SEECN: simulating complex systems using dynamic complex networks

Quax, R.; Bader, D.A.; Sloot, P.M.A.

DOI
10.1615/IntJMultCompEng.v9.i2.50
Publication date
2011
Document Version
Accepted author manuscript
Published in
International Journal for Multiscale Computational Engineering

SEECN: Simulating Complex Systems using Dynamic Complex Networks

Rick Quax David A. Bader* Peter M.A. Sloot

March 19, 2010

Abstract

Multiscale, multiphysics systems are too complex for traditional mathematical modeling and require numerical simulation, yet such systems arise everywhere from modeling the immune system and proteine interaction to epidemic spread in a human population. Unfortunately at present, researchers create their own ad-hoc programs for their particular study. To address this problem we present the Simulator for Efficient Evolution on Complex Networks (SEECN), an expressive simulator of complex systems that optimizes for both single-core and parallel performance. In SEECN, a complex network represents the system where the nodes and edges have specified properties which dictate the dynamics of the network over time. Our application is a detailed model of HIV spread among men who have sex with men and serves to show the simulator's expressiveness and to evaluate its performance.

1 Introduction

No generic, high performance simulator exists for multiscale, multiphysics and dynamic complex systems. Current simulators are either inefficient but expressive, or efficient but specific to a particular problem. Computational models for complex systems are needed since mathematical modeling is intractable due to the heterogeneity and dynamism of the agents and their interactions. Examples of such systems are the immune system, proteine interaction, flow of complex fluid, neuron interaction, and epidemic spread in a human population.

As a consequence, researchers create their own ad-hoc programs for specific applications. Much time is spent both creating the program and extending it to include more and more problems. In addition to expressiveness, the issues of efficiency and validation arise every time for each program. If one simulator would satisfy these three requirements – expressiveness, computational performance and centralized validation – researchers could focus their time on their application instead.

In this paper we present the Simulator for Efficient Evolution in Complex Networks, or SEECN[1], which models complex systems using dynamic complex networks where the nodes are agents with well-defined individual behav-

*College of Computing, Georgia Institute of Technology, USA.

[1]Pronounce SEECN as "season".

ior. Nodes and edges have fully specified parameters that dictate how the system evolves over discrete time steps, including addition and removal of nodes and edges, and changing parameters both independently (progression) and over edges (percolation).

A consequential concern with such an expressive approach is computing time, also called running time. Performance optimization is a challenge since complex networks often exhibit low degrees of locality and memory access patterns are not known a priori [8]. We show how SEECN reduces computational complexity by exploiting the widespread hierarchically modular community structure [18], improves cache efficiency by choosing an appropriate data structure and buffering edge traversals, and implements parallelization. Our experiments show that these improvements reduce the computing time to the extent that the simulator is practical even for large, detailed and highly entropic models.

Our show-case application is the HIV epidemic among men who have sex with men (MSM) because it is truly a multiscale complex system, where different processes drive the epidemic in parallel at different spatiotemporal scales, and because of its societal impact. The HIV model in this paper supports three progression stages, namely accute, asymptomatic and AIDS, each with a probability of being diagnosed and treated. All parameters such as transition probabilities are taken from literature and detailed in Reference 16.

This paper is organized as follows. In Section 2 we discuss related work with a special focus on performance while retaining a high degree of expressiveness. Then we provide a functional description of SEECN in Section 3 and define a basic (naïve) implementation in Section 4, which is less abstract and serves as a context for Section 5 where we present details of algorithms and improvements. Our experiments and results for evaluating the performance gains of our improvements are presented in Section 6 and we conclude in Section 7.

2 Related work

Many agent-based simulators exist [12, 17, 7], however only few focus on combining expressiveness with high performance. The work of Madduri [8] makes it clear that achieving high performance for various standard algorithms on complex networks, even when ignoring agents and interactions, is already a complicated task. For this reason, a common technique applied to physical systems is model reduction by using domain-specific symmetry and assumptions. Here we discuss selected work on simulators for detailed, dynamic complex systems with respect to the degree of expressiveness and performance, at different places in the spectrum.

A popular and mature agent-based simulator that explicitly supports dynamic, multiscale complex systems is Swarm [10]. However, no structural algorithmic effort has been reported to achieve single-core or parallel performance.

GLEaM [3] and EpiSimdemics [2] are examples of multiscale simulators that have been optimized, albeit for their application area of epidemics in populations. GLEaM simulates the spread of an epidemic in Europe, USA or even worldwide using two fixed scales of mobility: long-range air travel and short-range commuting travel. However, the granularity is a geographically defined subpopulation of at least cells of 15×15 minutes [4] with demographic statistics and is thus not agent-based, allowing GLEaM to run on high-end desktop

computers. Although GLEaM stores individual information it simulates no individual behavior.

In contrast, EpiSimdemics simulates individual movement in great detail. It simulates epidemics in urban settings using bipartite networks that connect people with locations and which are distilled using TRANSIMS [11]. To achieve performance, EpiSimdemics exploits disease-specific semantics in order to improve parallel efficiency; in this case, spatial and temporal interdependencies of agents are decoupled by enforcing the simulation time step to be smaller than some disease-specific threshold. Its parallelization technique is a variation on the classic Parallel Discrete Event Simulation (PDES) scheme and no single-core optimization is performed.

Moving away from application-specific simulators, two current projects aim to develop agent-based simulators for large systems but are still work in progress. EURACE [5] builds on the Flexible Large-scale Agent Modelling Environment (FLAME) and aims to optimize the communication and add dynamic load balancing, which sets it apart from other distributed simulators. The parallelization is automatic since models in FLAME are formally defined and the intermediate model representation can be partitioned. EURACE's application is a detailed simulation of the European economy although the technology appears applicable to other applications.

