IEEE Open Access Journal of
Power and Energy

Received 14 June 2022; revised 27 September 2022 and 12 November 2022; accepted 15 November 2022.
Date of publication 8 December 2022; date of current version 3 February 2023.

Digital Object Identifier 10.1109/OAJPE.2022.3227800

A Configurable Hierarchical Architecture for Parallel Dynamic Contingency Analysis on GPUs

CONG WANG[1] (Student Member, IEEE), SUANGSHUANG JIN[1] (Senior Member, IEEE), RENKE HUANG[2] (Senior Member, IEEE), QIUHUA HUANG[2] (Member, IEEE), AND YOUSU CHEN[2] (Senior Member, IEEE)

[1]School of Computing, Clemson University, North Charleston, SC 29410 USA
[2]Pacific Northwest National Laboratory, Richland, WA 99354 USA

CORRESPONDING AUTHOR: C. WANG (cong2@clemson.edu)

This work was supported by the U.S. Department of Energy (DOE) Advanced Grid Modeling (AGM) Program through the Pacific Northwest National Laboratory (PNNL) under Contract 20-201-xxxx-0974-205-2014159.

ABSTRACT Dynamic contingency analysis (DCA) for modern power systems is fundamental to help researchers and operators look ahead of potential issues, arrange operational plans, and improve system stabilities. However, since the system size and the number of contingency scenarios continue to increase, pursuing more effective computational performance faces many difficulties, such as slow algorithms and limited computing resources. This research accelerates the intensive computations of massive DCAs by implementing a two-level hierarchical computing architecture on graphical processing units (GPUs). The performance of the designed method is examined using four test systems of different sizes and compared with a CPU-based parallel approach. The results show up to 2.8x speedup using one GPU and 4.2x speedup using two GPUs, respectively. More accelerations can be observed once more GPUs are configured. It demonstrates that the proposed architecture can significantly enhance the overall computational performance of massive DCAs while maintaining a strong scaling capability under various resource configurations.

INDEX TERMS Dynamic contingency analysis, graphic processing unit, parallel computing, multiprocessing, speedup.

I. INTRODUCTION

DYNAMIC contingency analysis (DCA) is a critical function for Energy Management Systems (EMS) to identify potential physical limitations and system stability constraints in diverse scenarios with power system component failures [1]. Static contingency analysis (CA) only considers the system's steady-state behavior in a certain snapshot under different contingencies. However, DCA, which seeks to assess the ability of a grid to withstand cascading component failures or contingencies within a specified duration (a few seconds) [2], is paramount for the reliable operation of modern power systems, underpinning system stability and economic prosperity [3]. In massive DCAs, each grid configuration with component outages is evaluated as a scenario, whose feasibility is obtained by the solution of a power system dynamic simulation. If any collapsed consequences are detected, preventive and corrective operations should be immediately implemented based on the DCA results [4]. Finishing these tasks implies complex computations on system topology and dynamics. The time-consuming simulations result in a long wait time before the operators can assess the situation and make a timely decision. This latency hinders timely operations in power grids, where quick reactions are necessary to adjust plans and strategies. Therefore, a computationally efficient solution to massive DCAs is of great significance.

High-performance computing (HPC) leveraging advanced computing architectures holds great promise to overcome this computational barrier to fit the need for fast responses in online operations. Many existing parallelization approaches in accelerating the performance of power system DCA simulation are performed on the central processing unit (CPU)-based multi-core computers or clusters. For example, [5], [6], and [7] acquire desirable speedup of a single

dynamic contingency case using 64 cores. A two-level dynamic security assessment under uncertainty is introduced in [8]. Additionally, the dynamic contingency analysis tool (DCAT) developed in [9] bridges an outer-inner module to perform multiple contingencies, where the actual completion time is extremely host resource-dependent. In recent years, as the graphics processing unit (GPU) began to dominate the HPC area due to its exceptional computation capability, many intractable problems in science and engineering that have traditionally demanded a high-end supercomputer are now being solved on cost-effective GPUs alternatively [10], [11], [12], [13], [14]. In the power system domain, [15] proposes a novel method for transient stability simulation on multiple GPUs. [4] designs a hybrid computing architecture for real-time CA. Reference [16] uses a two-layered parallel static security assessment for large-scale power grids based on GPUs. Nonetheless, an advanced GPU-based implementation for parallel DCA is still missing.

This paper presents a configurable and hierarchical computing architecture for the first-ever accelerating massive DCAs in parallel on single or multiple GPUs. The higher level dispatches the available worker processes to the tasks at once. At its lower level, each single case in terms of GPU-optimized dynamic simulation is further programmed to use the provisioned resources on its attached process for data-level parallelism. The executions on one or more GPUs can be configured to host the DCA runs with enhanced flexibility and extended scalability. The main contributions of the paper are:

- Improved single DCA algorithm and GPU-based implementation using high-level computing libraries.
- The design for parallelizing massive DCAs through GPU resource partitioning on a hierarchical architecture.
- Configurable and universal adaptiveness of the proposed architecture with minimized hardware network communications.

The organization of the rest of this paper encompasses: 1) an overview of DCA's core algorithm and relevant GPU computing techniques in Section II; 2) the architecture design and code implementation for the data-level and task-level parallelisms on GPUs in Section III; 3) the case studies including the performance comparisons on CPU and GPU using different test systems, and the scalability analysis regarding extendable hardware configurations in Section IV; and 4) a conclusion of the current research outcomes with proposed future enhancement in Section V.

