# Facilitating Cellular-Automata-Based Computational Space for Parallel Multi-Agent Simulations

Naoya Suzuki*, Munehiro Fukuda† and Koichi Wada*
* Information Sciences and Electronics, University of Tsukuba, Tsukuba, Ibaraki, 305-8573, Japan.
E-mail: {nas, wada}@is.tsukuba.ac.jp
† Computing and Software Systems, University of Washington, Bothell, WA 98011, USA.
E-mail: mfukuda@u.washington.edu

*Abstract*— **The multi-agent simulation is viewed as the interaction of agents, and is frequently applied for analyzing their self-organization in complex systems. To decide their next behavior, such agents generally access spatial information that needs to be not only made available to the agents in local but also diffused over an entire simulation space. As an efficient implementation of information diffusion, we propose to combine the principle of cellular automata into the multi-agent simulation. There, agents walk over cellular automata that diffuse spatial information. To evaluate its efficiency, we have facilitated such information diffusion on top of the M++ parallel multi-agent simulator. Our performance experiment showed that implementing cellular automata in the Codi-1bit multi-agent simulation achieved 2.18-time speed-up when using 8 processors, as compared to the one with no support of cellular automata.**

## I. INTRODUCTION

The multi-agent simulation explores self-organization in complex systems through the interaction among simulation entities (or agents in the context of our paper), each having its own goal, behavior, and local information in a synthetic world [1]. This style of simulation has gained its popularity in areas such as artificial life [2], ecological simulation [3], and biological modeling, in which traditional mathematical techniques cannot always be applied effectively. In spite of its recognition, the multi-agent simulation has a scalability problem where millions of agents are required for simulation to increase its precision of computed results. Thus, of the major concerns is how to maintain competent performance for a large size of simulation problems.

One solution is parallel execution where agents are distributed over multiple computing nodes and are processed in parallel. The High Level Architecture (HLA) [4] is an IEEE-standardized technique to maintain a global simulation space among networked computers, each publishing local agents to and subscribing remote agents from the other processors. However, the problem is that agents are static to where they have been instantiated, which incurs a considerable amount of inter-processor communication among agents [5]. Because of this reason, HLA is better suited for distributed simulation with coarse-grain agents. As an alternative approach, we have agents migrate where they meet, so that all their communication takes place locally. On top of a cluster of computers, we have developed the M++ parallel multi-agent simulator in that agents are executed as threads, each capable of dynamically creating and autonomously navigating through an application-specific logical network [6]. In M++, a network node represents a simulation subspace, and a network link is used as a route from one node to another. Using M++, we have coded several simulation programs including Codi-1bit [7] that seeks for a neural network best fitted to a specific signal generation [8].

Although we have demonstrated the merits of designing each agent from its behavioral point of view, we have also noticed a general problem: the multi-agent simulation cannot always best model and execute the diffusion of spatial information. For example, Codi-1bit requires signals to be diffused over a neural network, which could be implemented by having an agent either send a signal message or carry signal information with it to other agents. This not only enlarges a semantics gap between simulation algorithm and its implementation, but also deteriorates the entire system performance.

To address this problem, we propose to combine the principle of cellular automata into the multi-agent simulation, where spatial information is diffused to adjacent subspaces every simulation time increment. We have facilitated such a cellular-automata-based computational space into M++. Specifically, all logical nodes include the *clocking()* abstract method that is implemented by a simulation programmer and is invoked by the system periodically in order to exchange spatial information with all neighboring nodes. We redesigned our Codi-1bit application with this new feature, and obtained convincing performance results as well as better programmability.

The main purpose of this paper is to demonstrate the merits of facilitating a cellular-automata-based computational space into parallel multi-agent simulations, by reprogramming and executing the Codi-1bit simulation on M++. The rest of our paper is organized as follows: Section 2 introduces the M++ execution model; Section 3 explains about the Codi-1bit algorithm and implementation; Section 4 evaluates the performance improvement; and Section 5 concludes our discussions.

