HPC-GAP: Design and Implementation of a Concurrency Model for GAP

Reimer Behrends

TU Kaiserslautern

User interface

- `!shell` – start a shell in a new thread.
- `!list` – list active threads.
- `!<n>` – switch to thread #n.
- `!break <n>` – interrupt thread #n.
- `!QUIT` – terminate GAP.

Basic multi-threading: tasks

Tasks are lightweight threads.

Function + *n* inputs => output

Tasks run asynchronously.

- `task := RunTask(func, arg_1, ..., arg_m);`
- `WaitTask(task_1, ..., task_m);`
- `k := WaitAnyTask(task_1, ..., task_m);`
- `result := TaskResult(task);`

Tasks: Example I

```
gap> t := RunTask(SortedList, [3, 2, 1]);;
gap> TaskResult(t);
[1, 2, 3]
```

Tasks: Example II

```
gap> tasks := List([1..20],
>       x -> RunTask(Factorial, x));;
gap> List(tasks, TaskResult);
```

Shared Memory

So far, we have no (or very limited) ways of sharing data between threads.

- May limit performance (copying huge objects between threads).
- May limit expressiveness (some concurrent algorithms require shared state).

Shared Memory: Regions

Regions manage mutual exclusion and immutability.

Regions form a partition of GAP objects.

- Thread-local regions.
- Public region.
- Read-only region.
- Shared region.

Use `RegionOf(obj)` to learn the region of `obj`.

Regions: Thread safety

Only one thread can have write access to a region at any given time*.

While no thread has write access to a region, an arbitrary number of threads can have read access to that region.

Regions can only read (or modify) objects in regions that they have read (or write) access to.

This avoids race conditions.

*: Except for the public region, which can therefore only contain objects that are inherently thread-safe.

Regions: Enforcing access

The GAP kernel is annotated with primitives (`ReadGuard(obj)` and `WriteGuard(obj)`) that check whether the current thread has access to `obj` and raises an error otherwise.

This makes it impossible to access a region that a thread doesn't have access to.

Basic region management

Default: New GAP objects end up in the current thread's thread-local region. Immutable objects become part of the public region.

Thread-local variables:

```
BindThreadLocal(name, value);
```

Functions are immutable and can safely be assigned to global variables.

Quick and dirty package conversion

Convert all global variables that don't reference functions to thread-local variables.

Caution: Replicates state for **each thread**. Can waste memory.

Immutability

Two different ways of expressing immutability.

- `MakeImmutable(obj)`: move to public region (most of the time).
- `MakeReadOnly(obj)`: move object (and subobjects) to read-only region.
- `MakeReadOnlyObj(obj)`: move object (without subobjects) to read-only region.

GAP: Difference between mathematical immutability (immutability of abstract state) and immutability of concrete state.

`MakeImmutable()` also requires that all subobjects be immutable.

Shared objects

Sharing data:

- `NewRegion()` or `ShareObj()` to create a new shared region.
- `MigrateObj()`, `AdoptObj()`, `IncorporateObj()` to move objects between regions.
- `atomic` statement to gain exclusive/read-only access to one or more regions.

Example: Immutability

```
BindGlobal("SmallPrimes",
    MakeImmutable([2, 3, 5, 7, 11, 13]));
```

Useful for constants. Because this is a common construct, there is a shortcut using the backquote character.

```
BindGlobal("SmallPrimes", `[2, 3, 5, 7, 11, 13]);
```

Example: Shared objects

```
l := ShareObj([]);

SharedListAdd := function(list, value)
  atomic readwrite list do
    Add(list, value);
  od;
end;

SharedListLength := function(list)
  atomic readonly list do
    return Length(list);
  od;
end;
```

Example: Readonly + shared objects

```
r := MakeReadOnly(rec(id := "foo",
                      cache := ShareObj([ ])));
```

We can access `r.id` without atomic and only need an `atomic` statement for `r.cache`.

Useful with `Objectify()`: Allows method dispatch without synchronization bottlenecks.

Note: `MakeImmutable()` doesn't work because of a shared subobject.

Advanced shared memory management

Regions are a simple, but sometimes crude and inefficient way for ensuring a basic form of thread-safety.

For more advanced forms of shared memory management, we begin with a (semi-)formal definition of thread-safety.

