A High-Level and Scalable Approach for Generating Scale-Free Graphs using Active Objects

Keyvan Azadbakht[1] Nikolaos Bezirgiannis[1] Frank S. de Boer[1] Sadegh Aliakbary[2]
[1]Centrum Wiskunde en Informatica, Amsterdam, Netherlands
{k.azadbakht, n.bezirgiannis, f.s.de.boer}@cwi.nl
[2]Faculty of Computer Science and Engineering, Shahid Beheshti University, Tehran, Iran
s_aliakbary@sbu.ac.ir

ABSTRACT

The Barabasi-Albert model (BA) is designed to generate scale-free networks using the preferential attachment mechanism. In the preferential attachment (PA) model, new nodes are sequentially introduced to the network and they attach preferentially to existing nodes. PA is a classical model with a natural intuition, great explanatory power and a simple mechanism. Therefore, PA is widely-used for network generation. However the sequential mechanism used in the PA model makes it an inefficient algorithm. The existing parallel approaches, on the other hand, suffer from either changing the original model or explicit complex low-level synchronization mechanisms. In this paper we investigate a high-level Actor-based model of the parallel algorithm of network generation and its scalable multicore implementation in Haskell.

CCS Concepts

•**Computing methodologies** → *Parallel programming languages;*

Keywords

Social Network, Actor Model, Multicore Processor, Parallel Algorithm, Cooperative Scheduling

1. INTRODUCTION

Social networks in the real world appear in various domains such as, among others, friendship, communication, collaboration and citation networks. Social networks demonstrate nontrivial structural features, such as power-law degree distributions, that distinguish them from random graphs. There exist various network generation models that synthesize artificial graphs that capture properties of real-world social networks. Some existing network generative models are the Erdos-Renyi (ER) [15] model of random graphs, the Watts-Strogatz (WS) [26] model of Small-world networks, and the Barabasi-Albert model of scale-free networks. Among these models, Barabasi-Albert model, which is based on Preferential Attachment [7], is one of the most commonly used models to produce artificial networks, because of its explanatory power, conceptual simplicity, and interesting mathematical properties [25]. The need for efficient and scalable methods of network generation is frequently mentioned in the literature, particularly for the preferential attachment process [2, 5, 8, 11, 16, 21–23, 25, 27]. Scalable implementations are essential since massive networks are important; there are fundamental differences between the structure of small and massive networks even if they are generated according to the same model, and there are many patterns that emerge only in massive networks [20]. Analysis of the large-scale networks is of importance in many areas, e.g. data-mining, network sciences, physics, and social sciences [6]. The property that we have focused on in this paper is the degree of the nodes and by preferential attachment (PA) we mean degree-based preferential attachment. In PA-based generation of the networks, each node is introduced to the existing graph preferentially based on the degrees of the existing nodes, i.e., the more the degree of an existing node, the higher the probability of choosing it as the target of a new connection.

The PA-based parallel and distributed versions of generating the scale-free graphs are based on a partitioning of the nodes and a parallel process for each partition which adds edges to its nodes. The edges are generated by random selection of target nodes. The data structure prescribed in the *Copy Model* [19] guarantees that the selection of the target is done consistently, e.g., the probability distribution of selecting the target nodes in the parallel version should remain the same as the distribution in the sequential one. However, from the point of view of the control flow, the following main problem arises: random selection requires synchronization between the parallel processes because of not-yet-resolved target nodes and the need for conflict resolution, namely, the selection of a node which has already been selected as a target of the given source node.

The main contribution of this paper is a high-level Actor-based model for the PA-based generation of networks which avoids the use of low-level intricate synchronization mechanisms. A key feature of the Actor-based model itself, so-called *cooperative scheduling*, however, poses a major chal-

SAC 2016,April 04-08, 2016, Pisa, Italy
©2016 ACM. ISBN 978-1-4503-3739-7/16/04…$15.00
DOI: http://dx.doi.org/10.1145/2851613.2851722

lenge to its implementation. In this paper we discuss the scalability of a multi-core implementation based on Haskell which manages cooperative scheduling by exploiting the high-level and first-class concept of continuations [24]. Continuations provide the means to "pause" (part of) the program's execution, and programmatically switch to another execution context; the paused computation can be later resumed. Thus, continuations can faithfully implement cooperative scheduling in a high-level, language-builtin manner.