## II. M++ EXECUTION MODEL

The M++ execution model is based on three different network layers as shown in Figure 1. The lowest is the *physical network* that defines underlying computing nodes,

(e.g. a Myrinet cluster of eight PCs in our implementation.) The middle is the *daemon network*, a completely-coupled communication network among daemon processes, each of which runs on a different computing node and manages agents. The highest is the *logical network*, an application-dependent computational network that is composed of nodes and links: the former objects are used as a working space where agents compute their next behavior, whereas the latter as a guide way that routes agents from one node to another.

Agents are represented as *self-migrating threads* (simply termed threads in the rest of our discussions) that are programmed in the M++ language, an agent-oriented language based on C++, are translated into C++ by the M++ translator, are compiled into executable threads, and are finally instantiated from a Unix shell prompt [9]. A thread is capable of constructing the logical network and navigating itself over the network with the following four keywords: (1) *create* instantiates a given node or a link; (2) *delete* deletes an existing node or a link; (3) *hop[along]* navigates a thread to a given node or along a link; and (4) *fork[along]* propagates a copy of thread to a given node or along a link. The network navigation is based on strong migration [10] where a thread keeps running in the same execution context even after its migration to a different node.

Nodes and links are both implemented as an ordinary C++ object. As a working space, a node permits threads to stop it by and to share its variables and methods. On the other hand, a link is intended as a guide way which threads can only pass along. Instead, threads can access variables and methods of a link incident to a node where they currently reside. Since nodes and links are passive objects, threads are responsible for diffusing their information, (i.e., carrying spatial information from node to node or link to link.) For better programmability, such information diffusion should be automated using the principle of cellular automata rather than threads, so that simulation designers can focus on implementing agents in threads.

The M++ has facilitated such cellular-automata based information diffusion, where all nodes provide the *clock()* abstract method that is implemented to access their incident links and is called by the M++ daemon every time a thread executes its *clocking* statement. With this *clocking* statement, all adjacent nodes can exchange their node information through their incident links at a time.

Figure 2 gives an example of M++ program. Upon an instantiation, a thread sets the *daemonId* variable to 0 in its constructor and starts executing the *main()* function that describes the corresponding agent behavior. It then creates the node 1 on the daemon 0, establishes the link 1 from the current working node to the node 1, and hops along the link 1 to the node 1 where it has all nodes invoke their *clock()* function as passing the argument 0.

An actual simulation with M++ is carried out by programming and injecting the following three classes of threads: the first is in charge of creating a simulation space by adding new nodes and links to the logical network; the second is programmed to periodically execute a *clocking* statement to trigger information exchange between adjacent nodes and to ultimately diffuse such node information throughout the space; and the third represents various types of agents.

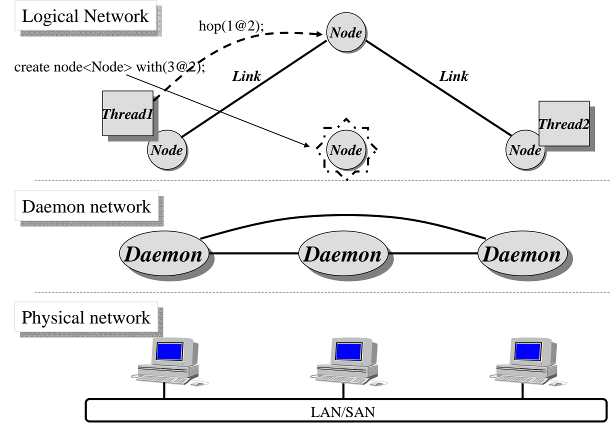


Fig. 1. M++ execution model

```
class Node {
public:
  void clock( int time ) {
    link[1]<Link>.data = time;
} };

class Link { public: int data; };

thread Thread {
public:
  Thread(): daemonId( 0 ) {}
  void main() {
    create node<Node>() with( 1@daemonId );
    create link<Link>() with( 1 )
      to ( 1@daemonId ) with ( 1 );
    hopalong( 1 );
    clocking 0;
  }
private:
  int daemonId;
};
```

