Hindawi
Scientific Programming
Volume 2022, Article ID 3886729, 13 pages
https://doi.org/10.1155/2022/3886729

Research Article

DSP, a Plug-and-Play Process-Based Distributed Algorithm Simulation Platform

Yifan Wang, Qianchuan Zhao, Hu Yan, and Wen Yang

Department of Automation, BNRist, Tsinghua University, Beijing 100084, China

Correspondence should be addressed to Qianchuan Zhao; zhaoqc@tsinghua.edu.cn

Received 19 October 2021; Accepted 26 January 2022; Published 10 February 2022

Academic Editor: Basilio B. Fraguela

In this paper, we propose DSP (Distributed algorithm Simulation Platform), a novel process-based distributed algorithm simulation platform to simulate real distributed systems for the design and verification of distributed algorithms. DSP consists of computer processes, and each process simulates an individual distributed node. A DSP process is mainly composed of a communication module, a computation module, an internal storage module, and an external interaction module. DSP is a flexible, versatile, and scalable simulation platform. It supports the testing of applications in various fields. Small-scale experiments can be done with a single personal computer, while large-scale experiments can be carried out through cloud servers. The greatest highlight of DSP is that it is plug and play, where nodes can be freely added or deleted during the simulation process. DSP is now open-sourced on GitHub, https://github.com/Wales-Wyf/Distributed-Algorithm-Simulation-Platform-DSP--2.0.

1. Introduction

With the rise of the Internet of Things, distributed systems are playing an increasingly important role in various areas, such as building systems [1] and communication systems [2]. Distributed simulations have attracted considerable attention, and research has proliferated from the 1970s to now. Distributed simulation can be classified as discrete-event simulation (DES) [3], real-time simulation [4], agent-based simulation (ABS) [5], hybrid simulation [6], etc. DES focuses on using parallel computing techniques to accelerate the execution of a discrete-event simulation program across multiple computers [7]. A DES simulator often uses conservative and optimistic approaches to manage time for synchronization. The most typical work is high-performance computing (HPC), using methods like GPU-clusters to substantially speed up the simulation [8]. Instead of considering time management, a real-time simulation concentrates on simulating a real-time progress or a system in the real world [9]. Among real-time simulators, High-Level Architecture is a representative standard to assemble distributed resources for large simulations and support the interoperation of distributed components. In HLA, simulation federates, animation federates, and other federates are connected by Runtime Infrastructure (RTI) to construct a federation for simulation [10]. Real-time simulators can also be used to analyze a real system [4]. ABS is based on the Agent-Based Model (ABM) [11], which describes the modeling and simulation of multiagent systems. ABS is composed of a number of agents and can simulate the individual actions of agents and interactions between agents. The most common distributed agent-based simulators are designed based on MPI (Message Passing Interface) [12]. A hybrid simulator is often composed of different categories of simulators mentioned above. As a result, some other researchers focus on how to combine several different types of simulators. One famous work is the Functional Mockup Interface (FMI), which delivers a tool-independent standard for model exchange and cosimulation [13].

Distributed simulation is essential and necessary for studying distributed systems for the following three reasons. Firstly, many real systems have not been built during the research process. Distributed applications need a virtual environment to be created and tested before its implementation. Secondly, a simulation platform is often cheap to establish and has high fault tolerance to avoid "expensive"

hardware damages. Thirdly, the scale expansion of a virtual environment is much easier than a real one.

In this paper, we mainly concentrate on simulating real distributed systems as much as possible for the design and verification of distributed algorithms. The origin of designing our simulator is to simulate a distributed building control system [1] before it is established. In this system, a building is divided into space units and intelligent devices. Each of them is controlled by a smart node called a Computing Process Node (CPN), and all these nodes form a distributed centerless network. The application development needs to be node-based and possible accidents in buildings should be supported in the simulator. According to the ideas illustrated in [14] and our own thinking, we chose ABS as our basic architecture for the following reasons:

(1) The problem has a natural representation as agents
(2) The interaction among agents is frequent and agents anticipate other agents' strategy decisions
(3) The agents can interact with users during the simulation process
(4) The topology of the multiagent network can be time-varying
(5) The synchronous and asynchronous communication can be supported between agents

The most frequently considered features of distributed simulation in the current work is versatility, scalability, accuracy, privacy, efficiency, etc. On the basis of our concerns, flexibility is taken as a practical consideration into account, which consists of three parts: (i) topology flexibility: the network topology can be generated arbitrarily and changed during the simulation process; (ii) algorithm flexibility: both synchronous and asynchronous complex algorithms can be supported; and (iii) fault simulation: scenes with problems such as communication delay, link failures, and node failures can be simulated.

The design of most distributed simulators can be summarized into three main components [5]: (i) simulation model: the architecture of a simulator to implement the simulation activities, containing computation, data storage, data communication, etc.; (ii) platform mechanisms: the mechanisms for users to simulate scenarios, including instantiating nodes through simulation model, establishing the overall topology, and expanding the simulation scale; and (iii) application implementation: the methods for users to implement distributed applications. Some simulators support only simple algorithms, while others support complex applications.

We first consider the MPI-based simulators. MPI-SIM is an early parallel simulator for MPI program simulation [15]. SimGrid is a popular distributed simulator for large-scale grid computing [5]. It mainly focuses on abstract distributed computing with its S4U interface, and MPI applications can be simulated through its SMPI interface. D-MASON is a distributed framework that can generate multiple objects based on the MASON library [16]. MPI-based simulators often use processes as simulation core and interprocess communication to transfer data. In this case, the network topology is actually fully connected where any two nodes can communicate. These simulators are versatile for diverse MPI applications and locally scalable with a high-performance computer. However, since interprocess communication is difficult to extend to multiple machines, cloud expansion is always not supported, and the topology is fixed during the simulation process.

Besides MPI-based simulators, many distributed simulation platforms are designed for research in specific fields. Christian et al. designed a distributed traffic simulation environment of cooperatively interacting vehicles [17]. Perkonigg et al. proposed MAC-Sim, an agent-based simulation for power grid analysis in the field of energy [18]. Felix delivered a distributed augmented reality simulation environment for medical training and implementing augmented reality applications [19]. These simulators are designed according to the corresponding domain knowledge and have different simulation models. Most of them have the problem of not being applied universally.

There are also other distributed simulation environments based on other technologies. Applying cloud computing, Rodrigo et al. delivered CloudSim [20], a framework that provides a simulation environment supporting cloud services. Alberto proposed a locally scalable P2P simulator, PeerSim [21], to run simple algorithms considering peer-to-peer computing.

A common limitation of current simulators is that the network topology is unchangeable during the simulation process. A scenario in which nodes join or leave the network cannot be simulated. Another problem is that only synchronous distributed algorithms can be implemented, and complex methods are usually unavailable in large-scale situations. Furthermore, majority of them cannot simulate delays or failures that happened in real systems.

