Asynchronous Programming in the Abstract Behavioural Specification Language

Azadbakht, K.

Chapter 3

Preferential Attachment on Distributed Systems

3.1 Introduction

Massive social networks are structurally different from small networks synthesized by the same algorithm. Furthermore there are many patterns that emerge only in massive networks [56]. Analysis of such networks is also of importance in many areas, e.g. data-mining, network sciences, physics, and social sciences [16]. Nevertheless, generation of such extra-large networks necessitates an extra-large memory in a single server in the centralized algorithms.

The major challenge is generating large-scale social networks utilizing distributed-memory approaches where the graph, generated by multiple processes, is distributed among multiple corresponding memories. Few existing methods are based on a distributed implementation of the Preferential Attachment model (PA, chapter 2) among which some methods are based on a version of the PA model which does not fully capture its main characteristics. In contrast, we aim for a distributed solution which follows the original PA model, i.e., preserving the same probability distribution as the sequential one. The main challenge of a faithful distributed version of PA is to manage the complexity of the communication and synchronization involved.

In a distributed version, finding a target node in order for the new node to make connection with may cause an unresolved dependency, i.e., the target itself is not yet resolved. However this kind of dependencies must be preserved and the to-be-resolved target will be utilized when it is resolved. How to preserve these dependencies and their utilization give rise to low-level explicit management of the dependencies or, by means of powerful programming constructs, high-level implicit management of them.

The main contribution of this chapter is a new distributed implementation of an ABS model of PA. In this chapter, we show that ABS can be used as a powerful programming language for efficient implementation of cloud-based distributed

applications.

This chapter is organized as follows: Section 3.2 elaborates on the high-level proposed distributed algorithm using the notion of cooperative scheduling and futures. In Section 3.3, implementation-specific details of the model are presented. Finally, Section 3.4 concludes the chapter.

Related Work. In chapter 2 (section 2.4) the existing related work for sequential and parallel implementations of PA is presented. The work in this chapter is inspired by the work in [4] where a low-level distributed implementation of PA is given in MPI: the implementation code remains closed source (even after contacting the authors) and, as such, we cannot validate their presented results (e.g, there are certain glitches in their weak scaling demonstration), nor compare them to our own implementation.

Since efficient implementation of PA is an important and challenging topic, further research is called for. Moreover, our experimental data are based on a high-level model of the PA which abstracts from low-level management of process queues and corresponding synchronization mechanism as used in [4].

In [68], a high-level distributed model of the PA in ABS has been presented together with a high-level description of its possible implementation in Java. However, as we argue in Section 3, certain features of ABS pose serious problems to an efficient distributed implementation in Java. In this chapter, we show that these problems can be solved by a run-time system for ABS in Haskell and a corresponding source-to-source translation. An experimental validation of a scalable distributed implementation based on Haskell is presented in [10].

3.2 Distributed PA

In this section, we present a high-level distributed solution for PA which is similar to the ones proposed for multicore architectures in [12] (chapter 2) and distributed architectures in [4, 68], in a sense that they adopt *copy model* introduced in [54] to represent the graph. The main data structure used to model the graph which represents the social network is given in section 2.3.1. We use the same data structure for distributed PA as well.

The sequential algorithm of PA based on copy model is fairly straightforward and the unresolved slots of the array are resolved from left to right. The distributed algorithms however introduce more challenges. First of all, the global array should be distributed over multiple machines as local arrays. The indices of the global array are also mapped to the ones in the local arrays according to the partitioning policy. Secondly, there is the challenge of *unresolved dependencies*, a kind of dependency where the target itself is not resolved yet since either the process responsible for the target has not processed the target slot yet or the target slot itself is dependent on another target slot (chain of dependencies). Synchronization between the processes

to deal with the unresolved dependencies is the main focus of this chapter. Next we present the basic synchronization and communication mechanism underlying our distributed approach and its advantages over existing solutions.