Even more ambitious is the goal of Argonne National Laboratory, developer of Repast Symphony [13]. Their aim is to reach exascale computing [14] by parallelizing simulations using 10^6 to 10^7 threads. Guiding the experimental development are three application domains: microbial biodiversity, cybersecurity, and the social aspects of climate change. Unfortunately no implementation details are available at the time of writing.

Promoting centralized development and validation effort is the focus of the Network Workbench tool (NWB) [15], a Java-based framework which contains public libraries of algorithms and networks. Its expressiveness and efficiency thus crucially depend on the available algorithms. In principle, SEECN could be part of the NWB. Currently it has no similar algorithms available.

Lastly we refer the interested reader to Reference 12 for an extensive review of available agent-based simulators. Five popular simulators were compared in Reference 17, among others with respect to performance. To the best of our knowledge, no simulator listed in these reviews investigates and optimizes performance to the degree of the related work discussed in this section.

3 Functional description

SEECN simulates complex systems by implementing dynamic complex networks in which nodes and edges are annotated by arbitrary parameters that dictate the evolution of the system. Its algorithms and memory lay-out are being optimized for single-core and parallel performance, while retaining expressiveness. The network is organized in a hierarchically modular fashion to facilitate multiscale models.

Nodes and edges are annotated by an arbitrary *state vector* and are input to *dynamics operators*. A state vector has predefined elements, each of which can be one of a predefined set of values. A dynamics operator, or simply operator, is then a stochastic mapping between higher-dimensional state vector space. This

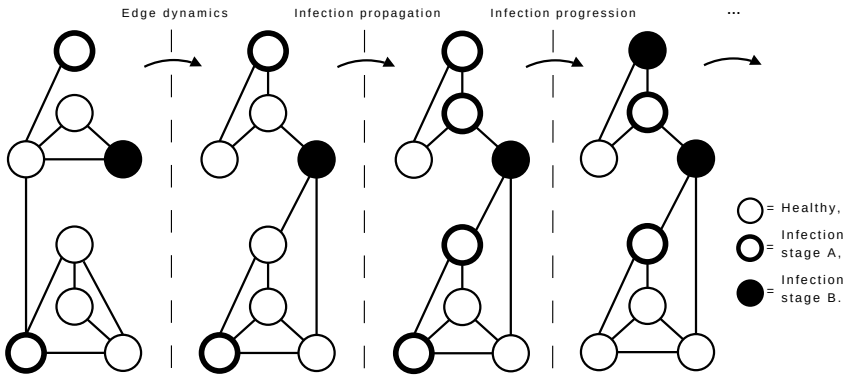

Figure 1: Example of dynamics operators being applied sequentially to a network in one time step. The network has two subhierarchies (or clusters) containing three nodes, each with more internal than external edges.

dimensionality depends on the type of operator. For instance, the local progression of nodes maps a single dimension to a single dimension, but a node's remote influence on another node over an edge maps a three-dimensional state vector space (two for the nodes, one for the edge) to a single dimension (that of the remote node). All unique operators executed in sequence evolve the network one time step, including adding and deleting nodes and edges, and changing properties of nodes and edges. (See Figure 1.) The implementation of an operator is a Markov Chain stochastic matrix containing transition probabilities. For efficiency, each matrix may be replaced by a function.

The way in which edges are generated is controlled by specifying a probability of creating an edge for every possible pair of nodes, which we call a *recipe* for simplicity and irrespective of its implementation. One recipe is used by the network generator to create the initial network; a usually different one is used for each time step to model edge dynamics.

To make it more precise, let a network of a set of edges E and a set of nodes V be $G = (V, E)$ with $N = |V|$ and S be a state vector space[2]. Then let $\mathcal{R} : V \times V \mapsto [0, 1]$ be a recipe and $\mathcal{W}_i : S^k \times S \mapsto [0, 1]$ be a $(k+1)$-dimensional transition probability matrix[3] where k is the number of relevant input state vectors. For example, for the local progression of a node's state $k = 1$ while for a node's influence on another node over an edge $k = 3$.

Further define $\mathcal{D}_i(G)$ a shorthand for $\mathcal{D}_i(G \mid \mathcal{R}, \mathcal{W}_i)$ as applying the ith dynamics operator $\mathcal{D}_i$ to network G. The effect of the operator is either transitioning state vectors based on every possible combination of nodes and edges (totalling k_i) using the transition matrix $\mathcal{W}_i$, or generating nodes and edges using the recipe $\mathcal{R}$, or a combination of the two[4]. Lastly, let $\mathcal{G}$ be a special dynamics operator that bootstraps the simulation by generating an initial, representative network.

[2]For ease of presentation we assume equal state vector space for nodes and edges. If they are unequal, one could create a superset S of both state vector spaces and let nodes and edges use the appropriate subset.

[3]For ease of presentation we assume that the state vector space includes a special state of being removed, e.g., 'isolated'.

[4]In fact, generating nodes and edges also requires a transition matrix since a state vector must be initialized.

A simulation in SEECN then evolves as

$$\begin{aligned} G_{t+1} &= \mathcal{D}_1\left(\mathcal{D}_2\left(\ldots \mathcal{D}_d\left(G_t\right)\ldots\right)\right) , \\ G_0 &= \mathcal{G}\left(\emptyset\right). \end{aligned}$$

where d is the number of dynamics operators and tth time step of T in total. Any statistic can then be calculated from the sequence $G_0, \ldots, G_T$, for instance the distribution of state vectors as a function of time.

At present, the most prominent shortcoming of SEECN is that the probability that a particular edge is created (the recipe) is independent of state vectors. For example, in HIV epidemic models for both male and female individuals there is no way to specify that sexual interaction between a male and female is more probable than between two males. We intend to implement this in the future in a computationally efficient manner. The current workaround is to let subsets of nodes coincide with communities of a single type of nodes, such as one range of nodes being implicitly male and another range for females, allowing the edge probabilities to be specified in the recipe. Such *community structure* is easily specified in our implementation of a recipe which is discussed in Section 5.

