Parallel Agent-Based Simulation on a Cluster of Workstations

Konstantin Popov¹, Vladimir Vlassov², Mahmoud Rafea¹, Fredrik Holmgren¹,
Per Brand¹, and Seif Haridi²

¹ SICS, Kista, Sweden. <http://www.sics.se>

² KTH/IMIT, Kista, Sweden. <http://www.imit.kth.se>

Abstract. We discuss a parallel implementation of an agent-based simulation. Our approach allows to adapt a sequential simulator for large-scale simulation on a cluster of workstations. We target discrete-time simulation models that capture the behavior of WWW. The real-world phenomena of emerged aggregated behavior of the Internet population is studied. The system distributes data among workstations, which allows large-scale simulations infeasible on a stand-alone computer. The model properties cause traffic between workstations proportional to partition sizes. Network latency is hidden by concurrent simulation of multiple users. The system is implemented in Mozart that provides multithreading, dataflow variables, component-based software development, and network-transparency. Currently we can simulate up to 10^6 Web users on 10^4 Web sites using a cluster of 16 computers, which takes few seconds per simulation step, and for a problem of the same size, parallel simulation offers speedups between 11 and 14.

1 Introduction

This paper discusses a *parallel implementation* of discrete-time *agent-based simulation* models. Such models are defined bottom-up, i.e. by defining individual behavior of components called agents. Our approach allows to adapt a sequential simulator for large-scale simulation on a cluster of workstations.

We target simulation models that capture the behavior of WWW. Web users are *active* and *stateful* agents connected with each other in a graph resembling the social network. Web sites are connected in a similar graph, forming a “landscape” where agents reside. At each time step, users exhibit certain behaviours such as visiting some of their bookmarked sites, exchanging information about Web sites in the “word-of-mouth” style, “web surfing” (i.e. visiting sites pointed by sites), and updating bookmarks. Once all users are simulated, the simulation proceeds to the next time step. In general, users access different sites at different time steps. Since users exchange information about sites, every user can acquire a reference and then access every site. The model parameters affect behavior of *individual* model components and their interaction. Running the simulation produces statistics which are used to study the *aggregated* behavior of the Internet population. In particular, the relationship between site popularity (i.e. number

of site visits) and the ratio of sites with that popularity, as well as the formation of clusters of popular sites are studied. Large-scale simulations with millions of model users may aid the understanding of the evolution of the Internet [9].

The simulator distributes data among workstations because of the memory consumption. Every workstation simulates its portion of model users. Simulation of 10^5 model users with 10^4 sites using in current model requires approximately 250Mb of main memory, while we target at least 10^6 users that would require *at least* 2.5Gb. More refined simulation models (for instance, reflecting the evolution of sites in terms of the quality of service, the type of content and the connectivity with other sites) require significantly more memory. On the other hand, we would like to speedup the execution of today's simulation runs – with 10^5 users for thousands of steps. The sequential simulator requires approximately 1 minute for 10^4 users for 10^2 steps on a 1GHz computer, so we could expect the parallel execution to finish in “hours” instead of “days”.

Model users can access arbitrary model sites, so the latter can be seen as *shared* data. This is different to simulation models such as computational fluid dynamics or neural networks, where model entities encapsulate their state entirely. In the worst case, distribution of shared data among computers causes network traffic proportional to the amount of that data. Moreover, users access *different* sites over time, as their bookmarks and bookmarks of their neighbours evolve, so that any distribution of sites aimed at minimizing the network traffic has to be dynamic. We opted for static partitioning because of the complexity and memory and run-time costs of a dynamic one.

The graphs that represent the social network of users and links between sites are “small-world” graphs, where every two nodes are connected through a rather short chain of links, while neighbour nodes are tightly connected with each other [23]. Intuitively, in order to meet these defining properties, such a graph must have a sufficient number of “long-distance” links. In a typical simulation run of our system, 30% of links point to remote nodes which are uniformly distributed over the entire set. Our heuristic for graph partitioning tries to keep tightly connected neighbours on the same workstation. We are unaware of an algorithm that optimizes partitioning with respect to links to remote nodes. Randomness of links to remote nodes causes the majority of them to cross the partition boundaries and thus cause network traffic. The amount of the traffic is proportional to the number of nodes in the graph. Very recently, however, we noticed a work on extraction of larger sub-graphs from a small-world graph [12]; we are studying its applicability for our system.

