Shared Memory Concurrency for GAP

Reimer Behrends
(TU Kaiserslautern)

behrends@mathematik.uni-kl.de

The GAP system [3], as it is introduced on the GAP Web site, is an open-source system for computational discrete algebra, with particular emphasis on computational group theory. It provides a programming language, an extensive library of functions implementing algebraic algorithms written in the GAP language as well as large data libraries of algebraic objects. The kernel of the system is implemented in C, and the library is implemented in the GAP language. Both the kernel and the library were originally sequential and did not support parallelism.

In the 4-year long EPSRC project "HPC-GAP: High Performance Computational Algebra and Discrete Mathematics" (`http://www-circa.mcs.st-and.ac.uk/hpcgap.php`), started in September 2009, a team of researchers at the University of St Andrews reengineered the GAP system to allow parallel programming in it using both shared and distributed memory approaches. In this article, we report on the results of the HPC-GAP project with respect to parallelizing execution on multicore systems with shared memory. The concurrency model that we employ aims at making concurrency facilities available to GAP users, while preserving the existing codebase (hundreds of thousands of lines of code and data) with as few changes as possible. To this end, the model preserves the appearance of sequentiality on a per-thread basis by containing each thread within its own data space, while making concurrent interaction possible through selective sharing of data and by allowing the migration of objects between data spaces.

Shared Memory: Regions

The fundamental concept through which HPC-GAP regulates concurrent interaction is that of a *region*. A region is a data space in memory; each GAP object belongs to precisely one region. Access to GAP objects is controlled through ownership of regions: in order to access objects within a region, a thread must first obtain exclusive or shared ownership of the region. As a rule, when a thread has exclusive ownership, it can read and modify objects within the region; when it has shared ownership, it can only read the objects. Violating these access rules results in a runtime error, thus disallowing access without ownership entirely.

This approach effectively makes data races – situations where two threads concurrently modify the same object or where one thread modifies an object while another reads it – impossible, eliminating what is typically the major source of software defects in multi-threaded systems. In this regard, our design differs from most mainstream programming languages, which tend to not enforce this property (see, e.g., [2]).

Each thread has an associated *thread-local region*, of which it always has exclusive ownership. In particular, objects in a thread's thread-local region can never be accessed by other threads.

When using only a single thread, the thread's behavior is essentially indistinguishable from a sequential system to the end user (though underlying libraries may employ concurrency). This design intentionally hides the complexities of parallelism from users who do not have the necessary expertise in parallel programming. They can safely continue to use the system as though it were sequential code while still benefiting from any parallelized libraries and packages that others have written. By having their actions compartmentalized within the main thread's region, their code will not interfere with any parallelized code and the parallelized code will not interfere with their code.

In addition, code that requires concurrent interaction can also create one or more *shared regions*. In order to obtain exclusive or shared ownership of a shared region, a thread must first acquire the read-write lock associated with the region (a read lock for shared ownership, a write lock for exclusive ownership). Shared regions are typically created through the `ShareObj(obj)` function, which creates a new shared region and moves `obj` inside.

Threads can claim shared or exclusive ownership to a region through a new control structure, the `atomic` statement. An atomic statement takes one or more arguments, optionally preceded by a `readonly` or `readwrite` descriptor. The thread then gains shared (`readonly`) or exclusive (`readwrite`) ownership for the duration of the `atomic` statement. Example:

```
atomic readonly obj, readwrite obj2 do
  # Thread has shared ownership of the region
  # containing <obj> and exclusive ownership
  # of the region containing <obj2>. Ownership
  # of both regions will be relinquished when
  # control flow reaches the "od" keyword at
  # the end of the code block.
od;
```

For convenience, constant data (esp. large tables) can be stored in a special *read-only region*. All threads have permanent shared ownership of the read-only region, meaning that they can access the data in it as though it were a shared region for which they hold a read lock, but without the extra overhead and inconvenience of having to aquire one each time. The `MakeReadOnly(obj)` primitive puts `obj` in the read-only region.

