Proceedings of the 2025 Winter Simulation Conference
E. Azar, A. Djanatliev, A. Harper, C. Kogler, V. Ramamohan, A. Anagnostou, and S. J. E. Taylor, eds.

DDA-PDES: A DATA-DEPENDENCE ANALYSIS PARALLEL DISCRETE-EVENT SIMULATION FRAMEWORK FOR EVENT-LEVEL PARALLELIZATION OF GENERAL-PURPOSE DES MODELS

Erik J. Jensen[1], James F. Leathrum, Jr.[1], Christopher J. Lynch[1,2], Katherine Smith[1,3], and Ross Gore[1,4]

[1]Department of Electrical and Computer Engineering, Old Dominion University, Norfolk, VA, USA
[2]Virginia Modeling, Analysis, and Simulation Center, Old Dominion University, Suffolk, VA, USA
[3]Virginia Digital Maritime Center, Old Dominion University, Suffolk, VA, USA
[4]Center for Secure and Intelligent Critical Systems, Old Dominion University, Suffolk, VA, USA

ABSTRACT

Utilizing data-dependence analysis (DDA) in parallel discrete-event simulation (PDES) to find event-level parallelism, we present the DDA-PDES framework as an alternative to spatial-decomposition (SD) PDES. DDA-PDES uses a pre-computed Independence Time Limit (ITL) table to efficiently identify events in the pending-event set that are ready for execution, in a shared-memory-parallel simulation engine. Experiments with AMD, Qualcomm, and Intel platforms using several packet-routing network models and a PHOLD benchmark model demonstrate speedup of up to 8.82x and parallel efficiency of up to 0.91. In contrast with DDA-PDES, experiments with similar network models in ROSS demonstrate that SD-PDES cannot speed up the packet-routing models without degradation to routing efficacy. Our results suggest DDA-PDES is an effective method for parallelizing discrete-event simulation models that are computationally intensive, and may be superior to traditional PDES methods for spatially-decomposed models with challenging communication requirements.

1 INTRODUCTION

Parallel Discrete-Event Simulation (PDES) methods typically rely on spatial decomposition (SD-PDES), where a simulation model is partitioned into separate logical processes (LPs) that can execute in parallel (Fujimoto 1990; Perumalla et al. 2022; Fujimoto et al. 2017; Rahman et al. 2017; Andelfinger and Cai 2022). SD-PDES is limited by simulation model decomposability and serial execution of events within an LP. In contrast, event-level PDES allows multiple independent events within a single LP to execute simultaneously, potentially offering greater parallelism than traditional SD-PDES methods. Event-level PDES may be particularly useful (1) for models with long-range communication patterns that would make LP-parallelization inefficient and (2) for monolithic models for which spatial decomposition is challenging.

In prior work, we introduced a novel approach for identifying independent events through data-dependence analysis (DDA) (Jensen et al. 2025). We defined mathematical frameworks for determining event readiness based on state-variable usage, scheduling dependencies, and timestamps. This laid the theoretical foundation for a new class of PDES that can dynamically identify ready events during simulation execution, regardless of model decomposability. Building on this foundation, we present the design, implementation, and performance of our Data-Dependence Analysis Parallel Discrete-Event Simulation (DDA-PDES) framework. Our contributions include:

- A parallel simulation engine that dynamically identifies ready events during runtime using DDA
- Algorithms and data structures for ready-event discovery and parallel execution

- A performance evaluation across multiple x86- and ARM-based hardware platforms
- Comparisons with the ROSS (Carothers et al. 2002) parallel simulation system using similar packet-routing network models that are challenging for SD-PDES

This work demonstrates that the theoretical foundations established in our earlier work can be implemented in a shared-memory high-performance simulation system. The results show that DDA-PDES can effectively identify and utilize parallelism at the event level, achieving significant speedup over serial execution and superior performance compared with SD-PDES, for some DES models.

The remainder of this paper is organized as follows: section 2 reviews the theoretical foundations of DDA-PDES and related work. Section 3 describes the architecture and implementation details of our framework. Section 4 outlines the experimental setup, including simulation models and hardware platforms. Section 5 presents performance results and analysis. Section 6 discusses conclusions and future work.

2 BACKGROUND AND THEORETICAL FOUNDATIONS

This section summarizes the foundational theoretical concepts underlying DDA-PDES, drawing from our previous work (Jensen et al. 2025). We discuss related work in event-level parallelization in section 2.2.

2.1 Data-Dependence Analysis in DES

Data-dependence analysis (DDA) is used to identify independent events in a discrete-event simulation model by examining state-variable (SV) usage and scheduling dependencies between events. The fundamental concept is that if two events do not have conflicting SV accesses, and cannot schedule intermediate events with conflicting SV accesses, they can be executed in any order without affecting the simulation outcome.

2.1.1 Identifying Ready Events

At a high level, ready events (REs) are defined as those events in the timestamp-ordered pending-event set $\mathbf{E}$, which will not be affected by preceding events in $\mathbf{E}$ and cannot affect preceding events in $\mathbf{E}$, if executed out of order. These events may execute immediately. Readiness for the k^{th} event in $\mathbf{E}$ requires mutual independence for each (j,k) pair in $\mathbf{E}$, where $j \in \{1,\ldots,k-1\}$, for $k>1$. The determination that two events in a pair are mutually independent requires knowledge of data dependencies and timing.