The system must be also maintainable, so that different web-simulation models could be captured within the same parallel simulation framework. Our colleagues who design and study models often change user behaviours and graph construction mechanisms. To meet this practical need, our system consists of a set of *modules* implementing behaviours of web users, constructing web sites, and also constructing users and sites graphs. The other part of the system is the *infrastructure* that holds collections of users and sites, and runs the simulation. Moreover, the interfaces between modules and the infrastructure part is

the *same* as in the sequential simulator, so our colleagues develop modules using the sequential system and we use the same modules in the parallel one.

Rapid prototyping of simulator modules is facilitated by our implementation platform, the Mozart system [13,21], available for both Un*x systems and Windows. Mozart implements Oz – a concurrent multi-paradigm programming language that offers symbolic computation with a variety of builtin data types, automatic memory management, and network-transparent distribution.

In the parallel system, the implementations of abstract data types (ADTs) that hold collections of users and sites *hide* the network from the simulation code. The data is partitioned between Mozart processes, but remote data can also be cached as long as the simulation model allows. The ADTs' implementations, in turn, are built atop of one single “distributed ADT” (**DistADT**) abstraction which completely *encapsulates* the operations over the network. When data is requested that does not belong to the local partition, a corresponding computer is contacted *inside* of the **DistADT** implementation. Since there are multiple simulation threads, network latency is automatically masked.

The parallel simulator can utilize different workstations. Our current model of data partitioning assigns partition sizes proportionally to the capacities of corresponding computers. Capacities are reversely proportional to the execution times of a certain fixed sequential simulation. The number of threads per Mozart process is a system-wide constant, which turned out to be good enough. The performance of the simulation has been improved by avoiding explicit and implicit synchronization between processes: barriers between time steps have been replaced by a more relaxed synchronization, and garbage collection in Mozart processes is initiated synchronously by the manager process.

We have already obtained promising results: on a cluster of 16 workstations simulating 10^6 users on 10^4 web sites, our simulator scales nearly perfectly if the shared data is stateless and replicated eagerly, nearly perfectly if it is replicated lazily, and deteriorates only by some 40% if data is stateful. If we interpolate the execution time to simulate 10^6 users on a single computer (the value we cannot directly obtain due to memory requirements), the simulation system offers the speedup of 14 with stateless replicated data, and 11 with stateful data.

2 Related Work

Our simulation models belong to the “discrete time system” class [25], which is a specialization of “discrete event system specification” (DEVS) class where time is continuous but only a finite number of events take place. Parallelization of discrete event models has received a lot of attention [25,8,6,16]. Our models are specified in terms of *pro-active behaviour* and *interaction* between entities rather than “events” and “response behaviour”.

It is acknowledged that most of the work on parallel discrete event simulation is done with the focus on parallel speedups, while the scalability, software engineering, and model reuse are the real issues for the modeler [18,24]. All these issues are our goals. Relatively little effort is devoted for large-scale simulation

and corresponding tools, with the notable exceptions of specially designed DEVS implementations [3], DaSSF(Dartmouth SSF, Scalable Simulation Framework)-related activities [4,17], as well as various attempts to build distributed simulators through *federation* of sequential ones [5]. We address the scalability issue by explicit distribution of data, and *encapsulating* the network operations within the implementations of collections of model users and sites.

Parallelizing a simulation involves at least the use of a specially designed simulation platform (e.g. [24]) that hides parallelization details, yet usually an adaptation of the model and some annotations for the simulation run-time system are necessary too [15,5]. Our approach is radically different: only the implementation of collections of agents (web users) and landscape locations (web sites) changes, while the implementations of the agent behaviours are kept unchanged.

Our work can be also assessed from the point of view of multi-agent simulation [10,20,1]. The difference to our work is twofold: while agents in multi-agent simulations appear to be more sophisticated than those in our case, we target orders of magnitude larger simulations. Another extreme are the so-called micro-simulations [2,14], where numerous agents are as simple as finite automata.

