SCIENCE CHINA
Technological Sciences

February 2013 Vol.56 No.2: 509–517
doi: 10.1007/s11431-012-5084-2

Parallel architecture and optimization for discrete-event simulation of spike neural networks

TANG YuHua[1,2], ZHANG BaiDa[2,1*], WU JunJie[2,1], HU TianJiang[3], ZHOU Jing[2,1] & LIU FuDong[2,1]

[1] *Department of Computer Science and Technology, School of Computer, National University of Defense Technology, Changsha 410073, China;*
[2] *State Key Laboratory of High Performance Computing, National University of Defense Technology, Changsha 410073, China;*
[3] *College of Mechatronic Engineering and Automation, National University of Defense Technology, Changsha 410073, China*

Received August 31, 2012; accepted November 2, 2012; published online December 3, 2012

Spike neural networks are inspired by animal brains, and outperform traditional neural networks on complicated tasks. However, spike neural networks are usually used on a large scale, and they cannot be computed on commercial, off-the-shelf computers. A parallel architecture is proposed and developed for discrete-event simulations of spike neural networks. Furthermore, mechanisms for both parallelism degree estimation and dynamic load balance are emphasized with theoretical and computational analysis. Simulation results show the effectiveness of the proposed parallelized spike neural network system and its corresponding support components.

spike neural network, discrete event simulation, intelligent parallelization framework

Citation: Tang Y H, Zhang B D, Wu J J, et al. Parallel architecture and optimization for discrete-event simulation of spike neural networks. Sci China Tech Sci, 2013, 56: 509–517, doi: 10.1007/s11431-012-5084-2

1 Introduction

It has long been the dream of mankind to develop a continuously self-learning computer able to make complex decisions as human do. Artificial neural networks, based on computer models of the human brain, have been constructed, and are successfully used in a wide range of fields. However, artificial neural networks perform poorly when they are presented with non-reasoning problems, such as object recognition and natural language understanding. Experimental evidence increasingly points towards this defect caused by the coding model—artificial neural networks use a rate-based coding model, whereas real neural networks are spike-based. For this reason, spike neural networks have attracted much interest. Compared to traditional artificial neural networks, spike networks use a spike-based coding model for a more accurate and detailed representation of the human brain.

There are many simulators and projects based on spike neural networks, e.g., GENESIS [1, 2], NEURON [3, 4], SpikeNET [5], PyNN [6], BRIAN [7, 8], NEST [9, 10], BNN [11], the Blue Brain Project [12] and the FACETS Project [13]. Many of them involve dozens of research institutions across a number of countries. Simulation of tens of millions of neurons, and the complexity of the synaptic connections between them, is computationally intensive and time-consuming, even for high-performance clusters. Figure 1 shows a schematic of the size of some representative animal brains in terms of the number of neurons. From this figure, we can see that there are too many neurons for per-

*Corresponding author (email: zhangbaida@gmail.com)

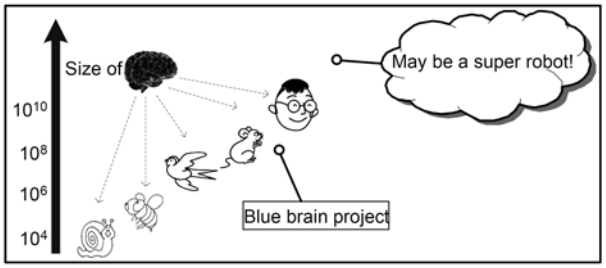

Figure 1 Brain size of humans and some animals. The axis gives the approximate number of neurons.

sonal computers to simulate even the brain of lower animals like the rat.

To simulate a spike neural network with the size of the human brain would require a storage capacity of up to several GB, and the simulation of a second of brain activity requires the calculation of approximately 10^{15} flops [14] for even the most simple integrate-and-fire (IF) [14] neuron model. If more complicated models are used, the required computation and storage rise rapidly. For example, the dynamic behavior of only 10000 neurons at most can be simulated despite using IBM's "Blue Gene" [12] super-computer. Therefore, parallelism is suggested and developed to model real-time, large-scale spike neural networks in this paper.

Parallel computing uses a variety of computing resources to solve one computational problem. Its main purpose is to accelerate the process to resolve large and complex problems, as well as to overcome the memory limitations of a single processor [15]. A lot of research has focused on the parallel simulation of spike neural networks [16, 17]. And simulators on several parallel platforms already exist, such as GPU [18] and FPGA [19]. In this article, we concentrate on simulations using a parallel computer. This provides the best performance, and is one of the most widely used parallel platforms [20]. For example, many simulators, such as GENESIS [1, 2], NEURON [3, 4], Neurosys [21], Spinnaker [22], and PCSIM [23] all support simulations on a parallel computer. However, users of the above simulator packages must draw upon their past computational experiences to partition the simulation task into suitable parallel processes [1, 3, 14, 21–29], which is undoubtedly a heavy burden for researchers of spike neural networks. This paper attempts to solve this problem automatically by some intelligent means.

