# Distributed Network Generation Based on Preferential Attachment in ABS

Keyvan Azadbakht(✉), Nikolaos Bezirgiannis, and Frank S. de Boer

Centrum Wiskunde & Informatica (CWI), Amsterdam, Netherlands
{k.azadbakht,n.bezirgiannis,f.s.de.boer}@cwi.nl

**Abstract.** Generation of social networks using Preferential Attachment (PA) mechanism is proposed in the Barabasi-Albert model. In this mechanism, new nodes are introduced to the network sequentially and they attach to the existing nodes preferentially where the preference can be based on the degree of the existing nodes. PA is a classical model with a natural intuition, great explanatory power and interesting mathematical properties. Some of these properties only appear in large-scale networks. However generation of such extra-large networks can be challenging due to memory limitations. In this paper, we investigate a distributed-memory approach for PA-based network generation which is scalable and which avoids low-level synchronization mechanisms thanks to utilizing a powerful programming model and proper programming constructs.

**Keywords:** Distributed programming · Social network · Preferential Attachment · Actor model · Synchronization

## 1 Introduction

Social networks appear in many domains, e.g., communication, friendship, and citation networks. These networks are different from random networks as they demonstrate structural features like power-law degree distribution. There exist certain models which generate artificial graphs that preserve the properties of real world networks (e.g., [1–3]), among which Barabasi-Albert model of scale-free networks, which is based on Preferential Attachment (PA) [3], is one of the most widely-used ones, mainly because of its natural intuition, great explanatory power and simple mechanism [4].

Generating network based on PA is inherently a sequential task as there is a sequence among the nodes in terms of their addition to the network. The nodes are added preferentially to the graph. The preference is the node degrees in the graph, i.e., the higher a node degree, the higher probability with which the new node makes connection.

Massive networks are structurally different from small networks synthesized by the same algorithm. Furthermore there are many patterns that emerge only in massive networks [5]. Analysis of such networks is also of importance in many


B. Steffen et al. (Eds.): SOFSEM 2017, LNCS 10139, pp. 103–115, 2017.
DOI: 10.1007/978-3-319-51963-0_9

areas, e.g. data-mining, network sciences, physics, and social sciences [6]. 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 PA model 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 paper is a new scalable distributed implementation of an ABS (Abstract Behavioral Specification) [7] model of PA. The ABS language is a high-level actor-based executable modeling language which is tailored towards modeling distributed applications and which supports a variety of tool-supported techniques for, e.g., verification [8] and resource analysis [9]. In this paper, we show that ABS also can be used as a powerful programming language for efficient implementation of cloud-based distributed applications. The underlying runtime system and compiler are written in the Haskell language integrating the Cloud Haskell API [10].

The paper is organized as follows: The description of ABS language and its Haskell backend is given in Sect. 2. Section 3 elaborates on the high-level proposed distributed algorithm using the notion of cooperative scheduling and futures. In Sect. 4, implementation-specific details and experimental results are presented. Finally, Sect. 5 concludes the paper.

*Related Work.* Efficient implementation of PA model has been investigated in, e.g., [4,11–15]. Some of these works still focus on the sequential approach (e.g., [4,11,12]). The main proposal of such methods is to adopt data structures which improve time and memory complexity. There are also parallel and distributed proposals: [13,14] do not fully capture the main properties expected in the original model of graph generation; [15] also requires complex synchronization and communication management.

Our work was inspired by the work in [15] 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 [15].

In [16] 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 Sect. 4, certain features of ABS pose serious problems to an efficient distributed implementation in Java. In this paper, we show that these problems can be solved by a runtime system for ABS in Haskell and a corresponding source-to-source translation. We do so by providing an experimental validation of a scalable distributed implementation based on Haskell.

## 2 ABS: The Modeling Framework

The Abstract Behavioral Specification language (ABS for short) [7] is a modeling language for concurrent systems. Its formal operational semantics permit the analysis [9], and verification [8] of complex concurrent models. Moreover, the ABS language is executable which means the user can generate executable code and integrate it to production—currently backends have been written to target Java, Erlang and Haskell [17] and ProActive [18] software.