Our approach to distributed simulation is novel practically as it does not involve explicit message passing as in [19,11], but rather provides a distributed implementation of the same abstract data types as in the sequential simulator. The programmer's obligation is to spawn the number of threads sufficient for hiding communication latency. Our approach is similar to the one of Nexus [7], where special "global pointers" are network-transparent entities.

3 The Simulation Model

The "word of mouth" (WoM) simulation model is a stochastic socio-economical model that addresses the real-world phenomena of the behavior of the Internet population [9]. The model of time is discrete. The model is defined in terms of Web *users* and Web *sites*. Users are modeled as stateful entities: in particular, they have short-term ("loyalty") and long-term ("bookmarks") memory. Sites have ranks representing their "goodness". Users and sites are interconnected by two separate *small world* (SW) graphs [23]. The users graph represents a social network of interaction between users, and the sites graph represents the Web. The topology of graphs and the number of nodes are fixed. We use the Watts-Strogatz model for building SW-graphs, where a SW-graph is based on a 1-d lattice, which models interconnections between users/sites according to their physical proximity. A fraction of links of the lattice is then randomly rewired.

At each step, each user exhibits a number of *behaviours*: (a) it visits a random subset of its bookmarked sites, (b) contacts a random set of its neighbour users and asks them about sites worth visiting (the "word of mouth" (WoM) mechanism), (c) surfs some links on some of the sites visited during behaviours *a* and *b*, and (d) evaluates sites visited during behaviours *a,b* and *c*, and updates the loyalty/bookmark lists. Users that are contacted during WoM behaviour must complete the previous simulation step, and must not complete the next step.

Some of them *may* have completed the current step; thus, users can be simulated in *any* order within a time step. Note that information about “interesting” sites is propagated through the social network. Best visited sites are accumulated in the short-term memory, and can eventually be promoted to the bookmarks.

Model parameters define the number of users/sites, constants for SW-graph construction, sizes of loyalty/bookmarks lists, and various constants controlling user behaviours. The simulation output is a sequence of statistic vectors containing numbers of visits per each site taken at every N^{th} step. The simulation has shown that the distribution of users per sites follows a universal power law [9]. Note that the model is *stochastic*. The simulation results did not change once a more relaxed synchronization was introduced, where users contacted during the WoM behaviour were allowed to have completed the *next* simulation step.

4 Oz and Mozart

Oz is a dynamically typed language with a variety of basic data types, including atoms, numbers, records and arrays. Builtin dictionaries allow to store and retrieve (*key, value*) pairs. Oz is a lexically-scoped language. Procedures in Oz are *first-class*, and generally have *environment* which *encapsulates* values from the procedure’s lexical scope. Concurrent *threads* in Oz can synchronize by *dataflow variables*. A dataflow variable *represents* a value that can be defined only once. If a value is not defined yet, threads that depend on it are transparently *blocked* by the Mozart run-time system until a value is defined. Defining a variable’s value is referred to *binding a variable*. Threads can also communicate via asynchronous message passing through *ports*. A port is an abstract data type with the “send a value” operation that provides a *partially ordered mailbox* for messages: messages sent by the same thread are retrieved from the box in the same order as they are sent.

Mozart provides for a *network-transparent* distribution of Oz programs: an application runs on a *network of computers* as it were one computer. This is supported by the Mozart’s *distribution subsystem*. Oz entities shared by threads running on different Mozart processes are called *distributed*. Stateless distributed entities are *replicated*, and stateful ones are kept consistent with the help of *access structures* that connect involved processes and are used by entity-type specific *distribution protocols*. When an operation on a distributed stateful entity is performed, distribution subsystems of involved processes exchange message(s) according to the entity’s protocol. Mozart differentiates between *stationary* and *mobile* entities. A ports, for example, is stationary: once created, it stays at a process where created; messages to the port are delivered to its “home” process.

5 The Simulator

Our parallel simulator is based on the sequential simulator which organization follows the one of the *Worker* on Figure 1. Simulation is controlled by the **Manager** component. **Simulator** executes behaviours for each user using data

from **Users** and **Sites** modules, and records statistics into **Statistics**. Model users and sites are represented as records, and are indexed. **Users** and **Sites** abstractions store representations under indices. Users’ neighbours, users’ bookmarks, and outgoing links from sites are represented as lists of indices.

