SIMULATING THE PROGRAMMABLE NETWORKS FOR HLA COMPATIBLE HIGH-PERFORMANCE SIMULATORS

Kayhan M. İmre
Department of Computer Engineering,
Hacettepe University
Ankara, TURKEY
E-mail: ki@hacettepe.edu.tr

KEYWORDS

Parallel discrete event simulation, network simulation, high performance programmable networks.

ABSTRACT

This paper explores a parallel discrete event simulator that simulates a programmable fat tree network. The programmable networks can be programmed to perform application specific tasks. The task explored in our research is a time management functionality offloaded to the network switches. Specifically, the network switches used for constructing the fat tree run Greatest Available Logical Time (GALT) computation. In this paper, this switch-based GALT computation is compared against two node-based GALT computations using the simulator developed.

INTRODUCTION

IEEE Standard for Modeling and Simulation (M&S) High Level Architecture (HLA) IEEE Std 1516™-2010 defines a standard for distributed simulation (IEEE Standard 1516.1-2010). HLA has well defined managements functions for implementing distributed simulations or integrating individual simulations as a single distributed simulation. HLA standard does not exclude high performance parallel simulations that require low latency interactions and high-level parallelism. The implementation of Run Time Infrastructure (RTI) of HLA is the key element for achieving high performance execution of an HLA compliant parallel simulation. One of the bottlenecks for achieving high performance RTI execution is advancing the simulation time of the joined federates. The federates are the parallel processes of an HLA compliant parallel simulation. The time regulating federates send time advance requests (TAR) to RTI, and in return, RTI responds such a request with time advance grant (TAG) callback. Each time regulating federate provides a time advancement plus a lookahead value. Behind the scenes, the RTIs join in a distributed computation to find the minimum of all *time+lookahead* values. This minimum value is named as Greatest Available Logical Time (GALT) that is no federate gets time advance grant beyond this value. As a consequence, the distributed GALT calculation becomes a global synchronization mechanism that harnesses the progress of federates. Improving GALT calculation time will increase the parallelism by decreasing the time spent for synchronization. This paper presents a simulation application developed as a part of a research that explores programmable networks for improving the performance of parallel applications. The simulation developed is used for evaluating network offloaded GALT computation and some other functions of RTI. Programmable switches are becoming available for performing application specific functions (Dang et al. 2015; Jin et al. 2017; Kaur et al. 2021). In the rest of the paper, the implementation of the simulator is presented. In the evaluation section, three different GALT algorithms are compared against each other using the simulator implementation.

SIMULATING PROGRAMMABLE NETWORK

The fat tree topology is one of the preferred topologies in high performance computing domain. This topology can be built from high speed standard network switches (Al-Fares et al. 2008) but constructing the fat tree topology from programable switches can improve the network beyond its connectivity rich characteristics. Some critical distributed computations with lightweight processing but requiring coordination of many participants are good candidates for implementing on the distributed network switches. The Greatest Available Logical Time (GALT) calculation of High Level Architecture (HLA) is one of those candidates. The performance of GALT calculation has a direct effect on the overall performance of a simulation application. The GALT calculation can be considered as a synchronization point that the joining federates of an HLA compliant simulation wait for the time advancement requests to be granted, and then, the federates can continue their local computations. The simulator presented in this paper is intended for investigating the behavior of the GALT calculation when it is implemented on a programable network. Along with switch-based GALT calculations, two other node-based GALT calculations were also implemented using the simulator to provide an equal ground for assessing performance characteristics.