Finally, HPC-GAP knows a *public region*, to which all threads have constant read and write access. This is the one exception to the rule that one has to have exclusive ownership in order to modify objects within a region. As a consequence, only objects that expose exclusively atomic operations to GAP (i.e., operations that cannot create data races) can be stored in the public region. The public region therefore only holds certain special kinds of objects (such as those needed by the various HPC-GAP synchronization primitives) as well as certain types of immutable objects. Functions, for example, are always contained in the public region[1], so that the same function can be called by multiple threads.

Objects can be migrated or copied between regions. Migration, unlike copying, is an inexpensive operation in that it simply changes the region membership descriptor of an object. Migration can be performed through the `MigrateObj(obj, target)` function, which migrates objects between arbitrary regions, or the `AdoptObj(obj)` function, which migrates an object to the current thread-local region.

Multi-threading

In order to use multiple threads, GAP programmers are encouraged to use the task library, which provides convenient abstractions over low-level thread management[2]. The task library uses the concept of futures [3] similar to mainstream programming languages such as C# and Java.

The primary interface provided consists of the `RunTask()` and `TaskResult()` functions:

```
task := RunTask(f, x1, ..., xn);
result := TaskResult(task);
```

`RunTask()` takes a function f and zero or more values x_i as its arguments. It then evaluates $f(x_1, \ldots, x_n)$ asynchronously in a different thread. `RunTask()` finishes immediately (i.e. while f is still being executed in parallel), returning a task descriptor as a result. The `TaskResult()` function takes a task descriptor as its sole argument, waits until that task has completed and returns the result of the underlying evaluation of $f(x_1, \ldots, x_n)$.

The task library provides additional primitives (such as waiting for the first of a set of tasks to finish, having the execution of tasks triggered by a condition, or the cancellation of tasks). But `RunTask()` and `TaskResult()` are the core primitives in that they allow the concurrent execution of arbitrary computations with an interface that is only slightly more complex than a regular function call.

Note that the evaluation of $f(x_1, \ldots, x_n)$ is performed in a separate thread; in order to make this possible, any thread-local arguments are implicitly copied to the new thread; likewise, the result (if thread-local) is copied back to the thread-local region of the thread performing the `TaskResult()` call.

Example: Parallel Matrix Multiplication

Figure 1 uses the standard parallel matrix multiplication algorithm to illustrate the above concepts.

```
 1 ParMatrixMultiplyRow := function(m1, m2, i)
 2   local result, j, k, n, s;
 3   result := [];
 4   atomic readonly m1, readonly m2 do
 5     n := Length(m1);
 6     for j in [1..n] do
 7       s := 0;
 8       for k in [1..n] do
 9         s := s + m1[i][k] * m2[k][j];
10       od;
11       result[j] := s;
12     od;
13   od;
14   return result;
15 end;
16
17 ParMatrixMultiply := function(m1, m2)
18   local tasks, result;
19   ShareObj(m1);
20   ShareObj(m2);
21   atomic readonly m1, readonly m2 do
22     tasks :=
23       List([1..Length(m1)],
24         i -> RunTask(ParMatrixMultiplyRow,
25                      m1, m2, i));
26     result := List(tasks, TaskResult);
27   od;
28   atomic readwrite m1, readwrite m2 do
29     AdoptObj(m1);
30     AdoptObj(m2);
31   od;
32   return result;
33 end;
```

Figure 1: *Parallel matrix multiplication.*

The `ParMatrixMultiply()` function in lines 17–33 performs the actual multiplication. It takes two square matrices as its arguments. The `ShareObj()` primitives in lines 19–20 create new shared regions for these matrices and migrate them inside. The `atomic` construct in lines 21–27 then acquires read locks for the two shared regions, granting the current thread shared ownership. The substance of the work occurs in lines 22–25, where one task is started for each row vector in `m1`. This is done through the standard `List()` function, which takes two arguments, a list and a function, and applies the function to each element in the list. This function performs a `RunTask()` call; thus, the result of `List()` is a list of task descriptors. The second `List()` invocation in line 26 then maps each of the task descriptors to their results, returning a list of the

[1] Obviously, this holds only for the function's *code*, which never changes, not any *data* that it operates on.