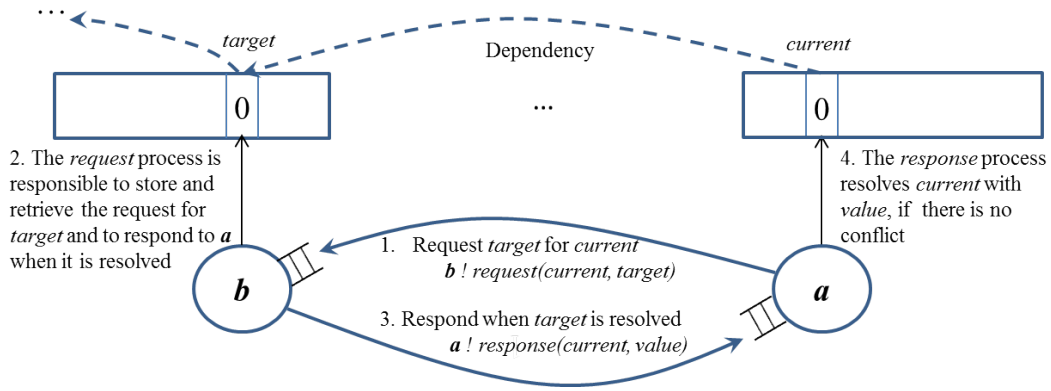
3.2.1 The Distributed ABS Model of PA

Two approaches are represented in Figure 3.1 which illustrate two different schemes of dealing with the unresolved dependencies in a distributed setting. In order to remain consistent with the original PA, both schemes must keep the unresolved dependencies and use the value of the target when it is resolved. Scheme A (used in [4]) utilizes message passing. If the target is not resolved yet, actor b explicitly stores the request in a data structure until the corresponding slot is resolved. Then it communicates the value with actor a. Actor b must also make sure the data structure remains consistent (e.g., it does not contain a request for a slot which is already responded).

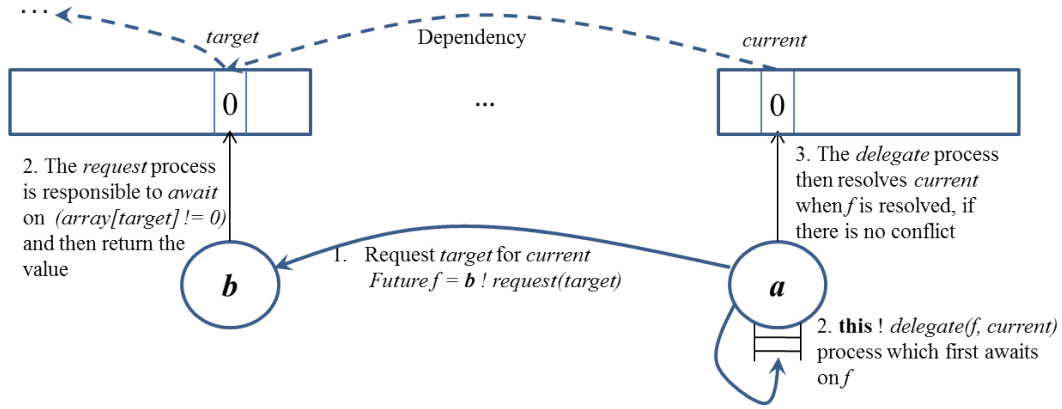
In addition to message passing, scheme B utilizes the notion of *cooperative scheduling*. Instead of having an explicit data structure, scheme B simply uses the *await* statement on $(target \neq 0)$. It suspends the request process until the target is resolved. The value is then communicated through the return value to actor a. Also *await* f? is skipped if the future f is resolved, and suspends the current process otherwise. This statement is used to synchronize on the return value of a called method. The above-mentioned await constructs eliminates the need for an explicit user-defined data structure for storing and retrieval of the requests. The following section describes an ABS implementation of the scheme B.

An ABS-like pseudo code which represents scheme B in the above section is given in Figure 3.2. The full implementation of the model is provided online[1]. The main body of the program, which is not mentioned in the figure, is responsible to set up the actors by determining their partitions, and sending them other parameters of the problem, e.g., n and m. Each actor then processes its own partition via *run* method. The function *whichActor* calculates and returns the index of the actor containing the target slot, based on n, m and the partitioning method. The request for the slot is then sent asynchronously to the actor and the future variable is sent as a parameter to the *delegate* function where the future value is obtained and checked for conflict. If there is no conflict, i.e., the new target is not previously taken by the source, then the slot is written with the target value. Recall that the one global array is divided into multiple local arrays, one per actor. Based on the partitioning method, n and m there is a mapping from the global indices to the local ones. The function *whichSlot* maps an index of the global array to the index of a local array. The *request* method is responsible to map the global index of the target to the local index function (via *whichSlot*) and *awaits* on it and returns the value once the slot is resolved. Note that, based on the same optimization of the array size discussed in chapter 2, the