Consider a simulation with a non-empty pending-event set $\mathbf{E}$, where each event is a specific instance of an associated event type. An event type defines the event routine, which determines state-variable updates and scheduling of new events. An event contains an event type $\mathtt{V}_i^G$ with a set of SVs, $\mathbf{S}_i \coloneqq \mathbf{I}_i \cup \mathbf{O}_i$, that are read ($\mathbf{I}_i$) or written ($\mathbf{O}_i$) during event execution, where the simulation model has n different event types, and $i \in \{1,\ldots,n\}$. There are three different ways in which event routine i may use SVs in $\mathbf{S}_i$. For each state variable $s_{i,\ell}$ in $\mathbf{S}_i$, where $\mathbf{S}_i$ has n_i state variables, and $\ell \in \{1,\ldots,n_i\}$, event routine i may

1. update $s_{i,\ell}$,
2. read $s_{i,\ell}$ to update an SV in $\mathbf{S}_i$, or
3. read $s_{i,\ell}$ to evaluate a condition, which determines conditional SV updating in $s_{i,\ell}$ or conditional event scheduling.

2.1.2 Direct and Indirect Data Dependencies

Data dependencies between event types can be direct or indirect. A direct data dependency (DDD) exists when two event types share state variables and at least one of them updates shared variables. Formally:

Definition 1 (Direct Data Dependency (DDD))

For two event types $\mathtt{A}$ and $\mathtt{B}$:

$$\mathtt{A} \overset{\text{DDD}}{\longleftrightarrow} \mathtt{B} \iff \mathbf{S}_{\mathtt{A}} \cap \mathbf{S}_{\mathtt{B}} = \mathbf{S}_{\mathtt{A}\cap\mathtt{B}} \neq \emptyset \ \wedge \ (\mathbf{S}_{\mathtt{A}\cap\mathtt{B}} \cap \mathbf{O}_{\mathtt{A}} \neq \emptyset \ \vee \ \mathbf{S}_{\mathtt{A}\cap\mathtt{B}} \cap \mathbf{O}_{\mathtt{B}} \neq \emptyset). \tag{1}$$

An indirect data dependency (IDD) occurs when one event type can schedule another event type, either immediately or intermediately, which then has a direct data dependency with a third event type:

Definition 2 (Indirect Data Dependency (IDD))

For three event types $\mathtt{A}, \mathtt{A}+,$ and $\mathtt{B}$:

$$\mathtt{A} \overset{\text{IDD}}{\longleftrightarrow} \mathtt{B} \iff \exists \mathtt{A}+ \mid \Big(\mathtt{A} \overset{\text{reach}}{\longrightarrow} \mathtt{A}+ \ \wedge \ \mathbf{S}_{\mathtt{A}+} \cap \mathbf{S}_{\mathtt{B}} = \mathbf{S}_{\mathtt{A}+\cap\mathtt{B}} \neq \emptyset \ \wedge \ (\mathbf{S}_{\mathtt{A}+\cap\mathtt{B}} \cap \mathbf{O}_{\mathtt{A}+} \neq \emptyset \ \vee \ \mathbf{S}_{\mathtt{A}+\cap\mathtt{B}} \cap \mathbf{O}_{\mathtt{B}} \neq \emptyset) \Big),$$

where $\mathtt{A} \overset{\text{reach}}{\Longrightarrow} \mathtt{A}+$ indicates that $\mathtt{A}$ can schedule $\mathtt{A}+$, immediately or intermediately. (2)

2.1.3 Event Conflicts and Independence Time Limits

Events a and b in a pending-event set $\mathbf{E}$, which have event types $\mathtt{A}$ and $\mathtt{B}$, are in conflict if:

1. $\mathtt{A}$ and $\mathtt{B}$ have a DDD, or
2. $\mathtt{A}$ and $\mathtt{B}$ have an IDD, and there is an event type $\mathtt{A}+$, such that $\mathtt{A}+$ is reachable from $\mathtt{A}$, $\mathtt{A}+$ and $\mathtt{B}$ have a DDD, and an $\mathtt{A}+$ event, $a+$, can be scheduled with a TS $t_{a+} \leq t_b$.

If a pair of events does not have an event conflict, they are mutually independent. A time limit specific to the event types, from which these events are instantiated, quantifies this independence:

Definition 3 (Event-Type-Pair Independence Time Limit (ITL))

For a model G with event types $\mathbf{V}^{\mathbf{G}} := \{\mathtt{V}_1^G, \ldots, \mathtt{V}_n^G\}$ and

alias types $\mathtt{A}, \mathtt{B}, \mathtt{B}'$, $\mathtt{A} \mapsto \mathtt{V}_i^G$, $\mathtt{B} \mapsto \mathtt{V}_j^G$, $\mathtt{B}' \mapsto \mathtt{V}_k^G$, $i, j, k \in \{1, \ldots, n\}$,

$$\mathbf{B}' := \left\{ \mathtt{V}_\ell^G \in \mathbf{V}^{\mathbf{G}} \mid \mathtt{A} \overset{\text{reach}}{\longrightarrow} \mathtt{V}_\ell^G \ \wedge \ \mathtt{V}_\ell^G \overset{\text{DDD}}{\longleftrightarrow} \mathtt{B}, \ \ell \in \{1, \ldots, n\} \right\} :$$

$$\delta_{\mathtt{A},\mathtt{B}}^G := \begin{cases} \text{time}\left(\mathtt{A} \overset{\text{min}}{\Longrightarrow} \mathbf{B}' \right), & \text{if } \mathbf{B}' \neq \emptyset, \\ \infty, & \text{otherwise,} \end{cases}$$

where $\mathtt{A} \overset{\text{min}}{\Longrightarrow} \mathbf{B}'$ is the minimum of all shortest paths,

from $\mathtt{A}$ to each $\mathtt{B}' \in \mathbf{B}'$, using minimum edge delays. (3)

2.1.4 Ready-Event Set

The set of ready events is defined mathematically in definition 4, for a specific DES model G, at a specific point in the simulation of G, using the terms defined as follows:

- G: the DES model, n event types,
- $\mathbf{V}^G$: the set of event types in G, size n,
- $\mathbf{E}$: the timestamp-sorted pending-event set in a simulation of G,