Parallel simulation is conducted by *workers*, each running several threads. **Users** and **Sites** modules hold now *partitions* of data, but provide transparent access to all data. Only locally stored users are simulated. **Users** and **Sites** are the *same abstractions* as in the sequential system but with different implementations, so the **Simulator** component is reused without changes. The **Manager** process constructs the WoM graphs for users and sites, and the representations of users and sites themselves. This data is initialized by the sequential **Users** and **Sites** components, so these components are reused as well. Once constructed and initialized, the data is divided into partitions and partitions are distributed to corresponding workers. Workers construct the distribution-aware **Sites** and **Users** modules containing the given partition data.

We have implemented both the strict and relaxed synchronization between workers, as described in Section 3. With the strict synchronization, there are barriers between simulation steps: the manager waits until all workers finish a step, then it initiates the next step. With the relaxed synchronisation, if a worker at a time step n contacts another worker at step m , then either $n = m$ or $n = m - 1$, i.e. a user that is contacted because of the WoM behaviour is allowed to have completed one step *ahead*. It is implemented as follows. At the beginning of a step n , each worker waits until all other workers report that they completed step $n - 2$, then it by itself tells other workers that it completed $n - 1$, and proceeds with the step n . It can be shown that the time in any two workers can differ by at most 1. Since synchronization messages and accesses to remote data go over the same channels, this guarantees that when a worker at step m receives a request from a worker at step n , then $n = m \pm 1$. The remaining case of $n = m + 1$ means the worker that received a request is lagging behind, so the request is queued until the next step.

Fig. 1. The Simulator.

6 Distributed Abstract Data Types

Users and **Sites** data abstractions of the parallel system contain local data and **GetRemote** procedures which provide access to data from remote partitions. **GetRemote** abstractions are provided by the generic **DistADT** abstract data type, hence the generic “**GetRemote**” name. Let us consider the **Users**

abstraction; the **Sites** abstraction is implemented similarly. **Users** provides the **GiveVote** method that calculates a WoM recommendation from a specified user. **GiveVote** takes a user index and returns a user index. Assume that a requested user happens to be non-local (arrow 1 on Figure 2), in which case **GetRemote** is used. It sends a request to a server on the right partition (arrow 2). The server uses that partition's **GiveVoteLocalUser** procedure (arrow 3). The output of **GetVoteLocalUser** is delivered backwards (arrows 4, 5 and 6) in a *container* sent by **GetRemote** along with the request. A container is implemented as a dataflow variable: it is declared by **GetRemote**, becomes distributed between the two processes when a request arrives to the server, and is bound to the result by the server. Eventually the value is delivered back (arrow 5) by the Mozart distribution subsystem.

Fig. 2. Distributed **Users** ADT.

The **GetRemote** abstraction is provided for **Users** partition by the **DistADT** data abstraction (see Figure 3). **DistADT** modules are shared among all partition processes. In our example, the server Oz port and process logically belong to the **DistADT** module serving the **Users** module. Although not

shown on picture, ports and servers of other **Users** partitions belong to the same **DistADT** module too. There is exactly one port/server pair per partition. The **DistADT** module encapsulates a map from user indices to their partition indices, so for any user index **GetRemote** can choose the correct port and send a request to the correct partition.

Fig. 3. **Users** ADT using **DistADT**.

In order to use a **DistADT** module, **Users** partitions have to *register*. Registration gives a **DistADT** module access to local partition data. In our example, the **GiveVoteLocalUser** procedure is the access abstraction. During registration a (stationary) Oz port is constructed and a local server (a thread that processes requests sent to the port) is started. **DistADT** modules are created by the manager and passed over to all partitions, so they become shared between partitions.

Note that invocation of `GetRemote` transparently returns a value of whatever type the server’s “`GetLocal`” (`GetVoteLocalUser` in our example) does. In this way, two (distinct) instances of the very same `DistADT` data abstraction are used for both `Users` and `Sites` ADTs.