Aware of these shortcomings, we present the DSP (Distributed algorithm Simulation Platform), a novel process-based distributed algorithm simulation platform. Each process simulates an individual distributed node. DSP has a server-based communication module, a dynamic topology management mechanism, and a complete design pattern with inline functions for distributed algorithms to solve the above problems. The comparison of the properties we pay attention to among the above simulation platforms is shown in Table 1.

1.1. Contributions. DSP differs from the previous distributed simulators according to its flexibility. DSP is also versatile and scalable.

(1) Flexibility: in each process, socket server threads are used for one-on-one communication, and a routing table is used to manage the neighbor relationship. Under these circumstances, a plug-and-play network topology can be implemented. Users can design custom distributed methods using sync/async communication functions. Delays and failures can be simulated on this platform.

Table 1: Comparison of distributed simulators.

	Versatility	Scalability		Flexibility			
		Local	Cloud	Plug and play	Async methods	Delay	Failure
MPI-SIM [12]	✓	✓					
SimGrid [3]	✓	✓				✓	
D-MASON [13]	✓	✓					
Traffic sim [14]			✓			✓	
MAC-sim [5]						✓	✓
MT sim [15]						✓	
CloudSim [16]	✓		✓				
PeerSim [17]	✓	✓					
DSP	✓	✓	✓	✓	✓	✓	✓

(2) Versatility: each process can be initialized to simulate nodes in any field by configuring parameters according to the user's demand.

(3) Scalability: each process is lightweight, and the node communication between different servers is convenient. By improving the computer performance or utilizing cloud servers, the simulation scale can be extended to a large scale, even for a complex distributed method.

2. Distributed Algorithm Simulation Platform (DSP)

In this section, the main structure, working mechanism, and function library are introduced. DSP is a process-based platform to simulate real distributed systems for developing and verifying distributed methods. The simulation object of DSP is a real distributed system composed of smart nodes. Each node is composed of an independent processor, a storage unit, and one-to-one communication interfaces. Each node can only communicate with its neighbors.

2.1. Structure of DSP. In this part, the structure of DSP is illustrated. DSP has a completely distributed framework comprised of computer processes. Each process has no global information and can only communicate with its neighborhoods. Each process can accept instructions from users and return the results of tasks. Figure 1 shows the simulation model of DSP.

2.1.1. Simulation Core: Process (Node). Nodes are simulated by computer processes that are basically composed of four modules: communication module, computation module, internal storage module, and external interaction module. Each node process has seven TCP socket servers, a data storage space, and a task thread trigger. Through importing topology information and configuring parameters, a node is instantiated with an ID as a unique identifier, an IP as a local address, a time-varying routing table to conserve communication information, and a data list to store data.

This design makes DSP versatile. Since a process is universal, nodes in DSP can simulate agents in different areas according to user demand with diverse imported data. Specifically, with the information of temperature, humidity, and number of people imported, a smart node to control a building area can be simulated. Similarly, distributed algorithms and applications in different fields can be researched and implemented in DSP, such as human evacuation methods and lighting control methods.

Furthermore, DSP is locally scalable and economical. A process occupies only a small number of resources in a computer. Hundreds of nodes can be simulated in a single cheap server. As a result, many experiments can be finished with only one computer, such as solving the distributed generation control problem in the power system. By utilizing a high-performance server, simulation nodes can grow to thousands or more.

2.1.2. Communication Module: TCP Socket Servers (Edge). The communication module comprises 6 TCP socket servers as communication channels for each node to transfer data with its neighbors. Each server is bound with a port. When two nodes are neighbors, they will both select an unused server to establish a one-to-one correspondence. During the communication process, one node will send the data to the corresponding server of its neighbors. This connection will hold unless a node/link failure happens or the network topology is changed. This design aims to imitate the hardware connection in a real system. A routing table is maintained to manage its neighbors' information in each process for dynamic topology management. Each element in the table conserves the information of one neighbor of the node. The routing table will be updated when its local topology is changed. Considering each node cannot hold too many hardware connections in a natural system, now we suppose the maximum number of neighbors of a node is 6. In fact, the number can be changed if we add the number of communication servers.

This design supports a general and flexible network topology in DSP. Users can create an arbitrary topology by configuring the routing table in the initialization part. Furthermore, when a node is detected to be deleted in the network, its neighbors can automatically adjust the routing table and break the connection of their corresponding communication servers. Similarly, a newly added node will only influence the routing tables of its neighbors. These events can even occur during simulations. These operations can be done at a small cost since they are local behaviors. As a result, a plug-and-play simulation environment is

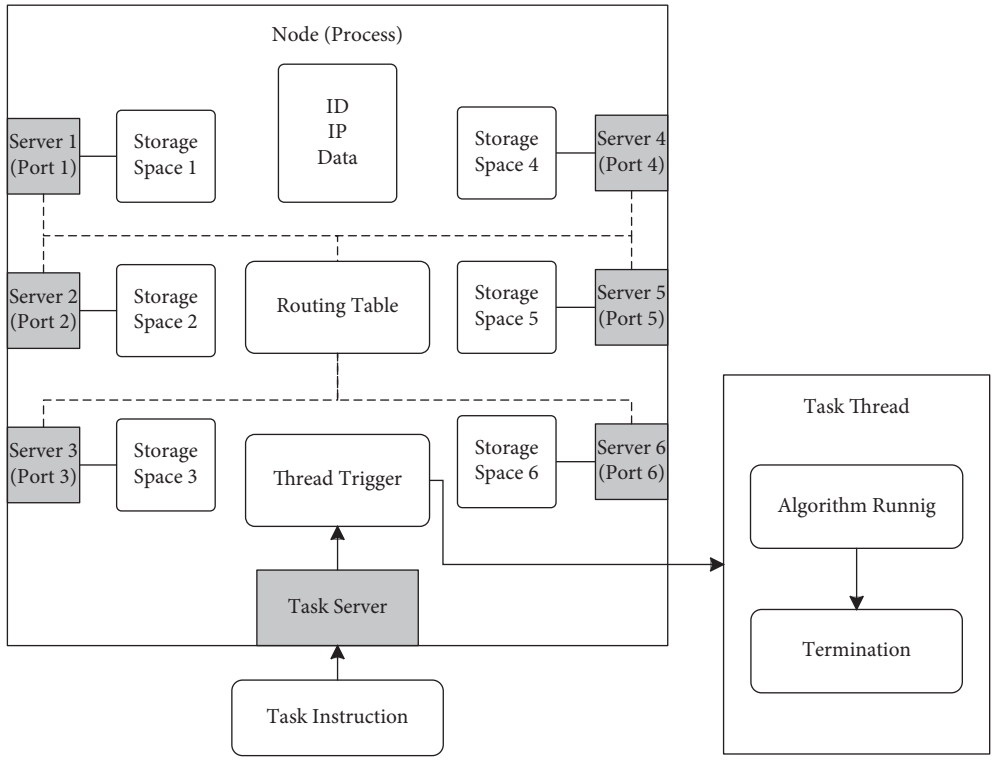

Figure 1: Simulation model of DSP.

implemented. By the way, the socket servers make randomized delays available, which provides an environment for the operation of distributed asynchronous algorithms.