ABS at its core is a purely functional programming language, with support for pure functions (functions that disallow side-effects), parametrically polymorphic algebraic datatypes (e.g. `Maybe<A>`) and pattern matching over those types. At the outside sits the imperative layer of the language with the Java-reminiscing class, interface, method and attribute definitions. Unlike Java, the objects in ABS are typed exclusively by interface with the usual nominal subtyping relations—ABS does not provide any means for class (code) inheritance. It also attributes the notion of *concurrent object group*, which is essentially a group of objects which share control [7]. Note that a complement to this notion where the active objects share the data, i.e., the message queue, instead of control is studied in [19].

Besides the common synchronous method calls to passive objects $o.m(\bar{e})$, ABS introduces the notion of concurrent objects (also known as active objects). These concurrent objects interact primarily via asynchronous method invocations and futures. An asynchronous method invocation is of the form $f = o!m(\bar{e})$, where $f$ is a future used as a reference to the return value of the asynchronous method call $m$. The method call itself will generate a process which is stored in the process queue of the callee object of the call. Futures can be passed around and can be queried for the value they contain. The query $r = f.get$ blocks the execution of the active object until the future $f$ is resolved, and returns its value. On the other hand, the statement *await f?* additionally releases control. This allows for scheduling of another process of the same active object and as such gives rise to the notion of *cooperative scheduling*: releasing the control cooperatively so another enabled process can be (re)activated. ABS provides two other forms of

releasing control: the `await b` statement which will only re-activate the process when the given boolean condition $b$ becomes true (e.g. `await this.field==3`), and the `suspend` statement which will unconditionally release control to the active object. Note that the ABS language specification does not fix a particular scheduling strategy for the process queue of active objects as the ABS analysis and verification tools will explore many (if all) schedulability options; however, ABS backends commonly implement such process queues with FIFO ordering.

Since we are interested in the implementation of a distributed ABS model, we utilize the cloud extension to the ABS standard language, as implemented in [17]. This extension introduces the *Deployment Component* (DC), which abstracts over the resources for which the ABS program gets to run on. In the simplest case, the DC corresponds to a Cloud Virtual Machine executing some ABS code, though this could be extended to include other technologies as well (e.g. containers, microkernels). The DC, being a first class citizen of the language, can be created (`DC dc1 = new AmazonDC(cpuSpec,memSpec)`) and called for (`dc1 ! shutdown()`) as any other ABS concurrent object. The DC interface tries to stay as abstract as possible by declaring only two methods `shutdown` to stop the DC from executing ABS code while freeing its resources, and `load` to query the utilization of the DC machine (e.g. UNIX load). Concrete class implementations to the DC interface are (cloud) machine provider specific and thus may define further specification (cpu, memory, or network type) or behaviour.

Initially, the Deployment Component will remain idle until some ABS code is assigned to it by creating a new object inside using the expression `o = [DC: dc1] new Class(...)`, where $o$ is a so-called remote object reference. Such references are indistinguishable to local object references and can be normally passed around or called for their methods. The ABS language specification and its cloud extension do not dictate a particular Garbage Collection policy, but we assume that holding a reference to a remote object or future means that the object is alive, if its DC is alive as well.

## 3 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 [20] and distributed architectures in [15,16], in a sense that they adopt *copy model* introduced in [21] to represent the graph. To this aim, the description of the main data structure used to model the graph which represents the social network is given. Next we present the basic synchronization and communication mechanism underlying our approach and its advantages over existing solutions.