Simulation can be divided into two categories according to the different driving factors—clock-driven simulation and event-driven simulation. In contrast to clock-driven simulation, the key advantage of event-driven simulation is that computation is only required at each spike point. In this paper, we consider only event-driven simulation models, and present a performance analysis model for spike neural networks. The focus of our paper is an intelligent method to estimate the best degree of parallelism and a dynamic load balancing method. We also prove two theorems, the optimal parallel degree theorem and the optimal load balancing theorem. A load balancing algorithm is also presented and implemented. Finally, a simulation framework for automatic parallelization is developed, based on the theory in this paper, allowing efficient parallel programs to be generated automatically. The experimental results show that the algorithm can effectively solve some of the issues with spike neural networks, namely their poor parallel simulation performance and the requirement for multi-user intervention. Typically, the simulation performance can be increased by 18.6%.

The rest of the paper is organized as follows. Firstly, Section 2 discusses the problem and the focus of our work. Section 3 then presents an intelligent method to estimate the best degree of parallelism and Section 4 describes our intelligent load balancing method. Some experimental results are presented in Section 5, and conclusions are given in Section 6.

2 Parallel simulation framework of a spike neural network

In the nervous system, the transmission and processing of all information are accomplished by neurons, which are formed by soma and several dendrites and axons starting from the soma [30, 31]. There are many connections between neurons, called synapses. Roughly speaking, the main computation completed by neurons is their excitability when the neuron cell body and dendrites receive a spike from other neurons. When the excitement level exceeds a certain threshold, the neuron itself will generate a spike that rapidly spreads along the axons, transmitting to other neurons through the synapses.

Simulation of a spike neural network involves rebuilding the electric process described above in a computer [14, 32] to complete some computational functions. A block diagram of a spike neural network is shown in Figure 2.

The simulation process is composed of the behavior of many neurons. Simulation of a single neuron consists of two parts, spike generation and spike propagation. In addition to

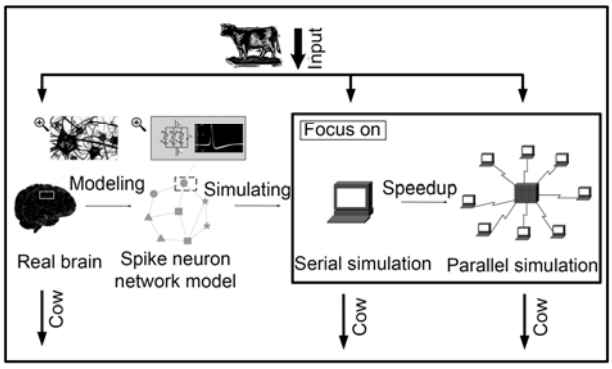

Figure 2 Block diagram of spike neural network simulation.

recording the state of neurons and synapses, the simulation system needs to maintain an event queue to record the spike events that will take place. At each simulation step, the event handler selects the earliest spike from the queue, updates the neuron state, and propagates the spike. Figure 3 shows a block diagram of the event-driven simulation code.

Compared with that on a PC, parallel simulation needs to solve two additional problems: (1) how many processors to use; (2) how to divide the task and the data.

The existing parallel neural network simulators task this work to the user, such as in NEURON [1, 2] and GENESIS [3, 4], but the user may be not familiar with this field, and so this could present a burden. The purpose of this study is to overcome the problem of neural network researchers not understanding parallel computing sufficiently well. Figure 4 shows the difference between our simulator and traditional systems. In a traditional parallel simulator, the number of processors P and the load balance must be assigned by the user, so that the parallel computing performance is limited by the user's experience. In this paper, we present a simulator that can automatically assign these parameters using an optimal processor number estimating algorithm and an intelligent load balancing algorithm; thus, the user does not even need to know that the parallel system exists.

For the convenience of the following discussion, we first give some definitions and symbols.

Definition 1 (speedup). The speedup $S(P)$ of a parallel simulator is defined as

$$S(P)=\frac{t_S}{t_P}, \tag{1}$$

where P is the number of parallel program execution processes, t_s is the simulation time when a single processor is used, and t_p is the simulation time when P processors are used.

```
1: WHILE (t<TotalTime)
2:    e=POP(Queue)
3:    e.iSpike(e.id)
4:    FOR (neuron n neighbored with e.id){
5:       updataSpike(n)
6:       insertSpike(n.spike)
7:    END FOR
8: END While
```

Figure 3 Main task of the event-driven simulation of spike neural network.

Definition 2 (computation communication factor). The Computation Communication Factor λ is defined as

$$\lambda=\frac{K(t_U+t_A+t_Q)}{t_C}, \tag{2}$$

where K is the average number of synaptic connections, and t_U, t_A, t_Q are time required to update the neuron state, time required for a spike event, time required to update the spike queue respectively. It can be seen that λ is related to the neuron model and computing system, but not to the processor number.

Definition 3 (P-way partition). A P-way partition $Q(N, P)$ of the set $\{1,\cdots, N\}$ denotes its division into P parts, each containing the same number of elements.