The first implementation choice was to make whether the simulator be sequential or parallel, and the latter was chosen to make implementation simple and scalable. The logical processes of the parallel implementation are implemented as processes of MPI-based (Message Passing Interface) parallel application. One-to-one mapping of logical processes to MPI processes provides

Communications of the ECMS, Volume 36, Issue 1,
Proceedings, ©ECMS Ibrahim A. Hameed, Agus Hasan,
Saleh Abdel-Afou Alaliyat (Editors) 2022
ISBN: 978-3-937436-77-7/978-3-937436-76-0(CD) ISSN 2522-2414

a good encapsulation. Additionally, one-to-one mapping of logical processes is achieved by mapping a logical process to a node to be simulated. A node can be either a programmable network switch or a computer. In Figure 1, a fat tree constructed from 4-port switches is depicted, and the numbers are MPI process identifications (i.e. ranks). There are total of 36 logical processes implement Chandy/Misra/Bryant null message algorithm to simulate the high-level behavior of the overall system (Chandy and Misra 1979; Bryant 1977; Fujimoto 2000). There are four logical processes (LPs) for simulating the core switches, eight LPs for the aggregate switches, eight LPs for the edge switches, and finally, there are sixteen computers represented by their LPs.

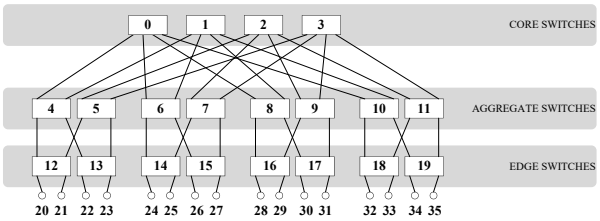

Figure 1: Fat tree for 16 compute nodes and MPI ranks.

In Figure 2, a larger fat tree constructed from 6-port switches is depicted. Each logical process can decide its role in the simulation by controlling MPI process rank, and then, each chooses its role in the simulation. There are four different roles and four corresponding LP types; Core switch, aggregate switch, edge switch and compute node (computer). Each LP can also locate its place in the topology using its own rank, and identify the MPI ranks that the LP exchanges messages (simulation events) with.

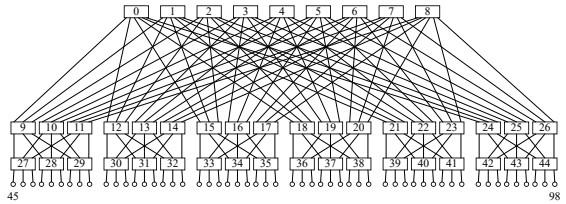

Figure 2: Fat tree for 54 compute nodes and MPI ranks.

After identifying the other LPs to communicate with, each LP assigns a pair of event FIFOs to each interaction with another LP, which also corresponds a communication link in the simulated fat tree. In Figure 3, up and downlink FIFOs assigned to communication links are shown for aggregate and edge switch LPs. In Figure 4, FIFOs are shown for core switch LPs have downlinks only. Finally, the LPs corresponding to compute nodes have only single links connected to edge switches (Figure 5). The events exchanged between LPs are used for simulating the communication packets transmitted over the communication links. The event data structure contains a payload field to transmit the real contents of the message packets simulated. This enables our simulator to run the real algorithms with real message contents while simulating the network behavior of the fat tree. The FIFOs inside the dash-lined boxes belong to LP's logic that simulates communication link queues.

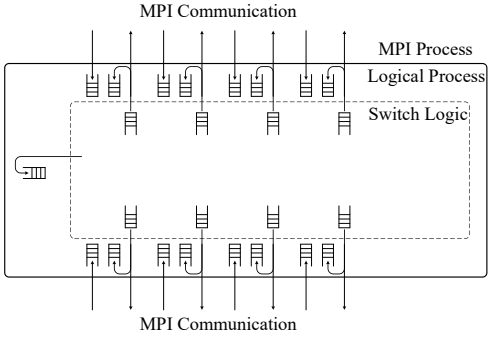

Figure 3: Logical Process FIFOs for aggregate and edge switches.

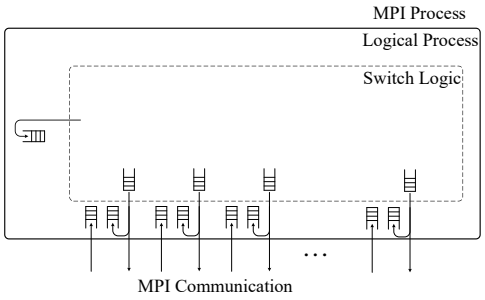

Figure 4: Logical Process FIFOs for core switches.