### 3.1 Array Representation of the Network Graph

In this paper, the social network is represented by the notion of graph, where the members of the network are the nodes and the connection between them are the edges. Generating a network based on Preferential Attachment is realized

by means of adding new nodes to the network preferentially. The preference is usually the degree of the nodes, that is, the higher the degree of a node, the higher probability that it makes connection with the new node. We assume there is a sequence between the nodes to be added to the network starting from 1 to $n$, each of which makes $m$ connections with the nodes in the existing graph. It implies that the initial state is a complete graph composed of the nodes 1 to $m+1$. $m$ is usually a small number.

Suppose node $u \in [m+2, n]$ is going to be attached to the existing graph with the nodes $[1, u-1]$. It is done by randomly selecting $m$ *distinct* nodes from $1, ..., u-1$, so that the probability of each node to be selected is proportional to its degree (to follow the PA model), that is, respectively $[p_1, ..., p_{u-1}]$ where

$$p_i = \frac{degree(i)}{\sum_{j=1}^{u-1} degree(j)} \qquad \sum_{i=1}^{u-1} p_i = 1$$

Figure 1 illustrates the array representation of the graph. Given the number of nodes $n$ and the number of connections per node $m$, the size of the array is known. As shown, $2m$ slots are allocated for the edges sourcing from a node, $u$ (in the figure, $m = 3$). The targets of $u$, represented by question mark (or later in implementation with *0*), are determined from the slots representing the edges sourcing from the nodes $[1, u-1]$ which are located previous to the node $u$. In order to generate the graph based on PA, the unresolved slots are resolved by randomly selecting the slots previous to the current node. The obtained values are then written as the targets of the current node, provided that there is no conflict between them. In case of conflict, the algorithm simply retries until all the targets are distinct for a specific node.

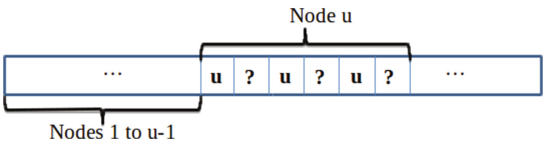



**Fig. 1.** The array representation of social network graph

The above-mentioned probability distribution is naturally applied through randomly selecting the slots with a uniform chance, since the number of slots keeping the value of a node is equal to its degree.

The sequential algorithm 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*, the one marked by $e$ in Fig. 2, a kind of dependency where the target itself is not resolved yet since either

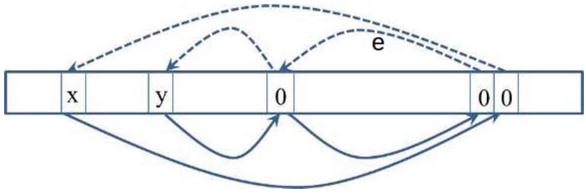



**Fig. 2.** Dependency and computation directions in the array

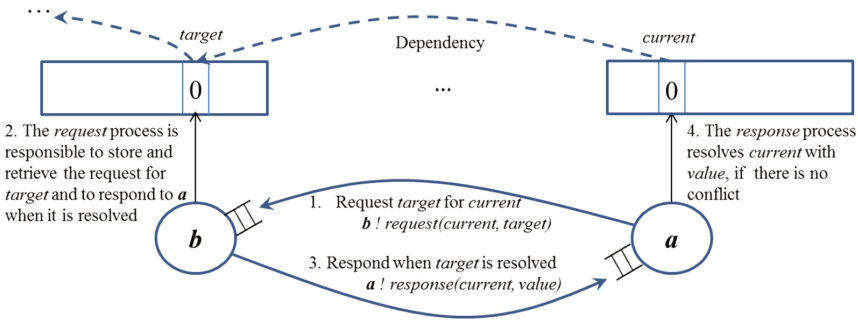
the process responsible for the target has not processed the target slot yet or the target slot itself is dependent on another target slot (chain of dependencies). Synchronization between the processes to deal with the unresolved dependencies is the main focus of this paper.

