Distributed Programming in Scala with APGAS

Philippe Suter  Olivier Tardieu  Josh Milthorpe
IBM T.J. Watson Research Center, Yorktown Heights, NY, USA
{psuter,tardieu,jjmiltho}@us.ibm.com

Abstract
APGAS (Asynchronous Partitioned Global Address Space) is a model for concurrent and distributed programming, known primarily as the foundation of the X10 programming language. In this paper, we present an implementation of this model as an embedded domain-specific language for Scala. We illustrate common usage patterns and contrast with alternative approaches available to Scala programmers. In particular, using two distributed algorithms as examples, we illustrate how APGAS-style programs compare to idiomatic Akka implementations. We demonstrate the use of APGAS places and tasks, distributed termination, and distributed objects.

Categories and Subject Descriptors D.1.3 [Programming Techniques]: Concurrent Programming—distributed programming

Keywords APGAS, Scala, Akka

1. Introduction

The APGAS programming model [10]—Asynchronous Partitioned Global Address Space—is a simple but powerful model of concurrency and distribution. It combines PGAS with asynchrony. In (A)PGAS the computation and data in an application are logically partitioned into places. In APGAS the computation is further organized into lightweight asynchronous tasks. With these, APGAS can express both regular and irregular parallelism, message-passing-style and active-message-style computations, fork-join and bulk-synchronous parallelism.

The X10 programming language [2] augments a familiar imperative, strongly-typed, garbage-collected, object-oriented language with the APGAS model. X10 and by extension APGAS have been used successfully to implement distributed applications running across tens of thousands of cores [13]. The recently developed APGAS library for Java [12] provides an alternative to X10 for programmers interested in the APGAS model but not willing to buy into a new programming language or development platform, which is not always possible or desirable.

To expose more programmers to APGAS, we propose to realize APGAS as an embedded domain-specific language for Scala. Scala welcomes library-based extensions and has pioneered alternative concurrency paradigms on the JVM, notably, the original actor library [5] and its more recent successor Akka. X10 shares ancestry and inspiration with Scala, and the facilities in Scala for library-defined language extensions make the code look almost exactly like X10 programs.

Section 1 describes the APGAS programming model and its realization in Scala. We then demonstrate two example programs: k-means in Section 2 and Unbalanced Tree Search in Section 3. Section 4 presents a preliminary performance evaluation, and Section 5 discusses selected implementation details.

2. Overview of APGAS in Scala

Terminology. A place is an abstraction of a mutable, shared-memory region and worker threads operating on this memory. A single application typically runs over a collection of places. In this work, each place is implemented as a separate JVM.

A task is an abstraction of a sequence of computations. In this work, a task is specified as a block. Each task is bound to a particular place. A task can spawn local and remote tasks, i.e., tasks to be executed in the same place or elsewhere.

A local task shares the heap of the parent task. A remote task executes on a snapshot of the parent task’s heap captured when the task is spawned. A task can instantiate global references to objects in its heap to work around the capture semantics. Global references are copied as part of the snapshot but not the target objects. A global reference can only be dereferenced at the place of the target object where it resolves to the original object.

A task can wait for the termination of all the tasks transitively spawned from it. Thanks to global references, remote tasks, and termination control, a task can indirectly manipulate remote objects.
Constructs. The two fundamental control structures in APGAS are `asyncAt`, and `finish`, whose signatures in the Scala implementation are:

```scala
def asyncAt(place: Place)(body: ⇒Unit) : Unit
def finish(body: ⇒Unit) : Unit
```

As is common in Scala libraries, we use by-name arguments to capture blocks.

The `asyncAt` construct spawns an asynchronous task at place `p` and returns immediately. It is therefore the primitive construct for both concurrency and distribution. The `finish` construct detects termination: an invocation of `finish` will execute its body and then block until all nested invocations of `asyncAt` have completed. The set of `asyncAt` invocations that are controlled comprises all recursive invocations, including all remote ones. This makes `finish` a powerful contribution of APGAS.

Because spawning local tasks is so common, the library defines an optimized version of `asyncAt` for this purpose with the signature:

```scala
def async(body: ⇒Unit) : Unit
```

We can use `async` for local concurrency. For instance, a parallel version of a Fibonacci number computation can be expressed as:

```scala
def fib(i: Int) : Long =
  if (i ≤ 1) i
  else {
    var a, b: Long = 0L
    finish {
      async { a = fib(i - 2) }
      b = fib(i - 1)
      a + b
    }
  }
```

In the code above, each recursive invocation of `fib` spawns an additional asynchronous task, and `finish` blocks until all recursive dependencies have been computed.