The rest of the paper is organized as follows. The description of the Actor-based modeling framework which is used to model the PA-based generation of massive networks is given in section 2. Section 3 elaborates on parallelizing the PA model. The implementation-specific details and the results for the proposed solution are presented in Section 4. Section 5 mentions the related works. Finally we conclude in section 6.

2. THE MODELING FRAMEWORK

The Actor-based modeling framework supports concurrency and synchronization mechanisms for concurrent objects. It is based on the ABS (Abstract Behavioral Specification) language [18] which is developed for designing executable models of distributed object-oriented systems . The ABS uses asynchronous method calls, futures, interfaces for encapsulation, and cooperative scheduling of method activations [12] inside concurrent objects [17]. This feature combination results in a concurrent object-oriented model which is inherently compositional and which supports various analysis tools [4], e.g., resource analysis [3].

Apart from a functional layer which includes algebraic data types and pattern matching, the modeling framework additionally features global arrays as a mutable data structure shared among objects which fits well in the multicore setting to decrease the amount of costly message passing, and also to simplify the model. We extended the ABS to support the feature. In general, this feature can cause complicated and hard-to-verify programs and thus the programmer should use the feature in a disciplined manner, such as provided by our model (which restricts the model to single write access), to avoid race conditions.

3. PARALLELIZING THE PA MODEL

In this section we present the solution for the PA problem utilizing the idea of active objects cooperating in a multicore setting. For the solution we adopt the *copy model*, introduced in [19]. We first introduce the main data structure of the proposed approach which is based on the graph representation in *copy model*. Next we present the basic synchronization and communication mechanism underlying our solution and its advantages over existing solutions.

3.1 The Graph Representation

We introduce one shared array, arr, as the main data structure that holds the representation of the graph. The array consists of the edges of the graph. Each (i, j) where $i, j > 0$ and $j = i + 1$, and $j \ mod \ 2 = 0$ shows an edge in the graph between $arr[i]$ and $arr[j]$ (Figure 1a).

According to the PA, each node is added to the existing graph via a constant number of edges (referred to as m) targeting *distinct* nodes. There is also an initial clique, a complete graph with the size of $m0$ where $(m0 \geqslant m)$, which is stored at the beginning of the array. Therefore the size of the array is calculated based on the number of nodes, num, and the number of edges that connect each new node to the existing distinct nodes, m. The connections of a new node are established via a probability distribution of the degrees of the nodes in the existing graph, that is, the more the degree of the existing node, the more the probability of choosing it as the target. For instance, if the node n is the new node to be added to the graph with the existing graph with $[1..n-1]$ nodes then, according to equation 1, the probability distribution of choosing the existing nodes is $[p_1..p_{n-1}]$. ($deg(i)$ gives the degree of the node i in the existing graph)

$$p_i = \frac{deg(i)}{\sum_{j=1}^{n-1} deg(j)} \qquad \sum_{i=1}^{n-1} p_i = 1 \tag{1}$$

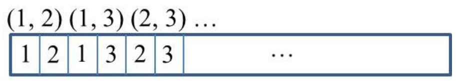
As we mentioned, the connections for the new node should be distinct. Therefore if a duplicate happens the approach retries to make a new connection until all the connections are distinct. This graph representation provides the above mentioned probability distribution since the number of occurrences of each node in the array is equal to its degree. As you see in figure 1b, in order to add node n to the existing graph containing $n-1$ nodes, with the assumption that $m = 3$, targets are selected randomly from the slots that are located previous to the node n. It is obvious that self-loop cannot happen, i.e., an edge whose source and target are the same. Figure 1c illustrates an optimization on the array so that the array only contains the targets of the edges since the sources for each node are calculable. The array is half size as the one in Figure 1b. Each slot in the array can have one of two states: *resolved* or *unresolved*. In the former case it contains the node number which is greater than zero, and in the latter it contains zero.