### 3.2 The Distributed ABS Model of PA

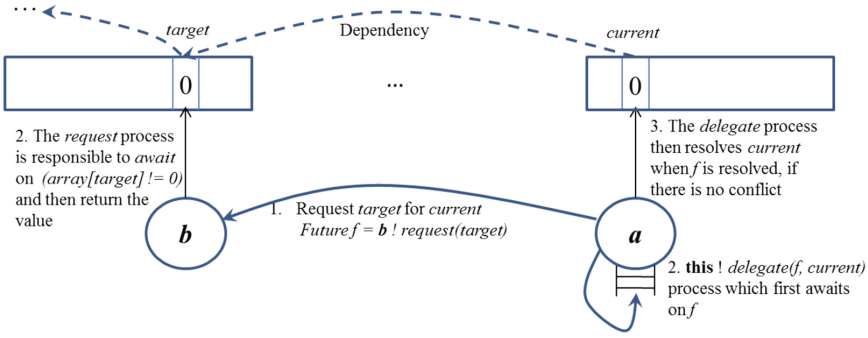
Two approaches are represented in Fig. 3 which illustrate two different schemes of dealing with the unresolved dependencies in a distributed setting. In order to remain consistent with the original PA, both schemes must keep the unresolved dependencies and use the value of the target when it is resolved. Scheme A (used in [15]) 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 $a$. The above-mentioned await construct eliminates the need for an explicit synchronization of the requests. The following section describes an ABS implementation of the scheme B and presents the performance results.

An ABS-like pseudo code which represents scheme B in the above section is given in Fig. 4. 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* returns the index of the actor containing the target slot. 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. The *request* method is responsible to map the global index of the target to the local index via *whichLocal* function and *await* on it and returns the value once the slot is resolved.




(a) Simple message passing




(b) Message passing with futures and cooperative scheduling

**Fig. 3.** The process of dealing with unresolved dependencies in an actor-based distributed setting

## 4 Implementation

The distributed algorithm of Fig. 4 is implemented directly in ABS, which is subsequently translated to Haskell code, by utilizing the ABS-Haskell [17] 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 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

```
1: Each actor O executes the following in parallel
2: Unit run(...)
3: for each node i in the partition do
4:     for j = 2 to 2m do j = j + 2 step
5:         target ← random[1..(i − 1) ∗ 2m]
6:         current = (i − 1) ∗ 2m + 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 (arr[localTarget] ≠ 0)
15:                                          ▷ At this point the target is resolved
16: return 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/(2m) ∗ 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
```

**Fig. 4.** The sketch of the proposed approach

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 [10], 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 [16], 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.

### 4.1 Implementing Delegation

The distributed algorithm described in Sect. 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 actors (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.

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 Fig. 5; 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 ($d$). 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.

```
1:  Unit run(...)
2:  for each node i in the partition do
3:      for j = 2 to 2m do j = j + 2 step
4:          target ← random[1..(i − 1) ∗ 2m]
5:          current = (i − 1) ∗ 2m + 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
```

**Fig. 5.** The modified run method with window of delegates.

### 4.2 Experimental Results

We ran the ABS-Haskell implementation of the PA algorithm by varying the graph size, on a distributed cloud environment kindly provided by the SURF foundation. The hardware consisted of identical virtual machines interconnected over a 10 Gbps ethernet network; each Virtual Machine (VM) was a single-core Intel Xeon E5-2698, 16 GB RAM running Ubuntu 14.04 Server edition. The runtime execution results are shown in Fig. 6; the execution time decreases while we add more VMs to the distributed system, which suggests that the distributed algorithm scales. However, still with 8 Virtual Machines the implementation cannot "beat" the execution time of 1 VM running PA sequentially; to achieve this we may need to include more VMs. The reason for this can be attributed to the significant communication overhead, since each worker will send a network packet for every request call made.

On the other hand, the memory consumption (Table 1) is more promising: a larger distributed system requires less memory per VM. For example with the largest tested graph size, a distributed system of 8 VMs requires approx. 2.5 times less memory per VM than a local system. This allows the generation of much larger PA graphs than would otherwise fit in a single machine, since the graph utilizes and is "distributed" over multiple memory locations. Finally, the repository at http://github.com/abstools/distributed-PA contains the ABS code for PA and instructions for installing the ABS-Haskell backend.