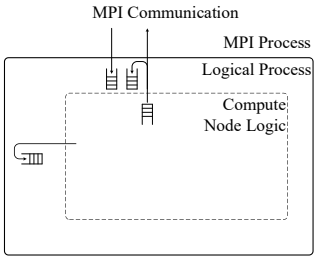

Figure 5: Logical Process FIFOs for compute nodes.

Finally, in each figure, the single FIFO standing on the left-hand side not attached to any link is used for storing events related to local processing timing. For example, in the compute node, the real application with real data can be executed but execution timing is simulated by the events that mark start and end time of a real computation. Such timings are not easy to measure using system clock but the instrumented code can predict execution timings and feed the FIFOs with proper events.

In Figure 6, the "main" function of the MPI process is given as an excerpt to show general structure of the simulator. After constructing the fat tree topology logically, each MPI process assigns up/down link FIFO pairs to communication links. Initially, each MPI process sends a null message to each link using "MPI_Isend" function which is an asynchronous message send function. In the main simulation loop, synchronous message receive function is used for waiting a message from a specific process (implementing an LP). After

making sure that there is no empty FIFO related to communication links, the algorithm finds the event with the minimum time, and consumes the events with the same time stamps. Since this is a typical implementation of Chandy/Misra/Bryant null message algorithm, there is no need to explain how the rest handles the null messages.

To simulate the programmability of communication switches, event triggered codes are placed in the "Consume_Event" function, and those codes implement application specific algorithms such as GALT calculation. The GALT calculation has several steps executed on the switches in a distributed manner. GALT calculation is triggered from a compute node by sending a time advance request (TAR) event to the edge switch LP. Upon receiving such event from a downlink, the edge switch LP checks the values received from downlinks previously, and if a new minimum value is calculated a new event is send to every uplink to inform the connected aggregate switch LPs. The aggregate switch LPs perform similar operation to inform the core switch LPs. When a core switch LP is reached, all the values from downlinks are checked. If a new minimum value is reached, this value is marked as the GALT value. In the next several steps, the GALT value is broadcasted downwards using downlinks until the event carrying the GALT value reaches a compute node. The compute node gets this event, creates a new event called "time advance grant" (TAG), and inserts the new event into the local application FIFO to finalize the GALT calculation.

```
void main(int argc, char* argv[])
{

// Definitions and Initializations  . . .

// The construction of the Fat tree  . . .

  for (int i = 0; i < ulcount; i++)   // Initiate uplink communications and
      Null_Event(Time, i, UPLINK);    // send NULL Events with current time

  for (int i = 0; i < dlcount; i++)   // Initiate downlink communication and
      Null_Event(Time, i, DOWNLINK);  // send NULL Events with current time

  while (continue_simulation) {
    for (int i = 0; i < ulcount; i++) // Wait for queues to become non-empty
      if (ulist[i].head == NULL) {    // Uplink
        MPI_Recv(&urecv_buff[i][0], RECV_BUFF_SIZE, MPI_CHAR,
                 ul[i], ucount_recv[i], MPI_COMM_WORLD, &ustatus[i]);
        ////////  Add event to FIFO . . .
      }
    for (int i = 0; i < dlcount; i++) // Wait for queues to become non-empty
      if (dlist[i].head == NULL) {    // Downlink
        MPI_Recv(&drecv_buff[i][0], RECV_BUFF_SIZE, MPI_CHAR,
                 dl[i], dcount_recv[i], MPI_COMM_WORLD, &dstatus[i]);
        ////////  Add event to FIFO . . .
      }
    ////////////////    Find Min . . .
    Time = min;
    for (int i = 0; i < ulcount; i++) // Consume events with current time
      Consume_Event(Time, i, UPLINK); // Uplink FIFO
    for (int i = 0; i < dlcount; i++) // Consume events with current time
      Consume_Event(Time, i, DOWNLINK); // Downlink FIFO

    Consume_Event(Time, 0, APPLICATION);// Application FIFO

    for (int i = 0; i < ulcount; i++)   // Send events with time + 1
      Null_Event(Time + 1, i, UPLINK);  // Send if Uplink is not busy
    for (int i = 0; i < dlcount; i++)
      Null_Event(Time + 1, i, DOWNLINK);// Send if Downlink is not busy
  }
  MPI_Finalize();
}
```

Figure 6: The "main" function of the MPI process implements LPs.