The sequential solution for the problem consists of processing the array from left to right to resolve all the slots. The parallel alternative, on the other hand, is to have multiple active objects processing partitions of the array in parallel. We distinguish between the following uses of indices. At the lowest level we have the indices of the slots. The next level is the id of the nodes. Each node contains slots. Finally at the top level we have the id of the partitions. Each partition contains nodes and consequently slots. In the proposed approach the partitions satisfy the following equations which express that the sets of indices of the partitions are mutually disjoint (equation 3); that their union is equal to the whole array (equation 2); furthermore, the sequence of slots of each node must be placed in one partition (equation 4) so that one active object resolves the new node and race conditions are avoided for the checking duplicates:

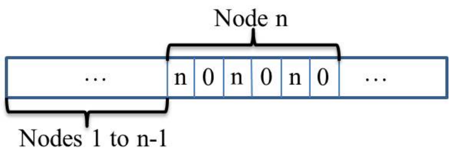
$$\bigcup_{i=1}^{w} par_i = G \tag{2}$$

$$\forall (1 \leq (i \neq j) \leq w). par_i \cap par_j = \emptyset \tag{3}$$

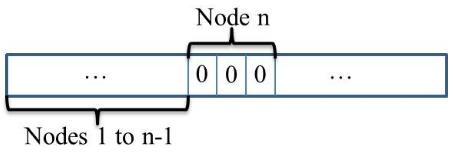
$$\forall i, j \in G. (node(i) = node(j)) \rightarrow (par(i) = par(j)) \tag{4}$$



(a) The array which represents the graph



(b) The nth node and its connections to the existing graph with $n-1$ nodes



(c) A memory usage optimization to the array based on *copy model*

Figure 1: The array representing the graph

where G is the global set containing all the indices of the shared array, w holds the number of partitions, par_i is the set which holds the indices of the ith partition of the array, $node(i)$ is a function that returns the node id to which the slot of the array with index i belongs, and $par(i)$ is a function that returns the partition id to which the index i belongs. Note that indices that belong to a specific node differ from the occurrences of that specific node in the array. The former indices are the slots that represents the edges that are created during introducing the *new* node to the graph, which its size is constant (denoted by m), while the latter changes during the graph generation.

There are different approaches to partition the array so that they hold the above equations, such as Consecutive and Round Robin Node Partitioning (CSP and RRP respectively). As it is shown in [2], RRP is more efficient and it is observed a better load balancing among processors as well as less unresolved chains of dependencies (Figure 2) which leads to less computational overhead. Therefore we have utilized RRP to partition the array among active objects.

3.2 Synchronization of Chains of Unresolved Dependencies

Each active object only resolves (i.e. writes to) the slots which belong to its own partition . Nevertheless it can read

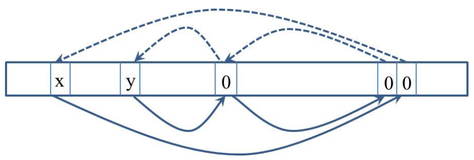
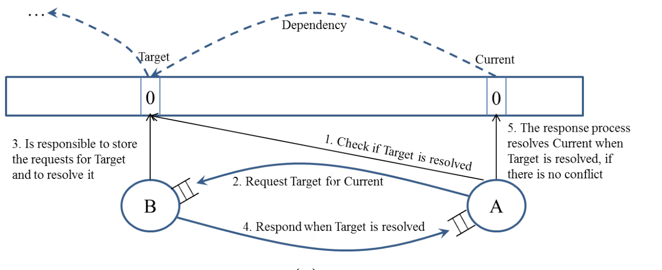
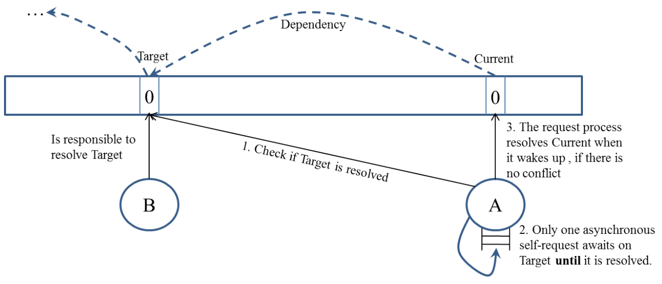