Moreover, the scale of DSP can be expanded on a large scale through cloud servers. The socket server mechanism makes the node communication between different servers with different operating systems utterly consistent with that of one computer. Through leasing or buying cloud servers, large-scale experiments to execute complex applications are available.

2.1.3. Computation Module: Thread Trigger. The computation module works as a thread trigger. The user-designed algorithms will be imported into each node. When a task execution instruction is received, it will be broadcast to all network nodes. Then the computation module in each node will start a new thread to run the corresponding preloaded algorithm. The thread will be alive until the algorithm stops. During the running process, a node will utilize the data transmitted by its neighbors to compute and send data to them. The thread trigger mode makes the multitask become achievable, which is our future work.

2.1.4. Internal Storage Module. The internal storage module is divided into two parts: process storage and database storage. The process storage conserves node information, topology information, temporary operation information, and neighbors' data. The neighbors' data storage space is divided into six parts, and the data in each part is updated based on the messages from the corresponding communication server. The database storage stores the running data generated by the imported algorithms and the debug information.

2.1.5. External Interaction Module. The external interaction module contains a socket server to communicate with the outside world as a task server. Users can send external task execution signals to nodes through task servers. After finishing the calculation, the results will be returned to the user.

2.1.6. Overall Architecture. From the structure described above, DSP is shown to be flexible: supporting plug-and-play networks, self-designed applications, and simulation of delays and failures; versatile: supporting multi-domain simulations; and scalable: improving the computer performance for local scalability and utilizing the cloud servers for cloud scalability.

In summary, the overall simulation architecture of DSP is shown in Figure 2, which is divided into three layers: cloud, server, and nodes. The three layers show an inclusive relationship.

Combined with the overall architecture, we consider the efficiency of DSP by dividing it into three parts: configuration, computation, and communication. DSP has high configuration efficiency. The whole configuration process can be regarded as arranging building blocks. Each block has the same model, but different topology and data information can be imported through initialization. Users need only a small amount of work to build a large network because of the universality of the blocks. The computation efficiency is

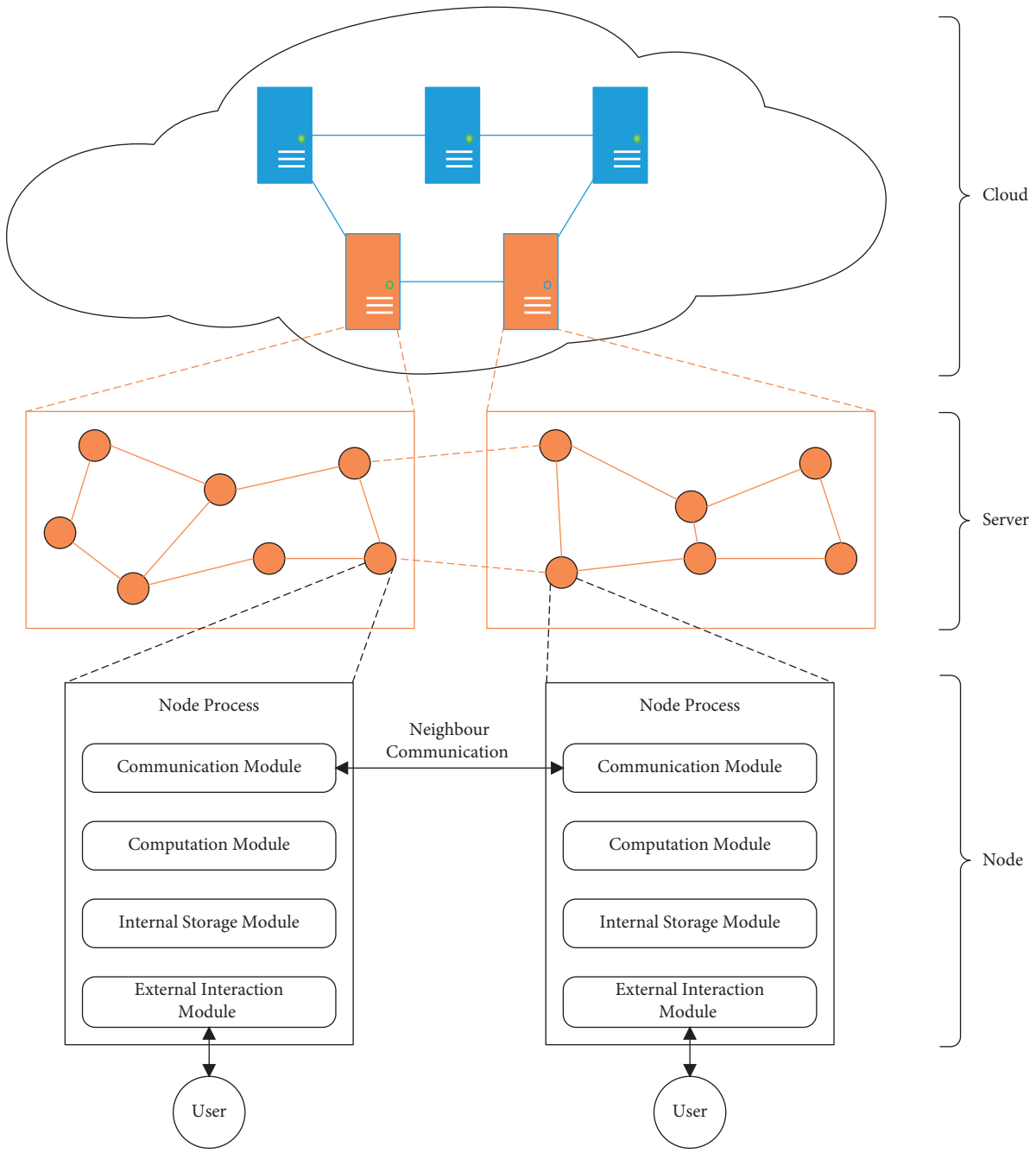

Figure 2: Overall architecture of DSP.

related to the number of running nodes and the server status. The resource allocation and scheduling of the processes are controlled by the operating system. At the beginning of the simulation, the OS will assign CPU cores and memory to all processes. During the simulation process, idle computer resources will be utilized for computation. Since many complicated applications are developed and tested in DSP, the most significant factor affecting the computation efficiency is often the algorithm complexity. The communication efficiency may be low due to the choice of socket server, because socket communication is slower than process communication and memory communication. However, this communication mechanism is necessary for one-on-one communication, the dynamic topology management mechanism, and cloud scalability. In fact, the synchronization process in large-scale agent-based networks costs much more time than the communication time, and the asynchronous algorithms run fast, which will be shown in the Experiments section.

