Improving the Scalability of SimGrid Using Dynamic Routing

Silas De Munck, Kurt Vanmechelen, and Jan Broeckhove

University of Antwerp,
Department of Computer Science and Mathematics,
Middelheimlaan 1, 2020 Antwerp, Belgium
silas.demunck@ua.ac.be

Abstract. Research into large-scale distributed systems often relies on the use of simulation frameworks in order to bypass the disadvantages of performing experiments on real testbeds. SimGrid is such a framework, that is widely used and mature. However, we have identified a scalability problem in SimGrid's network simulation layer that limits the number of hosts one can incorporate in a simulation. For modeling large-scale systems such as grids this is unfortunate, as the simulation of systems with tens of thousands of hosts is required. This paper describes how we have overcome this limitation through more efficient storage methods for network topology and routing information. It also describes our use of dynamic routing calculations as an alternative to the current SimGrid method which relies on a static routing table. This reduces the memory footprint of the network simulation layer significantly, at the cost of a modest increase in the runtime of the simulation. We evaluate the effect of our approach quantitatively in a number of experiments.

Keywords: Grid Computing, Grid Simulation, Scalability, SimGrid, Routing Algorithm, Boost Graph Library.

1 Introduction

Distributed and parallel processing techniques are common today in a wide range of applications. The increasing scale and complexity of distributed applications and systems necessitates research into more scalable and efficient algorithms and techniques for e.g. resource management and job scheduling. The evaluation of new algorithms on real testbeds is however impeded by their limited flexibility, controllability and availability. In addition, the costs for building and configuring large-scale testbeds are high.

For this reason, researchers turn to simulation to evaluate new algorithms and techniques, especially during the initial phases of development. A widely used simulation toolkit in this regard is SimGrid [1]. SimGrid is a toolkit providing functions for the simulation of distributed applications in heterogeneous distributed environments. It thereby targets platforms that range from a simple network of workstations to large-scale computational grids.

However, we have found SimGrid to have a scalability problem that impedes the simulation of large-scale systems such as grids. Currently, the size of the simulated networks is mainly limited by system memory. This memory limit is reached at roughly 4000 hosts on a machine with $8\,GB$ of memory. Although scaling beyond this limit is possible through virtual memory, this results in extensive swapping which cripples the simulator's performance. The large memory requirements of SimGrid are due to the memory-intensive manner of storing the routing information that is used in the simulated network. Another but less important problem is due to the structure of the *platform (description) files* that describe the network topology in SimGrid. These files are larger than necessary, resulting in significant startup costs of the simulation and decreased manageability of those files.

This paper introduces methods for improving the scalability of the SimGrid simulator. The network representation and the traffic routing functions in particular will be involved. SimGrid's original route information storage and lookup methods will be briefly discussed. Subsequently, different methods to improve them will be described and evaluated using a proof of concept implementation.

2 Scalability Issues

This section gives an overview of the scalability issues that we have identified in SimGrid. As mentioned before, the maximum attainable number of hosts in a simulation is limited by the available memory in the system. We take a look at the way SimGrid processes routing and topology information and how this influences the memory usage of the simulator.

2.1 SimGrid Simulation Infrastructure

The low-level network simulation in SimGrid is performed by the *SURF*-module. *SURF* provides the core functionalities to simulate a distributed system and is also responsible for the parsing and processing of the platform files. Another module, called *SIMIX*, is an intermediate layer between the low-level *SURF*-module and the high-level *MSG* user API. *SIMIX* uses *deployment (description) files* to describe a simulation scenario by assigning *processes*, which contain user-defined logic, to hosts. A typical SimGrid simulation run will first process the platform and deployment description files and then launch the simulation.

2.2 The Platform Files

SimGrid platform files are used to describe the topology of the network to be simulated. These files contain an XML-based description of hosts, network links and routing paths. The specified entities have properties like bandwidth or latency for links, and processing power for hosts. Routing information is statically specified by a list of links forming the path between each pair of hosts in the network. Note that the specified path from source to destination does not necessarily constitute the shortest path, the route can consist of any chain of links.

A recent revision of the platform file XML schema [2] has made it possible to specify a group of hosts using a *cluster* element. A cluster element in the platform file contains a template host configuration for the cluster hosts, that will be expanded to real hosts by SimGrid during the processing of the platform file. This functionality reduces the amount of redundant information and thus the size of the platform file. Specifying routes between clusters is also supported and will result in the generation of routes between the hosts in the source and destination clusters during the parsing process. This means that the simulation code itself has no knowledge about clusters, and that is the cause of a first inefficiency in the network model. Since the simulator only has knowledge about hosts, it has to know the specific routes from and to every host in the network.