Digression: Formal thread safety

Classical model: Hoare Logic

```
{ Precondition }
Code
{ Postcondition }
```

Example

```
 { x > 0 }
 x := x + 1;
{ x > 1 }
```

Concurrent world: Owicki-Gries

Concept of non-interference.

`Q` does not interfere with`{Pre} P {Post}` if: with one another if:

```
{ Pre }
P || Q
{ Post }
```

for all possible interleavings of `P` and `Q`*.

Problem: combinatorial explosion for number of proof obligations.

*: Simplified definition, see "An axiomatic proof technique for parallel programs" by S. Owicki and D. Gries, Acta Informatica, 1976 for the actual definition.

Interference: Example

Question: does the following hold?

```
{ x > 0 }
x := x + 1; || x := x + 1;
{ x > 2 }
```

Answer: No.

Managing correctness

Separation: Give each thread its own copy of the data.

Mutual exclusion/atomicity: Only one thread can access common data at the same time.

Read/write locks: Either one thread has exclusive access to data or any number of threads have read-only access to data.

Immutable data: Any number of threads

Idempotent operations: Primitives who can be applied repeatedly without changing the result.

Mutual exclusion != thread-safety.

Note: Mutual exclusion does not imply thread-safety.

Example:

```
shared := ShareObj([0]);
```

Thread code:

```
local t;
atomic shared do t := shared[1]; od;
t := t + 1
atomic shared do shared[1] := t; od;
```

Run by two threads concurrently, the end result is undefined.

Partial v. total correctness

- Partial correctness only says that a program doesn't compute any wrong results.
- Total correctness = partial correctness + program terminates.
- Concurrency specific issues: deadlock, starvation.

Advanced concurrency mechanisms

Goals:

- Simpler code.
    - Less region management boilerplate.
    - Accept increased memory usage to gain simplicity.
    - Accept reduced safety to gain simplicity/speed.
- More powerful constructs.
- Deadlock avoidance.

Atomic objects

Atomic objects solve the problem of having to write region management boilerplate for mutable objects with simple semantics.

Downside: They're unsafe. Nothing will protect you from race conditions. The reason why region management is apparently cumbersome at times is precisely so that one has to be explicit about the intended behavior, minimizing inadvertent bugs.

Atomic lists

Two flavors:

- Fixed size atomic lists (faster).
- Resizeable atomic lists (more flexible).

```
gap> al := AtomicList([1,2,3]);
gap> al[4] := 4;
gap> FromAtomicList(al);
gap> RunAsyncTask(function() al[1] := 314; end);
gap> FromAtomicList(al);

gap> al := FixedAtomicList([1,2,3]);
gap> al[4] := 4;
```

Atomic records

```
 gap> r := AtomicRecord();
 gap> r.x := 1;
 gap> !sh
--- Switching to thread 5
 [5] gap> r.z := 1;
 [5] gap> !0
--- Switching to thread 0
 FromAtomicRecord(r);
```

```
 gap> r := AtomicRecord(rec(x := 1, y := 2, z := 3));
 gap> RecNames(r);
 gap> r.("x");
```

Write-once semantics

Atomic objects can be given write-once semantics. Assigning to an entry more than once is ignored.

```
gap> r := MakeWriteOnceAtomic(rec(x := 1, y := 2));
gap> r.z := 3; r.x := 3;
gap> FromAtomicRecord(r);
```

The strict version raises an error:

```
gap> r := MakeStrictWriteOnceAtomic(rec(x := 1, y := 2));
gap> r.z := 3; r.x := 3;
gap> FromAtomicRecord(r);
```

Thread-local variables with constructors

```
BindThreadLocalConstructor("RNGState",
       ->RNGSeed(ThreadID(CurrentThread())));
```

Practical uses:

- Random number generators.
- Unique per-thread IDs.

Note: new syntax for parameterless functions.

Tasks: Cancellation

Task execution can be cancelled, but only cooperatively.

- `CancelTask(task)` requests cancellation.
- `OnTaskCancellation(func)` cancels task if cancellation was requested and returns `func()`.