II. BACKGROUND

With increasing renewable penetration and more frequent extreme weather events, there are exponential amounts of credible contingencies in power systems that should be analyzed in detail, imposing a substantial computational burden for real-time operations. This section reviews the general DCA mathematical model and parallel computing technologies on GPU, respectively.

A. SINGLE DYNAMIC CONTINGENCY ANALYSIS

The core computation in a single power system DCA is a dynamic simulation that generally consists of nodal admittance matrix (full $\mathbf{Y}$ matrix) manipulations and multiple time-step simulations. It requires a computationally intensive time-domain solution of numerous differential and algebraic equations (DAEs) for a short period of time (e.g., 30 seconds) as shown in (1),

$$\begin{cases} \dot{\mathbf{x}} = \mathbf{f}(\mathbf{x}, \mathbf{u}) \\ \mathbf{0} = \mathbf{g}(\mathbf{x}, \mathbf{u}) \end{cases} \tag{1}$$

where the vector $\mathbf{x}$ represents dynamic state variables such as generator rotor angles and speeds, and the vector $\mathbf{u}$ represents algebraic variables such as the network bus voltage magnitudes and phase angles, and real and imaginary parts of the bus voltage [17]. Given a power system with N buses and M generators, the algebraic equations in (1) can be represented by (2),

$$\mathbf{Y}_{NN} * \mathbf{V}_N = \mathbf{I}_N \tag{2}$$

where the vector of current injections at each bus $\mathbf{I}_N$ is the product of the full $\mathbf{Y}$ matrix $\mathbf{Y}_{NN}$ and the vector of bus voltages $\mathbf{V}_N$. To simplify the complexity of the matrix and achieve best matrix operation performance, for a power system with classical model and constant impedance load, $\mathbf{Y}_{NN}$ can be reduced to only contain generator internal buses, $\mathbf{Y}'_{MM}$. According to [18] and [19], (3) represents the logic of acquired reduced nodal admittance matrix (reduced $\mathbf{Y}$ matrix),

$$\mathbf{Y}'_{MM} = \mathbf{Y}_{MM} - \mathbf{Y}_{MN} * \mathbf{Y}_{NN}^{-1} * \mathbf{Y}_{NM} \tag{3}$$

where $\mathbf{Y}_{MM}$ is the matrix storing the resistance and reactance of generators, $\mathbf{Y}_{MN}$ is the links between generator internal buses and terminal buses, $\mathbf{Y}_{NN}$ contains constant load impedance and generator transient impedance, and $\mathbf{Y}_{NM}$ is the transpose of $\mathbf{Y}_{MN}$. Thus, the algebraic equations can be modified to (4), where the representations of current injections $\mathbf{I}'_M$ can be reduced to the product of $\mathbf{Y}'_{MM}$ and the vector of generator internal voltages $\mathbf{V}'_M$.

$$\mathbf{Y}'_{MM} * \mathbf{V}'_M = \mathbf{I}'_M \tag{4}$$

The equations of motion for an individual generator a in the complex system could be represented by (5) for a classical generator model.

$$\begin{cases} \dfrac{dw_a}{dt} = \dfrac{w_{sa}}{2H_a}(P_{ma} - P_{ea} - D_a(w_a - w_{sa})) \\ \dfrac{d\theta_a}{dt} = w_a - w_{sa} \end{cases} \tag{5}$$

Here H_a is the inertia constant, w_a is the speed, w_{sa} is the synchronous speed, P_{ma} and P_{ea} are the mechanical power input and active power at the air gap, D_a is the damping coefficient, and θ_a is the angular position of the rotor in the electrical radians with respect to synchronously rotating reference [6].

To solve the DAEs, the differential equation set in (1) needs to be first discretized into algebraic equations, which are then lumped with the original algebraic equations. The Modified Euler (ME) method [20] is usually used to solve these equations explicitly at each time step as shown in (6). The system balance is perturbed by a contingency scenario before (Pre), during (On), and after (Post) a transient disturbance or fault is applied. The $\mathbf{Y}'_{MM}$ generated regarding each phase is then involved in the numerical integration function.

$$\begin{cases} Predict: y_{i+1} = y_i + hf_i \\ Update: y_{i+1} = y_i + \frac{h}{2}[f_i + f_{i+1}] \end{cases} \tag{6}$$

DCA addresses each contingency scenario by deploying one aforementioned dynamic simulation and evaluating the system's dynamic trajectory. As a result, both the realization of parallel execution of multiple DCA cases, and the speeding-up of each single dynamic simulation that dominates one DCA run lead to the notable acceleration of the overall computational performance of massive DCA study.