Although the inclusion of the cluster element has already led to a significant reduction in the size of platform files, the need to fully specify the routes between all clusters in the platform files still results in rather large files. Furthermore, because *SURF* has no notion of clusters and routes between clusters, all the individual hosts of a cluster and the associated routes between them need to be expanded to an in-memory representation.

2.3 The Simulation Datastructures

During the parsing of the platform file, *SURF* stores the specified hosts and routes, as lists of links in a `xbt_dict_t`[1]. When the parsing is finished, the routing information is converted to one large two-dimensional array of size $n \times n$ (with n the number of hosts) containing the network link lists. Each array-element contains a list of pointers to the actual link data structures that make up the route from one host to another. This array contains all topology and routing information necessary to perform the simulation.

The use of the routing array has a substantial impact on the memory usage of the simulator. A significant amount of information in the route array is redundant, as the hosts of the same cluster have identical routes. Nevertheless, the advantage of this method is that the lookup of a specific route is very fast as it merely involves access to memory by direct indexing. The time and space complexity of the route lookup algorithm are respectively $O(1)$ and $O(n^2 l)$ where n is the number of hosts and l is the average number of links in a routing path.

3 Improving Scalability

As discussed before, the number of hosts in SimGrid is limited by memory. The root of this problem lies in the way the routing information is kept in memory and in the platform files. They both contain redundant routing information by defining all paths between all hosts, even though a specification of the links from

[1] `xbt_dict_t`: SimGrid uses its own dictionary data structure `xbt_dict_t` from the *XBT*-module which associates a `char*` with any `void*` data.

a host to its neighbours is sufficient to determine the routes between all pairs of hosts. By eliminating this redundancy in the routing table, scalability should improve significantly. We have implemented in SimGrid a number of routing algorithms to calculate routes dynamically. This eliminates the need for the extensive all-to-all routing table, reducing the memory footprint and the size of the platform files. In this section, we present a number of routing algorithms that calculate the required routing path at runtime. This consequently reduces the amount of memory needed and the size of the platform files.

Table 1 contains formulas that approximate the routing tables' memory usage for the different algorithms, where n is the number of hosts, l the average number of links in a path, m the average number of outgoing links per host, and c the number of cached entries (the maximum is n).

3.1 Floyd's Algorithm

We have implemented, in SimGrid, the Floyd-Warshall [3] algorithm for finding the shortest path between two hosts in the network. It uses an adjacency matrix that describes the network topology and it produces a predecessor matrix that holds the destination hosts' predecessor in the path between a pair of hosts, and a cost matrix that contains the total cost of that path. The cost and predecessor matrices are calculated before the start of the simulation itself. When this procedure is completed, the list of links composing the route can be easily constructed using the predecessor matrix.

The use of this algorithm results in a significant reduction in memory use. The size of route information data now only depends on the number of hosts, as opposed to the original algorithm where the length of the route path is an important factor. This reduction comes at a cost however, as the algorithm requires more time to initialize before the simulation starts. The route lookups during the simulation, compared to the current SimGrid implementation, are only slightly slower. The time and space complexity of the route initialization algorithm are now $O(n^3)$ and $O(n^2)$ respectively. The lookup of the route link list takes only $O(l)$ where l is the average number of links in a routing path.

Table 1. Routing table memory usage approximation formula's

Method	Mem. Usage Formula	Information stored in memory
Original	$n^2 l \times size_{ptr}$	list of (pointers to) links per host pair
Floyd	$n^2 \times (size_{double} + size_{int} + size_{ptr})$	path cost, predecessor ID of path and link-pointer per host pair
Dijkstra	$n \times (size_{ptr} + size_{int}) + nm$	host ID, adjacent host IDs and link-pointers
Dijkstra w. cache	$n \times (size_{ptr} + size_{int}) + nm + nc \times size_{int}$	host ID, adjacent host IDs, link-pointers and cached predecessor IDs of paths from source host

3.2 Dijkstra Shortest Path Algorithm

A standard Dijkstra shortest path algorithm [4] can be used from within the simulation itself, which eliminates the need for initialization of the routing table, and it will further reduce the memory footprint. For the implementation of this algorithm we have used the C++ Boost Graph Library (BGL) [5]. The algorithm produces a predecessor list for all routing paths starting from one source host to every other host in the network, based on a adjacency list. The reduction in memory usage is due to the use of an adjacency list instead of a matrix to represent the network as a graph.