Example:

```
gap> Read("demo/factor.g");
```

Tasks: Dependencies

Tasks can execute dependent on another's completion.

```
gap> task := RunTask(Sleep, 5);;
gap> task2 := ScheduleTask(task, ReturnTrue);;
gap> TaskResult(task2);
```

This is more efficient than `WaitTask()` within another task, which needs to save the state of the current task.

Deadlocks

In order to avoid deadlocks, regions actually follow a precedence hierarchy, and atomic nested statements need to acquire them in descending order. Therefore, we actually have more than one `NewRegion()` alternative.

```
NewRegion();              # top level (application code)
NewLibraryRegion();       # User libraries and GAP packages.
NewSystemRegion();        # System libraries.
NewKernelRegion();        # Kernel code.
NewInternalRegion();      # Minimal value.
NewSpecialRegion();       # Bypasses deadlock checks.
```

Passing multiple regions/objects to a single `atomic` statement is always safe, regardless of their relative precedence.

ShareObj() variants.

Similarly, `ShareObj()` has the same kinds of variants:

- `ShareObj()`
- `ShareLibraryObj()`
- `ShareSystemObj()`
- `ShareKernelObj()`
- `ShareInternalObj()`
- `ShareSpecialObj()`

And likewise for `ShareSingleObj()`.

Note that starting gap with `-Z` turns off these checks entirely.

Low-level primitives

- Channels
- Semaphores
- Synchronization variables
- Raw threads
- Disabling guards

Channels

Channels are FIFO objects that allow threads to exchange data directly.

They also allow for the automatic transfer between thread-local regions without writing `ShareObj()`/`AdoptObj()` boilerplate.

Channels themselves are in the public region and can be accessed concurrently by any number of threads.

Channel example

```
gap> ch := CreateChannel();
gap> SendChannel(ch, [1,2,3]);
gap> InspectChannel(ch);
gap> !sh
--- Switching to thread 5
[5] gap> ReceiveChannel(ch);
[5] [ 1, 2, 3 ]
[5] gap> RegionOf(last);
[5] <region: thread #5>
```

Channels with bounded capacity

```
gap> ch := CreateChannel(1);
gap> SendChannel(ch, "x");
gap> SendChannel(ch, "y"); # blocks
```

Channels with capacity 1 are useful for certain types of handshake semantics.

Additional channel functionality

Variant sending and receiving functions:

- `TryReceiveChannel(channel, default)`
- `MultiReceiveChannel(channel, amount)`
- `TrySendChannel(channel, value)`
- `MultiSendChannel(channel, list)`

Multiplexing:

- `ReceiveAnyChannel(channel_1, ..., channel_n)`
- `ReceiveAnyChannel(channel_list)`
- `ReceiveAnyChannelWithIndex(...)`

Semaphores

```
 gap> sem := CreateSemaphore(0);
 gap> WaitSemaphore(sem);
 gap> !sh
--- Switching to thread 5
 [5] gap> SignalSemaphore(sem);
 [5] gap> !0
--- Switching to thread 0
 gap>
```

Semaphore applications

- Idle wait on a condition (usually in conjunction with a region).
- Simulate low-level locks.
  - CreateSemaphore(1) to create a pseudo-lock.
  - WaitSemaphore() to lock it.
  - SignalSemaphore() to unlock it.
- Limit access to a resource with bounded capacity.

Synchronization variables

Synchronization variables are containers with write-once, blocking read semantics. Useful for data handshakes.

```
gap> var := CreateSyncVar();
gap> SyncIsBound(var);
false
gap> SyncWrite(var, [1,2,3]);
gap> SyncIsBound(var);
true
gap> SyncRead(var);
[ 1, 2, 3 ]
```

Note: the value will not be migrated. Use shared, read-only, immutable, or atomic objects. This is because the value may be read by multiple threads.

Raw threads

Starting a thread is expensive. You should only launch a new thread if you expect that it can do a significant amount of work.

```
gap> th := CreateThread(function()
>       Display("Ping!");
>     end);
gap> ThreadID(th);
gap> WaitThread(th);
```

Threads can be controlled by `KillThread()`, `PauseThread()`, and `ResumeThread()`.

Additional features

- Efficient serialization of GAP objects.
- Built-in hash tables/sets based on object identity.
- MPI bindings.
- ZeroMQ bindings.
- And more.

See documentation for details.