2.2. Working of DSP. In this part, we discuss the main working mechanism of DSP. The workflow is shown in Figure 3.

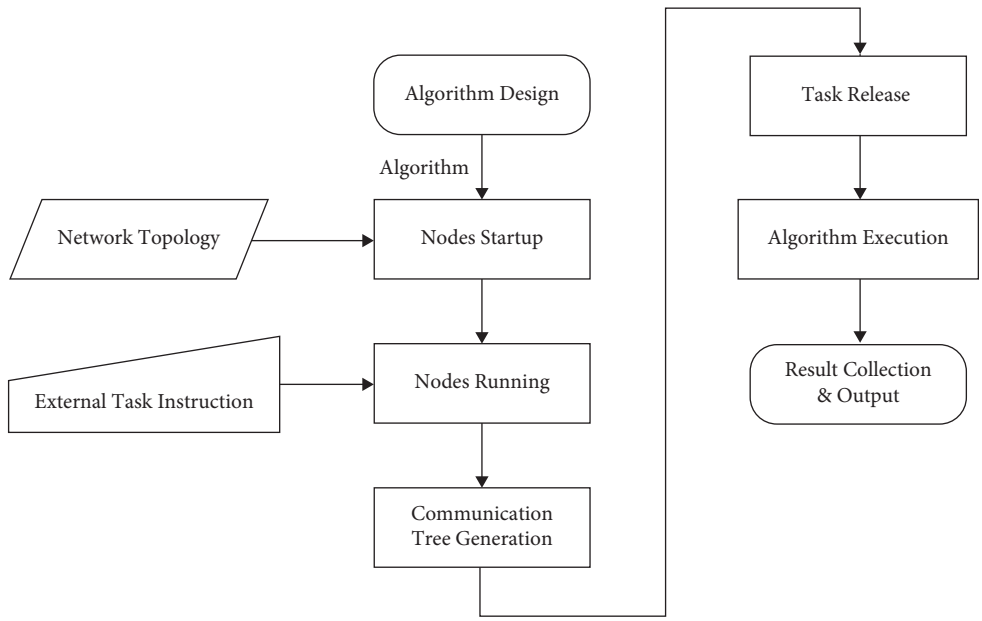

Figure 3: Workflow of DSP.

2.2.1. Design of Distributed Algorithm. DSP provides a Python template for users to design distributed algorithms. Python libraries are available, and several inline functions are offered, containing communication and debugging functions. The communication functions contain synchronous and asynchronous communication functions, so the design of synchronous and asynchronous algorithms can be supported. The debugging function lets users print the output or save the process variable information into the database. The code running on each node is required to be consistent.

2.2.2. Initialization of DSP. The initialization consists of two parts: topology creation and algorithm upload. Firstly, a topology list of JSON formats will be generated according to the network topology. The list contains the topology information of each node, including ID, IP, ports for communication servers and task servers, neighbors' information, and an initialized data list. Secondly, designed algorithms will be uploaded in advance. With the topology and algorithm information, the operating system will allocate computer resources to all nodes and start them. Then each node will find its neighbors through its local topology.

2.2.3. Task Release and Execution. When all nodes start, when the task server of one node receives a task execution instruction from a user, the calculation process will be started. The basic process is divided into four steps:

Step 1: the chosen node will be selected as the root node, and a communication spanning tree will be established.

Step 2: the root node will broadcast the task to all other nodes through the tree.

Step 3: the task's corresponding algorithm will be executed at each node. During the process, nodes will exchange messages with neighbors through the communication servers, and nodes will update variables according to local and neighbors' data.

Step 4: the results will be returned to the root node through the communication tree. The reorganized data will be sent back to users by the task server at the root node.

2.3. Function Library. In this part, we introduce several inline functions and system algorithms in DSP.

2.3.1. Synchronous Communication Function. The synchronous communication function can be used as *self.transmitData(self, data)*. Using this function, each node will send the "data" to all its neighbors and suspend until all the neighbors' data are received. This process ensures that all nodes stay in sync after calling this function.

2.3.2. Asynchronous Communication Function. The asynchronous communication function can be used as *self.send(self, data, delay)* or *self.sendDataToID(self, id, data, delay)*. For one node, the former one allows it to send "data" to all its neighbors while the latter one provides it a node-to-node communication with its neighbor named "id". "Delay" is the communication delay set by the users, which can be randomized or deterministic. The "data" is stored in a FIFO queue where historical data can be used. The algorithm running process will be immediately continued after the message transmission. As a result, different nodes will run the algorithm in different iteration rounds according to their

node status. Under these circumstances, developing asynchronous algorithms becomes available.

2.3.3. Debugging Function. The debugging function can be used as *self.sendDataToGUI(self, data)*. DSP has a simple GUI server for receiving information printed from the nodes. The "data" will be transformed into string format and be displayed on the interface in the form *"Node 'id': 'data'"*. Before the algorithm starts, users can set the variables to be recorded into the database for debugging.

2.3.4. System Algorithms. DSP provides some system algorithms for users to utilize in their self-design methods, mainly containing a spanning tree algorithm, the summation algorithm, the algorithm to find maximum or minimum, the shortest path algorithm, etc. The spanning tree algorithm can establish a spanning tree with a node specified as the root node. Each node can obtain its parent node and child nodes after running the algorithm. The summation algorithm and the algorithm to find the maximum or minimum enable each node to obtain the sum, maximum, or minimum of the required value. The shortest path algorithm makes a node find the shortest path to another node. All the algorithms mentioned above can be found in the "sample_code" folder.

3. Experiments

In this section, we show the application of DSP by giving two experiments. One is the distributed resource allocation problem, while the other is a 5000-node numerical example to find the solution of linear equations.

3.1. Distributed Resource Allocation. A distributed resource allocation problem is a particular distributed optimization problem where the optimization variables are individual but coupled with a global constraint. The mathematical form is shown as follows:

$$\begin{aligned} &\min \Sigma_{i=1}^{n} f_i(x_i), \\ &\text{s.t. } \Sigma_{i=1}^{n} x_i = C, \\ &\underline{c_i} \le x_i \le \overline{c_i}, i = 1, \cdots, n, \end{aligned} \tag{1}$$

where n is the number of nodes and C is the resource constraint. For each node i, f_i is the cost function and x_i is the resource allocated to the node i which is bounded by $[\underline{c_i}, \overline{c_i}]$. We designed an asynchronous distributed gradient-based algorithm (ADGD) to solve this problem by using DSP as the simulation environment [22].