Fig. 2. M++ thread framework

## III. CODI-1BIT ON M++

Using the Codi-1bit model, we have verified the effectiveness of cellular-automata-based information diffusion in the multi-agent simulation. It is a neural network simulation developed by the Advanced Telecommunication Research Institute in Japan [7]. The goal of this simulation is to build, grow, and evolve a cellular-automata-based neural network for solving various problems. One of the problems tackled is to obtain a neural network that emits a signal close enough to the $\sin\theta (0 \leq \theta \leq 2\pi)$ curve at a given observation point. The model prepares 30 different pieces of genetic code, for each of which it builds a cubic array of $24\times24\times18$ cells, initializes 48 cells as a neuron and the others as a blank, simulates the growth of neural network, (i.e., change blank cells in an axon or a dendrite cell) for the first 100 cycles, and charge the network with neural signals, (i.e., transfer signals from the 48 neurons along adjacent axon and dendrite cells) for the following 330 cycles. Thereafter, the model computes the

next generation of 30 genes through the processes of selection, crossover, and mutation. The original program was coded in C to find the best genetic code through the repetition of 10 generations.

Among the above computation phases, the 330-cycle signal transmission is a phase of information diffusion. A neural signal is periodically charged to each neuron, is dynamically diffused to the adjacent axons and dendrites according to their signal reactivity, is collected at a given neuron, and is checked if it matches the sine curve.

We have coded this signal transmission in two different versions of M++ program: one is the agent-based signal transmission where signals are carried by agents, (i.e, threads), and the other is the combination of the agent-based and the cellular automata-based signal transmission where signals are diffused through cellular automata periodically activated by threads. (The latter is abbreviated to the combination-based signal transmission in the following discussions.) In the both versions, the logical network is formed as a cubic mesh of nodes and links that are instantiated from the Brain and the Nerve class respectively.

Figure 3 shows the M++ code of the agent-based signal transmission. Starting from a NEURON-type node, a thread repeatedly checks the current node type, (i.e., NEURON, AXON, or DENDRITE) in *main()* and calls the corresponding rule function to propagate its copy along NERVE links to the adjacent nodes. For instance, in the *propagateByNeuronRule()* function, a thread checks whether the current node is active, (i.e., ready to transmit signals) or not. If it is active, the thread propagates its copy to the adjacent nodes in six different directions through *forkalong()*.

Figure 4 shows the M++ code of the combination-based signal transmission, where any pair of Brain nodes, each incident to the same Nerve link, read and write variables of this link. A *Signal* thread stays in each NEURON node and charges it periodically. This charge starts in *main()* where the thread executes a clocking statement to invoke the *clock()* method of all the nodes simultaneously. Each node checks its own type and calls the corresponding rule function so as to exchange its signal through the link variables with the adjacent nodes. For example, in *exchangeSignalByNeuronRule()*, a node writes 1 to the *signal* variable of six incident links that will be read by the adjacent nodes.

The combination-based signal transmission has two superiorities over the agent-based signal transmission. One is a narrower semantics gap between simulation algorithm and its implementation. While the multi-agent simulation is viewed as the interaction among agents, it is natural to consider spatial information, (i.e., signals in Codi-1bit) as passive data but not active agents. Another is better performance. Signals in the combination-based signal transmission are link variables and thus much lighter-weight instances than threads used in the agent-based transmission, because of which the combination-based transmission performs faster. In the next section, we will especially focus on this performance merit.

```
thread Signal {
private:
  int i, s;
  void propagateByNeuronRule() {
    if( node<Brain>.active ) {
      for( i = 0 ; i <= 5 ; i++ )
        forkalong( i );
  }
  // the following two resemble the above method
  void propagateByAxonRule(){ ... }
  void propageteByDenRule(){ ... }
public:
  void main() {
    for( s = 0 ; s < SignalSteps ; s++ ) {
      switch( node<Brain>.type ) {
        case NEURON:
          propagateByNeuronRule(); break;
        case AXON:
          propagateByAxonRule(); break;
        case DEND:
          propagateByDendRule(); break;
}  }  }  };
```