4 Basic Implementation

Before we explain our improvements for performance we first identify and discuss the memory structures and algorithms that are central to SEECN. The implementation as discussed here is naïve and forms the basis for our improvements of the next section. Our hope is that this section aids the reader in placing our improvements in perspective.

4.1 Memory structures

The important data structures in SEECN are discussed in turn and illustrated in Figure 2.

4.1.1 Recipe

In this paper we define an edge recipe, or recipe, as a datastructure that provides the probability of all edges being created between any possible pair of nodes. This term separates its function from its implementation, as opposed to terms such as edge probability matrix or adjacency matrix. Particularly, SEECN uses a different implementation of a recipe. Regardless of implementation, any recipe should imply some $N \times N$ matrix of edge probabilities.

4.1.2 Nodes

Nodes are stored in a 1-dimensional array of size N. Each node structure stores its number of edges and its *state vector* or state, which is an integer value and explained next.

A node's state may be defined by any number of properties and values, such as gender (male, female) and age (0 to 100). Each possible vector that assigns values to these properties, such as $(\text{male}, 34)$, is mapped bijectively onto a range of integer numbers. This minimizes the memory usage of state vectors and simplifies indexing into stochastic matrices.

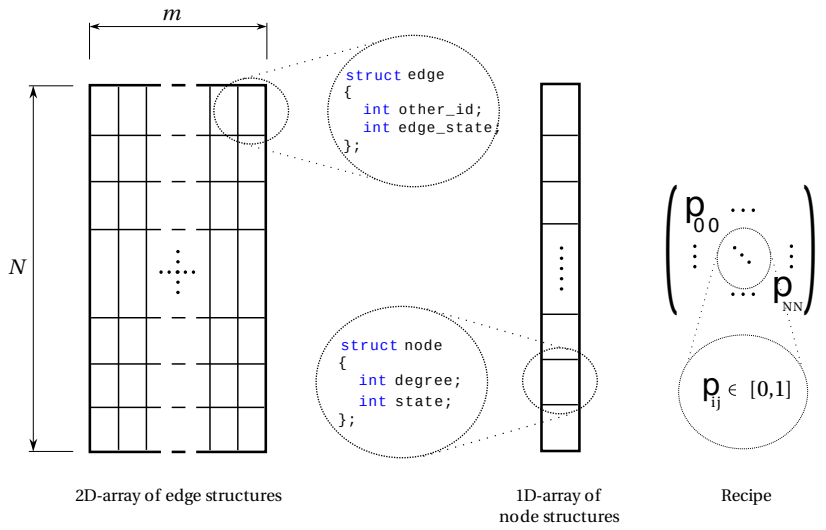

Figure 2: The three important data structures in SEECN are the node array, the edge matrix and the recipe. These illustrations are simplified for the purpose of presentation.

4.1.3 Edges

For each node, m edge structures are stored such that all edge structures are stored in a 2-dimensional $N \times m$ array. Each edge structure stores at least the node identifier of 'the other end'. SEECN currently assumes undirected edges, so each edge structure has a reciprocal.

4.2 Algorithms

As mentioned in Section 3, an execution of a simulation consists entirely of a collection of dynamics operators. We distinguish the following three types of operators which will simplify discussions about performance.

Node operators process all nodes, e.g., for disease progression and death. Node operators execute $\mathcal{O}(N)$ instructions[5].

Edge operators process all existing edges, e.g, for infection propagation. Edge operators execute $\mathcal{O}(|E|)$ instructions.

Recipe operators process all possible edges, e.g., for creating new edges for the network. Recipe operators execute $\mathcal{O}(N^2)$ instructions.

Formally, the stated asymptotic number of instructions above required assume that the number of possible state vectors scales sublinear to N. More practically speaking we assume that the number of *relevant* state vectors is constant and much smaller than N. If not, choosing a new state vector in disease progression and infection propagation may dominate computing time. In some

[5]We distinguish computing complexity, in terms of instruction count, from computing time complexity, which includes communication and memory access overhead. These quantities may differ depending on effects such as cache efficiency.

cases, the computing time can be greatly reduced by replacing a (partial) transition matrix by a function, especially if the transition probabilities are somehow regular with respect to state vector values.

To illustrate the basic implementation of SEECN we provide pseudo-code in Procedure 1. In Procedures 2, 3, and 4 we specify one example of each operator type.

Procedure 1 SIMULATEHIV(N, m, T)

```
states ← {('male', 'healthy'), ('male', 'infected'), ('male', 'AIDS')}
nodes ← array of N node structures
edges ← matrix of N × m edge structures
progressionProb ← matrix of |states| × |states| probabilities
propagationProb ← matrix of |states| × |states| × |states| probabilities
recipe ← matrix of N × N probabilities

NETWORKGENERATOR(nodes, edges)

for t = 0 to T − 1 do
  NODEOPERATOR_PROGRESSION(nodes, progressionProb)
  ...
  EDGEOPERATOR_PROPAGATION(nodes, edges, propagationProb)
  ...
  RECIPEOPERATOR_ADDEDGES(edges, recipe)
  ...
  print "In time step t the network is: (edges, nodes)"
end for
```

Procedure 2 NODEOPERATOR_PROGRESSION(nodes, progressionProb)

```
for i = 0 to N − 1 do
  nodes[i].state ← random state s with probability
  ...    progressionProb[nodes[i].state, s]
end for
```

Procedure 3 EDGEOPERATOR_PROPAGATION(nodes, edges, propagationProb)

```
for i = 0 to N − 1 do
  for e = 0 to nodes[i].degree do
    other_state ← nodes[edges[i].other_id].state
    nodes[i].state ← random state s with probability
    ...    propagationProb[other_state, nodes[i].state, s]
  end for
end for
```