[1] `https://github.com/kazadbakht/PA/blob/master/src/DisPA.abs`



(a) Simple message passing



(b) Message passing with futures and cooperative scheduling

Figure 3.1: The process of dealing with unresolved dependencies in an actor-based distributed setting

method *result(arr, target)* checks if the value for the index *target* is calculable then it returns the calculated value. Otherwise it checks the corresponding array index in the array and returns the value.

```
1: Each actor O executes the following in parallel
2: Unit run(...)
3: for each node i in the partition do
4:     for j = 1 to m do step
5:         target ← random[1..(i − 1)2m]
6:         current = (i − 1)m + j
7:         x = whichActor(target)
8:         Fut < Int > f = actor[x] ! request(target)
9:         this ! delegate(f, current)
10:
11:
12: Int request(Int target)
13: localTarget = whichSlot(target)
14: await (result(arr, localTarget) ≠ 0)
15:                                   ▷ At this point the target is resolved
16: return result(arr, localTarget)
17:
18:
19: Unit delegate(Fut < Int > f, Int current)
20: await f?
21: value = f.get
22: localCurrent = whichSlot(current)
23: if duplicate(value, localCurrent) then
24:     target = random[1..(current/m)2m]
25:                                   ▷ Calculate the target for the current again
26:     x = whichActor(target)
27:     Fut < Int > f = actor[x] ! request(target)
28:     this.delegate(f, current)
29: else
30:     arr[localCurrent] = value                                  ▷ Resolved
31:
32:
33: Boolean duplicate(Int value, Int localCurrent)
34: for each i in (indices of the node to which localCurrent belongs) do
35:     if arr[i] == value then
36:         return True
37: return False
```

Figure 3.2: The sketch of the proposed approach

3.3 Implementation

The distributed algorithm of Figure 3.2 is implemented directly in ABS, which is subsequently translated to Haskell code [10], by utilizing the ABS-Haskell [20] transcompiler (source-to-source compiler). The translated Haskell code is then linked against a Haskell-written parallel and distributed runtime API. Finally, the linked code is compiled by a Haskell compiler (normally, GHC) down to native code and executed directly.

The performance results of an experimental validation of the proposed approach in ABS-Haskell transcompiler is presented in [10]. The parallel runtime treats ABS active objects as Haskell's lightweight threads (also known as green threads), each listening to its own concurrently-modifiable process queue: a method activation pushes a new continuation to the end of the callee's process queue. Processes awaiting on futures are lightweight threads that will push back their continuation when the future is resolved; processes awaiting on boolean conditions are continuations which will be put back to the queue when their condition is met. The parallel runtime strives to avoid busy-wait polling both for futures by employing the underlying OS asynchronous event notification system (e.g. epoll, kqueue), and for booleans by retrying the continuations that have part of its condition modified (by mutating fields) since the last release point.

For the distributed runtime we rely on Cloud Haskell [32], a library framework that tries to port Erlang's distribution model to the Haskell language while adding type-safety to messages. Cloud Haskell code is employed for remote method activation and future resolution: the library provides us means to serialize a remote method call to its arguments plus a static (known at compile time) pointer to the method code. No actual code is ever transferred; the active objects are serialized to unique among the whole network identifiers and futures to unique identifiers to the caller object (simply a counter). The serialized data, together with their types, are then transferred through a network transport layer (TCP,CCI,ZeroMQ); we opted for TCP/IP, since it is well-established and easier to debug. The data are de-serialized on the other end: a de-serialized method call corresponds to a continuation which will be pushed to the end of the process queue of the callee object, whereas a de-serialized future value will wake up all processes of the object awaiting on that particular future.

The creation of Deployment Components is done under the hood by contacting the corresponding (cloud) platform provider to allocate a new machine, usually done through a REST API. The executable is compiled once and placed on each created machine which is automatically started as the 1st user process after kernel initialization of the VM has completed.

The choice of Haskell was made mainly for two reasons: the ABS-Haskell backend seems to be currently the fastest in terms of speed and memory use, attributed perhaps to the close match of the two languages in terms of language features:

Haskell is also a high-level, statically-typed, purely functional language. Secondly, compared to the distributed implementation sketched in Java [68], the ABS-Haskell runtime utilizes the support of Haskell's lightweight threads and first-class continuations to efficiently implement multicore-enabled cooperative scheduling; Java does not have built-in language support for algebraic datatypes, continuations and its system OS threads (heavyweight) makes it a less ideal candidate to implement cooperative scheduling in a straightforward manner. On the distributed side, layering our solution on top of Java RMI (Remote Method Invocation) framework was decided against for lack of built-in support for asynchronous remote method calls and superfluous features to our needs, such as code-transfer and fully-distributed garbage collection.