- $\mathbf{R}$: the set of ready events in a simulation of G,
- e_k: the k^{th} event in $\mathbf{E}$, $k \in \{1,\ldots,m\}$,
- $\mathtt{V}^G_{kk}$: the event type of e_k, $k \mapsto kk$, $kk \in \{1,\ldots,n\}$,
- t_k: the TS of event e_k,
- e_j: the j^{th} event in $\mathbf{E}$, $j \in \{1,\ldots,k-1\}$, for $k>1$,
- $\mathtt{V}^G_{jj}$: the event type of e_j, $j \mapsto jj$, $jj \in \{1,\ldots,n\}$,
- t_j: the TS of event e_j,
- $\delta^G_{jj,kk}$: the ITL for the event-type pair $(\mathtt{V}^G_{jj},\mathtt{V}^G_{kk})$, and
- $t_k - t_j < \delta^G_{jj,kk}$: a condition for e_k to be in $\mathbf{R}$, specifying that the TS difference between events e_k and e_j is less than the ITL for the event-type pair $(\mathtt{V}^G_{jj},\mathtt{V}^G_{kk})$.

Definition 4 (DDA-DES Ready-Event Set $(\mathbf{R})$)

$$\mathbf{R} := \left\{ e_k \in \mathbf{E} \;\middle|\; \begin{array}{l} \forall e_j \in \mathbf{E},\ j<k, \\ t_k - t_j < \delta^G_{jj,kk} \end{array} \right\} \tag{4}$$

2.2 Related Work on Event-Level Parallelization

Prior work on event-level DES parallelization was domain-specific, did not parallelize SV updates, or did not temporally bound event conflicts. Jones (1986), Jones et al. (1989) proposed an early alternative called "concurrent simulation," which finds parallelization through knowledge of scheduling behavior but cannot parallelize SV updating. Chen et al. (2012), Chen et al. (2014) introduced an out-of-order PDES simulator for the SystemC and SpecC system-level description languages, leveraging data-dependence analysis between events to execute them out of order, but this was limited to specific programming languages and applications. Kunz et al. (2012) experimented with probabilistic heuristics to detect causality relationships, but this approach appears useful only for specific types of simulations after a training phase. Quaglia and Baldoni (1999) introduced weak causality (WEC), which identifies causally-unordered events via SV-access analysis. WEC detects direct conflicts between event pairs but does not implement a mechanism for detecting indirect conflicts caused by scheduling dependencies. Unlike our approach, WEC is not a parallelization technique, and no experimental validation is provided. Stoffers et al. (2015) used data dependencies and scheduling dependencies to find event-level parallelism. However, their approach does not consider time-bounded event-type reachability, which may critically limit parallelism in many DES models.

Our work differs from these approaches by providing a general-purpose framework that can be applied to any discrete-event simulation model, which may be represented in an event graph, dynamically identifying ready events during runtime based on rigorous data-dependence analysis.

3 IMPLEMENTATION OF DDA-PDES

This section describes the implementation of our DDA-PDES framework, including the data structures and algorithms that facilitate DDA-PDES.

3.1 Independence Time Limit (ITL) Table Generation

A critical component of our approach is the Independence Time Limit (ITL) table, which provides constant-time lookup of causality relationships between event types. For a model with n event types, this table has $O(n^2)$ size but provides $O(1)$ access time. Prior to generating the ITL table, some components must be explicitly defined in the simulation model, for each event type i: (1) the sets of input and output SVs $\mathbf{I}_i$ and $\mathbf{O}_i$, and (2) the minimum scheduling delays for all successor events. Correctness of these definitions is

essential to maintain causality. Currently, these components are defined manually for each event type, but it may be possible to automate this key process. The generation of the ITL table occurs in three phases, as shown in algorithm 1 and algorithm 2, not counting the preliminary generation of the Floyd-Warshall (F-W) shortest-paths table. The algorithms presented here improve upon the earlier work in Jensen (2023) by extending functionality to a broader class of DES models.

Phase 0 determines which vertices are reachable from each vertex in the event graph. Phase 1 then computes the initial ITL values by analyzing which events can affect each other through state variable access patterns. For each vertex pair (j,k), the minimum time needed for an earlier j-type event to affect a later k-type event, through direct or indirect data dependencies, is the phase-1 ITL value. Phase 2 then refines these values by considering, for each vertex pair (h,i), how a later i-type event may affect immediately an earlier h-type event or any events an earlier h-type event may schedule before the execution of the later i-type event. The outer loop of each phase is independent and parallelizable. Our implementation uses OpenMP for parallelization. The ITL-generation algorithm's time complexity is $O(n^3)$, but this is decoupled from simulation performance. Using the AMD system described in section 4.2, ITL-table generation including F-W takes about 0.6 s and 264 s for 512-node and 4096-node 3D Torus NoC models, respectively, using baseline algorithms. The 4096-node time can be reduced to 48 s with CPU-optimized code and to 9 s using an NVIDIA RTX 3070 GPU. Note, if any event routine i is modified such that $\mathbf{I}_i$ or $\mathbf{O}_i$ is changed, or a minimum scheduling delay is changed, the ITL table must be updated.

3.2 DDA Calendar Queue

Key to our parallel execution strategy is DDA-CQ, a novel parallel calendar queue for managing the pending-event set, designed to support efficient parallel operations in the DDA-PDES framework. Our design utilizes several key features:

1. **Contiguous Memory Layout**: Unlike traditional calendar queues that use arrays of linked lists, our implementation stores events in a single contiguous array indexed by bucket ID and offset, which facilitates efficient ready-event identification and parallel event scheduling.
2. **Parallel Access**: Contention is minimized through strategic use of atomic operations for bucket indices, using spin locks only for bucket overflow lists and the far-future list.
3. **Adaptive Bucket Width**: The queue dynamically adjusts bucket width to maintain a useful number of pending events in each bucket, avoiding sparsity and overflow.