Procedure 4 RECIPEOPERATOR_ADDEDGES(nodes, edges, recipe)

```
for i = 0 to N − 1 do
  for j = i to N − 1 do
    r ← random number ∈ [0.0, 1.0)
    if r < recipe[i, j] and (i, j) ∉ edges then
      edges[i, nodes[i].degree].other_id = j
      edges[j, nodes[j].degree].other_id = i
      nodes[i].degree ← nodes[i].degree + 1
      nodes[j].degree ← nodes[j].degree + 1
    end if
  end for
end for
```

5 Improvements

A primary concern with detailed individual-based simulations is computing time. Many real networks are approximately scale-free [1] which are notoriously difficult to partition and whose access patterns are highly irregular. In addition, typical applications include parameter-searching and impact evaluation of parameters which require that multiple simulation executions are combined to obtain statistical significance. Due to the immense search space, some type of computational steering of experiments is often used such as simulated annealing. In conclusion, a simulator should run fast in order to be useful.

Our improvements focus on computational complexity, cache efficiency and parallelization. Firstly, recipe operators dominate computation time and become a bottleneck for larger networks even though such operators are cache efficient. The second bottleneck is the number of cache misses, primarily due to edge operators because the simulator performs many random memory accesses per little computation. Finally, an obvious but non-trivial improvement is parallelization.

5.1 Reducing Computational Complexity

Many complex networks have some community structure such that many nodes are (partly) similar in terms of network dynamics. This could be in the nature of the system that is being modeled, but it could also result from lack of data and assumptions. If we reduce the size of the recipe to store only the necessary information, we reduce the required amount of computation and memory.

A common and generic way to express such community structure is *hierarchical modularity* [18]. See Figure 3. For our purpose it is important that a hierarchically modular network recursively divides sets of nodes into 'subhierarchies' in such a way that two nodes in the same subhierarchy have equal edge probabilities towards any other node outside the subhierarchy. An example model where this is useful is simulating an epidemic in a country. Data may specify how many people travel from city A to city B, but not specify more details. In this case equal edge (proximity) probability may be assumed between any resident of city A to any resident of city B. In contrast, each city may have data on how many people travel from one city area to the other, providing more

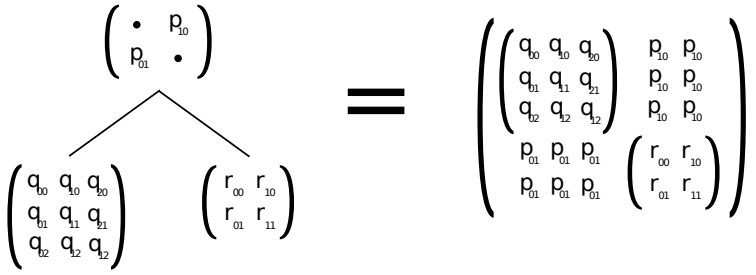

Figure 3: Any recipe must imply a $N \times N$ matrix or probabilities while the implementation may vary. In SEECN, the recipe is a tree-structure describing the hierarchical modularity of the network.

details only within each subhierarchy.[6]

For recipe operators we can exploit the regularity imposed by hierarchical recipes. In contrast, the complexities of node and edge operators cannot be reduced assuming that every existing node and edge is relevant for the evolution of the epidemic. The following observation holds even if edge probabilities would depend on state vectors since the total number of state vectors is constant for varying N.

Many (more than constant in N) edge probabilities in a recipe are equal if C grows sublinear in N, where C is the number of communities[7]. Since edge probabilities are equal for all nodes within a community, recipe operators compute in terms of communities instead of individual nodes. For example, to generate edges for a node its degree[8] is drawn from a suitable distribution since each community's contribution to a node's degree is binomially distributed. For larger community sizes (larger networks) these binomial distributions approach Poisson distributions that sum up. Then, each edge's neighbor node id is selected by first selecting a community id $[0, C)$ and then a random node id within that community.

Thus, recipe operators require $\mathcal{O}(NC)$ instructions per time step. In worst-case, though, C grows linear in N in which case the computational complexity remains $\mathcal{O}(N^2)$. In the experiments of this paper we use Kronecker networks [6], for which $C = \log_2 N$ [16].

5.2 Improving Cache Efficiency

Random memory access dominates computing time due to visiting and traversing edges, both of which require special attention [8] and are discussed in turn. Node and recipe operators can visit node structures and random number generators in order, so this section focuses on edge operators.

[6] A 1-level hierarchy requires N^2 probabilities and thus our recipe implementation preserves expressiveness.

[7] We define a community as a subhierarchy that is not divided further, i.e., a leaf node in the hierarchical (tree-structured) recipe description.

[8] A node's degree is defined as its number of incident edges.

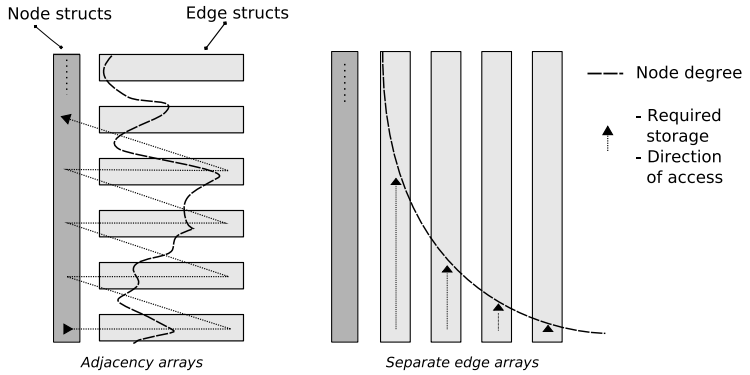

Figure 4: A comparison of memory access patterns between the adjacency array and JDS for a scale-free network.