The Dijkstra shortest path algorithm yields a significant reduction of the memory needed. The space complexity of this algorithm is $O(nm)$, with m the average number of outgoing links per host. This algorithm results in a linear instead of quadratic increase in memory usage as a function of the number of hosts. The time complexity in this case adds up to $O(n\, log\, n)$ for each route lookup.

3.3 Dijkstra with Caching

The Dijkstra algorithm is a substantial improvement when it comes to memory usage, but it has a large impact on runtime performance. To improve this situation, we have used a cache for the calculated predecessor array. All routes from one source host to all other hosts are cached at the first lookup of a path from the source host. The space complexity then becomes $O(nm + cn)$, with m the average outgoing links and c the cache size. Time complexity is still $O(n\, log\, n)$ for the first calculation of a route and $O(log\, c)$ for all succeeding route lookups from the cache, where c is the size of the cache. A C++ Standard Template Library (STL) map [6], is used for the cache. The effect of this cache on actual memory use and runtime performance depends on the simulated scenario.

3.4 Necessary Topology Changes

Implemention of the aforementioned algorithms requires some changes to SimGrid, as SimGrid allows one to describe a route as a chain of links in the platform files where the links do not necessary have real hosts (or other hops) in between them. Our route calculation algorithms are not capable of working with such a route description, because they calculate a routing path as a composition of single links between hosts. The specification of the routes as a chain of links does not allow one to derive the network topology graph, as two hosts in the same cluster can be configured to have a completely different route to the same destination.

Another problem is that SimGrid implicitly creates links when a cluster is specified. To still be able to use the cluster declaration syntax in the platform files, the generation of links for clusters in *SURF* was altered. SimGrid currently only creates a backbone link in a cluster, connecting all of its hosts. The backbone link should then be used in the platform files to link the cluster hosts to a gateway

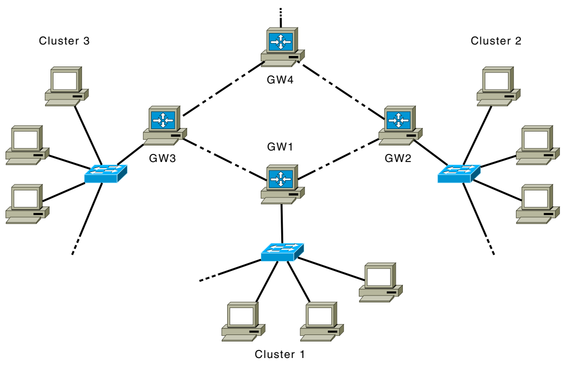

Fig. 1. New cluster network topology

connecting to the rest of the network. This way, all incoming and outgoing traffic has to pass through the cluster's gateway host. To perform a fair comparison against the original SimGrid routing method, this network topology change is also used with the original routing scheme. Figure 1 shows this new topology scheme graphically.

4 Experiments

We compare the algorithms of the previous section to the original approach used in SimGrid. We evaluate the differences in performance, memory usage and disk space requirements. Our evaluation has been carried out with the `masterslave_forward` example that comes with the SimGrid source code. This example simulates a number of *master* hosts sending tasks to other *slave* hosts. Platform files of various sized networks with their corresponding deployment files were generated. The deployment files define 1 host for each cluster that functions either as master or as a slave, and each master has 4 other slave hosts. Each master distributes 20 "tasks" among its slaves. The tests are carried out on a Linux 64-bit machine with 32 GB of RAM. The results are taken from one testrun, as we haven't found a considerable variance between different runs.

4.1 Platform File Size

As discussed before, the new routing scheme eliminates the need for a full specification of all routing paths through the network. Only the links between two adjacent hosts need to be specified. Platform files were generated using a modified version of the Java `PlatformGenerator` from the "contributed section" of the SimGrid source code repository. The platform generator was used to produce platform files that comply with the Barabási–Albert (BA) network topology model [7]. The generated platform files are based on a given number of clusters and a network topology fulfilling the BA-laws. Each cluster connects its hosts through a gateway host to the rest of the network.