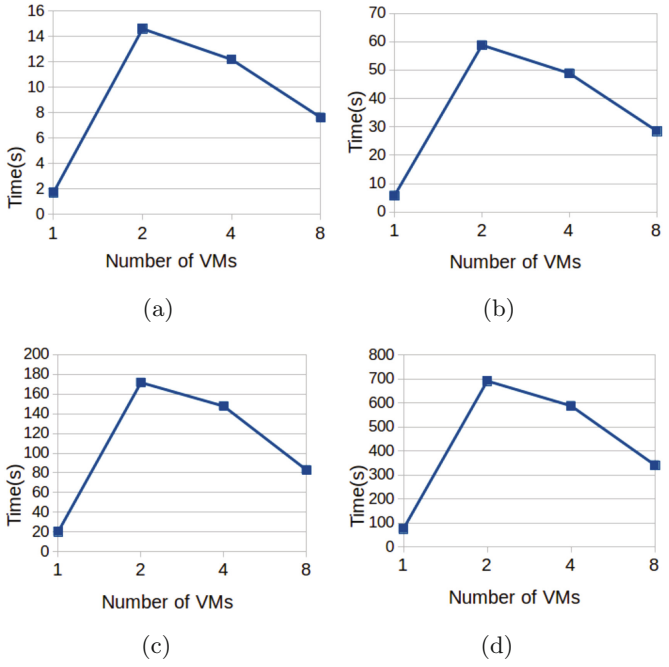



**Fig. 6.** Performance results of the distributed PA in ABS-Haskell for graphs of $n = 10^6$ nodes with degree $d =$ (a) 3, (b) 10 and $n = 10^7$ nodes with degree $d =$ (c) 3, (d) 10.

**Table 1.** Maximum memory residency (in MB) per Virtual Machine.

| Graph size | Total number of VMs | | | |
|---|---|---|---|---|
| | 1 | 2 | 4 | 8 |
| $n = 10^6, d = 3$ | 306 | 423 | 313 | 229 |
| $n = 10^6, d = 10$ | 899 | 1058 | 644 | 411 |
| $n = 10^7, d = 3$ | 1943 | 2859 | 1566 | 874 |
| $n = 10^7, d = 10$ | 6380 | 9398 | 4939 | 2561 |

## 5 Conclusion and Future Work

In this paper, we have presented a scalable, 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 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.

**Acknowledgments.** Partly funded by the EU project FP7-612985 UpScale (http://www.upscale-project.eu) and the EU project FP7-610582 ENVISAGE (http://www.envisage-project.eu). This work was carried out on the Dutch national HPC cloud infrastructure, a service provided by the SURF Foundation (http://surf.nl).

## References

1. Erdös, P., Rényi, A.: On the central limit theorem for samples from a finite population. Publ. Math. Inst. Hungar. Acad. Sci **4**, 49–61 (1959)
2. Watts, D.J., Strogatz, S.H.: Collective dynamics of 'small-world' networks. Nature **393**(6684), 440–442 (1998)
3. Barabási, A.L., Albert, R.: Emergence of scaling in random networks. Science **286**(5439), 509–512 (1999)
4. Tonelli, R., Concas, G., Locci, M.: Three efficient algorithms for implementing the preferential attachment mechanism in Yule-Simon stochastic process. WSEAS Trans. Inf. Sci. Appl. **7**(2), 176–185 (2010)
5. Leskovec, J.: Dynamics of Large Networks. ProQuest, Ann Arbor (2008)
6. Bader, D., Madduri, K., et al.: Parallel algorithms for evaluating centrality indices in real-world networks. In: International Conference on Parallel Processing, ICPP 2006, pp. 539–550. IEEE (2006)