Figure 2: An example of the general sketch of dependencies (right to left) and computations (left to right)


…
Dependency
Target
Current
0
0
3. Is responsible to store the requests for Target and to resolve it
1. Check if Target is resolved
5. The response process resolves Current when Target is resolved, if there is no conflict
2. Request Target for Current
B
A
4. Respond when Target is resolved

(a)


…
Dependency
Target
Current
0
0
Is responsible to resolve Target
1. Check if Target is resolved
3. The request process resolves Current when it wakes up , if there is no conflict
B
A
2. Only one asynchronous self-request awaits on Target until it is resolved.

(b)

Figure 3: Two different solutions for the PA problem (the second one is the proposed approach)

all the slots throughout the array. In the parallel solution, an active object may select a slot as the target which is not resolved yet since either the other active object responsible for the target slot has not processed it or the target slot may wait for another target itself (see dependency chains in figure 2). The way waiting for unresolved slots is managed is crucial for the complexity of the model and its scalability. Next we describe the two main approaches to deal with unresolved dependencies (Figure 3):

Synchronization by communication: Active object A processes its own partition of the array and for each unresolved randomly selected slot it sends a request to the object B responsible for the target. When object B processes the request, it checks whether the slot is resolved. If it is not then it stores the information of the request (e.g. the sender id, the slot requiring the value of the target) in a corresponding data structure. Because B is the only object which writes to the target slot when it is resolved, it suffices that B answers all the stored requests waiting for the resolved target by broadcasting the value of the slot. As such this approach exploits the wait-notify pattern rather than busy-wait polling, and it can be efficient depending on how the programmer implements the data structure. However, this approach involves a low-level ad-hoc management of the requests through the explicit user-defined implementation of the storage and retrieval of the requests. Note that in this approach there are *exactly* two messages that have to be passed for each request for an unresolved slot (Figure 3a).

Synchronization by Cooperative Scheduling: Active object A processes its own partition of the array and for each unresolved randomly selected slot it sends an asynchronous *self* request for the target value. When object A schedules and processes the request it checks whether the slot is resolved or not (by a Boolean condition) and if not it *awaits* on this condition. This means that the request process is *suspended*. It is notified when the boolean condition evaluates to *true* and stored back in the object's queue of active processes (Figure 3b). This approach also avoids busy-waiting and follows the wait-notify pattern. However, the key feature in this approach is the use of cooperative scheduling in which the executing process of an active object can release conditionally the control cooperatively so that another process from the queue of the object can be executed. The continuation of the process which has conditionally released the control will be stored into a separate queue of suspended processes. These processes are stored again in the object's queue for execution when they are notified. The Haskell implementation of this mechanism takes care of the low-level storage, execution and suspension of the processes generated by asynchronous messages. The ABS code itself, see below, remains high-level by means of its programming abstractions describing asynchronous messaging and conditional release of control.

3.3 The Actor-based Model of PA

The main part of the encoding of our proposed approach is depicted in Figure 4. The worker objects are active objects which resolve their corresponding partitions. To this aim, each worker goes through its own partition and it checks a randomly selected target for each of its slots (note that m denotes the number of connections, or slots in the array, per node). If the target slot is already resolved then the worker takes the value and resolves the slot of the `current` index in case there is no conflict. If it is not resolved yet then it calls the *request* method asynchronously. The *request* method awaits on the target until it is resolved. Then it uses the value of the resolved target to resolve the current slot, if there is no duplicate. In case of a duplicate, the algorithm selects another target randomly in the same range as the previous one.