The distributed generation control problem under the IEEE-30 bus case is considered as an example [23]. In this case, the system is composed of 6 generators and 30 buses. x_i (MW) is the power generation produced by generator i, and f_i ($/hr) can be approximated by a quadratic function $f_i(x_i) = a_{i_2} x_i^2 + a_{i_1} x_i + a_{i_0}$. The parameters are shown in Table 2, and the network of the IEEE-30 bus case is shown in Figure 4. The total power demand C is 291.76 MW.

We simulate the above case on DSP in a personal computer with Intel I7-8750 CPU and 16 GB memory. The local topology of node 1 (Gen #1) is shown as below as a JSON form:

```
{"ID" : 1,
"IP" : "localhost",
"PORT" : [10000, 10001, 10002, 10003, 10004, 10005, 10006],
"adjID" : [2, 4],
"datalist" : {"para" : [0, 2.0, 0.00375], "bound" : [50, 200], "amount" : 291.76}},
```

(2)

where "ID" is the node ID, "IP" is the local address, "PORT" is the port list for communication servers and task servers, "adjID" is the list of its neighbors' IDs, and "datalist" is a set of its initialization parameters. Each node possesses its topology information in the same format, and through the topology list, the network can be constructed by DSP. Our preloaded ADGD can be tested, and the optimal solution can be achieved based on the network. The convergence curves and results are shown in Figure 5 and Table 3. The total time cost to simulate 100 iterations is 3.93s. The execution process is fast since ADGD is an asynchronous method and has low time complexity.

As mentioned before, DSP is flexible and supports a plug-and-play topology. ADGD is tested in the 3-phase experiment below. Phase 1 is a normal case. In Phase 2, a node failure happens, that is, Generator 5 is broken. In Phase 3, Generator 5 is repaired and reconnected to the network. The topology changes happen at the (10s, 20s). Nodes in DSP will detect the neighbor change and the network topology will be updated. Figure 6 displays the topology changing process. Then ADGD will rerun the spanning tree algorithm to construct a new communication tree and continue the optimization process based on it. The spanning tree changing process and the algorithm running process are separately shown in Figures 7 and 8. The optimal points in different phases will be obtained at t_1, t_2, t_3. Consequently, the ability of algorithms to adapt to changing topology can be tested in DSP.

Table 2: Parameters of the generation control problem.

Gen	a_{i_0}	a_{i_1}	a_{i_2}	$\underline{c_i}$	$\overline{c_i}$
#1	0	2.0	0.00375	50	200
#2	0	1.75	0.0175	20	80
#3	0	1.0	0.0625	15	50
#4	0	3.25	0.00834	10	35
#5	0	3.0	0.025	10	30
#6	0	3.0	0.025	12	40

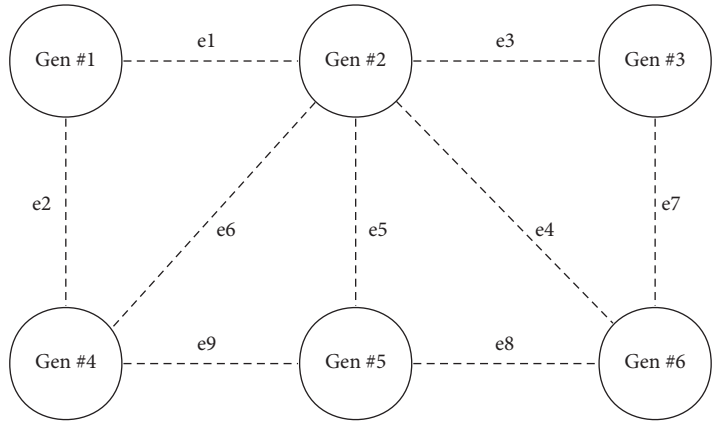

Figure 4: Network of the IEEE-30 bus case.

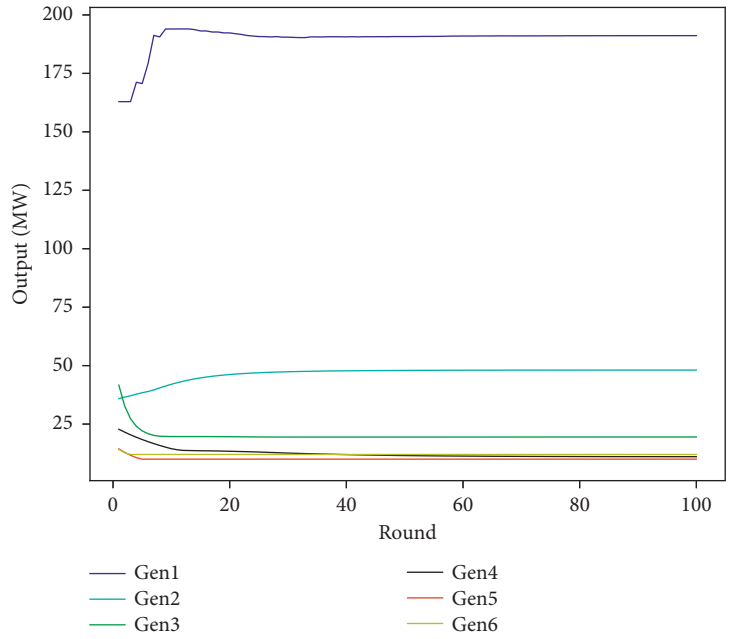

Figure 5: Convergence curve of ADGD under IEEE-30 bus case.

Table 3: Convergence results of ADGD under IEEE-30 bus case.

Gen #1	Gen #2	Gen #3	Gen #4	Gen #5	Gen #6	$P_d - \sum P_i$	Cost	Time
191.166	48.107	19.470	11.017	10.000	12.000	0.000	796.140	3.93

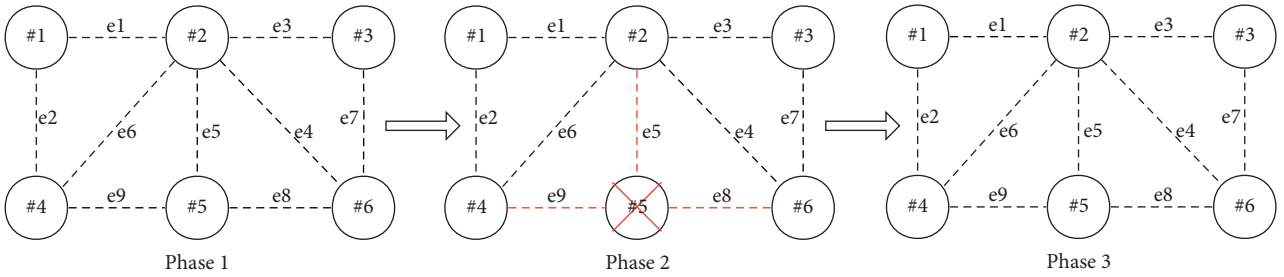

Figure 6: Networks under changing topology.