Table 2. SimGrid platform file sizes for an increasing number of clusters in the network

Clusters	Original Size	New Size	% of Original
10	44 K	11 K	25.1 %
100	5.5 M	126 K	2.2 %
200	22.2 M	254 K	1.1 %
300	49.6 M	399 K	0.8 %
400	95.7 M	527 K	0.5 %
800	422.4 M	1.0 M	0.2 %

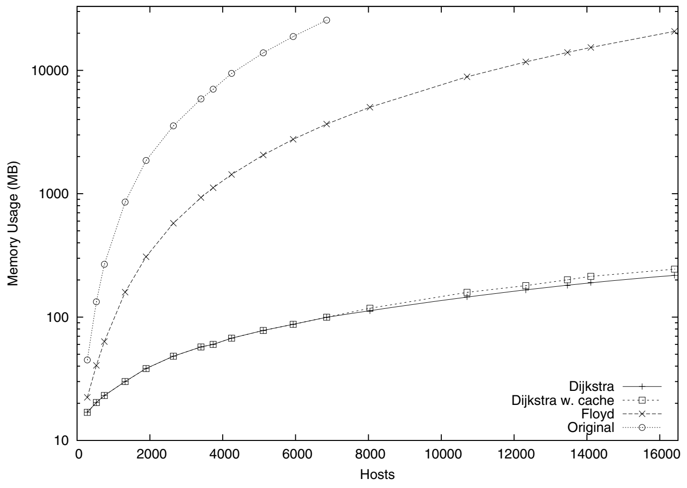

Fig. 2. SimGrid's memory usage in relation to the number of hosts in the network (logarithmic scale)

The second and third column in Table 2 show the platform file sizes respectively for the original and the new platform file structure for different cluster counts. The last column contains the percentage of the new against the original file size. The introduction of a new platform file structure significantly reduces the file size to a fraction of the original size, but it implies losing the ability to describe an arbitrary topology (e.g. a hypergraph).

4.2 Memory Footprint

Memory usage with each of the algorithms is measured by monitoring the `VmSize` value of the running process' status file in the Linux `proc` file system. The `VmSize` is the size of the total address space of a process (not including reserved regions) [8]. The highest value that appeared in this file during a process run is used for the evaluation.

Figure 2 depicts SimGrid's memory usage in relation to the number of hosts[2] specified in the platform file on a logarithmic scale. It is clear that the memory usage of the original SimGrid network representation increases very fast. For example, 20 GB of memory is used at about 6000 hosts. All other algorithms show a significant reduction in memory usage, thus allowing for the simulation of larger networks. The use of Floyd's algorithm significantly reduces the rate of increase and the two variants of Dijkstra's algorithm do so even more. With Floyd's algorithm the simulation requires 20 GB at about 16000 hosts, an improvement of more than 250 %. Furthermore, the Dijkstra algorithms are able to handle up to 16000 hosts with about 200 MB of memory.

4.3 Runtime Performance

To test the runtime performance of the algorithms, the startup time of the simulator and the time it takes to calculate or lookup the routes has to be taken into account. The size of the network affects the startup time for the orginal scheme and Floyd's algorithm, but it has no impact for the Dijkstra schemes. However for the route lookup, the behavior is exactly the opposite. In that case, Dijkstra takes significantly more time to perform the route calculation compared to the other algorithms. Consequently, the amount of simulated traffic as well as the size of the network will influence the performance.

As there is no standard benchmark for the simulator, we focus on the performance of the platform initialization and the route lookups, instead of the duration of a full simulation. We measure the time it takes to complete the initialization functions of SimGrid, namely the `MSG_create_environment` and `MSG_launch_application` functions for the parsing and processing of the platform and deployment files respectively, and the amount of time spent looking up a routing path. We have wrapped the route lookup in a function to be able to profile accurately. This new function is called from the `communicate` function in *SURF*. For the lookup time, we consider the total amount of time spent in the wrapper function divided by the number of calls.

Initialization Performance. Table 3 clearly shows that both variants of Dijkstra's algorithm result in the shortest initialization sequence. The Floyd algorithm requires a significant amount of time to initialize the predecessor and cost matrices. However, storing these on disk can remove this runtime cost for simulations that reuse the same network topology. SimGrid's original routing algorithm is significantly slower in initialization than both Dijkstra variants, particularly for increasing number of hosts.

[2] In the tested development version of SimGrid the number of hosts appeared to be doubled internally compared to the amount of hosts specified by the platform files, resulting in a memory usage that is 4 times as high as it should be for the original and the Floyd route algorithm. This explains the difference between Fig. 2 and the formulas in Table 1.