EVALUATION

In this section, the performance evaluations of three different GALT calculation approaches are presented. Firstly, three different GALT calculations are implemented using the simulator presented in this paper. The first one as explained in the paper is switch based GALT calculation. The second GALT calculation approach is a conventional one that uses compute nodes to reach the result. This approach benefits from fat tree topology characteristics by calculating local GALT values in the subtrees, recursively, and then, achieving the final GALT value in a compute node. The compute node holding the GALT value broadcasts this value to every other compute node using broadcast capability provided by the network. The third GALT calculation approach is broadcast-based, each compute node broadcasts its TAR request to everyone. Then, each compute node individually calculates the GALT value from the messages received.

The first evaluation of three approaches are carried out using a quiet network to measure pure performances of each GALT calculation approach. In the second part of the evaluation, the federates running on compute nodes are logically allocated in a two-dimensional space, and using data distribution management of HLA, each federate receives events from eight neighbors. While the federates are communicating using this publish-subscribe mechanism, the GALT calculation is triggered to measure its performance under different network traffic loads. The evaluation of three different GALT calculation approaches under various network loads are presented in the second part.

In Figure 7, the results of three different GALT calculation in a quiet network for three different network sizes are presented. For switch size 4, there are 16 compute nodes, and fat tree-based approach (FTAR) is the worst, broadcast-based approach (BTAR) is better, and programmable switch-based approach (TAR) is the best. For switch size 6, there are 54 compute nodes, the broadcast-based approach becomes the worst. For switch size 8, there are 128 compute nodes, the broadcast-based approach gets much worse, while TAR and FTAR keep their performances steady for all network sizes.

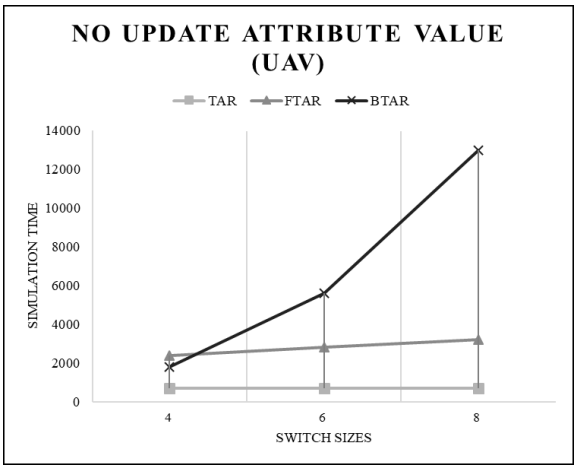

Figure 7: Three GALT algorithms for three different network sizes.

In the second part of the evaluation, additional network traffic is generated to see how it affects GALT calculations. In Figure 8, each compute node (i.e. federate running on it) sends an event to each neighbor causing each compute node receiving eight events from eight neighbors. The performances of switch-based (TAR) and fat tree-based (FTAR) calculations scale well while broadcast-based (BTAR) calculation gets much worse. In Figure 9, each compute node (i.e. federate running on it) sends five events to each neighbor causing each compute node receiving forty events from eight neighbors. The performances of TAR and FTAR are nearly the same.