We define an edge visit as a predictable memory access for an edge structure by a dynamics operator. Edge operators visit all existing edges once and in some order in every time step. An edge traversal differs from an edge visit in that it is a memory access for an edge structure as a result of some dynamics, for instance to spread a disease from one node to the other. Edge traversals are thus model-dependent and unpredictable in advance.

5.2.1 Visiting edges

Network dynamism constrains the choice of data structure and suggests the classic adjacency matrix (see left of Figure 4). To remain efficient, all memory should be pre-allocated and not be (de)allocated continuously, which would fragment memory. The adjacency matrix stores edges contiguously per node and reserves equal capacity $D > 0$ for each node. Visiting all edge structures of one node is 'in cache'. However, particularly in scale-free networks only a small fraction of nodes uses its full capacity, in which case visiting the first edge structure of the next node is (almost) always a cache miss. Thus, the adjacency matrix incurs an expected N cache misses when visiting all existing edges of a network once.

An alternative is the Jagged Diagonal Storage (JDS) [19] which stores node x's ith edge in the ith array of size N edge structures (of which there are D) at element x out of N (see right of Figure 4). Essentially the direction of storage of edge structures is inverted. In JDS, all edge structures are visited per array of size N, and a cache miss occurs due to a node of which the ith edge is accessed while $B \geq 1$ previous nodes had less than i edges, for some architecture-dependent cache line size B.

To minimize this probability we initially renumber nodes on prior expected degree. Intuitively, in the first portion of an edge structure array, an edge structure is probably occupied but the same is true for any of the B previous edge structures. In the latter portion a cache miss is improbable because each edge structure is empty with high probability, in which case it trivially does not incur a cache miss. The performance gain of renumbering nodes depends on the degree distribution implied by the recipe.

The performance gain of the JDS and renumbering is experimentally evalu-

ated in Section 6.1.

5.2.2 Traversing edges

The second source of cache misses is due to *traversing* edges, which we address both by queuing and sorting *updates* and preventing traversals altogether. An edge is traversed when an operator needs to access the memory of a neighboring node, for instance to change its state as a result of infection spreading. In a naïve implementation the number of cache misses for T edge traversals is $\mathcal{O}(T)$ in general. The performance benefit of our improvements are evaluated in Section 6.3.

Instead of immediately performing unpredictable memory writes, operators queue updates which dictate state changes to a central queue. This process is cache efficient because the memory access pattern for queuing is sequential. When an operator has finished, the queue is sorted on receiving node identifier and the updates are 'applied' in order, splitting the traversal problem into a sorting part and an edge visiting part. The latter problem is addressed by the JDS and becomes more efficient as the density T/N grows. For improved efficiency the updates of multiple independent operators may be combined before sorting, increasing the density T/N.

The sorting part becomes the bottleneck and actually performs worse than the naïve implementation in asymptotic terms, requiring $\mathcal{O}(T \log T)$ cache misses. Nevertheless there is a large reduction of the constant factor in this complexity: 'cache-aware' sorting algorithms incur $\mathcal{O}(T/B(\log T/B)/(\log T/B))$ cache misses for cache size B and memory size M.[9] Because of this, in practice queuing and sorting updates can well be more efficient than the naïve implementation even though asymptotically it performs worse.

In addition to queuing updates we prevent a fraction of edge traversals by storing a neighbors state in the edge structure of each node. In this way, computing the result of remove influences on a particular node is cache efficient and requires no traversals, but once a node's state changes all edge structures of its neighbors must be updated. In the worst case all nodes change state so that all edges must be traversed to update the neighbor's copy of the state: this is the number of edge traversals needed if no state copies were stored in edge structures. The efficiency of this technique depends on how often a node's state changes and therefore on the specific model, so we have not evaluated its performance gain in experiments.

5.3 Parallelization

Because edge traversals dominate computing time for large networks and usually $|E|$ grows at least linear in N, we partition the network with respect to edge structures. In contrast, node operators are relatively inexpensive because they access all nodes in order. Moreover, node structures must already be accessed for edge operators, e.g., to update a node's degree after edge removal or addition.

Therefore we partition the network at the granularity of nodes, in contiguous ranges, while balancing the expected total number of edges to be handled by each

[9] This expression is for a simple RAM-model with one cache layer. Although its details may change for different architecture models, cache and memory sizes are constant in terms of T so the computing time complexity of sorting remains $\mathcal{O}(T \log T)$.

process. Although SEECN could exploit the hierarchical community structure to partition the network more optimally, the current partitioning algorithm is simplified by assuming no assortativity[10] beyond that of expected degree; in fact, Kronecker generates such networks [16]. This assumption implies that a node x with expected degree $\langle d_x \rangle$ expectedly incurs $\langle d_x \rangle \,/\, \langle d_y \rangle$ more interprocess edges than node y, and therefore the optimal partition is to balance the expected number of edges to handle by any process i out of p. The probability that an edge connects two nodes of different processes is $(p-1)/p$.

In terms of implementation, few changes need be made. Most prominently, each process' update queue (Section 5.2.2) must be split into p queues, one for each process. At intermediate steps the queues are transferred at once while benefiting from high communication bandwidth, which would not be possible without buffering. Exploiting bandwidth is important because the size of a queue scales as $\mathcal{O}(|E|)$ for increasing N, so communication would otherwise become the bottleneck.

6 Experiments and results

In this section we perform two types of experiments. Firstly, we compare the performance of the JDS to that of the classic adjacency array in Section 6.1. Then, in Section 6.2, we describe our experiments of an HIV epidemic model among MSM and show the results for single-core and parallel performance in Sections 6.3 and 6.4, respectively.

The experiments are run on a Linux cluster of 680 dual-core Intel Xeon nodes at 3.4 GHz. Although a compute node consists of multiple cores we only allocate one process per compute node, homogenizing communication overhead. We implement SEECN in C++, parallelize it with MPI, and compile it with GCC 4.1.2 and MPICH2 1.0.8. We use TCP/IP over Infiniband for communication. SEECN's code is not tuned to the cluster's architecture and uses the standard STL sorting algorithm.