Note that the calls to the request method in lines 8 and 22 are asynchronous (denoted by exclamation mark) and synchronous (denoted by dot) respectively. The asynchronous call is introduced so as to spawn one process per each unresolved dependency. In the synchronous call, however, there is no need to spawn a new process since the current process is already introduced for the corresponding unresolved dependency. Note that suspension of such a process thus involves in general an entire call stack, which poses one of the major challenges to the implementation of ABS, but which is dealt with in Haskell by the high-level and first-class concept of continuations (described in more detail below).

```
 1: Each active object $O$ executes the following in parallel
 2: run(...) : **void**
 3: **for each** Node $i$ in the partition **do**
 4:     **for** $j = 1$ *to* $m$ **do**
 5:         $target \leftarrow$ random$[1..(i-1)*m]$
 6:         $current = (i-1)*m+j$
 7:         **if** $arr[target] = 0$ **then**
 8:             **this** ! $request(current, target)$
 9:         **else if** $duplicate(arr[target], current)$ **then**
10:             $j = j-1$          ▷ Repeat for the current slot $j$
11:         **else**
12:             $arr[current] = arr[target]$          ▷ Resolved
13:
14:
15: request($target : Int, current : Int$) : **void**
16: **await** $(arr[target] \neq 0)$
17:                         ▷ At this point the target is resolved
18: $value = arr[target]$
19: **if** $duplicate(value, current)$ **then**
20:     $target = random[1..target/m*m]$
21:                 ▷ Calculate the target for the current again
22:     **this**.$request(target, current)$
23: **else**
24:     $arr[current] = value$          ▷ Resolved
25:
26:
27: duplicate($value : Int, current : Int$) : $Boolean$
28: **for each** $i$ in (indices of the node to which $current$
    belongs) **do**
29:     **if** $arr[i] == value$ **then**
30:         **return** True
31: **return** False
```

Figure 4: The sketch of the proposed approach

4. IMPLEMENTATION AND RESULTS

The implementation of the proposed PA algorithm is done in the ABS language with its actively developed *abs2haskell* backend [9]. The backend translates ABS source-to-source down to Haskell code. Haskell was chosen because of its relative good execution speed and the straightforward mapping of ABS features to equivalent in Haskell. ABS, as described in Section 2, is a statically-typed, purely-functional language at its core with an object-oriented layer and strict evaluation; comparatively, Haskell is a statically-typed, purely-functional language with a by-default lazy evaluation strategy. The translation of the functional core becomes a one-to-one mapping, where we add on-top support for interfaces (through Haskell typeclasses) and subtyping (through Haskell existential types).

During execution, the de-facto Haskell compiler (GHC) employs for its runtime system a *M:N* hybrid threading model, where M lightweight (in terms of memory) threads are mapped to N SMP-enabled kernel threads through a time-sharing fashion (usually *M>N*). In our setting, each active object (actor) becomes such a lightweight thread, and instead ABS processes become coroutines, which are even more lightweight, suspendable computations implemented on top of Haskell's first-class continuations. The *continuation* is a language-builtin construct to suspend the current execution of a process in the program and store its "future" execution simply as data; the data include the process' closure (a function together with its environment, i.e. free variables) and the current call stack. Coroutines (in our case ABS processes) are cooperatively scheduled: a running coroutine may decide to suspend its execution and its associated active object (thread) will pick another process to run based on a scheduling strategy for processes (round-robin by default). In contrast, ABS objects are preemptively scheduled by the Haskell runtime system to take advantage of multicore parallelism.

The ABS code for the algorithm maintains a global, mutable, $O(1)$, boxed array: each array-cell is an ABS future coupled with a set of active objects (their thread references) as “listeners”. An ABS process will suspend its execution until the future of the array cell is resolved; the active object that resolves the future will inform the set of listeners to wake up the corresponding suspended processes. This extension of future-arrays is integrated naturally in the ABS ecosystem through the `await` on-boolean-condition. Finally, we use a random-number library with each active object having each own, separate random-number generator for performance reasons.

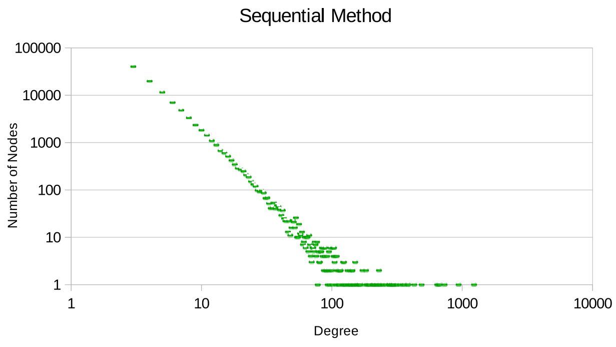
Results: We run the program of the PA-based generation of networks in ABS2Haskell based on the proposed approach on SURFsara cluster on a 16 core processor 2.30 GHz (Intelő Xeonő CPU E5-2698 0) with 128GB of memory [1].