B. PARALLEL COMPUTING ON GPU

1) DATA-LEVEL PARALLELISM

Modern GPUs have a high data parallel architecture design, which is composed of many cores sharing the same control unit (e.g., 5,120 cores in a Tesla V100 GPU [21]). The primary computing unit of an NVIDIA GPU is called *thread*. All indexed threads grouped in a *block* perform Single Instruction with Multiple Data (SIMD) since each thread in charge of one array element executes the identical formula in parallel. The GPU memory (registers, shared memory, cache, and global memory) is accessible to the threads for performance optimization purposes. A *grid* containing a bunch of threads of blocks is the largest defined structure for GPU programming. These resource definitions, memory layout, and related computations can be controlled by writing a kernel through CUDA [22], a computing platform that allows the software to relieve the intensive CPU load with GPU. In this paper, as the GPU is favorable for the data-level parallel implementation, a single DCA algorithm in terms of dynamic simulation is optimized and accelerated by coding in CUDA Python [23], where the program takes advantage of the Python ecosystem, CUDA driver and runtime APIs, and extensive GPU data structure and parallel computing libraries (e.g., CuPy [24], Numba [25], etc.).

2) TASK-LEVEL PARALLELISM

Currently, large CPU-based distributed systems or clusters are dominating event-driven computing. The focus of using a portable device is still devoted to data-level parallelism, as discussed above. Thus, the concurrency of events is not usually the top priority due to traditional GPU design natures. Recent NVIDIA GPUs (Tesla V100 or A100 [26]) built with Volta architecture or later are capable of allowing applications from multiple processes to overlap, achieving resource partitioning and fast task-level parallelism. In this research, the configurable implementations of such are done through the use of CUDA-aware Message Passing Interface (MPI) [27], a communication standard in parallel computing. It is responsible for launching and designating processes to different dynamic contingencies. Moreover, Multi-Process Service (MPS), which can benefit performance if the GPU computing capacity is underutilized by a single application process, enables each MPI job to have its own memory space but cooperate with the others without interference on the same GPU [28].

Algorithm 1 Original $\mathbf{Y}_{NN}$ and $\mathbf{Y}'_{MM}$ Formations

Input: system data (β_1, β_2), fault condition (nb, fb, Bb, rx)
Output: pre, on, and post fault $\mathbf{Y}'_{MM}$
Original $Y\_reduce()$ for ${\mathbf{Y}'_{MM}}^{i}$ ($i = pre, on, post$)
1: Compute full $\mathbf{Y}$:
$\mathbf{Y}_{NN}{}^{i} = [\beta_1, nb, fb, Bb, rx]^i$
2: Construct $\mathbf{Y}_{MM}$, $\mathbf{Y}_{MN}$, and $\mathbf{Y}_{NM}$:
$[\mathbf{Y}_{MM}, \mathbf{Y}_{MN}, \mathbf{Y}_{NM}]^i = [\beta_2]^i$
3: Solve Eq. (3) for reduced $\mathbf{Y}$:
${\mathbf{Y}'_{MM}}^{i} = [\mathbf{Y}_{NN}, \mathbf{Y}_{MM}, \mathbf{Y}_{MN}, \mathbf{Y}_{NM}]^i$

III. PARALLEL IMPLEMENTATIONS

This section presents a two-level computing architecture, which is designed to account for boosting DCA computations on single or multiple GPUs through both task and data-level parallelism with advanced algorithms and communication control strategy.

A. OPTIMIZATIONS OF SINGLE DCA ALGORITHM

According to Section II. A, the reduced $\mathbf{Y}$ matrix needs to be recalculated once a fault occurs during the dynamic simulation. As shown in Algorithm 1, the standard code version implemented in PST [29], which we use as a baseline reference, reruns each step in the formation of $\mathbf{Y}_{NN}$ and $\mathbf{Y}'_{MM}$ in $Y\_reduce()$ function. It assumes each $\mathbf{Y}'_{MM}$ is a result of two types of variables:

1) β_1 and β_2, which are constant, represent the two collections of power system input, respectively;
2) fb (fault far bus), nb (fault near bus), Bb (shunt susceptance), and rx (reactance) govern the fault condition to be applied in a scenario.

Whereas given the fact that the variables changed for applying a fault are not modifying system parameters, Algorithm 1 can be optimized to mitigate the workload by effectively recycling and calling intermediate results from the pre-fault phase, such as $\mathbf{Y}_{NN}$, $\mathbf{Y}_{MM}$, $\mathbf{Y}_{MN}$, and $\mathbf{Y}_{NM}$. Algorithm 2 demonstrates the new algorithm to obtain all three $\mathbf{Y}'_{MM}$. The $Y\_init()$ function creates necessary arrays at once and retains the unchanged matrices from $\mathbf{Y}_{NN}$ on pre-fault. Consequently, in $Y\_red()$, the operations for on-fault and post-fault can make use of them directly. Compared to the original program, it reduces from 33 logical steps and 69 significant matrix computations to only 11 and 37,

Algorithm 2 Optimized $\mathbf{Y}_{NN}$ and $\mathbf{Y}'_{MM}$ Formations

Input: system data (β_1,β_2), fault condition (nb, fb, Bb, rx)
Output: pre, on, and post fault $\mathbf{Y}'_{MM}$