Definition 4 (impartiality rate). For a graph $G(V, E)$ and G's P-way partition $Q(N, P)$ (where N is the number of nodes in G), the Impartiality Rate IR is defined as

$$IR(Q,G)=\sum_{i=1}^{N}\left(\max_{1\le j\le P} g(i,j)\Big/\sum_{j=1}^{P} g(i,j)\right), \tag{3}$$

where $g(i, j)$ is the number of vertices in the set $\{k|(i, k)\in E, k\in j\text{th part of } Q(N, P)\}$

Table 1 gives a complete list of symbols and notation.

We now discuss the main innovations of this paper, i.e., the estimation of the optimal number of processors and the intelligent load balancing mechanism.

3 Intelligent estimation method for the degree of parallelism

Because of communication and synchronization overheads, the performance of parallel simulation is not always improved by the use of more processors. Therefore, the first problem that a parallel simulator must resolve is the number of processors needed to achieve the optimal performance. In traditional simulators, this number is assigned by users ac-cording to their prior experience with the system at hand.

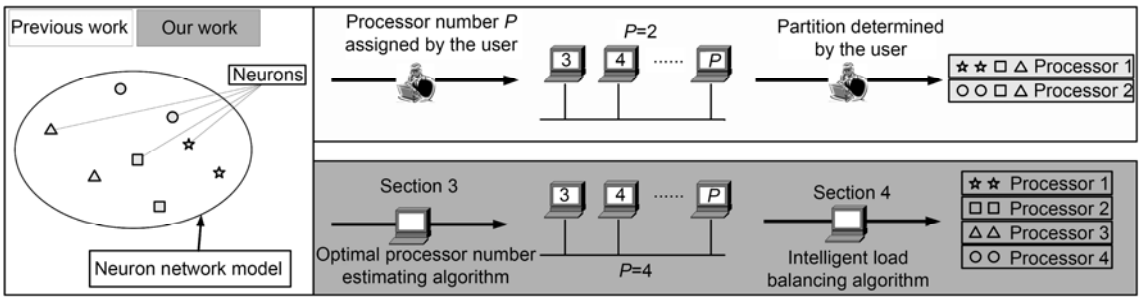

Figure 4 Workflow of traditional parallel systems and our proposed system.

Table 1 Notation used in this paper.

Symbol	Meaning
K	Average number of synaptic connections
$d(i)$	Number of postsynaptic neurons of the ith neuron
N	Number of neurons
M	Synapse number
F	Neuron spike rate
P	Number of processors
$S(P)$	Speedup of parallel programs when P processors are used
λ	Computation communication factor
t_U	Time required to update the neuron state
t_A	Time required for a spike event
t_Q	Time required to update the spike queue
t_C	Time required to compare and send spike-tick
t_S	Single-processor simulation time
$t_P(P)$	Parallel simulation time when P processors are used
$t_1(P)$	Computation time for parallel simulation
$t_2(P)$	Time required in parallel simulation for inter-processor communication
$Q(N,P)$	P-way partition of integers $\{1,\cdots, N\}$
$IR(Q,G)$	Impartiality rate of partition Q of graph G
$POST(i)$	Post-synaptic neuron collection of neuron i

This section presents an intelligent method that automatically determines the optimal number of processors, requiring no user intervention.

Theorem 1 (optimal parallel degree theorem). Parallel event-driven simulation of a spike neural network can achieve maximum speedup when the number of processors is equal to the computation communication factor λ.

Proof. In a parallel simulation, the total runtime can be written as

$$t_P(P) = t_1(P) + t_2(P), \tag{4}$$

where t_1 is the computational cost and t_2 is the communication cost. We will analyze the runtime in terms of these two parts. All the symbols used are listed in Table 1.

Firstly, let us consider the computational cost.

A neural network can generate $F\times N$ spikes in each biological time unit. This result is independent of the use of parallel simulation. The computation time for each spike interval is $K(t_U + t_A + t_Q)$ and the total computation time in each unit of biological time is

$$t_S = F \cdot N \cdot K(t_U + t_A + t_Q). \tag{5}$$

If these tasks are evenly assigned to P processors (uneven distribution will be discussed in the next section), each processor simply performs $1/P$ of the total tasks. The computation time is

$$t_1(P) = \frac{F \cdot N \cdot K(t_U + t_A + t_Q)}{P}. \tag{6}$$

Next, let us consider the communication part of the parallel simulation time.

When event-driven simulation is parallelized, each processor must communicate with others at every time step to determine which processor is responsible for the earliest spike event. Thus, $2P$ sending operations and $(P-1)$ comparing operations are required to implement this communication at each step. The overhead can be expressed as $\log(P)\cdot t_C$, i.e.,

$$t_2(P) = F \cdot N \cdot \log(P) \cdot t_C. \tag{7}$$

Therefore, the total parallel simulation time can be expressed as

$$\begin{aligned} t_P(P) &= t_1(P) + t_2(P) \\ &= \frac{F \cdot N \cdot K(t_U + t_A + t_Q)}{P} + F \cdot N \cdot \log(P) \cdot t_C. \end{aligned} \tag{8}$$