6.1 Cache efficiency: JDS and adjacency array

We compare the cache efficiency of the JDS and the adjacency array on random scale-free networks with power-law exponents 1.6, 2.0 and 3.0. For each experiment a network is generated and all its edges are visited once. Figure 5 shows the speedup factors of JDS over the adjacency array as a function of network size.

Clearly, for scale-free networks the JDS performs better for all but the smallest networks. We expect that the step function-like shape is an artefact of the memory architecture but have not further investigated its exact cause. This pattern could explain the sudden increase of the 1.6-curve at 2^{20} nodes, namely that the other curves would follow the same trend with a delay similar to in the range $N \in [2^{16}, 2^{18}]$. The adjacency array experiments could not go beyond 2^{20} nodes because the adjacency array would run out of memory, most likely because it allocates a single, contiguous block of memory whereas the JDS allocates D arrays for maximum possible degree D. Here, $D = 120$.

[10] Assortativity refers to a statistical bias in edge probabilities depending on node properties, for instance the bias of age in forming relationships.

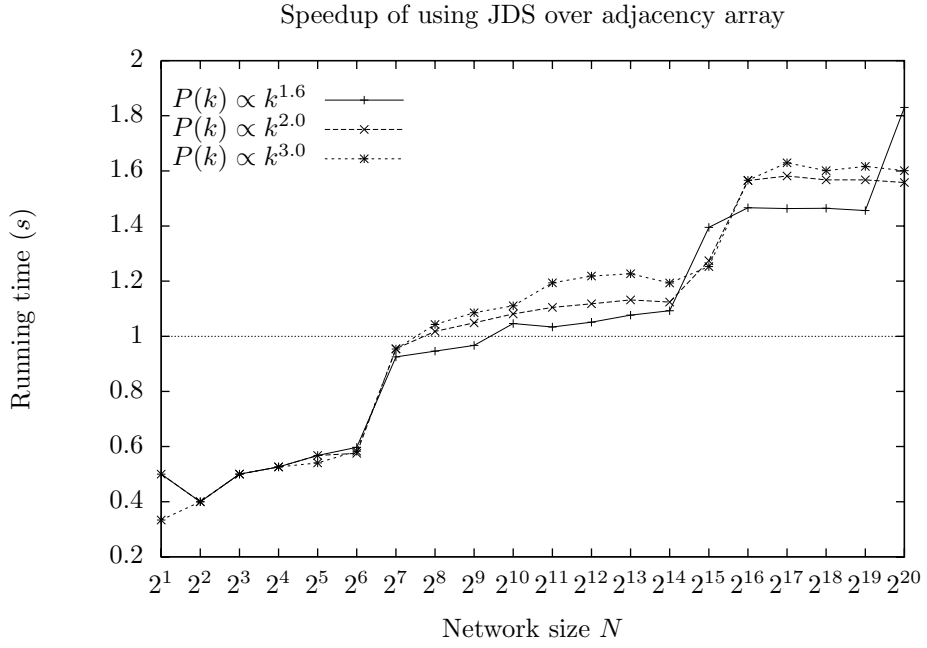

Figure 5: The sequential computing times using the adjacency array divided by the computing times using the JDS, for random scale-free networks with power-law exponents 1.6, 2.0 and 3.0. The computing time for each data structure, N and power-law exponent is the average of 10 generated networks with 5 complete edge visits per network. Note that the x-axis has a logarithmic scale.

6.2 HIV epidemic among MSM

We perform benchmark experiments using one, four and sixteen processors. The epidemic model is fairly complex and represents HIV in a population of MSM. It assumes a hierarchical network with power-law exponent 1.6 [20] and classifies nodes as healthy, acute, (un)treated asymptomatic, or (un)treated AIDS, each of which have distinct infectiousness, life expectancy and duration of relationships. The details are presented elsewhere [16] where good qualitative correspondence with historical data was found.

These experiments use Kronecker networks as a surrogate for the community structure of sexual contacts among MSM. A Kronecker network of 2^n nodes has $n-1$ levels where hierarchies recurse into two subhierarchies in a structured way. In short, the network has an approximately scale-free degree distribution and adds additional community structure. Mahdian et al. [9] have shown that some real social networks were adequately approximated by a Kronecker network. A generator for Kronecker networks takes four parameters which were taken from Reference 16.

To be able to distinguish the distinct influences on performance we have implemented the following three variants.

cached The nodes in the network data structure are not renumbered on sorted expected degree and the update buffers are not sorted before they are