$Y\_init()$ for ${\mathbf{Y}'_{MM}}^{pre}$
1: Compute and save pre-fault $\mathbf{Y}$:
${\mathbf{Y}_{NN}}^{pre} = \beta_1[nb,fb,Bb,rx]^{pre}$
2: Construct and save pre-fault $\mathbf{Y}_{MM}$, $\mathbf{Y}_{MN}$, and $\mathbf{Y}_{NM}$:
$[\mathbf{Y}_{MM},\mathbf{Y}_{MN},\mathbf{Y}_{NM}]^{pre} = [\beta_2]^{pre}$
3: Solve Eq. (3) for reduced $\mathbf{Y}$:
${\mathbf{Y}'_{MM}}^{pre} = [\mathbf{Y}_{NN},\mathbf{Y}_{MM},\mathbf{Y}_{MN},\mathbf{Y}_{NM}]^{pre}$
$Y\_red()$ for ${\mathbf{Y}'_{MM}}^{on,post}$
4: Compute on and post-fault full $\mathbf{Y}$:
${\mathbf{Y}_{NN}}^{on,post} = \beta_1 {\mathbf{Y}_{NN}}^{pre}\ [nb,fb,Bb,rx]^{on,post}$
5: Solve Eq. (3) for on and post-fault reduced $\mathbf{Y}$:
${\mathbf{Y}'_{MM}}^{on,post} = \mathbf{Y}_{NN}^{on,post}[\mathbf{Y}_{MM},\mathbf{Y}_{MN},\mathbf{Y}_{NM}]^{pre}$

```
@cuda.jit
def sim(arguments)
   tid = cuda.grid(1) and tid < number of generators;

   At each time-step:
     Fault determination;
      ...
     1. Vectorized operation for the current
        injection algebraic equation in (4)

        I'M[tid] = 0.
        for i in range(number of generators):
            I'M[tid] += Y'MM[tid,i]*V'M[i]
      ...
     2. Element-wise operation for AB approximation
        in (7) to solve differential equation in (5)

        θ(step+1)[tid] = θ(steps)[tid]+ h*
                         (23*dθ(steps)[tid]-
                          16*dθ(steps-1)[tid]+
                           5*dθ(steps-2)[tid])/12
```

LISTING 1. **Numba kernal sim() for DAE solving.**

respectively, achieving a significant reduction in the computational load.

For solving a set of DAEs, the ME method in (6) is substituted with the Adams-Bashforth (AB) method in (7). Instead of predicting and updating the state variables in each iteration, AB only requires a one-time approximation by utilizing the values from current and previous steps, which theoretically provides a two times speedup regardless of hardware constraints due to one less operation.

$$y_{i+1} = y_i + \frac{h}{12}[23f_i - 16f_{i-1} + 5f_{i-2}] \tag{7}$$

B. ACCELERATION OF SINGLE DCA

The acceleration of a single DCA can be achieved by leveraging GPU resources. In $\mathbf{Y}_{NN}$ and $\mathbf{Y}'_{MM}$ formations, GPU has the capability over CPU to accomplish large matrix operations. The data structure written through CuPy fully supports the array functionalities. For example, the significant computations in (3) are completed with the call of GPU-based matrix-matrix dot product [30] and LU decomposition [31], [32] for sparse linear system solving. The implementation offers opportunities for automatically parallelizing the program for speedup without modifying the original algorithm.

The DAE solving function, where the algebraic or state variables of generators in (4) and (5) are independent of each other, contains vectorized and element-wise computations in each time step. Therefore, it is well-suited for data-level parallelism on GPU. These operations can be customized to merge the calculations into a kernel ($sim()$) through Numba to avoid the waste of multi-kernel launches repeatedly. As shown in Listing 1, the network equation solving and AB approximation can be fit into a 1-D grid of 1-D blocks. To perform SIMD, we make every indexed thread (tid) responsible for computing the dynamics of each generator. By looping over the columns of $\mathbf{Y}'_{MM}$ and $\mathbf{V}'_M$, the current injection $\mathbf{I}'_M$ is computed. The approximated next step angle position ($\theta(\text{steps}+1)[\text{tid}]$) can be obtained by taking advantage of the results from the current and last two time steps.

C. ACCELERATION OF MASSIVE DCAs USING MULTIPROCESSING

The concept of multiprocessing refers to task-level parallelism, which generates multiple processes and allocates tasks between them. The design in this paper takes full advantage of multiprocessing as massive DCA runs are also independent of each other. We implement MPI to initialize a group of processes and distribute tasks. MPS is used to allow function executions from different processes to overlap on the device, achieving GPU partitions with higher resource utilization and faster elapsed time. It ensures the simultaneous scheduling of work submitted by individual processes and stays out of the critical execution path. The communication functionality between the process and server is enclosed within the CUDA binaries.

Fig. 1 shows the entire implementation methodology on GPUs with a detailed illustration of the critical executions on a single GPU. The dynamic simulation program for analyzing a single contingency scenario is elaborated in Section II. According to Algorithm 2, it can be separated into three components ($Y\_init()$, $Y\_red()$, and $sim()$). Suppose there are x scenarios that need to be solved in total, and a machine has n available processes for one GPU ($x \gg n$). Thus, there should be x/n iterations to complete all cases. In the beginning, the main process P_0 launches $Y\_init()$ function to parse the system input data and contingency settings. Then the program executes the steps in Algorithm 2 on GPU to get the full $\mathbf{Y}$ matrix and reduced $\mathbf{Y}$ matrix in pre-fault phase. The task-level parallelism starts from the information broadcast to all processes ($P_0, P_1, \ldots, P_{n-1}$). In this study, the MPI application is able to submit GPU-based work directly to the device by activating MPS. We formulate the resource provisioning strategy to equally partition the GPU available threads to each process (active thread percentage to $100\%/n$ per process). Each process then locally generates and takes its