[2] Of course, low-level thread and concurrency primitives are also available for advanced concurrency constructs.

row vectors of the product of the matrices. In lines 28–31, the two arguments are migrated back to the current thread's thread-local region; the `AdoptObj()` calls effectively undo the `Shareobj()` calls in lines 19–20.

The actual multiplication operation for each row vector takes place in the `ParMatrixMultiplyRow` function in lines 1–15. This is the basic matrix multiplication algorithm for a given row vector. The only difference compared to the sequential version is the `atomic` statement in lines 4–13, which gives the task temporary shared ownership of the matrix regions. Note that this is necessary even though the main thread had already done the same: the tasks have no knowledge of what the main thread is currently doing and thus have to also claim shared ownership on their own. Because all tasks require only read access to the matrices, shared ownership is sufficient and all tasks can execute in parallel (as long as the hardware permits).

Conclusion

A public beta release of HPC-GAP is planned for the near future. If you would like access to the current pre-release version, please contact the author or the GAP Group.

References

[1] GAP – Groups, Algorithms, and Programming, Version 4.7.5; 2014 `http://www.gap-system.org`

[2] Hansen, P. B. Java's Insecure Parallelism. *SIGPLAN Notices*, v.34 n.4, p.38–45, April 1999.

[3] Friedman, D. P. and Wise, D. S. The Impact of Applicative Programming on Multiprocessing. *1976 International Conference on Parallel Processing*, p.263–272.

The GAP package `SingularInterface`

M. Barakat, M. Horn, F. Lübeck, O. Motsak, M. Neunhöffer, H. Schönemann
(TU Kaiserslautern, JLU Gießen, RWTH Aachen University, TU Kaiserslautern, triAGENS GmbH, TU Kaiserslautern)

`barakat@mathematik.uni-kl.de, max.horn@math.uni-giessen.de, frank.luebeck@math.rwth-aachen.de, motsak@mathematik.uni-kl.de, max@9hoeffer.de, hannes@mathematik.uni-kl.de`

What is `SingularInterface`?

The GAP package `SingularInterface` is a highly efficient and robust unidirectional low-level interface to SINGULAR [2, 3]. It is the outcome of an intensive collaboration between core developers of both systems.

The goal of this interface is to map all of SINGULAR's powerful functionality into GAP. To achieve this it automatically wraps *all* SINGULAR datatypes and exports *all* of SINGULAR's interface procedures to GAP.[1] Furthermore, all procedures of any contributed library can be loaded on demand.[2]

This package is a rather "faithful" image of SINGULAR; it does not make an extensive attempt for a better integration of SINGULAR into the GAP ecosystem. This is intentionally left to other packages, which are free to realize this in different ways.

The development of `SingularInterface` has reached a β-phase and is already actively used in some research projects. We hope to attract more users in the near future, whose feedback will be crucial for a successful further development.

How to get it?
To download and install `SingularInterface` please follow the instructions on

`http://gap-system.github.io/SingularInterface/`

If you are reading this article, say, more than one year in the future, and have a recent GAP installation, then hopefully you already have a working version of this package.

To check that the package has been successfully installed, start GAP and type:

```
gap> LoadPackage( "SingularInterface" );
true
```

To see all imported procedures type:

```
gap> SI_<press TAB twice>
```

The SINGULAR library "`standard.lib`" is loaded by default. To see all imported SINGULAR library procedures type:

```
gap> SIL_<press TAB twice>
```

To load any other library, e.g. "`matrix.lib`", type:

```
gap> SI_LIB( "matrix.lib" );
true
```

[1] With the prefix "`SI_`" prepended to their names.
[2] They appear in GAP with the prefix "`SIL_`" prepended to their names.