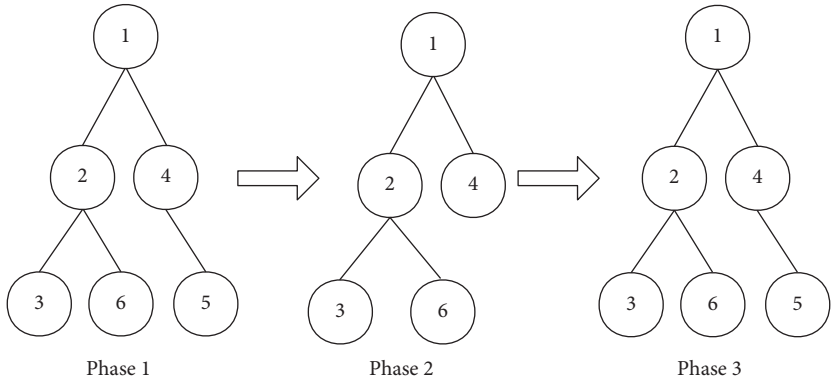

Figure 7: Spanning trees under changing topology.

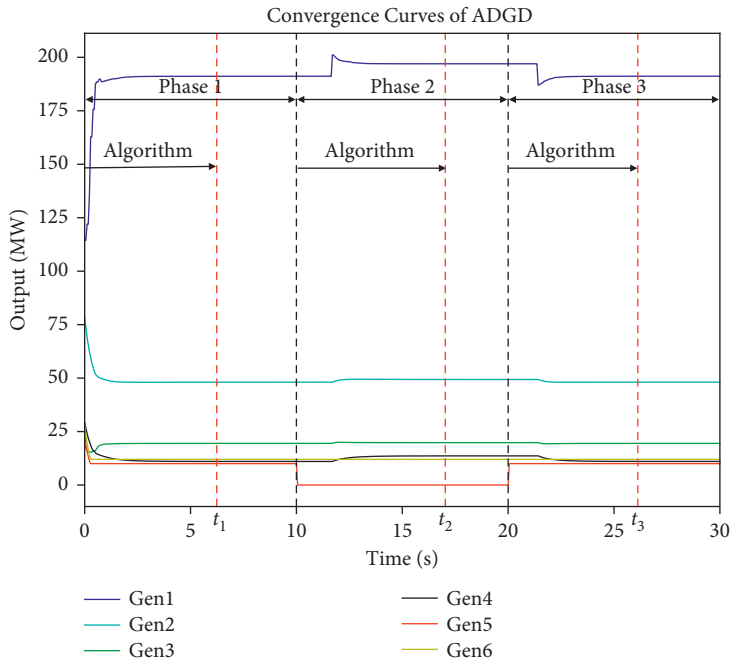

Figure 8: Convergence curves under changing topology.

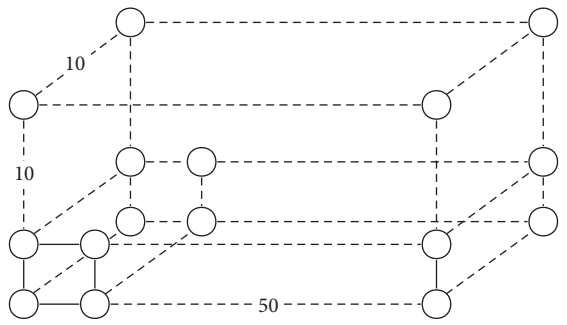

Figure 9: $50 \times 10 \times 10$ cube topology.

```
wyf@caslx:~$ ps -eo pid,psr,%cpu,%mem,vsz,rss,command |grep -w 'DSP_Process.*00'
21222 243  0.1  0.0 701392 33896 python3 DSP_Process.py 100
22330 212  0.1  0.0 701392 34076 python3 DSP_Process.py 200
23422 239  0.1  0.0 701392 33740 python3 DSP_Process.py 300
24522  91  0.1  0.0 701392 33724 python3 DSP_Process.py 400
25631 208  0.1  0.0 701392 33948 python3 DSP_Process.py 500
26722 215  0.1  0.0 701392 33820 python3 DSP_Process.py 600
27830  88  0.1  0.0 701392 33584 python3 DSP_Process.py 700
28922 222  0.1  0.0 701392 33836 python3 DSP_Process.py 800
30022 245  0.1  0.0 701392 33936 python3 DSP_Process.py 900
31130 236  0.1  0.0 701392 33832 python3 DSP_Process.py 1000
32222 226  0.1  0.0 701392 33764 python3 DSP_Process.py 1100
33322 237  0.1  0.0 701392 33740 python3 DSP_Process.py 1200
34422 224  0.1  0.0 701392 33820 python3 DSP_Process.py 1300
35522 112  0.1  0.0 701392 33792 python3 DSP_Process.py 1400
36622 101  0.1  0.0 701392 33948 python3 DSP_Process.py 1500
37722 223  0.1  0.0 701392 33860 python3 DSP_Process.py 1600
38823 221  0.1  0.0 701392 33896 python3 DSP_Process.py 1700
39930 108  0.1  0.0 701392 33896 python3 DSP_Process.py 1800
41022 120  0.1  0.0 701392 33876 python3 DSP_Process.py 1900
42121 250  0.1  0.0 701392 33784 python3 DSP_Process.py 2000
43222 110  0.1  0.0 701392 33976 python3 DSP_Process.py 2100
44322 249  0.1  0.0 701392 33832 python3 DSP_Process.py 2200
45422 228  0.1  0.0 701392 33924 python3 DSP_Process.py 2300
46522 221  0.1  0.0 701392 33928 python3 DSP_Process.py 2400
47622 111  0.1  0.0 701392 33856 python3 DSP_Process.py 2500
48717 205  0.1  0.0 701392 33724 python3 DSP_Process.py 2600
49823 194  0.1  0.0 701392 33820 python3 DSP_Process.py 2700
50923  85  0.1  0.0 701392 33872 python3 DSP_Process.py 2800
52014 234  0.1  0.0 701392 33956 python3 DSP_Process.py 2900
53129 146  0.1  0.0 701392 33608 python3 DSP_Process.py 3000
54222  68  0.1  0.0 701392 33720 python3 DSP_Process.py 3100
55322 216  0.1  0.0 701392 33888 python3 DSP_Process.py 3200
56425 228  0.1  0.0 701392 33824 python3 DSP_Process.py 3300
57521 233  0.1  0.0 701392 33604 python3 DSP_Process.py 3400
58622 115  0.1  0.0 701392 33900 python3 DSP_Process.py 3500
```

Figure 10: Node operation of DSP.