Substituting the expressions for t_S and t_P into our definition of speedup, we obtain

$$S(P) = \frac{t_S}{t_P} = \frac{1}{\log(P)/\lambda + 1/P}. \tag{9}$$

Because $S(P)$ is convex for a given λ, a maximum value exists. The derivative of $S(P)$ can be inferred from eq. (9) as follows:

$$\begin{aligned} S(P)' &= \left(\frac{1}{K \cdot \log(P)/\lambda + 1/P}\right) \\ &= \frac{-1}{(K \cdot \log(P)/\lambda + 1/P)^2}\left(\frac{1}{P \cdot \lambda} - \frac{1}{P^2}\right). \end{aligned} \tag{10}$$

Supposing P_{max} is the number of processors that maximizes $S(P)$, then P_{max} must satisfy $S(P)' = 0$, i.e.,

$$\frac{-1}{(K \cdot \log(P_{max})/\lambda + 1/P_{max})^2}\left(\frac{1}{\lambda P_{max}} - \frac{1}{P_{max}^2}\right) = 0. \tag{11}$$

The first term on the left-hand side of eq. (11) cannot be 0, so the only possibility is that the second term is 0.

As P_{max} is not 0, the equation can be simplified to

$$\left(\frac{1}{\lambda} - \frac{1}{P_{max}}\right) = 0. \tag{12}$$

Solving eq. (12), we obtain $P_{max}=\lambda$, i.e., parallel simulation achieves maximum speedup when the number of processors is λ, and the maximum speedup is then $\lambda/(1+\log(\lambda))$.

Theorem 1 implies that the parallel simulation of a spike neural network achieves optimal performance when the number of processors is equal to the computation communication factor λ. This means that the parallelized degree of a spike neural network simulation has a ceiling, and this ceiling is proportional to the computation communication factor. When the neural network model and the parallel computer have been determined, this ceiling is determined. The significance of the theorem is that it provides a guide-

line for designers to select an appropriate number of processors.

Figure 5 shows the change in speedup, according to eq. (9), with respect to the number of processors for different values of λ. From the figure, we can see that larger values of λ give a better parallel simulation performance.

Figure 6 shows the maximum speedup that can be achieved by the parallel simulation of a spike neural network when different λ values are given. From the figure, we can see that λ must be large enough to obtain a good performance from the parallel simulation, because linear speedup can only be attained when λ is considerably greater than the number of processors.

Theorem 1 implies that the closest integer to the computation communication factor λ is the optimal number of processors. Figure 7 shows a two-part method for measuring computation communication factor λ. The first part acquires the computational overhead of the neurons, and the second part finds the communication delay. The computational overhead is found by running a pre-defined neural network simulation on a single processor, and the communication delay is acquired using the Ping-Pong facility.

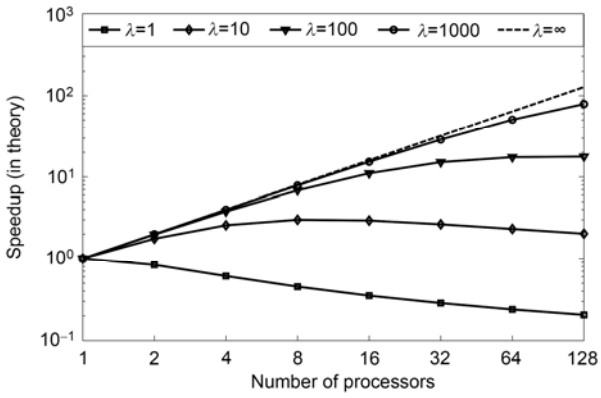

Figure 5 Speedup with respect to the number of processors for different values of λ.

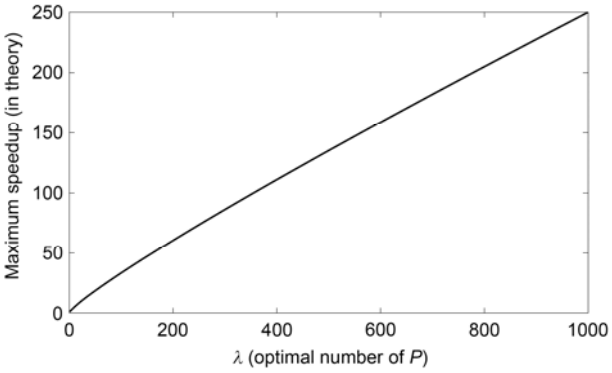

Figure 6 Optimal speedup with respect to λ.

1: **Input**: neuron model, average number of neighbors K
2: **Output**: computation communication factor λ
3: record runtime T_1 of neural network with n neurons on single processor
4: record the number of spikes N
5: record number of updating M.
6: record the communication time T_2 between processors
7: $\lambda = K \cdot T_1 / T_2 \cdot N / M$

Figure 7 Intelligent algorithm to estimate the optimal number of processors.