Fig. 3. M++ code of agent-based signal transmission

```
class Nerve { public: int signal; }

class Brain {
private:
  bool active; int i;
  void exchageSignalByNeuronRule() {
    if( active )
      for( i = 0 ; i <= 5 ; i++ )
        link<Nerve>[i].signal = 1;
  }
  // the following two resemble the above method
  void exchangeSignalByAxonRule() { ... }
  void exchangeSignalByDendRule() { ... }
public:
  int type;
  void clock( int t ) {
    switch( type ) {
      case NEURON:
        exchageSignalByNeuronRule(); break;
      case AXON:
        exchangeSignalByAxonRule(); break;
      case DEND:
        exchangeSignalByDendRule(); break;
} } };

thread Signal {
 int s;
public:
  void main() {
    for( s = 0 ; s < SignalSteps ; s++ ) {
      signalToNeuron(); // charge Neuron nodes
      clocking s; // issues clocking mechanism
} } };
```

Fig. 4. M++ code of combination-based signal transmission

## IV. Performance

We have compared the simulation performance between the agent-based and the combination-based signal transmission in the Codi-1bit model, under the test environment summarized in Table I. The evaluation was conducted by repeating four generations as changing the number of processors from 1 to 8.

As shown in Figure 5, the combination-based signal transmission yielded better performance than the agent-based transmission when we used four or more processors. In particular, it achieved 2.18-time speed-up as compared to the agent-based transmission in execution with eight processors.

The agent-based transmission itself improved its performance 1.15-time better when using two processors, which however slowed down as adding more processors. This performance degradation is correlated with a trade-off between the number of threads per machine and that of thread migrations over the system, (i.e., between the parallelization effect and the communication overhead.) Adding more processors

TABLE I
TEST ENVIRONMENT

| Features | Descriptions |
|---|---|
| CPU | Athlon 1GHz（FSB 266MHz) |
| Memory | DDR-SDRAM 256MB |
| Network card | Myrinet M2MPCI32B |
| Switch | Myrinet M2M-DUAL-SW8 |
| OS | Solaris8 |
| Compiler | gcc-2.95.3 |
| #CPU consisting a cluster | 8 |

reduces the number of threads allocated per machine but incurs more thread migrations. The latter overhead is proportional to the number of inter-processor links that grows rapidly as more processors are added. In fact, the total number of inter-processor links is 432 in two processors, 1296 in four processors, and 1728 in eight processors. A two-processor configuration still gained parallelization effect that overcame the cost of thread migration. However, using four or eight processors deteriorated the performance due to the rapidly growing number of inter-processor links, (e.g., three times and four times larger than using two processors respectively), which incurred the proportional number of thread migrations.

On the other hand, the combination-based signal transmission achieved better than the agent-based transmission in four- and eight-processor configurations. It is due to a trade-off between the number of *clock()* function calls and that of inter-processor links. As shown in Figure 5, a single processor execution took more time in the combination-based than in the agent-based transmission. This is because it had to call the *clock()* function of all $24\times24\times18$ nodes. However, the number of such function calls can be reduced by adding more processors. While the number of inter-processor links grows on the other hand, the link size in Codi-1bit is 200 bytes, which is 2/3 of the thread size, (i.e., 316 bytes.) Thus, communication overhead is less in the combination-based than in the agent-based transmission. Furthermore, there is a big difference in the number of active threads between the agent-based and the combination-based transmission. The former needs approximately 150 threads that are created, propagated to other nodes, and destructed every simulation cycle. In contrast, the latter uses only 48 threads, each fixed to a given neuron cell and charging it with a signal periodically, which thus causes no thread migrations. Therefore, the combination-based transmission can minimize the cost of thread creation, destruction, migration, and context switch. As a result, these factors brought better parallelism in execution with eight processors.