Both the “*data to computation*” and “*computation to data*” approaches to parallel execution are supported by `DistADT`, and used in our system. In the `GiveVote` example, the vote is computed by the process holding the data. Computing a vote requires analysis of the bookmark list, while both the query and the response messages exchanged between processes contain only an index. Thus, it is beneficial to move the computation to data. On the other hand, the complex algorithm of user simulation requires representations of several sites visited by the user. Those sites can even belong to different partitions. Thus, site representations (data) are moved to the partition where simulation takes place.

7 Performance

We have evaluated our system on a cluster of 16 AMD Athlon 1900+ computers connected by the 100Mbit switched Ethernet. Scalability and speedup properties are presented on Figure 4. The “no caching/replication” case refers to the situation when model sites that reside on a remote worker are retrieved over the network every time they are needed. The “caching” version keeps retrieved sites for the duration of the model time step. In the version “with replication”, all workers have their own copies of *stateless* model sites. While in the WoM model the sites are stateless, our experiments show how the simulator would behave if that data *were* stateful. Since the *majority* of data is remote even with 16 workers, the system should scale well also beyond that size. The system also delivers respectful speedups when it counts *most* – for larger problem sizes.

Fig. 4. Scalability and Speedups.

Figure 5 shows the impact of multithreading. Given the properties of the simulation model and the best-case communication latency in Mozart, we have also estimated the number of threads required to hide communication latency using the (analytical) deterministic model of processor utilization in a multithreaded

architecture (see e.g. [22])¹. The model predicts that approximately 100 threads is needed to hide the latency, which is in compliance with our experimental data.

The next plot shows that sporadical load imbalance due to e.g. activities of other processes in OSeS is a no big issue for our simulator. We emulated such kind of imbalance by deliberate over/underloading of one worker during initial partitioning. Note also that *underloading* a worker, which apparently causes its more frequent preemption by the OS, does *not* have a “cascading” effect on other workers. We are investigating this issue further.

Fig. 5. Number of threads, load balance, and synchronization.

Finally, the effect of “relaxed” synchronization discussed in Sections 3 and 5 is presented: one can see quite a significant improvement, in particular for *larger* simulations. Given that a barrier in Mozart can require ca. 50msec, so on 100 steps ca. 5 seconds could be saved just by eliminating the barriers, the improvement is larger than we have expected for *large* simulations. Our hypothesis here is that relaxed synchronization allows to “smooth out” *temporal* imbalance of work *between* adjacent time steps.

8 Conclusions

We have shown a design of a parallel simulator for discrete-time agent-based simulation models targeted for studying the evolution of WWW. Distribution of data among computers is mandatory due to memory requirements, but efficient parallel simulation is not trivial because of the large amount of spatially and temporally irregular inter-process communication. At the same time, the simulator must be easily adapted for a set of different simulation models.

The parallel simulator runs on both a network of workstations and a multi-processor, and shows good scalability and speedup for up to 1M of model agents. The simulator is small and easy to maintain. It is implemented in Oz/Mozart that supports symbolic, concurrent and network-transparent programming, all necessary for this work. The evaluation of the simulation on a cluster of workstations is still going on, concurrently with integration of new simulation models.

¹ $N_{threads} = L/(R + C) + 1$, where L and R are the mean times for communication latency and thread run time between remote memory accesses, respectively, and C is the thread context switch time.

Acknowledgments. We are thankful to all members of our lab for the creative environment, ubiquitous support and constant encouragement. Special thanks go to Petros Kavassalis and Stelios Lelis from ICS-FORTH, Heraklion, Crete, Greece. The work is done within the EU project iCities (IST-1999-11337).