Another common pattern is to execute a computation remotely and block until the desired return value is available. For this purpose, the library defines:

```scala
def at[T:Serialization](place: Place)(body: ⇒T) : T
```

Handling failures. Remote invocations can fail, for instance if the code throws an exception or if the process hosting the place terminates unexpectedly. The error handling model of APGAS is to surface errors up to the first enclosing `finish`, which throws an exception. The critical property that APGAS maintains is happens-before invariance: failures cannot introduce execution orderings that are not possible under regular execution conditions [3, 4]. Detailed examples of resilient benchmarks are beyond the scope of this paper.

In the following sections, we highlight some APGAS patterns in two concrete benchmarks, and provide contrast with the actor paradigm as expressed in Akka.

3. Distributed k-means Clustering

The `k`-means benchmark uses Lloyd’s algorithm [6] to divide a set of points in a `d`-dimensional space into `k` disjoint clusters. Given an arbitrary set of initial clusters, the algorithm iterates over the following steps:

1. For each point, assign that point to whichever cluster is closest (by Euclidean distance to the cluster centroid).
2. For each cluster, update the centroid (the arithmetic mean of all points assigned to that cluster).

Distributed computation is straightforward: each process holds a portion of the points and computes cluster assignments and centroid contributions for each point. At each iteration, a master process collects all centroid contributions, computes the aggregates, checks if the computation has converged, and if not, communicates the updated values to all workers.

Figure 1 shows the main structure of a distributed `k`-means computation with APGAS. The state is split between the master’s view of 1) the centroids and 2) the contributions being collected, and the workers’ place-local memory, comprising a subset of points and the local view of the centroids. The place-local memory is held in local, of type `GlobalRef[LocalData]`.

---

1 The name comes from the fact that a `GlobalRef` is available globally, even though it points to place-local objects.
The structure of the computation, including the distribution aspect, is fully explicit in the code: the outermost `while` loop iterates until convergence, the `for` loop spawns an activity to be run asynchronously at each place as indicated by `asyncAt`, which in turn spawns a remote activity at the master place to combine the place’s local view with the master’s view. Finally, `finish` ensures that all remote work has completed before proceeding to the next iteration. An aspect of the code that can be harder to grasp is the movement of data: the value of `currentCentroids` is sent from the master to a worker by letting the variable be captured in the closure passed to `asyncAt`. Note that while `local` is a GlobalRef and is therefore never serialized implicitly, we use apply to dereference it and thus pass a copy of the data of type `LocalData` to the master process in the nested `asyncAt`. Finally, note that the code that adds the contribution of a worker to the master values is synchronized to avoid data races.

For contrast, Figure 2 shows the related parts of an actor-based implementation of k-means clustering using Akka. Almost as a dual to the APGAS implementation, the movement of data is entirely explicit, but the control flow must be inferred from the flow of messages: the master actor sends itself `Run` messages to continue the computation, and must keep count of how many `Updated` messages it received from workers to know when an iteration is complete. There is no need for data synchronization, as the model enforces that message processing within an actor is always a sequential operation.

### 4. Unbalanced Tree Search (UTS)

The UTS benchmark measures the rate of traversal of a tree generated on the fly using a splittable random number generator [9]. The problem specification describes several cryptographic laws for computing the number of children of a node and their hashes. This results in trees that are deterministic but unbalanced in unpredictable ways.

A sequential implementation of UTS is straightforward: the code maintains a work list of nodes to expand, and repeatedly pops one and adds its children to the list. It terminates when the list is empty. In contrast, a parallel and distributed implementation of UTS is a challenge because of imbalance. We implement distributed work stealing with lifelines [11].

**Distributed Algorithm.** A fixed collection of workers collaborate on the traversal. The workers are organized in a ring. Each worker maintains a work list of pending nodes to visit and count of nodes already traversed. Each worker primarily processes its own list, following the sequential algorithm.
If the list becomes empty, the worker tries to steal nodes from another random worker. If this fails because the victim’s work list is empty as well, the worker sends a request to the next worker in the ring—its lifeline—and stops. If this lifeline now has or later obtains nodes to process, it deals a fraction of these nodes to the requester. One work list is initialized with the root node of the traversal. The traversal is complete when all workers have stopped and there are no deal messages from a lifeline in flight. The sum of the node counts is computed at that point.