3.2. Large-Scale Experiment. We consider the problem of solving large-scale linear equations in DSP. The problem can be formulated as follows:

$$\begin{aligned} Ax &= b, \\ A &= \left(A_{ij}\right)_{n\times n}, \\ b &= \left(b_1, L, b_n\right)^T, \end{aligned} \tag{3}$$

where A is the coefficient matrix and b is the target vector. In this experiment, the total number of nodes is 5000. We denote N_i as the neighbor list of node i and assume that the matrix parameters of node i are only related to itself and its neighbors, $A_{ij} = 0\,(j \notin N_i \text{ and } j \neq i)$. In this experiment, we set $A_{ii} = 8$, $A_{ij} = 1\,(j \in N_i)$ and $b_i = |N_i| + 8$ for each node i. The answer is easily obtained as $x = 1$. The topology is a $50 \times 10 \times 10$ cube, which is shown in Figure 9.

We simulated this experiment on the DSP on two high-performance servers. One is with two AMD EPYC 7702 CPUs (128 cores, 256 threads) and 1 TB memory, while the other one is with two AMD EPYC 7402 CPUs and 512 GB memory. We use the former server to run 3500 nodes and the latter server to run 1500 nodes. Each node refers to a Python process, and an independent core runs many nodes. Figure 10 partially shows some basic information of the node operation on the former server when these nodes are just started.

The topology information is basically identical to the above example, except that the "datalist" contains $A_{ij}\,(j = 1, \cdots, n)$ and b_i. In the simulation environment, we apply a

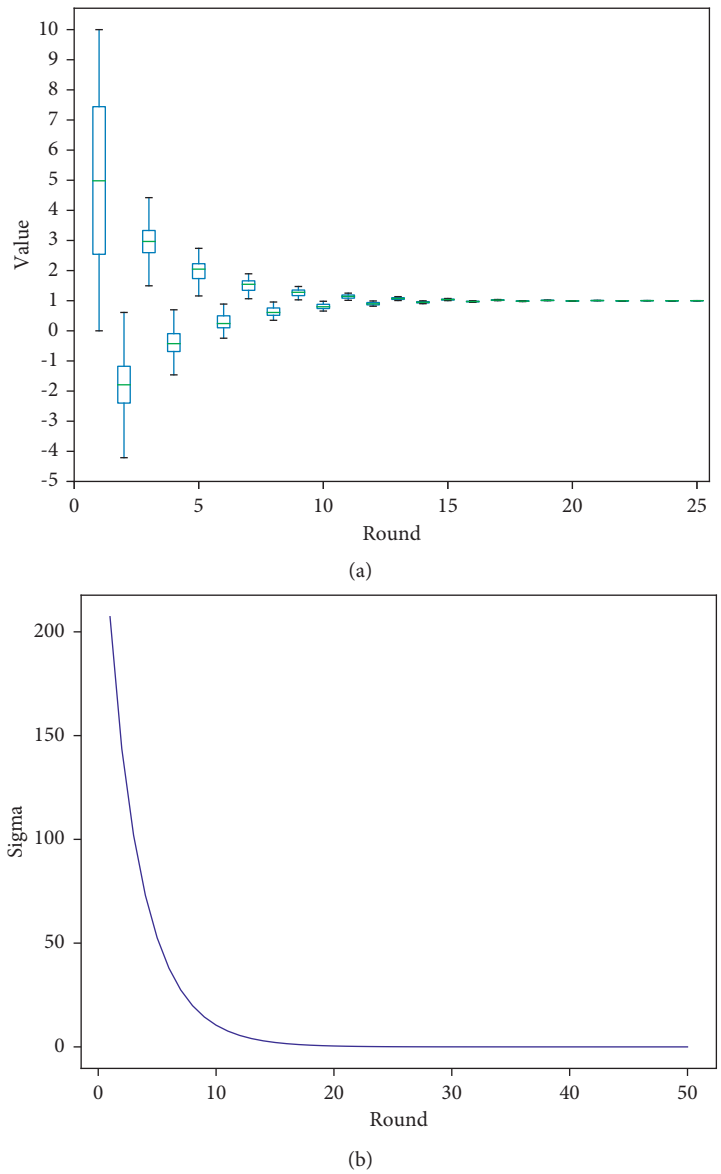

Figure 11: Convergence condition of the Jacobi method. (a) Convergence of the whole network; (b) distance from the optimal solution.

simple distributed implementation of the traditional Jacobi method introduced in [24] to solve these linear equations (since $A_{ij} = 0\,(j \notin N_i \text{ and } j \neq i)$). The updated formula is shown as follows:

$$x_i^{(k)} = \frac{1}{A_{ii}}\left\{\Sigma_{j\in N_i}\left(-A_{ij}x_j^{(k-1)}\right) + b_i\right\}(i = 1, 2, \cdots, n). \quad (4)$$

The initial value of each x_i is set to be a random value in $[0, 10]$. We select the box diagram to describe the convergence condition of the whole network in Figure 11(a). The distance between the process solution and the optimal solution is plotted in Figure 11(b). The total time cost is 138.35 s for 50 iterations. It is relatively slow since this Jacobi method is synchronous, and nodes expend unnecessary time on waiting for the network synchronization. This also proves the importance of asynchronous algorithms.

This experiment shows that DSP can work well under the scale of 5000 nodes. Through renting or buying more high-performance servers, more significant experiments can be implemented. The results indicate that the adaptability and robustness of the distributed algorithms can be effectively examined on the DSP.

4. Conclusions

In this paper, DSP, a process-based distributed algorithm simulation platform, is proposed to simulate real distributed systems. DSP is an agent-based distributed simulator that is flexible, versatile, and scalable. The main innovation of DSP is that it can support plug-and-play networks and the design of complex algorithms. Now the main structure and basic modules have been completed. The experiments show that DSP is suitable for developing distributed methods and

testing them in diverse scenes. DSP can be applied in various research areas for designing and verifying distributed applications. In fact, many students in our laboratory and cooperative colleges are using it. There are also many algorithms proposed for building systems and HVAC systems produced based on DSP. We are currently focusing on developing the multitasking management mechanism for simulating a more complex scenario where multiple methods interact. A more user-friendly GUI is also being continuously improved. In the future, we will mainly consider improving the efficiency of DSP, including how to accelerate computation by utilizing computing resources on the cloud and how to speed up communication by using efficient communication mechanisms. Another work we are trying is to apply DSP in hardware like Raspberry Pi for carrying out semiphysical simulation. Encapsulation and privacy issues will also be discussed [25].

Data Availability

The DSP software is available at https://github.com/Wales-Wyf/Distributed-Algorithm-Simulation-Platform-DSP--2.0.

Conflicts of Interest

The authors declare that they have no conflicts of interest.