Table 3. Total time measurement of the platform and deployment file initialization

Clusters	Hosts	Dijkstra	Dijkstra w. cache	Floyd	Original SimGrid
10	280	74 *ms*	73 *ms*	772 *ms*	654 *ms*
50	1319	343 *ms*	353 *ms*	71874 *ms*	16555 *ms*
90	2646	806 *ms*	779 *ms*	608 *s*	68376 *ms*
150	4242	1215 *ms*	1213 *ms*	2578 *s*	183 *s*
180	5112	1483 *ms*	1435 *ms*	4450 *s*	267 *s*
250	6854	1907 *ms*	1968 *ms*	10867 *s*	515 *s*

Table 4. Average duration of a route lookup

Clusters	Hosts	Dijkstra	Dijkstra w. cache	Floyd	Original SimGrid
10	280	7629 μs	367 μs	3 μs	2 μs
50	1319	39403 μs	1672 μs	4 μs	2 μs
90	2646	91058 μs	3854 μs	4 μs	2 μs
150	4242	145 *ms*	6094 μs	5 μs	2 μs
180	5112	173 *ms*	7759 μs	5 μs	2 μs
250	6854	239 *ms*	9828 μs	5 μs	2 μs

Route Lookup Performance. For the route lookup performance, we have measured the execution time of the route calculation function and compared these timings in Table 4. The Floyd algorithm and the original routing algorithm have a negligible route lookup cost, compared to the duration of other calculations in the `communicate` function and the time required for solving SimGrid's analytical network model. Both Dijkstra algorithms perform much worse in this regard. Still, the cache makes a significant difference. The efficiency of the route cache depends on the scenario that is simulated, more specifically on the number of network messages that are sent from the same host. We have measured an increase in average computation time per communication by a factor of 500 for Dijkstra and by a factor of 25 for cached Dijkstra.

5 Future Work

Although our improvements are significant, still more efficient methods can be developed. These may however require more fundamental changes to SimGrid. A more advanced way of representing the network topology in memory can further reduce the memory usage and also improve performance. If SimGrid has a real notion of a clusters and its associated nodes, routing can be carried out between clusters or cluster gateways, partially ignoring the nodes. In a hierarchical network representation, it would also be possible to use different routing algorithms on different levels, each optimized for their specific network topology or application.

6 Conclusion

The simulation of large-scale distributed systems using SimGrid is currently impeded by intensive memory usage of SimGrid's network representation and routing facilities. In addition, the platform files that describe large network configurations require a significant amount of parsing time and disk space. An advantage however, is the low runtime cost of route lookups which is independent of the number of hosts in the network.

In support of simulating large-scale systems with SimGrid, we have presented a number of routing algorithms that considerably reduce the amount of memory required for simulating the network. We have demonstrated that route calculation with Floyd's routing algorithm significantly reduces the memory footprint. Startup costs of the simulation however are heavily affected by the number of hosts, while the route calculation is almost as fast as the original route lookup method. The Dijkstra algorithm results in a major decrease in memory usage, but induces a runtime cost for calculating the routes at each route lookup. In order to mitigate this, a cached version of Dijkstra's algorithm was introduced that induces this cost only the first time a route is asked for, at the cost of slightly higher use of memory. The introduction of these algorithms results in a classical space-time tradeoff. In this regard, we have shown the implications for each algorithm in terms of memory usage and runtime cost.

Acknowledgements

We gratefully thank Arnaud Legrand and Martin Quinson of the SimGrid development team for their constructive comments and inspiring discussions.

References

1. Casanova, H., Legrand, A., Quinson, M.: SimGrid: A generic framework for large-scale distributed experiments. In: Proceedings of the 10th IEEE International Conference on Computer Modelling and Simulation (UKSIM/EUROSIM 2008), Washington, DC, USA, pp. 126–131. IEEE Computer Society, Los Alamitos (2008)
2. Frincu, M.E., Quinson, M., Suter, F.: Handling very large platforms with the new simgrid platform description formalism. Technical Report 0348, INRIA (2008)
3. Floyd, R.W.: Algorithm 97: Shortest path. Commun. ACM 5(6), 345 (1962)
4. Dijkstra, E.W.: A note on two problems in connection with graphs. Numerische Mathematik 1, 269–271 (1959)
5. Siek, J., Lee, L.: The boost graph library (BGL) (2000)
6. Austern, M.H.: Generic Programming and the STL: Using and Extending the C++ Standard Template Library. Addison Wesley Longman Inc., Amsterdam (1999)
7. Barabasi, A.L., Albert, R.: Emergence of scaling in random networks. Science 286(5439), 509–512 (1999)
8. Bovet, D., Cesati, M.: Understanding the Linux Kernel, 3rd edn. O'Reilly Media, Inc., Sebastopol (2005)