3.3 Parallel Execution Strategy

The execution strategy uses OpenMP to manage a thread pool that executes ready events in parallel. The main execution loop is implemented as shown in algorithm 3. In each execution iteration, parallel worker threads process events in the first bucket of the calendar queue. Each thread examines the events and determines their readiness by consulting the Independence Time Limit (ITL) table. An event is ready if, for each preceding event in the timestamp-sorted first bucket, the event-pair ITL is greater than the timestamp delta. In the parallel implementation, the ready-event set is restricted to the first bucket in the DDA-CQ.

`CheckHasEvents()` identifies the first bucket, partitions the first bucket to group non-empty events together at the front, sorts these events by timestamp, and establishes a partition point that serves as the working bucket size. `UpdateFirstBucket()` resets executed first-bucket events, integrates any buffered immediate events into the first bucket, and triggers adaptive operations including bucket width adjustments and bucket expansion.

Algorithm 1: Independence Time Limit (ITL) Table - Phase 0 & 1.

Input: $shortestPaths, I_s, O_s$
Output: ITL, Rs

$ITL \leftarrow$ zeros matrix of size $numVertices^2$
// Independence Time Limit table
$Rs \leftarrow$ array of $numVertices$ empty lists
// Reachability sets
// Phase 0: Compute reachable vertices
for $l \leftarrow 0$ **to** $numVertices - 1$ **do**
  $lmPaths \leftarrow$ empty list
  for $m \leftarrow 0$ **to** $numVertices - 1$ **do**
    if $shortestPaths[l][m] < \infty$ **then**
      append m to $lmPaths$
  $Rs[l] \leftarrow lmPaths$
// Phase 1: Compute initial ITL values
for $k \leftarrow 0$ **to** $numVertices - 1$ **do**
  $S_k \leftarrow \text{SetUnion}(I_s[k], O_s[k])$
  // State variable set S_k = Input SVs $\cup$ Output SVs
  $U_{S_k} \leftarrow$ empty list // Vertices that can update SVs in S_k
  for $l \leftarrow 0$ **to** $numVertices - 1$ **do**
    $O_{l\_S_k} \leftarrow \text{SetIntersection}(O_s[l], S_k)$
    if $O_{l\_S_k}$ *is not empty* **then**
      append l to U_{S_k}
  for $j \leftarrow 0$ **to** $numVertices - 1$ **do**
    $X_{jk} \leftarrow \text{SetIntersection}(Rs[j], U_{S_k})$
    // Vertices reachable from j that affect k
    if X_{jk} *is not empty* **then**
      $T_{jx} \leftarrow$ empty list
      for *each x in* X_{jk} **do**
        append $shortestPaths[j][x]$ to T_{jx}
      $ITL[j][k] \leftarrow \min(T_{jx})$
    else
      $ITL[j][k] \leftarrow \infty$
return ITL, Rs

Algorithm 2: Independence Time Limit (ITL) Table - Phase 2.

Input: $shortestPaths, ITL, Rs$
Output: ITL

// Phase 2: Refine ITL values
for $h \leftarrow 0$ **to** $numVertices - 1$ **do**
  for $i \leftarrow 0$ **to** $numVertices - 1$ **do**
    $Z_i \leftarrow$ empty list // Vertices that i can affect immediately (ITL = 0)
    for $l \leftarrow 0$ **to** $numVertices - 1$ **do**
      if $ITL[i][l] = 0$ **then**
        append l to Z_i
    $X_{hi} \leftarrow \text{SetIntersection}(Rs[h], Z_i)$
    // Reachable vertices with immediate effect
    if X_{hi} *is not empty* **then**
      $T_{hi} \leftarrow$ empty list
      for *each x in* X_{hi} **do**
        append $shortestPaths[h][x]$ to T_{hi}
      if $\min(T_{hi}) < ITL[h][i]$ **then**
        $ITL[h][i] \leftarrow \min(T_{hi})$
        // Update if phase-two value is smaller
return ITL

Algorithm 3: Parallel Out-of-Order Simulation Execution.

```
   Input: ITL table, event buckets
   Output: Simulation statistics
1  simTime ← 0 run ← CheckHasEvents(true)
2  UpdateFirstBucket(simTime)
   // Begin parallel region
3  while run do
4      buckets, startIdx, partitionPoint ← GetFirstBucket()
       // Parallel for loop with dynamic scheduling
5      for j ← 0 to partitionPoint − 1 in parallel do
6          if buckets[startIdx + j].status = 0 then
7              eventReady ← true
8              vertexID ← buckets[startIdx + j].vertexID
9              timestamp ← buckets[startIdx + j].time
10             for i ← 0 to j − 1 do
11                 depVertexID ← buckets[startIdx + i].vertexID
12                 depTimestamp ← buckets[startIdx + i].time
13                 indepLimit ← ITL[vertexID][depVertexID]
14                 if timestamp − depTimestamp ≥ indepLimit then
15                     eventReady ← false
16                     break
17             if eventReady then
18                 Execute(buckets[startIdx + j])
       // Synchronization point - single thread updates
19     UpdateFirstBucket(simTime)
20     run ← CheckHasEvents()
```

4 EXPERIMENTAL SETUP

Multiple simulation models and multiple platforms were used to evaluate the performance of our DDA-PDES implementation. The DDA-PDES simulation framework and models were developed in C++ and compiled with the GNU Compiler Collection (GCC) using `O3` optimization.

4.1 Simulation Models

To evaluate our DDA-PDES framework, we implemented multiple packet-routing network models with different topologies and capabilities, and also the PHOLD benchmark. Each model exercises different aspects of our event-level parallelism approach, with varying complexity and communication patterns. The 2D and 3D models feature adaptive routing, similar to strategies employed for network-on-chip (NoC) packet routing in (Ascia et al. 2008; Mak et al. 2010; Trik et al. 2021).

PHOLD Benchmark: Standard PDES benchmark with fully-connected topology, 4096 nodes.

Ring Network: One-dimensional ring with bidirectional packet routing, 4096 nodes (8192 event types).