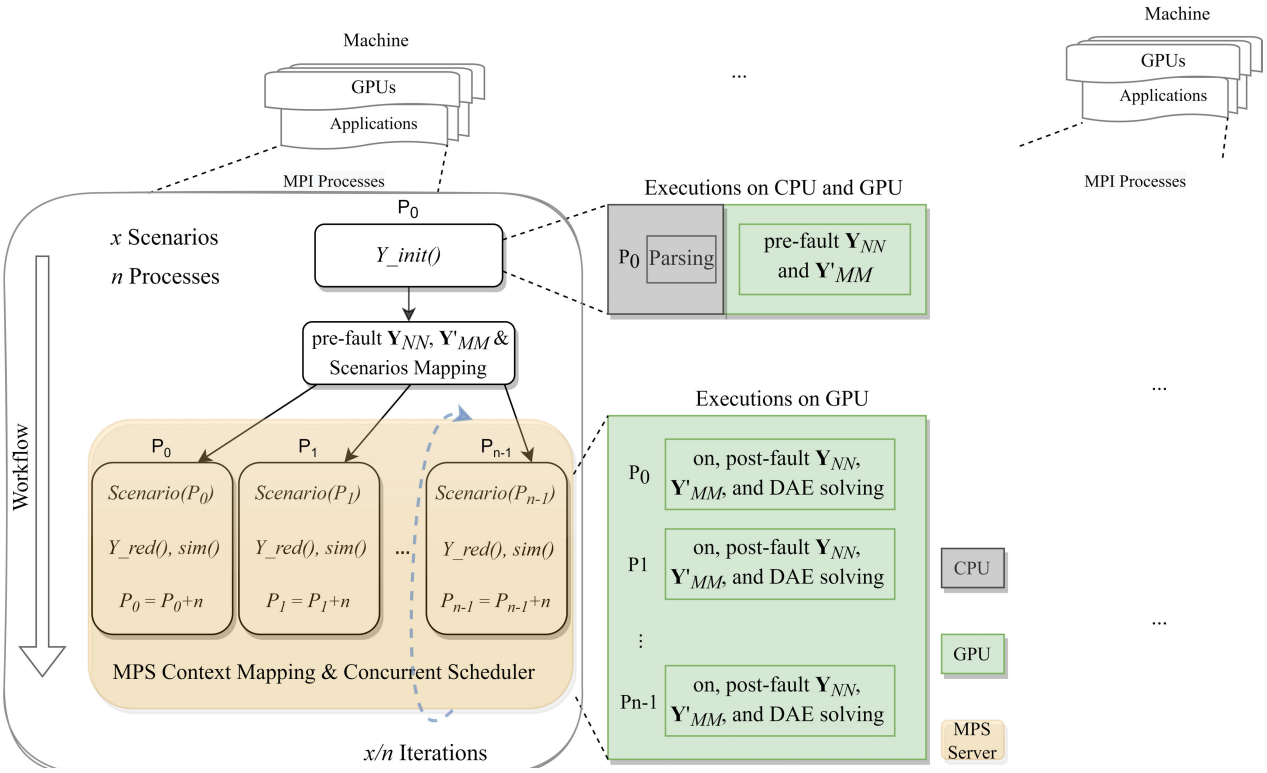

FIGURE 1. Configurable hierarchical architecture for accelerating multiple dynamic contingencies on GPUs. On a single GPU, the main process (P_0) initializes the contingencies and distributes them with the global pre-fault results to the other processes for task-level computations.

own contingency to compute the corresponding matrices for on-fault and post-fault conditions in $Y\_red()$ simultaneously. Finally, each one with $\mathbf{Y}'_{MM}$ matrices enters the kernel $sim()$, to complete the network equation solving and numerical integration of the dynamic state variables.

The architecture is configurable since it also supports multi-GPU concurrency with controls for mapping DCA jobs over many processes on a GPU cluster. Each independent device is set to have an equal number of scenarios to run in parallel. No information transfer is needed. Furthermore, more performance boost can be achieved by aggregating more computing resources from other machines. The input control can evenly separate and dispatch the DCA cases to every GPU device on a scalable computing platform. One machine can act as an independent system, only allowing internal communications between processes within each attached GPU. In other words, each GPU should initialize the pre-fault condition with its assigned DCA scenarios to provide requirements for the local processes later. There is no action for controlling and sharing the global information from one host to the others, although the interactions can be fulfilled through a conventional higher-layer MPI global communicator if needed. This idea reduces the communication overhead in the network and improves overall performance.

In summary, the proposed GPU-based architecture has novel merits as follows:

- Hierarchical task and data-level parallelism to enable massive DCA studies with fast dynamic simulations.
- High portability and reconfiguration capability to run on a single GPU or multiple GPUs if applicable.
- Minimal required communication and overhead incurred between CPU and GPU due to maximized deployment of all simulations on GPUs.

IV. RESULTS AND ANALYSIS

In this section, the speedup of the proposed parallel DCA architecture using different power system cases is presented. We analyze the performance scalability based on test results.