The program is verified using a set of test cases (e.g. checking for the resolution of *all* edges of the graph and checking duplicates for the final graph). Figure 5 illustrates the degree distribution of two networks: one generated by sequential method which follows the original model of PA, and the other one by our proposed parallel method. According to this experiment, the degree distribution of the graphs generated by our proposed method follows a power-law degree distribution. In Figure 6, the performance and the scalability of the program is depicted for different input parameters. The performance of the program is good in comparison with the performance of the efficient sequential implementation of the PA in abs2haskell.

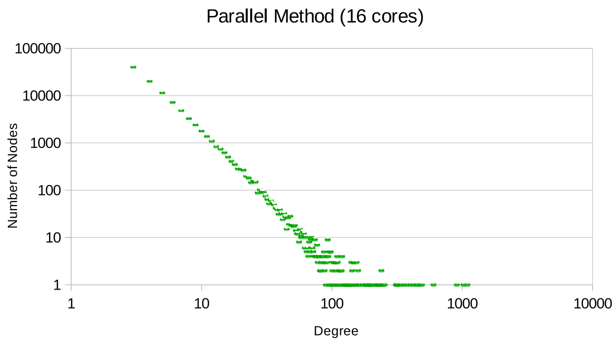
Looking at the performance results, one point worth mentioning is the super-linear speedup observed when going from 1-core to a 2-core execution for any of the 4 distinct runs. We speculate that this can most likely be attributed to the great effect a multi-level CPU cache can have on a multicore setup. Specifically for our case and granted our SURFSara experimentation system, a doubling in number of cores leads to the doubling of the size of L1 and L2 cache (the shared L3 cache stays the same). This results to less overall cache misses on a 2-core setup, which greatly adds to the performance, hence the super-linear speedup. However, this effect is only clearly observable when transitioning from 1-core to 2-core; after 2 cores, the parallel threading overhead overshadows any larger-cache benefit. Still, this remains just a speculation; we are planning to investigate more on the reason and the impact a cache behaviour can have over the PA graph generation.

5. RELATED WORK

There exist some attempts to develop efficient implementations of the PA model [2,5,8,16,21,22,25,27]. Some existing works focus on more efficient implementations of the sequential version [5,8,25]. Such methods propose the utilization of data-structures that are efficient with respect to memory consumption and time complexity. Few existing methods are based on a parallel implementation of the PA model [2,22,27], among which some methods [22,27] are based on a version of the PA model which does not satisfy its basic criteria (i.e., consistency with the original model). The approach



(a)



(b)

Figure 5: The degree distribution of a network with 10^5 nodes and $m = 3$ generated by the sequential and the proposed parallel methods

in [2] requires complex synchronization and communication management and generates considerable overhead of message passing. This stems from that this latter approach is not developed for a multicore setting but for a distributed one. However our focus is to have a high-level scalable and parallel implementation of the original PA model utilizing the computational power of multicore architectures.

6. CONCLUSION AND FUTURE WORK

We showed that the PA-based generation of networks allows a high-level and scalable multicore implementation using the ABS language and its Haskell backend that supports cooperative multitasking via continuations and multicore parallelism via its lightweight threads. Due to space limitations we omitted a comparison with the use of the Encore programming language which is a new object-oriented parallel programming language based on actors [10] and which is based on a runtime system in C.