References

1. Anderson, J.: A Generic Distributed Simulation System for Intelligent Agent Design and Evaluation. In: Proc. of the 10th Int. Conf. on AI, Simulation, and Planning in High Autonomy Systems (2000) 36–44
2. Cetin, N., Nagel, K., Raney, B., Voellmy, A.: Large-Scale Multi-Agent Transportation Simulations. In: Proc. of the Workshop on Cellular Automata, Neural Networks and Multi-Agent Systems in Urban Planning, Milan, Italy (2001)
3. Chow, A.C.H., Zeigler, B.P.: Parallel DEVS: A Parallel, Hierarchical, Modular, Modeling Formalism. Proc. of the 26th Winter Simulation Conf. (1994) 716–722
4. Cowie, J., Liu, H., Liu, J., Nicol, D., Ogielski, A.: Towards Realistic Million-Node Internet Simulations. In: Proc. of PDPTA'99, Las Vegas, NV, USA (1999)
5. Ferenci, S.L., Perumalla, K.S., Fujimoto, R.M.: An Approach for Federating Parallel Simulators. In: Proc. of PADS'00, Bologna, Italy. IEEE CS Press (2000) 63–70
6. Ferscha, A.: Parallel and Distributed Simulation of Discrete Event Systems. In: Parallel and Distributed Computing Handbook. McGraw-Hill (1996) 1003–1041
7. Foster, I., Kesselman, C., Tuecke, S.: The Nexus Approach to Integrating Multithreading and Communication. J. of Paral. and Dist. Computing **37** (1996) 70–82
8. Fujimoto, R.M.: Parallel and Distributed Simulation Systems. Wiley (2000)
9. Lelis, S., Kavassalis, P., Sairamesh, J., Haridi, S., Holmgren, F., Rafea, M., Hatis-tamatiou, A.: Regularities in the Formation and Evolution of Information Cities. In: Second Kyoto Workshop on Digital Cities. LNCS 2362, Springer (2001) 41–55
10. Logan, B., Theodoropoulos, G.: The Distributed Simulation of Multiagent Systems. Proceedings of the IEEE **89** (2001) 174–185
11. Lowenthal, D.K., Freeh, V.W.: Architecture-Independent Parallelism for Both Shared- and Distributed-Memory Machines Using the Filaments Package. Parallel Computing **26** (2000) 1297–1323
12. Matsuo, Y.: Clustering using Small World Structure. Proc. of the Int. Conf. Knowledge-Based Intelligent Inform. and Engineer. Systems, Crema, Italy (2002)
13. Mozart Consortium: The Mozart Programming System. <http://www.mozart-or.oz/>
14. Nagel, K., Rickert, M.: Parallel Implementation of the TRANSIMS Micro-Simulation. Parallel Computing **27** (2001) 1611–1639
15. Nicol, D.M., Heidelberger, P.: Parallel Execution for Serial Simulators. ACM Transactions on Modeling and Computer Simulation (TOMACS) **6** (1996) 210–242
16. Nicol, D.M.: Principles of Conservative Parallel Simulation. In: Proc. of the 28th Winter Simulation Conf., Coronado, CA, USA. ACM Press (1996) 128–135
17. Nicol, D.M., Liu, J.: Composite Synchronization in Parallel Discrete-Event Simulation. IEEE Transactions on Parallel and Distributed Systems **13** (2002) 433–446
18. Page, E.H.: Beyond Speedup: PADS, the HLA and Web-Based Simulation. In: Proc. of PADS'99, Atlanta, GA, USA. IEEE Society Press (1999) 2–9
19. Strumpen, V.: Software-Based Communication Latency Hiding for Commodity Workstation Networks. In: Proc. of ICPP'96. CRC Press (1996) 146–153
20. Uhrmacher, A.M., Gugler, K.: Distributed, Parallel Simulation of Multiple, Deliberative Agents. Proc. of PADS'00, Bologna, Italy. IEEE CS Press (2000) 101–108

21. Van Roy, P., Haridi, S.: Concepts, Techniques, and Models of Computer Programming. MIT Press (2004) (to appear)
22. Vlassov, V., Ayani, R.: Analytical Modeling of Multithreaded Architectures. *J. of Systems Architecture* **46** (2000) 1205–1230
23. Watts, D.L.: Small Worlds: the Dynamics of Networks Between Order and Randomness. Princeton University Press, New Jersey, USA (1999)
24. Wieland, F., Blair, E., Zukas, T.: Parallel Discrete-Event Simulation (PDES): A Case Study in Design, Development, and Performance Using SPEEDES. In: Proc. of PADS'95, Lake Placid, NY, USA. IEEE CS Press (1995) 103–110
25. Zeigler, B.P., Praehofer, H., Kim, T.G.: Theory of Modeling and Simulation: Integrating Discrete Event and Continuous Complex Dynamic Systems. Academic Press (2000)