7. Johnsen, E.B., Hähnle, R., Schäfer, J., Schlatte, R., Steffen, M.: ABS: a core language for abstract behavioral specification. In: Aichernig, B.K., Boer, F.S., Bonsangue, M.M. (eds.) FMCO 2010. LNCS, vol. 6957, pp. 142–164. Springer, Heidelberg (2011). doi:10.1007/978-3-642-25271-6_8
8. Din, C.C., Bubel, R., Hähnle, R.: KeY-ABS: a deductive verification tool for the concurrent modelling language ABS. In: Felty, A.P., Middeldorp, A. (eds.) CADE 2015. LNCS, vol. 9195, pp. 517–526. Springer, Heidelberg (2015). doi:10.1007/978-3-319-21401-6_35
9. Albert, E., Arenas, P., Correas, J., Genaim, S., Gómez-Zamalloa, M., Puebla, G., Román-Díez, G.: Object-sensitive cost analysis for concurrent objects. Softw. Test. Verification Reliab. **25**(3), 218–271 (2015)
10. Epstein, J., Black, A.P., Peyton-Jones, S.: Towards haskell in the cloud. In: ACM SIGPLAN Notices, vol. 46, pp. 118–129. ACM (2011)
11. Atwood, J., Ribeiro, B., Towsley, D.: Efficient network generation under general preferential attachment. Comput. Soc. Netw. **2**(1), 1 (2015)
12. Batagelj, V., Brandes, U.: Efficient generation of large random networks. Phys. Rev. E **71**(3), 036113 (2005)
13. Yoo, A., Henderson, K.: Parallel generation of massive scale-free graphs. arXiv preprint arXiv:1003.3684 (2010)
14. Lo, Y.C., Li, C.T., Lin, S.D.: Parallelizing preferential attachment models for generating large-scale social networks that cannot fit into memory. In: Privacy, Security, Risk and Trust (PASSAT), 2012 International Conference on and 2012 International Confernece on Social Computing (SocialCom), pp. 229–238. IEEE (2012)
15. Alam, M., Khan, M., Marathe, M.V.: Distributed-memory parallel algorithms for generating massive scale-free networks using preferential attachment model. In: Proceedings of SC13: International Conference for High Performance Computing, Networking, Storage and Analysis, p. 91. ACM (2013)
16. Şerbănescu, V., Azadbakht, K., de Boer, F.: A java-based distributed approach for generating large-scale social network graphs. In: Pop, F., Kołłodziej, J., Di Martino, B. (eds.) Resource Management for Big Data Platforms, pp. 401–417. Springer, Cham (2016)
17. Bezirgiannis, N., Boer, F.: ABS: a high-level modeling language for cloud-aware programming. In: Freivalds, R.M., Engels, G., Catania, B. (eds.) SOFSEM 2016. LNCS, vol. 9587, pp. 433–444. Springer, Heidelberg (2016). doi:10.1007/978-3-662-49192-8_35
18. Henrio, L., Rochas, J.: From modelling to systematic deployment of distributed active objects. In: Lluch Lafuente, A., Proença, J. (eds.) COORDINATION 2016. LNCS, vol. 9686, pp. 208–226. Springer, Heidelberg (2016). doi:10.1007/978-3-319-39519-7_13
19. Azadbakht, K., de Boer, F.S., Serbanescu, V.: Multi-threaded actors. arXiv preprint arXiv:1608.03322 (2016)
20. Azadbakht, K., Bezirgiannis, N., de Boer, F.S., Aliakbary, S.: A high-level and scalable approach for generating scale-free graphs using active objects. In: Proceedings of the 31st Annual ACM Symposium on Applied Computing, pp. 1244–1250. ACM (2016)
21. Kumar, R., Raghavan, P., Rajagopalan, S., Sivakumar, D., Tomkins, A., Upfal, E.: Stochastic models for the web graph. In: 2000 Proceedings of 41st Annual Symposium on Foundations of Computer Science, pp. 57–65. IEEE (2000)