A. TEST SYSTEMS AND HARDWARE CONFIGURATION

Three real power systems (IEEE39b, IEEE145b, and Polish3120b) with different scales are selected to evaluate the computational performance. An artificially created synthetic system (ATF900b) is also applied to reference an increased density of system topology. Table 1 summarizes the differences between these systems. The tripping scenarios (faults) are made during a 30-second dynamic simulation period and cleared after 0.1 seconds for each DCA case. The step width for approximation is 0.005 seconds. All the working programs are tested on Clemson University's supercomputing facilities' Palmetto Cluster [33], which is a Linux-based environment comprised of 2,115 compute nodes (totaling

TABLE 1. Test power systems with different scales.

Systems	Buses	Generators	Algebraic Variables	State Variables
IEEE39b	39	10	20	60
IEEE145b	145	50	100	300
Polish3120b	3120	93	186	558
ATF900b	900	300	600	1800

TABLE 2. Test systems vs. execution time of single DCA in different studies.

Versions	IEEE39b	IEEE145b	Polish3120b	ATF900b
[34], [35]	2.8s	0.13s	8.8s	10.1s
Our GPU	1.1s	1.2s	1.6s	1.7s

32,600 CPU cores), including 639 nodes equipped with 2 Tesla V100 GPUs, and 34 nodes each have 2 Tesla A100 GPUs.

B. SINGLE DCA ON ONE GPU

In parallel computing, the performance of a program is bounded by the non-parallelization portion of the program and influenced by the computation-to-communication ratio, i.e., computation intensity vs. communication overhead. The CUDA initialization and system data transfer between host and device are unavoidable factors for our single DCA run. We have examined our program with a computing node consisting of 8 Intel Xeon(R) Gold 6148@2.40 GHz cores with 32 GB RAM and 1 Tesla V100 16 GB GPU, which is a typical configuration for testing the performance. For benchmarking purposes, we compare our results with the other two research outcomes in [34] and [35], where GPU is utilized as an accelerator to boost specific portions of a dynamic simulation with Algorithm 1. For example, [34] uses low-level CUDA programming to handle LU decomposition on GPU. Reference [35] implements portable compiler directives on Fortran code to offload element search and assignments for dense matrix manipulations. The testing results of each power system are demonstrated in Table 2. Our implementation is the fastest among the three GPU-based studies mainly for three reasons:

1) The proposed dynamic simulation algorithm in Algorithm 2 significantly reduces the workload of $\mathbf{Y}_{NN}$ and $\mathbf{Y}'_{MM}$ computations;
2) The high-level data structure usage and advanced all-GPU parallel implementations discussed in Section III. B improve the overall computational performance;
3) Although the systems vary in size, all test cases can fit into GPU to make use of its exceptional computing resources.

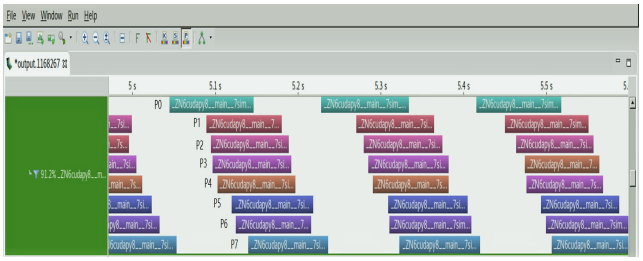

FIGURE 2. Overlapped kernel *sim*() between processes in several scenario solving loops for IEEE145b system.

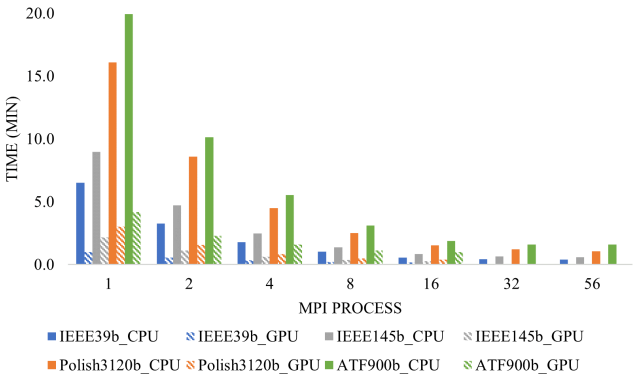

FIGURE 3. Execution time and scalability for accelerating 1,024 dynamic contingencies on GPU and CPU.

C. MULTIPLE DCAs ON A SINGLE COMPUTING NODE

For accelerating DCAs on GPU using the proposed multiprocessing approach, we request a maximum of 56 CPU cores, 372 GB memory, and 2 Tesla V100 GPUs on a single computing node. There are 1,024 scenarios to be solved for each test system. As the number of computing processes increases, more cases can be run in parallel, and the computational time is expected to decrease until a specific threshold is reached. Fig. 2 offers a glance at the *sim*() Numba kernel when the process number is set to 8. According to the timeline profiling, kernel executions are highly overlapped within a GPU, which exhibits great concurrency with high throughput as desired. The Volta MPS server supports 48 concurrent contexts per device at its maximum if the allocated GPU memory is within the limit, but the real allocation is subject to problem size and task intensity. In our case, we can fit 22 tasks per GPU device for the smallest IEEE39b system but only 16 for the largest ATF900 system. The performance of our GPU-based hierarchical architecture is compared against a pure CPU-based approach. The main difference between the two approaches is that the CPU simulator includes individual contingency workflow in Algorithm 1 and achieves data-level parallelism through CPU-based multi-threading techniques.