Future work will be dedicated toward further optimizations of the Haskell runtime system for the ABS. Other work of interest is to formally restrict the use of shared data structures in the ABS to ensure encapsulation. One particular approach is to extend the compositional proof-theory of concurrent objects [14] with foot-prints [13] which capture write accesses to the shared data structures and which can be used

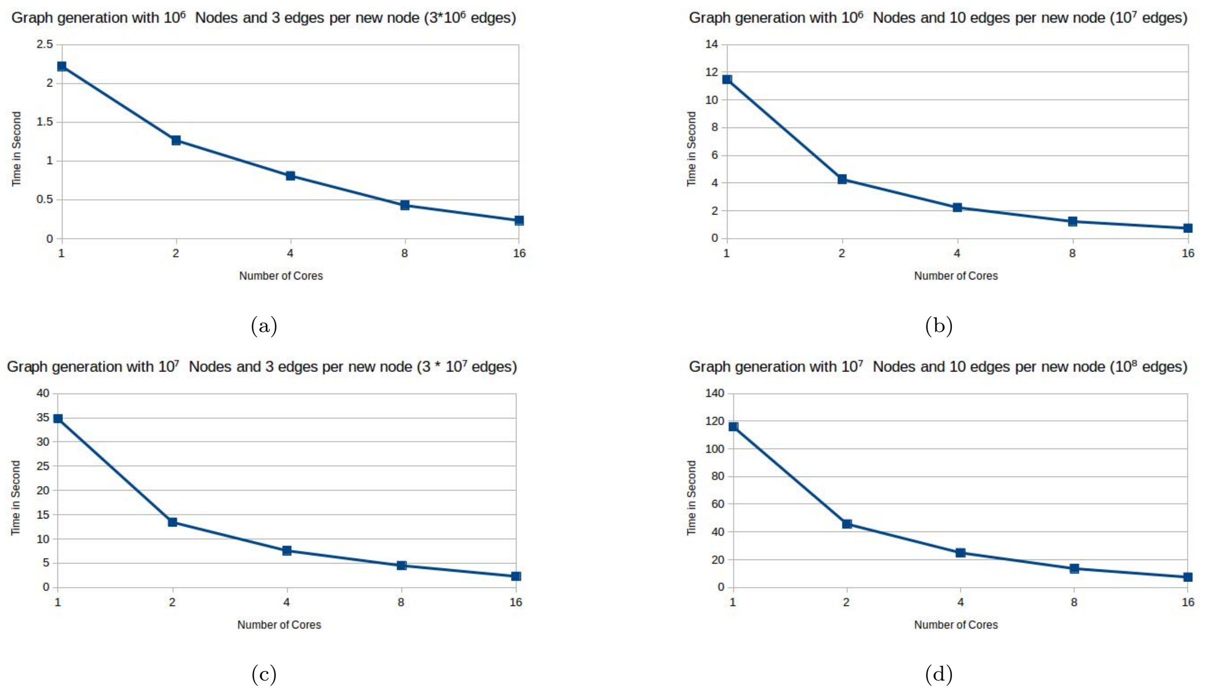


Figure 6: The performance and scalability results of the proposed approach in ABS2Haskell

to express disjointness of these write accesses.

7. ACKNOWLEDGEMENTS

Partly funded by the EU project FP7-612985 UpScale: From Inherent Concurrency to Massive Parallelism through Type-based Optimizations (`http://www.upscale-project.eu`). Partly funded by the EU project FP7-610582 ENVISAGE: Engineering Virtualized Services (`http://www.envisage-project.eu`). This work was carried out on the Dutch national e-infrastructure with the support of SURF Foundation. The authors would like to thank Tobias Wrigstad and Dave Clarke for their helpful comments and suggestions.

8. REFERENCES

[1] Surfsara. https://surfsara.nl/.

[2] M. Alam, M. Khan, and M. V. Marathe. 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*, page 91. ACM, 2013.

[3] E. Albert, P. Arenas, J. C. Fernández, S. Genaim, M. Gómez-Zamalloa, G. Puebla, and G. Román-Díez. Object-sensitive cost analysis for concurrent objects. *Softw. Test., Verif. Reliab.*, 25(3):218–271, 2015.

[4] E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini. Cost analysis of object-oriented bytecode programs. *Theor. Comput. Sci.*, 413(1):142–159, 2012.

[5] J. Atwood, B. Ribeiro, and D. Towsley. Efficient network generation under general preferential attachment. *arXiv preprint arXiv:1403.4521*, 2014.

[6] D. Bader, K. Madduri, et al. Parallel algorithms for evaluating centrality indices in real-world networks. In *Parallel Processing, 2006. ICPP 2006. International Conference on*, pages 539–550. IEEE, 2006.

