinguishable from sequential code
  - Accesses only thread-local data directly
  - May use parallelised libraries with a sequential API
- If you want more, we provide skeletons:
  - result := ParList(list, x -> f(x), NumWorkers);
  - ParList() splits ‘list’ in segments, migrating them to workers
  - Worker threads perform x ->f(x) thread-locally
  - Results migrated back to main thread

A closer look at migration



- Thread 1: *SendChannel(ch, a);*
- Thread 2: *b := ReceiveChannel(ch);*
- Object is migrated:
  - From thread 1's thread-local data space
  - Via the channel's shared data space
  - To thread 2's thread-local data space
- Note: Variables *a* and *b* both reference the same object at the end
  - Not classic message passing
  - Race conditions possible?

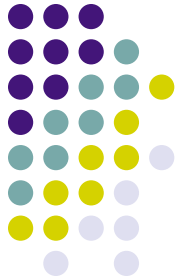
Data race protection



- Primitives protect any access to an object
  - WriteGuard(): Requires exclusive access
  - ReadGuard(): Requires shared access
  - Both check the object's data space descriptor
- **All** object accesses are:
  - Either: protected by the appropriate primitive
  - Or: statically verified to be safe.

$\Rightarrow$ Data races cannot occur

Example



- This code produces an error:

  local c, z;

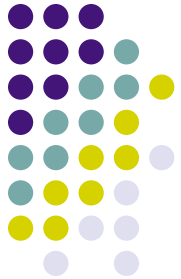
  c := 2;

  z := ParList([1..100], x->x+c, NumWorkers);

- Why?
  - c is in the main thread's thread-local data space
  - Worker threads try to access it
- Solution:

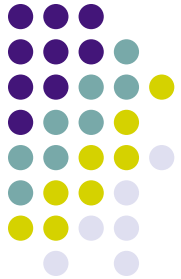
  z := ParList([1..100], Publish(x->x+c), NumWorkers);

Deadlock protection



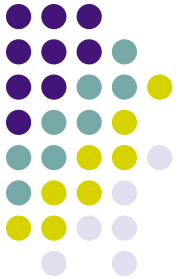
- There must be a partial order ‘>’ on shared data spaces
- Users do not need to specify this partial order
- If a thread holds a lock on DS1 while it acquires a lock on DS2, DS1 > DS2 must hold
- If not, an error is raised
- GAP kernel tracks sequences of lock operations
- Ensures that there are no cycles in that relation

Summary



- Thread-local data spaces: sequential core
- Shared & public data spaces, migration: provide concurrency
- Protection from data races and deadlocks
- Skeletons to encode parallel algorithms

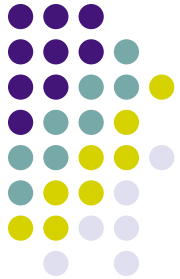
Non-interference



- Owicki-Gries (Acta Informatica 1976)
- Principle: Actions in one process don't invalidate assertions in another
- A, B programs
- B doesn't interfere with A:

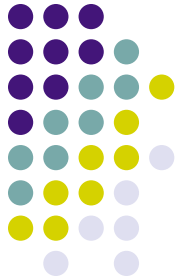
$\{Pre_A\}\ A\ \{Post_A\}$
$\Rightarrow\ \{Pre_A\}\ A \parallel B\ \{Post_A\}$

Ensuring non-interference



- Programmers may not formally prove correctness, but still consider assertions informally
  - "What properties hold at what point in the program"
- Minimize proof obligations for non-interference
- Make non-interference the default
- Make interference explicit
- Alert programmer to potential interference
- Composability of correctness proofs

Software Transactional Memory



- Performance concerns
  - High constant overhead
  - Works well for some problems, not for others
- Open nesting needed?
  - Unintuitive, requires expertise
- We can still use STM for fine-grained locking
  - “Adaptive Locks: Combining Transactions and Locks for Effective Concurrency”, T. Usui, R. Behrends, J. Evans, Y. Smaragdakis, PACT ‘09.