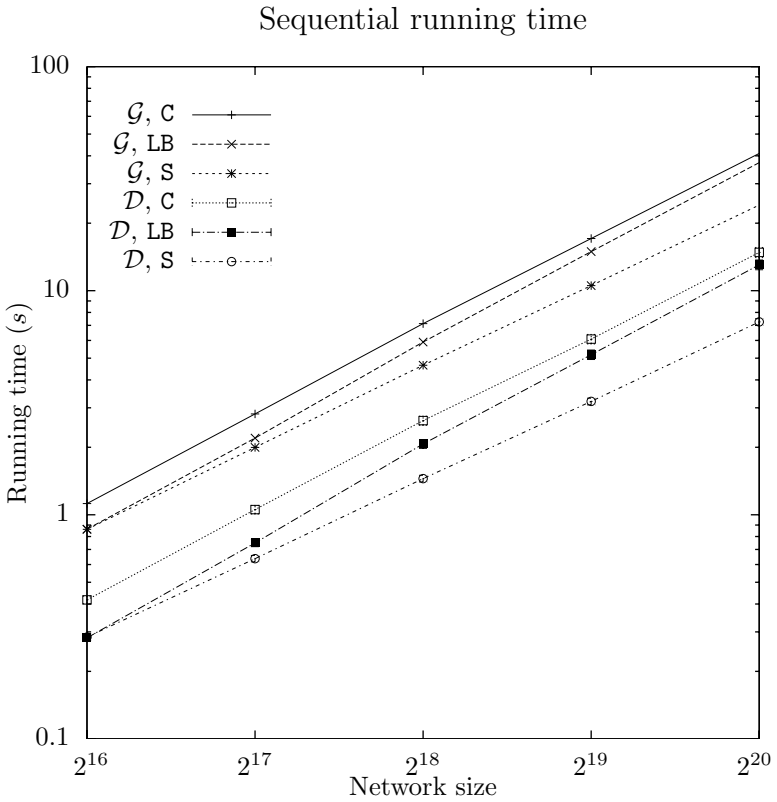

Figure 6: The sequential computing times (on logarithmic scale) of generating the network ($\mathcal{G}$) and performing all other dynamics in a single time step ($\mathcal{D}$). All data points are averages of five runs, and each run had 100 time steps of dynamics (standard error is too small to show).

applied to the network.

load-balanced The nodes in the network data structure are renumbered on sorted expected degree, but update buffers are still not sorted before they are applied to the network. The renumbering enables load-balancing for networks with no further assortativity beyond expected degree. It makes edge visiting efficient but edge traversal remains inefficient.

sorted The nodes in the network data structure are renumbered on sorted expected degree and update buffers are sorted before they are applied to the network. Both visiting and traversing edges become more efficient.

6.3 Single-core performance

The sequential experiments show the impact of cache efficiency of both visiting (`load-balanced`) and traversing (`sorted`) edges, for increasing N. The results are shown in Figure 6. We could not experiment with network sizes of over 2^{20} because a single processor would run out of memory. The JDS was used for all experiments.

The results suggest indeed that the computing time scales as $|E|$, justifying parallelization schemes with respect to the number of edges. The logarithmic

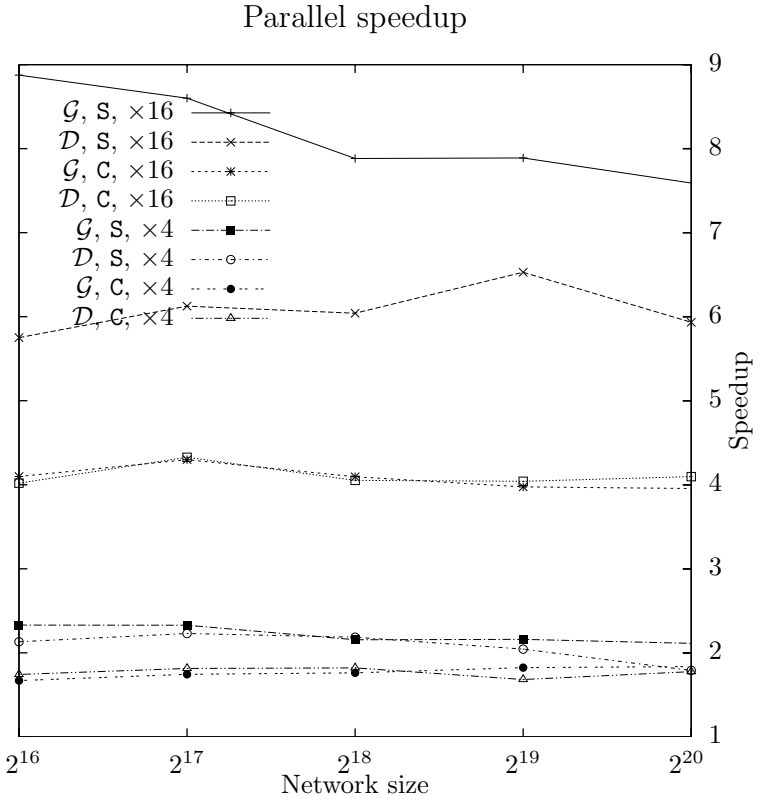

Figure 7: The parallel speedups (on logarithmic scale) of generating the network ($\mathcal{G}$) and performing all other dynamics in a single time step ($\mathcal{D}$) using 4 and 16 processors. All data points are averages of five runs, and each run had 100 time steps of dynamics (standard error is too small to show).

computing time plots of the `sorted` generator and dynamics operators have slopes of 2.26 and 2.30. The theoretically expected slope for the logarithmic edge count is 2.3 (not shown) because in the present case $|E| \propto 2.3^{\log_2 N}$ [16], in the absence of a maximum degree.

It is also interesting that random (unsorted) edge traversals that are cache efficient without sorting queues decreases with increasing network size, because both slopes for the `load-balanced` algorithm are about 2.45. This suggests that the computational cost of random edge traversals eventually outweigh that of edge visits and that sorting the update queue is quite effective in practice.

Further, the performance benefit of renumbering nodes and visiting them in order is significant for smaller networks ($N \leq 2^{16}$), for which sorting queues has no benefit. The number of updates per time step grows as $|E|$, so the average number of updates per node increases, whereas the number of initial visits (i.e., not caused by an update) remains constant. Consequently, for larger networks ($2^{16} < N \leq 2^{20}$) the benefit of cache efficient initial visits diminish while the benefit of sorting updates and performing them in order increases.

6.4 Parallel performance

For the parallel case we show the speedup of the `cached` and `sorted` algorithms over the same range as for the sequential case in Figure 7. The JDS was used for all experiments. Clearly, balancing the number of nodes is less efficient than balancing edge counts. For an increasing number of processors, `sorted`'s efficiency curves (not shown) are about 0.5 and almost constant; in contrast, `cached`'s efficiency curves drop from 0.4 to 0.25.

The speedups remain roughly constant over our range of N, which would be the case if both processing and communication time would be dominated by the number of updates and therefore by $|E|$. Communication overhead that is sublinear in $|E|$ in the absence of additional assortativity is not possible in general (without load imbalance), so we consider our parallelization efforts successful.