[7] A.-L. Barabási and R. Albert. Emergence of scaling in random networks. *science*, 286(5439):509–512, 1999.

[8] V. Batagelj and U. Brandes. Efficient generation of large random networks. *Physical Review E*, 71(3):036113, 2005.

[9] N. Bezirgiannis and F. S. de Boer. ABS: a high-level modeling language for Cloud-Aware Programming. In *Proc. SOFSEM '16*. Springer, 2016. To appear.

[10] S. Brandauer, E. Castegren, D. Clarke, K. Fernandez-Reyes, E. B. Johnsen, K. I. Pun, S. L. T. Tarifa, T. Wrigstad, and A. M. Yang. Parallel objects for multicores: A glimpse at the parallel language encore. In *Formal Methods for Multicore Programming - 15th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM 2015, Bertinoro, Italy, June 15-19, 2015, Advanced Lectures*, pages 1–56, 2015.

[11] D. Chakrabarti, Y. Zhan, and C. Faloutsos. R-mat: A recursive model for graph mining. In *SDM*, volume 4, pages 442–446. SIAM, 2004.

[12] F. S. De Boer, D. Clarke, and E. B. Johnsen. A complete guide to the future. In *Programming Languages and Systems*, pages 316–330. Springer, 2007.

[13] F. S. de Boer and S. de Gouw. Being and change:

Reasoning about invariance. In *Correct System Design - Symposium in Honor of Ernst-Rüdiger Olderog on the Occasion of His 60th Birthday, Oldenburg, Germany, September 8-9, 2015. Proceedings*, pages 191–204, 2015.

[14] J. Dovland, E. B. Johnsen, and O. Owe. Verification of concurrent objects with asynchronous method calls. In *2005 IEEE International Conference on Software - Science, Technology and Engineering (SwSTE 2005), 22-23 February 2005, Herzelia, Israel*, pages 141–150, 2005.

[15] P. Erdös and A. Rényi. On the central limit theorem for samples from a finite population. *Publ. Math. Inst. Hungar. Acad. Sci*, 4:49–61, 1959.

[16] D. Gregor and A. Lumsdaine. The parallel bgl: A generic library for distributed graph computations. *Parallel Object-Oriented Scientific Computing (POOSC)*, page 2, 2005.

[17] M. P. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(3):463–492, 1990.

[18] E. B. Johnsen, R. Hähnle, J. Schäfer, R. Schlatte, and M. Steffen. Abs: A core language for abstract behavioral specification. In *Formal Methods for Components and Objects*, pages 142–164. Springer, 2012.

[19] R. Kumar, P. Raghavan, S. Rajagopalan, D. Sivakumar, A. Tomkins, and E. Upfal. Stochastic models for the web graph. In *Foundations of Computer Science, 2000. Proceedings. 41st Annual Symposium on*, pages 57–65. IEEE, 2000.

[20] J. Leskovec. *Dynamics of large networks*. ProQuest, 2008.

[21] Y.-C. Lo, H.-C. Lai, C.-T. Li, and S.-D. Lin. Mining and generating large-scaled social networks via mapreduce. *Social Network Analysis and Mining*, 3(4):1449–1469, 2013.

[22] Y.-C. Lo, C.-T. Li, and S.-D. Lin. 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)*, pages 229–238. IEEE, 2012.

[23] S. Nobari, X. Lu, P. Karras, and S. Bressan. Fast random graph generation. In *Proceedings of the 14th International Conference on Extending Database Technology*, pages 331–342. ACM, 2011.

[24] J. C. Reynolds. The discoveries of continuations. *Lisp and symbolic computation*, 6(3-4):233–247, 1993.

[25] R. Tonelli, G. Concas, and M. Locci. Three efficient algorithms for implementing the preferential attachment mechanism in yule-simon stochastic process. *WSEAS Transactions on Information Science and Applications*, 7(2):176–185, 2010.

[26] D. J. Watts and S. H. Strogatz. Collective dynamics of 'small-world'networks. *nature*, 393(6684):440–442, 1998.

[27] A. Yoo and K. Henderson. Parallel generation of massive scale-free graphs. *arXiv preprint arXiv:1003.3684*, 2010.