2D Irregular NoC Network: Mixed topology with local mesh and express links. Parameterized probability of local connectivity. Features adaptive routing based on current n-hop-neighbor queue states and propagated historical queue information from incoming packets, 4096 nodes (8192 event types).

3D Torus NoC Network: Regular wrapped mesh with six neighbors per node. Features the same adaptive routing as the 2D Irregular NoC Network models, 4096 nodes (8192 event types).

Table 1 provides an event-routine analysis from the Python Lizard library (Terryin/Lizard), to estimate the computational intensity of each model and the speedup capability with the DDA-PDES implementation.

Table 1: Cyclomatic complexity and size metrics for DES model event routines.

Model	NLOC	CCN	Tokens
PHOLD	13	3	119
Ring	111	36	811
2D Irregular NoC	462	119	3495
3D Torus NoC	453	126	4240

4.2 Hardware Platforms

We conducted experiments on three different hardware platforms:

- **AMD Ryzen 5 3600**: x86_64 architecture, 6 cores, 4.2 GHz max clock, 192 KiB L1d cache × 6, 192 KiB L1i cache × 6, 3 MiB L2 cache × 6, 32 MiB L3 cache × 2, 16 GB DDR4 3200 MHz memory, Manjaro Linux 24.0.8, GCC 14.2.1
- **Qualcomm Snapdragon X Elite**: aarch64 architecture, 12 cores, 3.8 GHz max clock, 1.1 MiB L1d cache × 12, 2.3 MiB L1i cache × 12, 144 MiB L2 cache × 12, 14 GB LPDDR5X 4224 MHz memory, WSL 2 running Ubuntu 24.04.2 LTS on Windows 11, GCC 13.3.0
- **Intel Xeon Gold 6148**: x86_64 architecture, 40 cores (2 sockets, 20 cores/socket), 2.4 GHz max clock, 1.3 MiB L1d cache × 40, 1.3 MiB L1i cache × 40, 40 MiB L2 cache × 40, 55 MiB L3 cache × 2, 384 GB DDR4 2666 MHz memory, Rocky Linux 9.3 (Blue Onyx), GCC 11.4.1

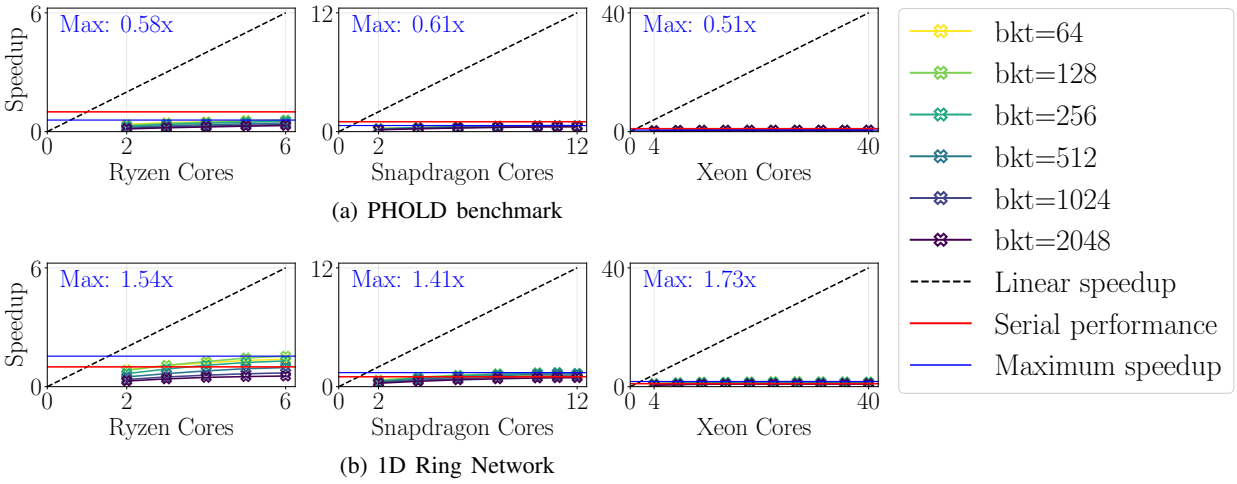
5 RESULTS AND ANALYSIS

This section presents the performance results of our DDA-PDES implementation across different hardware platforms and simulation models, and provides comparisons with similar SD-PDES model simulations. Each experimental parallel run is validated against a corresponding serial run, comparing: (1) the final simulation-clock time, (2) the final SV values, (3) the number of event executions in total and by type, and (4) if applicable, the global mean packet delay, to full double precision. All parallel runs are consistent with corresponding serial runs, based on these criteria. In our previous work (Jensen et al. 2025) that evaluated the available parallelism in DES models, similar to those used in this work, ready events were identified and executed serially, out-of-order. In that work, all 1,900 serial out-of-order runs were found to be consistent with corresponding serial in-order runs, comparing event-type-level simulation traces that logged the timestamp and values of all input and output SVs at each event execution.