1) SCALABILITY WITH ONE GPU

The overall scalability of both single GPU-based and CPU-based multiprocessing implementations for four test systems are plotted in Fig. 3. As we expected, by offloading all DCA

TABLE 3. Speedup comparison using GPU-based architecture on a single computing node (one GPU and two GPUs) against CPU-based parallel implementations with different process settings.

Test Systems		IEEE39b		IEEE145b		Polish3120b		ATF900b	
Number of GPU(s)		1	2	1	2	1	2	1	2
Speedup over CPU-based Parallel Implementations with Various Number of Processes	1	37.6	52.3	32.1	46.0	42.2	64.8	20.2	39.6
	16	3.2	4.5	3.0	4.3	4.0	6.2	1.9	3.7
	32	2.4	3.4	2.3	3.3	3.1	4.8	1.6	3.1
	56	2.2	3.1	2.1	3.0	2.8	4.2	1.6	3.1

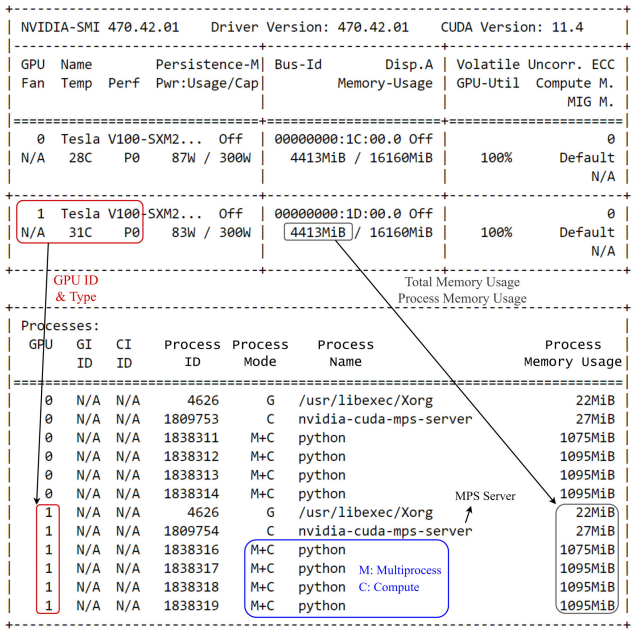

FIGURE 4. MPS server launch with 4 MPI processes pinned on each Tesla V100 GPU.

simulations to a GPU, the performance is improved over the CPU-based one. The computational time is reduced across all test systems regardless of the number of processes. It can be observed that all test systems can reach faster or equivalent executions but with much fewer processes and memory than the CPU approach. For instance, using 56 CPU cores, the parallel CPU-based simulator can solve 1,024 scenarios for Polish3120b within 1.1 min. However, a faster execution (0.8 min) can be achieved on a GPU by launching only four processes. Moreover, at 16 processes, the GPU program accomplishes all the tasks in only 0.4 min, 4x times faster than the CPU. The considerable performance improvement comes from the optimized algorithm, the designed high-efficient GPU-based implementations, and GPU's superior processing capability for data parallelism.

2) SCALABILITY WITH TWO GPUs

Fig. 4 demonstrates the successful implementation of multi-processing on a GPU cluster. As illustrated, two Tesla V100 GPUs are on the current computing node. Each GPU has its own MPS server started, and the four processes with unique IDs are running to solve four independent DCA scenarios in parallel. Compared to the single GPU, since the maximum allowed processes can be doubled, half of the task loads are distributed to the added device, and the computational performance is linearly improved. Table 3 benchmarks the speedups using the CPU-based parallel simulator with different configurations (1, 16, 32, and 56 processes) against the peak performance of the GPU-based simulator using one GPU (16 processes) and two GPUs (32 processes). The Polish3120b system exhibits the best speedup over CPU-based parallel implementation, where a boost of 2.8x with one GPU and 4.2x with two GPUs is recorded.

TABLE 4. Comparisons using different numbers of GPUs to solve 1,024 dynamic contingencies for ATF900b system.

Node(s)	Total GPUs	Total Processes	Cases on Each GPU	Time	Speedup
1	1	16	1024	62.9s	1.0
1	2	32	512	32.3s	1.9
2	4	64	256	17.2s	3.7
4	8	128	128	9.9s	6.4

D. MULTIPLE DCAs ON DIFFERENT COMPUTING NODES ACROSS A DISTRIBUTED SYSTEM

We assume that the more resources available, the more tasks can be run concurrently to accelerate the computations. In another test for better acceleration, we solve 1,024 contingencies by taking advantage of distributed computing facilities. The hardware configurations of each node remain the same as in the single-node test.

Table 4 presents the solution time using 16 processes on each GPU but from a different number of nodes for the ATF900b power system. By requesting four nodes and eight GPUs on a cluster, a speedup of 6.4 is recorded compared to the performance of one GPU on a single computing node. The scalability is nearly linear due to no network communications between the nodes. Each device is only responsible for dealing with the local interactions between its processes. The testing results indicate our approach has extendable and configurable nature, as mentioned in Section III. Hence, the GPU multiprocessing method can further improve the performance of massive DCA simulations if more computing power is available.