4 Intelligent load balancing mechanism

Load balancing is a key issue in parallel computing. If the loads between processors are not balanced, the performance of the simulator will suffer a sharp decline. In traditional neural network parallel simulators, loads must be partitioned by the user. If the user is unfamiliar with parallel computing, and wrongly partitions the task, then the system performance will be poor. This burdens the user, and does not make best use of the potential of parallel simulation. Figure 8 shows the situations in which the load is balanced and unbalanced. Figure 8(a) shows the event queue and part of the neural network used by this example, in which the arrows are from pre-synapse neurons to post-synapse neurons. Figures 8(b) and 8(c) show two different partitioning strategies, where the circles are labeled according to which neuron they are updating. Dark circles denote tasks to be computed during this time step and light-colored circles denote tasks to be done in a future step. Tasks are arranged in rows to represent the different processors, and the horizontal axis represents time. It can be seen from Figure 8(b) that the load is evenly distributed across all processors, i.e., the eight neurons updating tasks in each spike cycle are evenly distributed across the four processors. In Figure 8(c), however, the eight neurons updating tasks on each spike cycle are only assigned to two of the four processors, while the other two processors have no tasks assigned. Although four neurons updating tasks are initially assigned to each processor in Figures 8(b) and 8(c), the load becomes very unbalanced in Figure 8(c) when dynamic situation is considered.

Theorem 2 (optimal load balance theorem). In an event-driven parallel simulation of a spike neural network, the partition strategy with the lowest *IR* value will perform best, and the lower bound of *IR* is

$$\sum_{i=1}^{N} 1 / P, \tag{13}$$

where N is the total number of neurons, P is the number of processors.

Proof. Suppose that the function $g(i, j)$ returns the number of post-synaptic neurons in the jth processor of neuron i. The spike propagation and updating cost will then be proportional to

$$\left(\max_j g(i, j)\right) \Big/ \sum_{j=1}^{P} g(i, j). \tag{14}$$

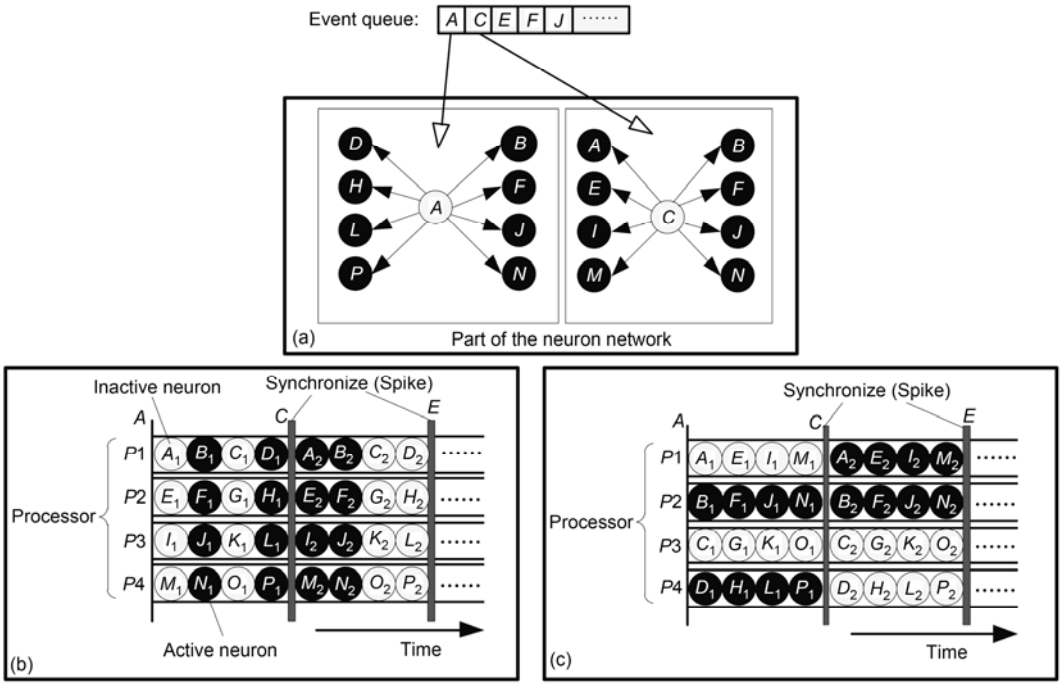

Figure 8 Key issues in the load balance mechanism. (a) Snapshot of the event queue, as well as part of the neural network; (b) a perfect load balancing strategy; (c) an imbalanced load strategy. Dark circles denote active tasks, light circles denote inactive tasks.

Thus, the total simulation time can be expressed by the following formula:

$$t_{\mathrm{P}}(P) = t_1(P) + t_2(P) = t_{\mathrm{S}}\left(\max_j g(i,j)\right) \Big/ \sum_{j=1}^{P} g(i,j) + t_2(P), \tag{15}$$

where t_2 is the cost of communication related to the number of processors in the event-driven model. Therefore, t_{p} is minimized when $\sum_{i=1}^{N}(\max_j g(i,j)) / \sum_{j=1}^{P} g(i,j)$ attains its minimum value.

For any partition Q of graph G, the lower limit of $(\max_j g(i,j)) / \sum_{j=1}^{P} g(i,j)$ is $1/P$. This can be inferred from $\sum_{j=1}^{P}(g(i,j) / \sum_{j=1}^{P} g(i,j))$ and $g(i, j)>1$ for any i and j. The minimum value can only be reached when $g(i, j) = d(i)/P$. Therefore, for any partition Q of G, the inequality $IR(Q) \geqslant \sum_{i=1}^{N} 1/P$ is true.