7 Conclusion

In this paper we make first steps in investigating whether the temporal dynamics of detailed, agent-based models connected by multiscale complex networks can indeed be simulated in reasonable computing time. In particular, we discuss and implement algorithmic improvements and evaluate the impact of buffering edge traversals, sorting these traversals, and parallelizing with balanced numbers of nodes or edges. For a representative epidemic model with scale-free network structure we find a constant sequential computing time reduction of almost a factor 2 for renumbering nodes and sorting and buffering updates, and a constant parallel efficiency of roughly 0.5. Additional to the sequential speedup is the improved cache efficiency of an alternative data structure, yielding up to a factor 2 for larger network sizes. As a result, a detailed simulation of HIV among one million persons in a hierarchically modular and scale-free network over 25 years (100 time steps) takes only two minutes using 16 processing nodes. In conclusion, we suggest that for simulating detailed, agent-based models connected by multiscale complex networks, experimentation can be a powerful alternative to mathematical modeling, and that our computational performance make even computational steering methods using such systems feasible.

8 Acknowledgement

The authors (Quax and Sloot) would like to acknowledge the financial support of the European Union through the projects DynaNets (`www.dynanets.org`), EU project no. FET-233847, and EpiWork (`www.epiwork.eu`), EU project no. FET-231807. Baders research is supported in part by NSF Grant CNS-0614915, Pacic Northwest National Laboratorys Center for Adaptive Supercomputing Software and by MIT Lincoln Laboratory.

References

[1] Albert-László Barabási and Reka Albert. Emergence of scaling in random networks. *Science*, 286:509, 1999.

[2] Christopher L. Barrett, Keith R. Bisset, Stephen G. Eubank, Xizhou Feng, and Madhav V. Marathe. Episimdemics: an efficient algorithm for simulating the spread of infectious disease over large realistic social networks. In *SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, pages 1–12, Piscataway, NJ, USA, 2008. IEEE Press.

[3] Vittoria Colizza, Alain Barrat, Marc Barthelemy, and Alessandro Vespignani. Predictability and epidemic pathways in global outbreaks of infectious diseases: the SARS case study. *BMC Medicine*, 5(1):34, 2007.

[4] Socioeconomic data and applications center (SEDAC). Gridded population of the world, version 3 (gpwv3) and the global rural-urban mapping project (grump), September 2009.

[5] Christophe Deissenberg, Sander van der Hoog, and Herbert Dawid. EURACE: A massively parallel agent-based model of the European economy. *Applied Mathematics and Computation*, 204(2):541 – 552, 2008. Special Issue on New Approaches in Dynamic Optimization to Assessment of Economic and Environmental Systems.

[6] Jurij Leskovec, Deepayan Chakrabarti, Jon Kleinberg, and Christos Faloutsos. Realistic, mathematically tractable graph generation and evolution, using Kronecker multiplication. In *PKDD 2005*, pages 133–145. Springer, 2005.

[7] Charles M. Macal and Michael J. North. Tutorial on agent-based modeling and simulation. In *WSC '05: Proceedings of the 37th conference on Winter simulation*, pages 2–15. Winter Simulation Conference, 2005.

[8] Kamesh Madduri. *A High-Performance Framework for Analyzing Massive Complex Networks*. PhD thesis, Georgia Institute of Technology, 2008.

[9] Mohammad Mahdian and Ying Xu. Stochastic Kronecker graphs. Technical report, Proceedings of the 5th Workshop on Algorithms and Models for the Web-Graph, 2007.

[10] Nelson Minar, Rogert Burkhart, Chris Langton, and Manor Askenazi. The Swarm simulation system: A toolkit for building multi-agent simulations. Working Papers 96-06-042, Santa Fe Institute, June 1996.

[11] Kai Nagel, Richard L. Beckman, and Christopher L. Barrett. TRANSIMS for transportation planning. In *In 6th Int. Conf. on Computers in Urban Planning and Urban Management*. Addison-Wesley, Reading,Massachusetts, 1999.

[12] Cynthia Nikolai and Gregory Madey. Tools of the trade: A survey of various agent based modeling platforms. *Journal of Artificial Societies and Social Simulation*, 12(2):2, 2009.

[13] M. J. North, T. R. Howe, N. T. Collier, and J. R. Vos. The Repast Simphony runtime system. In *Proceedings of the Agent 2005 Conference on Generative Social Processes, Models, and Mechanisms*, 2005.

[14] M. J. North, C. M. Macal, G. W. Pieper, and C. G. Drugan. Agent-based modeling and simulation for exascale computing. *SciDAC Rev.*, 2008.

[15] NWB Team. Network Workbench Tool. Indiana University, Northeastern University, and University of Michigan, http://nwb.slis.indiana.edu, 2006.

[16] Rick Quax. Modeling and simulating the propagation of infectious diseases using complex networks. Master's thesis, Georgia Institute of Technology, 2008.

[17] Steven F. Railsback, Steven L. Lytinen, and Stephen K. Jackson. Agent-based simulation platforms: Review and development recommendations. *SIMULATION*, 82(9):609–623, 2006.

[18] Erzsébet Ravasz and Albert-László Barabási. Hierarchical organization in complex networks. *Phys. Rev. E*, 67(2):026112, Feb 2003.

[19] Youcef Saad. Krylov subspace methods on supercomputers. *SIAM J. Sci. Stat. Comput*, 10:1200–1232, 1989.

[20] A. Schneeberger, C. H. Mercer, S. A. Gregson, N. M. Ferguson, C. A. Nyamukapa, R. M. Anderson, A. M. Johnson, and G. P. Garnett. Scale-free networks and sexually transmitted diseases: a description of observed patterns of sexual contacts in britain and zimbabwe. *Sex Transm Dis*, 31(6):380–387, June 2004.