3.3.1 Implementing Delegation

The distributed algorithm described in Section 3 uses the concept of a *delegate* for asynchronicity: when the worker actor demands a particular slot of the graph array, it will spawn asynchronously an extra delegate process (line 9) that will only execute when the requested slot becomes available. This execution scheme may be sufficient for preemptive scheduling concurrency (with some safe locking on the active object's fields), since every delegate process gets a fair time slice to execute; however, in cooperative scheduling concurrency, the described scheme yields sub-optimal results for sufficient large graph arrays. Specifically, the worker actor traverses its partition from left to right (line 3), spawning continuously a new delegate in every step; all these delegates cannot execute until the worker actor has released control, which happens upon reaching the end of its `run` method (finished traversing the partition). Although at first it may seem that the worker actors do operate in parallel to each other, the accumulating delegates are a space leak that puts pressure on the Garbage Collector and, most importantly, delays execution by traversing the partitioned arrays "twice", one for the creation of delegates and one for "consuming them".

A naive solution to this space leak is to change lines 8,9 to a synchronous instead method call (i.e. `this.delegate(f,current)`). However, a new problem arises where each worker actor (and thus its CPU) continually blocks waiting on the network result of the request. This intensely sequentializes the code and defeats the purpose of distributing the workload, since most processors are idling on network communication. The intuition is that modern CPUs operate in much larger speeds than commodity network technologies. To put it differently, the worker's main calculation is much faster than the round-trip time of a request method call to a remote worker. Theoretically, a synchronous approach could only work in a parallel setting where the workers are homogeneous processors and requests are exchanged through shared memory with memory speed near that of the CPU processor. This hypothesis requires further investigation.

```
1: Unit run(...)
2: for each node i in the partition do
3:     for j = 1 to m do step
4:         target ← random[1..(i − 1)2m]
5:         current = (i − 1)m + j
6:         x = whichActor(target)
7:         Fut < Int > f = actor[x]! request(target)
8:         aliveDelegates = aliveDelegates + 1
9:         this! delegate(f, current)
10:        if aliveDelegates = maxBoundWindow then
11:            await aliveDelegates <= minBoundWindow
```

Figure 3.3: The modified run method with window of delegates.

We opted instead for a middle-ground, where we allow a window size of delegate processes: the worker process continues to create delegate processes until their number reaches the upper bound of the window size; thereafter the worker process releases control so the delegates have a chance to execute. When only the number of alive delegate processes falls under the window's lower bound, the worker process is allowed to resume execution. This algorithmic description can be straightforwardly implemented in ABS with boolean awaiting and a integer counter field (named *this.aliveDelegates*). The modification of the `run` is shown in Figure 3.3; Similarly the `delegate` method must be modified to decrease the *aliveDelegates* counter when the method exits.

Interestingly, the size of the window is dependent on the CPU/Network speed ratio, and the Preferential Attachment model parameters: nodes (n) and degree (d). In [10], the performance results of the PA model presented in this chapter in the Haskell backend are given. We empirically tested and used a fixed window size of $[500, 2000]$. Finding the optimal window size that keeps the CPUs busy while not leaking memory by keeping too much delegates alive for a specific setup (cpu,network,n,d) is planned for future work.

3.4 Conclusion and Future Work

In this chapter, we have presented a high-level distributed-memory algorithm that implements synthesizing artificial graphs based on Preferential Attachment mechanism. The algorithm avoids low-level synchronization complexities thanks to ABS, an actor-based modeling framework, and its programming abstractions which support *cooperative scheduling*. The experimental results for the proposed algorithm presented in [10] suggest that the implementation scales with the size of the distributed system, both in time but more profoundly in memory, a fact that permits the generation of PA graphs that cannot fit in memory of a single system.

For future work, we are considering combining multiple request messages in a single TCP segment; this change would increase the overall execution speed by having a smaller overhead of the TCP headers and thus less network communication between VMs, and better network bandwidth. In another (orthogonal) direction, we could utilize the many cores of each VM to have a parallel-distributed hybrid implementation in ABS-Haskell for faster PA graph generation.