Because the number of possible partitions suffers a 'combinatorial explosion', finding the partition with the smallest *IR* value is quite difficult. Let us assume that N is the number of neurons, P is the number of processors, and Q is the number of neurons assigned to each processor, i.e., N/P. Therefore, there are $C(N, Q)$ choices for the first partition, $C(N–Q, Q)$ choices for the second partition, ⋯, and $C(Q, Q)$ for the last partition. The rank of all partitions should not be counted, i.e., we should divide by $P!$ Thus, the total number of possible partitions is

$$\frac{1}{P!}\binom{N}{Q}\binom{N-Q}{Q}\cdots\binom{Q}{Q}. \tag{16}$$

The number of feasible solutions is so large that it is impossible to find the optimal solution in the limited time. For example, when N=100, P=4, the number of possible choices is 6.72×10^{55}. Therefore, we present a heuristic algorithm to solve this optimization problem, named A-A algorithm. The principle of the A-A algorithm is shown in Figure 9. Consider the set of vertices to be divided into two parts initially, i.e., set A and set B are separate. Firstly, identify a subset A_1 in set A and a subset B_1 in set B. A new partition $[A', B']$ is then obtained by exchanging subset A_1 with B_1, and the value of $IR([A', B'])$ is less than $IR([A, B])$. We can implement this idea using a function $\Delta IR(v)$, which takes a vertex as input and returns the gain in *IR*. It can be expressed as follows:

$$\Delta IR(v) = \sum_{v_i \in preset(v)} \Delta IR(v, v_i), \tag{17}$$

where $preset(v)$ means the set of all vertices connected by an edge to vertex v. The function $\Delta IR(v, v_i)$ can be described as follows. Suppose there is an edge from w to v, and that the vertices with edges from w are distributed across the partitions according to $k_1, k_2, \cdots, k_n$. Vertex v is moved from partition s to partition t, such that the distribution becomes

$k_1, k_2, \cdots, k_s-1, \cdots, k_t+1, \cdots, k_n$. $\Delta IR(v, w)$ is then computed as

$$\Delta IR(V,W)=\begin{cases}1/d(W), & k_t = k_{\max},\\ -1/d(W), & k_s = k_{\max} \,\&\, k_t < k_{\max}-1,\\ 0, & \text{otherwise},\end{cases} \quad (18)$$

where $k_{\max}$ is the maximum of $k_1, k_2, \cdots, k_n$.

The main idea of the A-A algorithm is to start at an arbitrary partition, and then reduce the value of *IR* gradually by continuously exchanging vertices. When the value of *IR* cannot be reduced by an exchange of adjacent vertices, a vertex pair is selected at random for exchange. This process continues until some end condition is met.

Figure 10 shows the pseudocode of the intelligent load balancing A-A algorithm, where the function *RandomPartition* returns a random partition of the input graph *G*. The gain from exchanging a vertex pair is computed by eq. (17) on Line 5 of Figure 10. Because load balancing is a nondeterministic polynomial (NP) problem, the End condition on Line 7 can have several choices, e.g., that the total *IR(G)* cannot be improved any more. The function *maxgainVP* returns the vertex pair whose exchange gives the maximal gain and the function *updatinggain* updates the gain of all vertex pairs after the vertex exchange.

5 Experiments

In order to verify the analysis model described above, we perform the following experiments on the Tianhe-1A parallel computer [33], which includes two 2.93 G Intel Xeon X5670 CPUs and 24 GB of RAM on each node. The simulation is a 4×10^6 IF neuron network using the model Smith et al. [34].

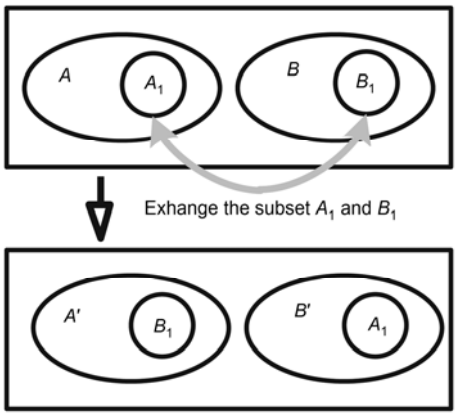

Figure 9 Diagram of the intelligent load balancing algorithm.

```
1: INPUT: Graph G for the neural network tasks, number of processors P
2: OUTPUT: A P-way partition of G, named Q, with IR(Q) lower enough
3: Q=RandomPartition(G, P);
4: For (Each vertex pair){
5:   computing the gain According to eq. (17)
6: }//end For
7: While (End condition){
8:   maxgainVP()
9:   updatinggain()
10: }//end While
```

Figure 10 Pseudocode for the intelligent load balancing algorithm.

An object-oriented method, whereby each neuron is treated as an object, is used to implement our parallel simulator. A model can be added to the simulator by adding a new neuron class under some predefined interface. The relations between neurons can then be configured in two ways. One is via a relation function, which takes two neuron IDs as input and outputs their relationship. The second method uses a configuration file, in which all the neuron pairs and their connection are listed. When a new neural network model is needed, the user needs only to define the new neuron relationship. Other elements of the system, such as parameter computing, completion of other work needed in the parallel simulation, parameter calculation, and the division of tasks, are completed automatically by the simulator.