Acknowledgments

This work received supports from National Key Research and Development Project of China under Grant 2017YFC0704100, entitled New Generation Intelligent Building Platform Techniques, National Natural Science Foundation of China (nos. 61425027 and 62192751), the 111 International Collaboration Program of China under Grant BP2018006, the BNRist Program under Grant no. BNR2019TD01009, and the National Innovation Center of High Speed Train R&D Project (CX/KJ-2020-0006). The authors appreciate intensive feedback from our project team users on New Generation Intelligent Building Platform Techniques including Dr. Ziyan Jiang, Prof. Jinchuan Xing, Prof. Qiliang Yang, Prof. Jili Zhang, Prof. Junqi Yu, Prof. Zhenya Zhang, Prof. Qiansheng Fang, Prof. Xi Chen, Prof. Qingshan Jia, Dr. Huai Li, and Dr. Zhen Yu and also from other research teams including Prof. Ming Wang, Prof. Yunchu Zhang, and Prof. Zhiyong Xing.

References

[1] Q. Zhao and Z. Jiang, "Insect intelligent building (I2B): a new architecture of building control systems based on Internet of Things (IoT)[J]," *International Conference on Smart City and Intelligent Building*, vol. 890, pp. 457–466, 2018.

[2] P. Baran, "On distributed communications networks[J]," *Communications Systems IEEE Transactions on*, vol. 12, no. 1, pp. 1–9, 2009.

[3] R. M. Fujimoto, "Parallel and distributed simulation systems [C]," in *Proceedings of the Simulation Conference Proceedings, 1999 Winter*, IEEE, Phoenix, AZ, USA, 1999.

[4] A. V. Brito, H. Bucher, H. Oliveira et al., "A distributed simulation platform using HLA for complex embedded systems design[C]," in *Proceedings of the IEEE/ACM International Symposium on Distributed Simulation & Real Time Applications*, Chengdu, China, Oct.2016.

[5] H. Casanova, A. Giersch, A. Legrand, M. Quinson, and F. Suter, "Versatile, scalable, and accurate simulation of distributed applications and platforms," *Journal of Parallel and Distributed Computing*, vol. 74, no. 10, pp. 2899–2917, 2014.

[6] A. Anagnostou, A. Nouman, and S. Taylor, "Distributed hybrid agent-based discrete event emergency medical services simulation[C]," in *Proceedings of the Winter Simulations Conference*, Washington, DC, USA, 8-11 Dec. 2013.

[7] R.. Fujimoto, "Parallel and distributed simulation," in *Proceedings of the 2015 Winter Simulation Conference (WSC)*, Huntington Beach, CA, USA, 6-9 Dec. 2015.

[8] Z. Fan, F. Qiu, A. Kaufman, and S. Yoakum-Stover, "GPU cluster for high performance computing," in *Proceedings of the SC'04: Proceedings of the 2004 ACM/IEEE conference on Supercomputing*, Pittsburgh, PA, USA, 6-12 Nov. 2004.

[9] S. J. E. Taylor, "Distributed simulation: state-of-the-art and potential for operational research," *European Journal of Operational Research*, vol. 273, no. 1, pp. 1–19, 2019.

[10] J. S. Dahmann, R. M. Fujimoto, and R. M. Weatherly, "The department of defense high Level architecture," in *Proceedings of the 29th conference on Winter simulation*, Georgia, Atlanta, USA, December 7 - 10, 1997.

[11] M. Villani, "Multi-agent systems. Simulation and applications (computational analysis, synthesis, and design of dynamic models series)[J]," *The Journal of Artificial Societies and Social Simulation*, vol. 12, no. 4, 2009.

[12] L. W. Gropp and E. Skjellum, "Using MPI – portable Parallel Programming with the Message Passing Interface 2e[J]," *Scientific Programming*, vol. 5, no. 3, pp. 275-276, 2014.

[13] T. Blochwitz, M. Otter, M. Arnold et al., "The functional mockup interface for tool independent exchange of simulation models," in *Proceedings of the 8th International Modelica Conference*, Dresden, Germany, 2011.

[14] P. O. Siebers, C. M. Macal, J. Garnett, D. Buxton, and M. Pidd, "Discrete-event simulation is dead, long live agent-based simulation!" *Journal of Simulation*, vol. 4, no. 3, pp. 204–210, 2010.

[15] R. Bagrodia, E. Deelman, and T. Phan, "Parallel simulation of large-scale parallel applications," *International Journal of High Performance Computing Applications*, vol. 15, no. 1, pp. 3–12, 2001.

[16] G. Cordasco, V. Scarano, and C. Spagnuolo, "Distributed MASON: a scalable distributed multi-agent simulation environment[J]," *Simulation Modelling Practice and Theory*, vol. 89, 2018.

[17] M. Hanai, T. Suzumura, A. Ventresque, and K. Shudo, "An adaptive VM provisioning method for large-scale Agent-based traffic simulations on the cloud," in *Proceedings of the 2014 IEEE 6th International Conference on Cloud Computing Technology and Science*, Singapore, 15-18 Dec.2014.

[18] F. Perkonigg, D. Brujic, and M. Ristic, "MAC-Sim: a multi-agent and communication network simulation platform for smart grid applications based on established technologies[C]," in *Proceedings of the Smart Grid Communications (SmartGridComm), 2013 IEEE International Conference*, Vancouver, BC, Canada, 21-24 Oct. 2013.

[19] F. G. Hamza-Lup, J. P. Rolland, and C. Hughes, "A distributed augmented reality system for medical training and simulation [J]," 2018, https://arxiv.org/abs/1811.12815.

[20] R. N. Calheiros, R. Ranjan, A. Beloglazov, C. A. F. De Rose, and R. Buyya, "CloudSim: a toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms," *Software: Practice and Experience*, vol. 41, no. 1, pp. 23–50, 2011.

[21] A. Montresor and M. Jelasity, "PeerSim: a scalable P2P simulator[C]," in *Proceedings of the Proceedings P2P 2009, Ninth International Conference on Peer-to-Peer Computing*, IEEE, Seattle, Washington, USA, 9-11 September 2009.

[22] Y. Wang, Q. Zhao, and X. Wang, "An asynchronous gradient descent based method for distributed resource allocation with bounded variables," *IEEE Transactions on Automatic Control*, p. 1, 2021.

[23] O. Alsac and B. Stott, "Optimal load flow with steady-state security," *IEEE Transactions on Power Apparatus and Systems*, vol. PAS-93, no. 3, pp. 745–751, 1974.

[24] A. Margaris, S. Souravlas, and M. Roumeliotis, "Parallel implementations of the Jacobi linear algebraic systems solve [C]," in *Proceedings of the Balkan Conference on Informatics*, Sofia, Bulgaria, March 2014.

[25] J. D. Watson and F. H. C. Crick, "Molecular structure of nucleic acids: a structure for deoxyribose nucleic acid," *Nature*, vol. 171, no. 4356, pp. 737-738, 1953.