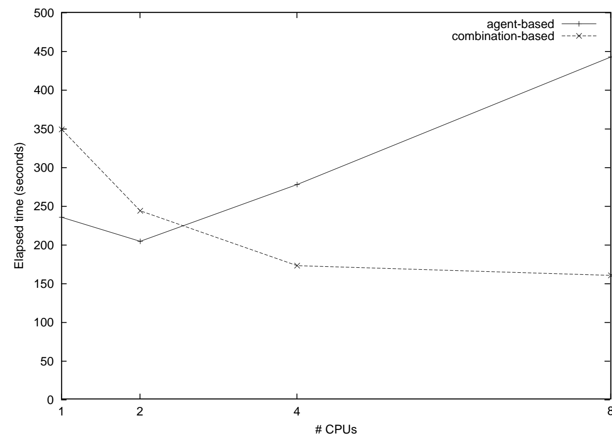


Fig. 5. Codi-1bit performance

## V. CONCLUSION

In this paper, we have proposed the combination-based signal transmission that combines the principle of cellular automata into the multi-agent simulation, where spatial information is diffused to adjacent subspaces every simulation time increment. To evaluate our proposed scheme, we have programmed the Codi-1bit simulation in both the agent-based and the combination-based signal transmission using M++, and have compared their code and performance. As discussed in section IV, the combination-based signal transmission improved simulation performance especially when the size of spatial information to be exchanged, (i.e., the M++ link size) is smaller than the thread size. We have also demonstrated that the use of cellular automata is effective for reducing the number of threads to be spawned, destructed, and migrated, which considerably improves the entire simulation performance.

Our next plan is to develop a visualization toolkit that eases on-going status checks and result presentations of agents and cellular automata.

## REFERENCES


[1] J. Ferber, *Multi-Agent Systems An Introduction to Distributed Artificial Intelligence*. Addison-Wesley, 1999.

[2] R.Collins and D.Jefferson, *Artificial Life II*. Addison-Wesley, 1992, ch. Antfarm, pp. 579–601.

[3] J. M.Epstein and R. L.Axtell, *Growing Artificial Societies*. Cambridge, MA 02142-1493: MIT Press, 1996.

[4] *IEEE Std 1516-2000, IEEE Standard for Modeling and Simulation (M&S) High Level Architecture (HLA)*. New York, NY 10016-5997: IEEE, 2000.

[5] B. Logan and G. Theodoropolous, “The distributed simulation of multi-agent systems,” in *Proc. of the IEEE 89(2)*, 2001, pp. 174–185.

[6] N.Suzuki, M.Fukuda, and L.F.Bic, “Self-migrating threads for multi-agent applications,” in *Proc. of IEEE Computer Society International Workshop on Cluster Computing*, Melbourne, Australia, December 1999, pp. 221–228.

[7] Norberto, E.Nawa, M.Korkin, and H. garis, “Atr’s cambrain project: The evolution of large-scale recurrent neural network modules,” in *Proc. of PDPTA’98*, Las Vegas, NV, July 1998, pp. 1087–1094.

[8] M. Fukuda, N. Suzuki, L. M. Campos, and S. Kobayashi, “Programmability and performance of m++ self-migrating threads,” in *Proc. of IEEE Int’l Conference on Cluster Computing*, Newport Beach, CA, October 2000, pp. 331–340.

[9] M. Fukuda and N. Suzuki, “M++ user’s manual,” University of Washington, Bothell, WA 98011, Technical Report available in http://dept.washington.edu/dslab/m++/user_man.ps, September 2002.

[10] G. Cugola, C. Ghezzi, G. P. Picco, and G. Vigna, “Analyzing mobile code languages,” in *Mobile Object Systems: Towards the Programmable Internet*. Springer-Verlag: Heidelberg, Germany, 1997, pp. 93–110.