5.1 Speedup Analysis

Figures 1 and 2 show the speedup achieved by our DDA-PDES implementation on the Ryzen, Snapdragon, and Xeon platforms. Serial performance is measured from two different serial implementations, using two different pending-event set data structures: (1) a C++ `std::multiset`, and (2) a fixed-width calendar queue (Brown 1988). For each testing configuration, the mean of five parallel runs is compared against the best serial run, which may be from either serial implementation. For the more computationally-lightweight models, the PHOLD benchmark and the ring network, the calendar queue implementations are about 20%

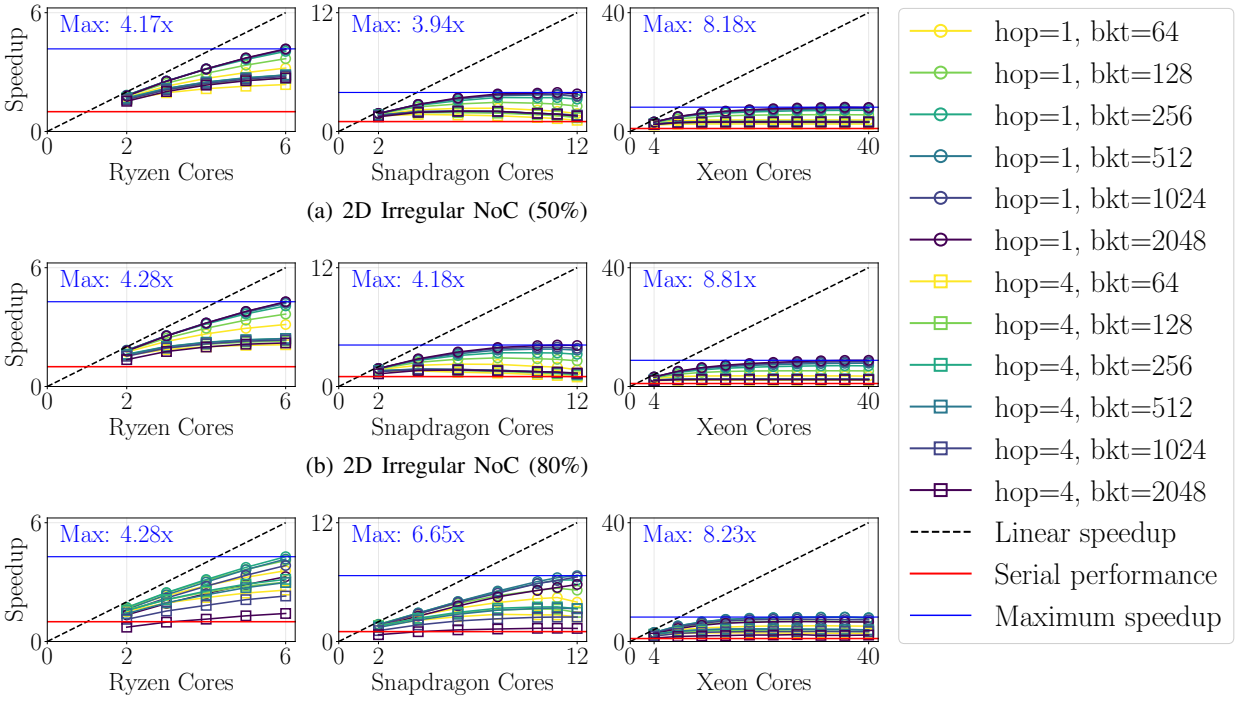
and 13% faster than the `std::multiset` implementations. For the other network models, the differences are minimal, as performance is not bound by event-set management.



(a) PHOLD benchmark

(b) 1D Ring Network

Figure 1: Grouped speedup plots for the PHOLD benchmark and 1D ring network. Multiple calendar queue bucket sizes are tested. Speedup is non-existent or minimal with lightweight events.



(a) 2D Irregular NoC (50%)

(b) 2D Irregular NoC (80%)

(c) 3D Torus Network

Figure 2: Grouped speedup plots for the 2D irregular NoC networks (50% and 80% local connectivity) and 3D torus topology, 1-hop-aware and 4-hop-aware configurations. Multiple calendar queue bucket sizes are tested. DDA-PDES can parallelize the computationally-intensive routing algorithms.

The PHOLD and Ring models find no speedup and minimal speedup, respectively, as the parallelization overhead is more significant than the event routines. The 2D and 3D models, which feature computationally-

intensive adaptive routing algorithms, find maximum speedup of 4.28, 6.65, and 8.82 on these platforms with 6, 12, and 40 cores. Maximum parallel efficiency is 0.91. Generally, the 1-hop-aware models parallelize better than the 4-hop-aware models because there are more independent events for the threads to execute. Similarly, using larger bucket sizes exposes the threads to more ready events, but performance returns diminish with increasing bucket size, as the ready-event identification process has $O(n^2)$ time complexity.

5.2 ROSS Comparison

For comparison with SD-PDES, we implemented similar 3D torus models for the ROSS simulation system (Carothers et al. 2002). Figure 3 shows a comparison between our DDA-PDES implementation and ROSS for the 3D torus network model with both 1-hop-aware and 4-hop-aware configurations. The ROSS models were designed to minimize message passing by storing queue-size updates in shared memory, for LPs belonging to the same process. Further, a parameter defines how often an LP has to broadcast queue-size updates to inter-process LPs, based on how much its queue size has changed. Additionally, queue-size update messages are allowed to be stale, similar in scale to the defined lookahead value. All ROSS runs use 8 MPI processes; using more than 8 processes resulted either in slower run times, likely due to poor decomposition, or failure due to out-of-memory errors. The DDA model is able to use up to 12 cores and achieves increased speedup beyond 8 cores, so that data is included in fig. 3.