We use a face detection application as the benchmark for performance analysis. The computation communication factor λ is estimated by the method in Figure 7. The values of t_C, t_U, t_A, and t_Q measured under the IF model on the Tianhe-1A platform are

$$t_C = 1.15\ \mu s,\ t_U + t_A + t_Q = 0.12\ \mu s.$$

We compare results with values of $K = 80$ and $K = 400$.

In order to verify the performance analysis model for parallel simulation, the computation and communication runtimes are examined for different numbers of processors. The histograms in Figures 11(a) and 11(c) illustrate the runtime for parallel simulation with different numbers of processors. In each bar-group, the left bar is the runtime when the A-A intelligent load balancing strategy is used, and the right bar is the runtime when a random load balancing strategy is used. The curves in Figures 11(b) and11(d) show the speedup factor due to parallel simulation with different numbers of processors when two load balancing algorithms are used. The theoretical speedup value from eq. (9) is also plotted.

It can be seen from Figures 11(b) and 11(d) that eq. (9) accurately describes the actual speedup when the communication cost is small compared with the computation cost. Once communication accounts for the vast majority of simulation time, the actual performance curve gradually deviates from the theoretical one. This is caused by complex communication behavior when the number of processors is large. More deviation can be seen in Figure 11 (b) than in Figure 11(d) for the posterior one with larger λ. We can also see from Figure 12 that, when the number of processors is less than λ, the performance is significantly improved, but when the number of processors is much larger than λ, there is little improvement in performance. This is because the load balancing algorithm improves the balance of the computing tasks, but when the number of processors is much

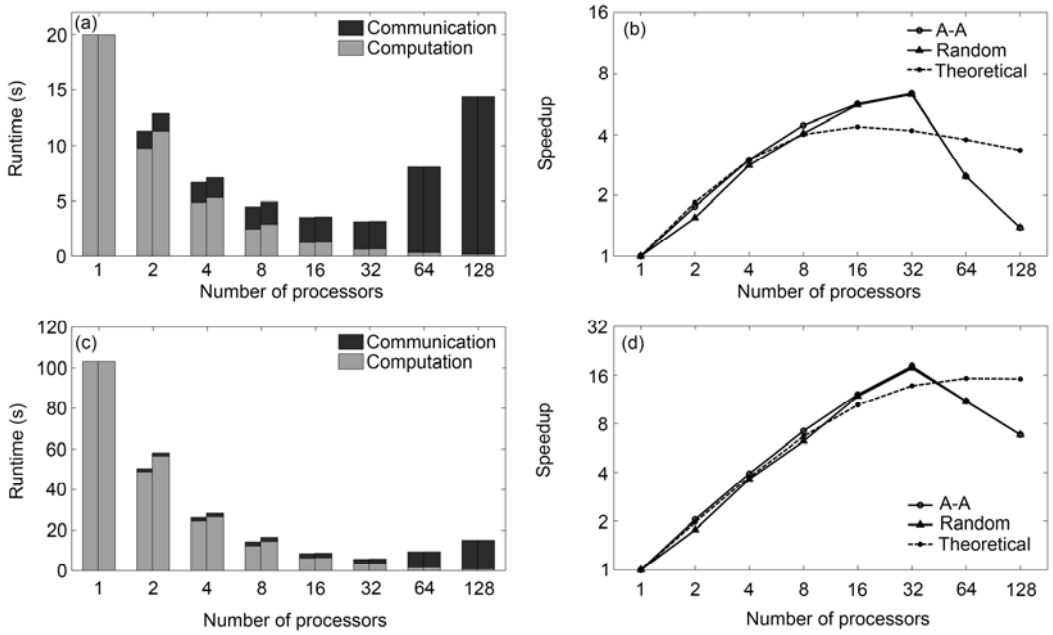

Figure 11 Runtime and speedup factor of parallel simulation with different numbers of processors. In (a) and (c), the left-hand bar for each processor number denotes the runtime with the A-A intelligent load balancing strategy and the right-hand bar denotes the runtime with a random load balancing strategy. (a) Runtime with K = 80, λ=16.68; (b) speedup with K = 80, λ=16.68; (c) runtime with K = 400, λ=83.48; (d) speedup with K = 400, λ=83.48.

larger than λ, the communication time accounts for the majority of the whole simulation time, and so any load imbalance has less impact on the performance.

6 Conclusions

The simulation of event-driven spike neural networks is an important tool for neural computing science. By parallelizing such networks, we can greatly increase the size, accuracy, and speed of simulation. In this paper, we first presented a performance model for parallelized event-driven simulation. Based on this model, we proved two theorems related to the design of an optimal simulator. Consequently, an intelligent algorithm was designed to assign the optimal number of processes and partition the simulation tasks. Under this framework, the user does not require any knowledge of parallel computing technology, and can use our parallel simulator as simply as a PC.

This work was supported by the National Natural Science Foundation of China (Grant Nos. 61003082, 60921062, 61005077).