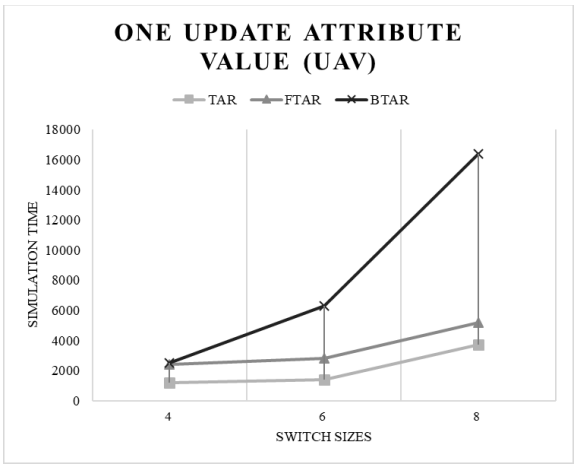

Figure 8: Three GALT algorithms for three different network sizes and light network traffic.

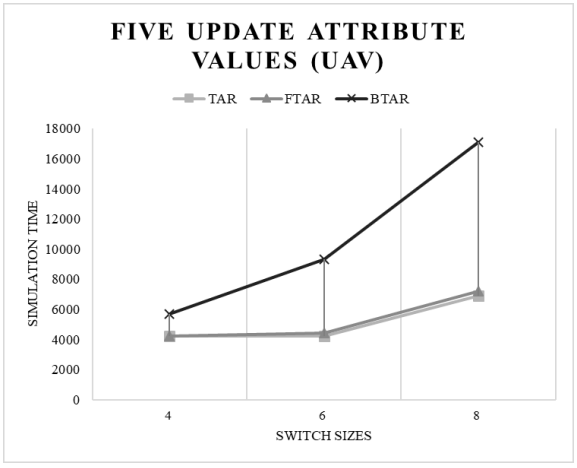

Figure 9: Three GALT algorithms for three different network sizes and heavy network traffic.

From Figure 10 to Figure 12, the performances of TAR and FTAR approaches are evaluated. Similar to previous measurements, different communication loads are tested for switch size 8 with 128 compute nodes only. In three figures, performance results for three different software related overhead values are presented. The software related overhead includes message preparation cost and handling costs of network software layers. Using some experimental values (α=100, α=400 and α=800), two GALT calculation approaches (TAR and FTAR) are compared. Since switch-based approach avoids such software related overheads, it performs much better when such overheads are high. As a general trend, when the network traffic gets heavier, the performance difference between switch-based (TAR) and fat tree-based (FTAR) calculations gets narrower.

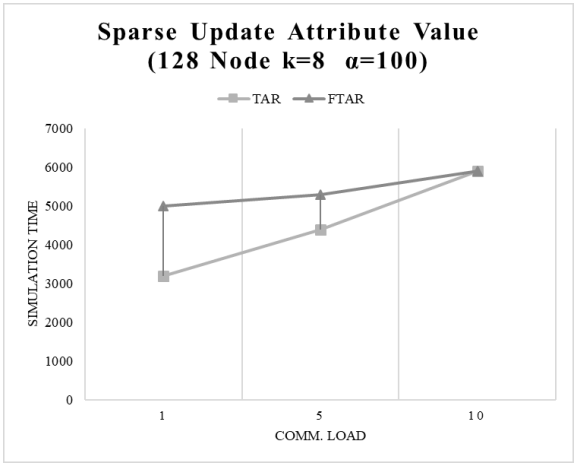

Figure 10: TAR and FTAR approaches with low software overhead.

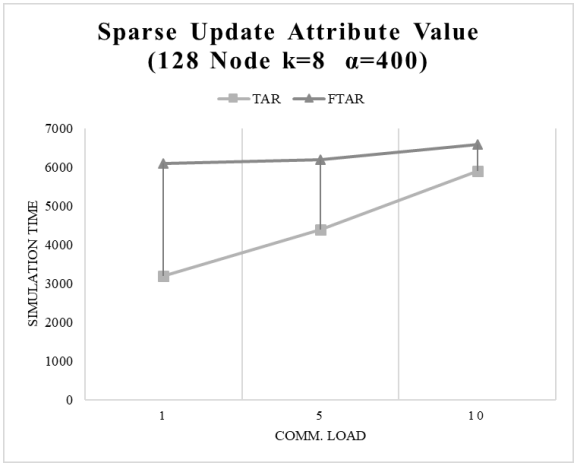

Figure 11: TAR and FTAR approaches with moderate software overhead.