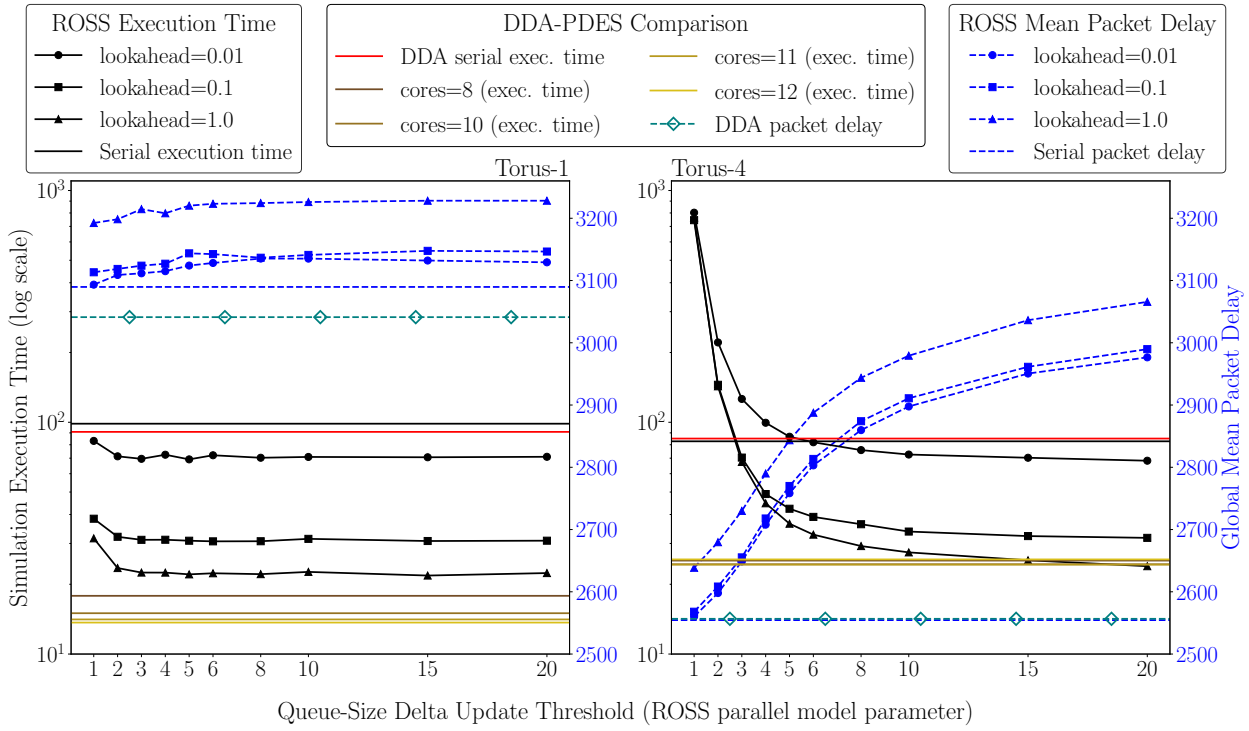


Figure 3: Performance comparison between DDA-PDES and ROSS for 3D torus network model, 1-hop-aware (Torus-1) and 4-hop-aware configurations (Torus-4). The ROSS parallel model uses (1) a queue-size delta update threshold parameter and (2) a queue-size update message delay similar to lookahead, to alleviate communication burden while potentially maintaining routing efficacy. The fully shared-memory DDA model does not have to make these compromises.

The 1-hop ROSS model can parallelize somewhat effectively, if some degradation to the packet-routing capability is permitted. For example, using a queue-size delta update threshold of 2 and a lookahead of

Table 2: Speedup and error for Torus-4 on ROSS, multiple lookahead and update threshold values.

Update Threshold	Lookahead = 0.01		Lookahead = 0.1		Lookahead = 1.0	
	Speedup	Error (%)	Speedup	Error (%)	Speedup	Error (%)
1.0	0.10	0.22	0.11	0.53	0.11	3.29
2.0	0.37	1.70	0.57	2.11	0.58	4.90
3.0	0.66	3.69	1.18	3.94	1.23	6.87
⋮	⋮	⋮	⋮	⋮	⋮	⋮
10.0	1.14	13.43	2.45	13.95	3.01	16.63
15.0	1.18	15.50	2.56	15.93	3.26	18.86
20.0	1.21	16.52	2.61	17.04	3.46	20.01

0.1, the ROSS model achieves a speedup of 3.08 with a mean packet delay error of 0.94%. On the same platform, the comparable DDA model achieves a speedup of up to 6.65, with no deviation from the serial mean packet delay. However, for the 4-hop model, the ROSS model can achieve significant speedup only with significant degradation to the routing capability. For example, using a lookahead of 1.0 and an update threshold of 10, speedup is 3.01, and mean packet delay error is 16.63%. On the same platform, the comparable DDA model achieves a speedup of 3.5, with no deviation from the serial mean packet delay. Complete metrics for the ROSS 4-hop torus model are in table 2.

6 CONCLUSION AND FUTURE WORK

We presented the design and performance evaluation of Data-Dependence Analysis Parallel Discrete-Event Simulation (DDA-PDES), a simulation system that builds on theoretical foundations for identifying independent events through data-dependence analysis. Our implementation is a shared-memory simulation engine that dynamically detects and executes ready events in parallel. Rather than utilizing spatial decomposition and multiple logical processes, simulation threads parallelize model execution using a precomputed lookup table that encapsulates event dependencies. Not limited to a narrow model class, DDA-PDES can parallelize a wide range of DES models, given they have event routines that are sufficiently computationally intensive to outweigh the overhead of the parallelization method.

Our experimental results across three hardware platforms and multiple simulation models demonstrate that DDA-PDES can achieve significant speedup over serial execution and superior performance compared with SD-PDES, for some DES models. The approach may be beneficial for models with complex data dependencies that are not easily decomposable into separate logical processes, including high-resolution atomic models such as biomedical models with fast and slow processes. Future work may explore: (1) parallelization of additional classes of problems, which feature long-range dependencies and computationally-intensive events, (2) integration of shared-memory DDA parallelization into SD-PDES methods, and (3) automated methods for identifying SV sets and minimum scheduling delays, for expediency and correctness.

REFERENCES

Andelfinger, P., and W. Cai. 2022. "Advanced Tutorial: Parallel and Distributed Methods for Scalable Discrete Simulation". In *2022 Winter Simulation Conference (WSC)*, 268–282 https://doi.org/10.1109/WSC57314.2022.10015291.

Ascia, G., V. Catania, M. Palesi, and D. Patti. 2008. "Implementation and Analysis of a New Selection Strategy for Adaptive Routing in Networks-on-Chip". *IEEE Transactions on Computers* 57(6):809–820.