1 Goddard N H, Hood G. Parallel Genesis for Large-scale Modeling. New York: Plenum Press, 1997
2 Bower J M, Beeman D, Wylde A M. The book of GENESIS: Exploring Realistic Neural Models with the GEneral NEural SImulation System. New York: Springer-Verlag, 1998
3 Migliore M. Parallel network simulations with NEURON. J Comput Neurosci, 2006, 21(2): 119–129
4 Carnevale N T, Hines M L. The NEURON Book. Cambridge: Cambridge University Press, 2006
5 Delorme A. SpikeNET: A simulator for modeling large networks of integrate and fire neurons. Neurocomputing, 1999, 26: 989–996
6 Davison A P. PyNN: a common interface for neuronal network simulators. Front Neuroinf, 2008, 2: 1858–1876
7 Goodman D, Brette R. Brian: a simulator for spiking neural networks in Python. Front Neuroinf, 2008, 2: 138–147
8 Goodman D F, Brette R. The brian simulator. Front Neurosci, 2009, 3(2): 192–201
9 Gewaltig M O, Diesmann M. NEST (neural simulation tool). Scholarpedia, 2007, 2(4): 1430–1437
10 Eppler J M. PyNEST: a convenient interface to the NEST simulator. Frontiers Neuroinf, 2008. 2: 372–388
11 Alamdari A S. Biological Neural Networks (BNNs) Toolbox for MATLAB: User Guide. General Public License, 2004
12 Markram H. The blue brain project. Nature Rev Neurosci, 2006, 7(2): 153–160
13 Meier K. The FACETS Project. Wikipedia, 2005, 3(3): 202–215
14 Brette R. Simulation of networks of spiking neurons: A review of tools and strategies. J Comput Neurosci, 2007, 23(3): 349–398
15 Buyya, R. High Performance Cluster Computing: Architectures and Systems, vol 1. Upper SaddleRiver, NJ: Prentice Hall, 1999
16 Strey A. A comparison of OpenMP and MPI for neural network simulations on a SunFire 6800. Adv Parallel Comput, 2004. 13: 201–208
17 Morrison A. Advancing the boundaries of high-connectivity network simulation with distributed computing. Neural Comput, 2005, 17(8): 1776–1801
18 Nageswaran J M, Dutt N. A configurable simulation environment for the efficient simulation of large-scale spiking neural networks on graphics processors. Neural Networks, 2009, 22(5): 791–800
19 Cheung K, Schultz S R, Leong P H. A parallel spiking neural network simulator. International Conference on Field-Programmable Technology. Sydney: ACM, 2009. 78–91
20 Hennessy J L, Patterson D A, Goldberg D. Computer Architecture: A Quantitative Approach. San Francisco: Morgan Kaufmann. 2003
21 Pacheco P, Camperi M, Uchino T. Parallel neurosys: A system for the simulation of very large networks of biologically accurate neurons on parallel computers. Neurocomputing, 2000, 32: 1095–1102

22 Khan M M. SpiNNaker: mapping neural networks onto a massively-parallel chip multiprocessor. IEEE International Joint Conference on Neural Networks. Hong Kong: IEEE, 2008. 2849–2855
23 Pecevski D, Natschl T, Schuch K. PCSIM: a parallel simulation environment for neural circuits fully integrated with Python. Front Neuroinf, 2009, 3: 113–124
24 Plesser H. Efficient parallel simulation of large-scale neuronal networks on clusters of multiprocessor computers. European Conference on Parallel and Distributed Computing. Rennes: IRISA, 2007. 672–681
25 Wilson E C, Goodman P H, Harris F C. Implementation of a biologically realistic parallel neocortical-neural network simulator. Proceedings of the 10th SIAM Conference on Parallel Process for Scientific Computing. Philadelphia: SIAM, 2001. 12–23
26 Nageswaran J M. Efficient simulation of large-scale spiking neural networks using CUDA graphics processors. International Joint Conference on Neural Networks. Atlanta: IEEE, 2009. 287–295
27 Hines M L, Eichner H, Schurmann F, Neuron splitting in compute-bound parallel network simulations enables runtime scaling with twice as many processors. J Comput Neurosci, 2008, 25(1): 203–210
28 Lobb C J. Parallel event-driven neural network simulations using the Hodgkin-Huxley neuron model. Workshop on Principles of Advanced and Distributed Simulation. Monterey: IEEE, 2005. 102–108
29 Mohraz K, Schott U, Pauly M. Parallel simulation of pulse-coded neural networks. IMACS World Congress on Scientific Computation, Modelling and Applied Mathematics. Berlin: IEEE, 1997. 23–28
30 Izhikevich E M. Simple model of spiking neurons. IEEE T Neural Network, 2004, 14(6): 1569–1572
31 Vreeken J. Spiking neural networks, an introduction. Institute for Information and Computing Sciences, Utrecht University Technical Report UU-CS-2003-008, 2002
32 Gerstner W, Kistler W M. Spiking neuron models. vol 15. Cambridge, UK: Cambridge University Press, 2002
33 Yang X J. The TianHe-1A supercomputer: its hardware and software. J Comput Sci Technol, 2011. 26(3): 344–351
34 Brette R. Exact simulation of integrate-and-fire models with exponential currents. Neural Comput, 2007, 19(10): 2604–2609