Each worker can be in one of three states; work: the worker is processing nodes from its work list, wait: the worker is attempting to steal nodes from a random victim and waiting for the result, and idle: the worker has signaled its lifeline and stopped.

**Implementation in APGAS.** We focus here on two aspects of the implementation: active messages and termination. Figure 3 shows a fraction of the Worker class. When a worker has run out of work and stealing has failed, the protocol dictates that it goes into idle mode and signals the next worker in the ring that it has done so. This corresponds in the code to the completion of the run() task by the invocation of lifelineReq(). This second method implements an active message pattern: the execution of lifeline.set(true) happens at place nextInRing. This works because the implicit this captured in the closure has type PlaceLocal and is therefore resolved to the Worker instance unique to the destination place. Reactivation of a worker that has gone idle is achieved in a similar way; its lifeline runs:

```
asyncAt(prevInRing) { lifelineDeal(newWork) }
```

This, as shown in Figure 3, spawns a task that enters run().

Distributed termination detection is notoriously difficult to implement correctly and efficiently. For instance in UTS, observing that all workers are idle does not guarantee that the traversal is complete as messages containing nodes to process might still be in flight. In our code, however, a single invocation of finish solves the problem. We invoke our distributed computation from the first place as

```
finish { worker.run() }
```

As shown in Figure 3, when a worker goes into idle mode, the corresponding task completes. Since finish guards all tasks transitively, it terminates exactly when the last work item has been exhausted.

**Implementation with Akka.** Because Akka embraces explicit messaging and actors that act as state machines, the code follows the protocol description very closely. For instance, the code corresponding to a worker being reactivated by its lifeline is:

```
case LifelineDeal(wl) ⇒
    workList.merge(wl); become(working); self ! Work
```

A significant challenge, however, lies in detection termination. We implemented a protocol where workers that go into idle mode additionally communicate to a central worker how many times they have sent lifeline messages, and by aggregating all counts, the central worker can detect when no messages are in flight.

### 5. Performance Evaluation

We ran our APGAS and Akka implementations of k-means and UTS on a 48 core machine, measuring the performance of configurations with 1, 2, 4, 8, 16, and 32 workers. For the APGAS programs, the number of workers corresponds to the number of places. For the Akka programs, n workers correspond to n + 1 actors: both benchmarks use the idiom of a master actor supervising the workers and detecting termination, as described in Sections 2 and 3. Because we are primarily interested in the scaling profile of our applications, we normalize the performance by the number of workers.

We ran our Akka programs by allocating one process for each worker actor, and using akka-remote for communication. This configuration is close to APGAS in terms of communication constraints, the problem input size to 32 million 4-dimensional points and 5 centroids, and measured performance as the number of iterations per second. The core computational code (determining the closest centroid for each point) is common to the benchmarks. Figure 4 shows the effect of scaling the number of workers for the APGAS and Akka implementations (note the tight scale). The scaling profiles are overall similar, with an initial improvement in per-worker throughput, possibly due to increased available memory bandwidth when using multiple sockets.

---

1. Places in APGAS are currently only realized as separate processes.
For UTS, we measured the rate of traversal of a tree of $4^{2^b}$ billion nodes, in millions of nodes per second (Mn/s). Most of the computational work is hashing, for which the code is shared. Figure 5 shows that the scaling profiles are similar for the two implementations.

6. Implementation Status

The APGAS library is implemented in about 2,000 lines of Java 8 code, with a Scala wrapper of about 200 lines. It uses the fork/join framework for scheduling tasks in each place. The library exposes its ExecutorService, making it possible in principle to develop applications that use APGAS in cooperation with Scala futures. Distribution is built on top of the Hazelcast in-memory data grid [1]. APGAS relies on Hazelcast to assemble clusters of JVMs and invoke remote tasks.

The Scala layer defines the Serialization type class as a mechanism to handle all Scala types uniformly, converting them to types compatible with java.io.Serializable, as required by Hazelcast. An alternative would be to bypass Java serialization entirely and use, e.g., pickling [7].

Another possible improvement is the handling of capture in closures: environment capture is a mechanism central to APGAS, but is error prone. The problem is well-known and the X10 compiler, for instance, handles it with custom warnings. In APGAS for Scala, using spores with properly defined headers [8] would help clarify the movement of data between places.

7. Conclusion

APGAS is a concurrent and distributed programming model where the structure of computation and distribution is fully explicit. Our work brings this model to Scala. We demonstrated the coding style through examples, showing that the resulting programs, while following a different structure, are comparable in complexity and performance to actor-based implementations.

References