Brown, R. 1988. "Calendar Queues: A Fast 0 (1) Priority Queue Implementation for the Simulation Event Set Problem". *Communications of the ACM* 31(10):1220–1227.

Carothers, C. D., D. Bauer, and S. Pearce. 2002. "ROSS: A High-Performance, Low-Memory, Modular Time Warp System". *Journal of Parallel and Distributed Computing* 62(11):1648–1669.

Chen, W., X. Han, C.-W. Chang, G. Liu, and R. Dömer. 2014. "Out-of-Order Parallel Discrete Event Simulation for Transaction Level Models". *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 33(12):1859–1872.

Chen, W., X. Han, and R. Dömer. 2012. "Out-of-Order Parallel Simulation for ESL Design". In *2012 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. March 12th-16th, Dresden, Germany, 141-146.

Fujimoto, R. M. 1990. "Parallel Discrete Event Simulation". *Communications of the ACM* 33(10):30–53.

Fujimoto, R. M., R. Bagrodia, R. E. Bryant, K. M. Chandy, D. Jefferson, J. Misra, *et al*. 2017. "Parallel discrete event simulation: The making of a field". In *2017 Winter Simulation Conference (WSC)*, 262–291 https://doi.org/10.1109/WSC.2017.8247793.

Jensen, E. J. 2023. "An Algorithm for Finding Data Dependencies in an Event Graph". In *Modeling, Simulation and Visualization Student Capstone Conference*. April 20th, Suffolk, VA, USA, 1-13.

Jensen, E. J., J. Leathrum, James, C. Lynch, K. Smith, and R. Gore. 2025. "Out of Order and Causally Correct: Ready-Event Discovery through Data-Dependence Analysis". In *Proceedings of the 39th ACM SIGSIM Conference on Principles of Advanced Discrete Simulation*, SIGSIM-PADS '25, 88–98. New York, NY, USA: Association for Computing Machinery https://doi.org/10.1145/3726301.3728416.

Jones, D., C.-C. Chou, D. Renk, and S. Bruell. 1989. "Experience With Concurrent Simulation". In *1989 Winter Simulation Conference Proceedings*, 756–764 https://doi.org/10.1109/WSC.1989.718751.

Jones, D. W. 1986. "Concurrent simulation: an alternative to distributed simulation". In *Proceedings of the 18th Conference on Winter Simulation*, WSC '86, 417–423. New York, NY, USA: Association for Computing Machinery https://doi.org/10.1145/318242.318468.

Kunz, G., M. Stoffers, J. Gross, and K. Wehrle. 2012. "Know Thy Simulation Model: Analyzing Event Interactions for Probabilistic Synchronization in Parallel Simulations". In *5th International ICST Conference on Simulation Tools and Techniques 2012 (SIMUTools 2012)*. March 19th-23rd, Desenzano del Garda, Italy, 119-128.

Mak, T., P. Y. Cheung, K.-P. Lam, and W. Luk. 2010. "Adaptive Routing in Network-on-Chips Using a Dynamic-Programming Network". *IEEE Transactions on Industrial Electronics* 58(8):3701–3716.

Perumalla, K., M. Bremer, K. Brown, C. Chan, S. Eidenbenz, K. S. Hemmert, *et al*. 2022. "Computer Science Research Needs for Parallel Discrete Event Simulation (PDES)". Technical report, Lawrence Livermore National Lab.(LLNL), Livermore, CA, USA.

Quaglia, F., and R. Baldoni. 1999. "Exploiting Intra-Object Dependencies in Parallel Simulation". *Information Processing Letters* 70(3):119–125.

Rahman, S., N. Abu-Ghazaleh, and W. Najjar. 2017. "PDES-A: A Parallel Discrete Event Simulation Accelerator for FPGAs". In *Proceedings of the 2017 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation*. May 24th-26th, Singapore, Republic of Singapore, 133-144.

Stoffers, M., T. Sehy, J. Gross, and K. Wehrle. 2015. "Analyzing Data Dependencies for Increased Parallelism in Discrete Event Simulation". In *Proceedings of the 3rd ACM SIGSIM Conference on Principles of Advanced Discrete Simulation*. June 10th-12th, London, UK, 73-74.

Terryin/Lizard. "Lizard: A Simple Code Complexity Analyzer". https://github.com/terryyin/lizard. Accessed: 10 April 2025.

Trik, M., S. Pour Mozaffari, and A. M. Bidgoli. 2021. "Providing an Adaptive Routing Along with a Hybrid Selection Strategy to Increase Efficiency in NoC-Based Neuromorphic Systems". *Computational Intelligence and Neuroscience* 2021(1):8338903.

AUTHOR BIOGRAPHIES

ERIK J. JENSEN is a modeling and simulation engineering Ph.D. candidate at Old Dominion University. Research activity includes parallel simulation, machine learning for LLM response validation, high-performance computing, and virtual laboratory environments for STEM education. His email address is ejens005@odu.edu, and his software is at https://github.com/ejensODU/.

JAMES F. LEATHRUM is an Associate Professor in the Department of ECE at Old Dominion University. He earned a Ph.D. in Electrical Engineering from Duke University. His research interests include simulation software design, distributed simulation, simulation-based test & evaluation, and simulation education. His email address is jleathru@odu.edu.

CHRISTOPHER J. LYNCH is a Research Assistant Professor at the Virginia Modeling, Analysis, and Simulation Center (VMASC) at Old Dominion University. He holds a Ph.D. in modeling and simulation. His email address is cjlynch@odu.edu.

KATHERINE SMITH is a Research Assistant Professor at the Virginia Digital Maritime Center at Old Dominion University (ODU). She has a Ph.D. in modeling and simulation engineering. Her email address is k3smith@odu.edu.

ROSS GORE is a Research Associate Professor at the Center for Secure and Intelligent Critical Systems at Old Dominion University (ODU). His website is https://rossgore.github.io/ and his email address is rgore@odu.edu.