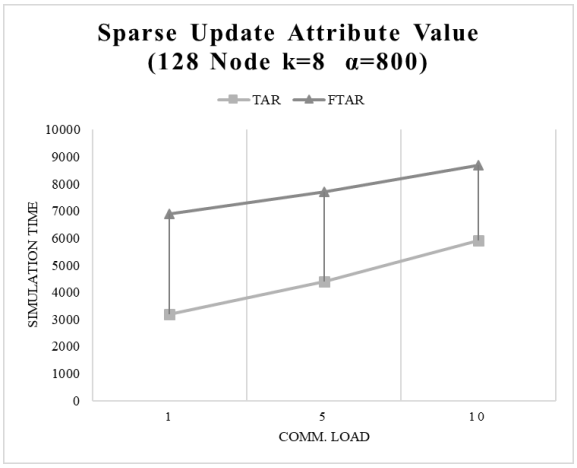

Figure 12: TAR and FTAR approaches with high software overhead.

CONCLUSION

The simulator presented in this paper simulates a simulation infrastructure that offloads some of its functionality to a programmable network. The programmable fat tree network performs GALT calculation on the programmable switches. The preliminary performance results of three different GALT calculation approaches are projected using the simulator presented in this paper. These preliminary experiments show that offloading time advancement calculation to the network switches will help to increase the performance of HLA compliant parallel and distributed simulations. The simulator is developed using MPI parallel programming library to accommodate real workloads running as a part of compute node LPs. The simulation events not only contain simulation related data but also real application data encapsulated in the payload field. This system can easily scale up using parallel computers both to shorten the execution time and to overcome memory limitations.

REFERENCES

Al-Fares M.; Loukissas A.; and Vahdat A. 2008. “A scalable, commodity data center network architecture”. SIGCOMM Comput. Commun. Rev. 38, 4 (October 2008), 63–74.

Bryant, R.E. 1977. “Simulation of Packet Communications Architecture Computer Systems”. MIT-LCS-TR-188, Massachusetts Institute of Technology.

Chandy K. and Misra J. 1979. “Distributed Simulation: A Case Study in Design and Verification of Distributed Programs”. IEEE Transactions on Software Engineering. Vol. 5. No. 5. 440–452.

Dang H.T.; Sciascia D.; Canini M.; Pedone F.; and Soulé R. 2015. “NetPaxos: consensus at network speed”. In Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research (SOSR '15). Association for Computing Machinery, New York, NY, USA, Article 5, 1–7.

Fujimoto R. M; 2000. “Parallel and Distributed Simulation Systems”, Wiley Interscience.

IEEE Standard 1516.1-2010. “Standard for Modeling and Simulation (M&S) High Level Architecture (HLA) - Federate Interface Specification.”

Jin X.; Li X.; Zhang H.; Soulé R.; Lee J.; Foster N.; Kim C.; and Stoica I. 2017. “NetCache: Balancing Key-Value Stores with Fast In-Network Caching”. In Proceedings of the 26th Symposium on Operating Systems Principles (SOSP '17). Association for Computing Machinery, New York, NY, USA, 121–136.

Kaur S.; Kumar K.; and Aggarwal N. 2021. “A review on P4-Programmable data planes: Architecture, research efforts, and future directions”, Computer Communications, Volume 170, 109-129.

AUTHOR BIOGRAPHY

KAYHAN M. İMRE received his B.Sc., M.Sc., degrees in Computer Engineering from Hacettepe University, Ankara, Turkey and his Ph.D. degree in Computer Science from University of Edinburgh, Scotland, in 1985, 1987 and 1993 respectively. He is an Associate Professor at the Computer Engineering Department, Hacettepe University, Ankara. His research interests are in parallel processing, parallel and distributed simulation and real-time systems.