V. CONCLUSION AND FUTURE WORK

Traditional power system CA and DCA research heavily rely on high-end CPU-based HPC facilities. This paper discusses a first-ever two-level hierarchical parallel architecture to accelerate massive DCA using GPU devices. The dynamic simulation for solving an individual DCA is optimized on GPU to achieve data-level parallelism. The architecture using the multiprocessing method makes multiple tasks run concurrently within each device. Compared to the CPU-based parallel solution, which requires much more computing resources, the GPU-based architecture achieves even better results by solving 1,024 scenarios in all test systems. It is concluded that the approach possesses the robustness and extensibility to enhance the overall computational performance for massive DCAs under various GPU configurations.

In the future, an integrated application will be built with: 1) fine-tuned CUDA dynamic simulators to satisfy more complex systems and models for a single DCA; 2) real-time data visualization by leveraging the graphics interoperability features of GPU; and 3) a generalized GPU-based multiprocessing architecture to accommodate other related research topics in power system domain.

ACKNOWLEDGMENT

Clemson University is acknowledged for the generous allotment of computing time on Palmetto Cluster.

REFERENCES

[1] H. Bulat, D. Franković, and S. Vlahinić, "Enhanced contingency analysis—A power system operator tool," *Energies*, vol. 14, no. 4, p. 923, Feb. 2021.

[2] A. Mittal, J. Hazra, N. Jain, V. Goyal, D. P. Seetharam, and Y. Sabharwal, "Real time contingency analysis for power grids," in *Proc. Eur. Conf. Parallel Process.* Cham, Switzerland: Springer, 2011, pp. 303–315.

[3] X. Li, P. Balasubramanian, M. Sahraei-Ardakani, M. Abdi-Khorsand, K. W. Hedman, and R. Podmore, "Real-time contingency analysis with corrective transmission switching," *IEEE Trans. Power Syst.*, vol. 32, no. 4, pp. 2604–2617, Jul. 2017.

[4] S. Huang and V. Dinavahi, "Real-time contingency analysis on massively parallel architectures with compensation method," *IEEE Access*, vol. 6, pp. 44519–44530, 2018.

[5] S. Jin, Z. Huang, R. Diao, D. Wu, and Y. Chen, "Parallel implementation of power system dynamic simulation," in *Proc. IEEE Power Energy Soc. Gen. Meeting*, Jul. 2013, pp. 1–5.

[6] S. Jin, Y. Chen, D. Wu, R. Diao, and Z. Huang, "Implementation of parallel dynamic simulation on shared-memory vs. distributed-memory environments," *IFAC-PapersOnLine*, vol. 48, no. 30, pp. 221–226, 2015.

[7] S. Jin, Z. Huang, R. Diao, D. Wu, and Y. Chen, "Comparative implementation of high performance computing for power system dynamic simulations," *IEEE Trans. Smart Grid*, vol. 8, no. 3, pp. 1387–1395, May 2017.

[8] Y. Chen, B. Palmer, P. Sharma, Y. Yuan, B. Mathew, and Z. Huang, "A high performance computational framework for dynamic security assessment under uncertainty," in *Proc. IEEE Electron. Power Grid (eGrid)*, Nov. 2018, pp. 1–5.

[9] Y. Chen, K. Glaesemann, X. Li, B. Palmer, R. Huang, and B. Vyakaranam, "A generic advanced computing framework for executing windows-based dynamic contingency analysis tool in parallel on cluster machines," in *Proc. IEEE Power Energy Soc. Gen. Meeting (PESGM)*, Aug. 2020, pp. 1–5.

[10] J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone, and J. C. Phillips, "GPU computing," *Proc. IEEE*, vol. 96, no. 5, pp. 879–899, May 2008.

[11] W. Liu, B. Schmidt, G. Voss, and W. Müller-Wittig, "Streaming algorithms for biological sequence alignment on GPUs," *IEEE Trans. Parallel Distrib. Syst.*, vol. 18, no. 9, pp. 1270–1281, Sep. 2007.

[12] R. Weber, A. Gothandaraman, R. J. Hinde, and G. D. Peterson, "Comparing hardware accelerators in scientific applications: A case study," *IEEE Trans. Parallel Distrib. Syst.*, vol. 22, no. 1, pp. 58–68, Jan. 2011.

[13] W. J. van der Laan, A. C. Jalba, and J. B. T. M. Roerdink, "Accelerating wavelet lifting on graphics hardware using CUDA," *IEEE Trans. Parallel Distrib. Syst.*, vol. 22, no. 1, pp. 132–146, Jan. 2011.

[14] M. G. Awan et al., "Accelerating large scale de novo metagenome assembly using GPUs," in *Proc. Int. Conf. High Perform. Comput., Netw., Storage Anal.*, 2021, pp. 1–11.

[15] V. Jalili-Marandi, Z. Zhou, and V. Dinavahi, "Large-scale transient stability simulation of electrical power systems on parallel GPUs," *IEEE Trans. Parallel Distrib. Syst.*, vol. 23, no. 7, pp. 1255–1266, Jul. 2012.

[16] D. Chen, H. Jiang, Y. Li, and D. Xu, "A two-layered parallel static security assessment for large-scale grids based on GPU," *IEEE Trans. Smart Grid*, vol. 8, no. 3, pp. 1396–1405, May 2017.

[17] S. Jin and D. P. Chassin, "Thread group multithreading: Accelerating the computation of an agent-based power system modeling and simulation tool—C GridLAB-D," in *Proc. 47th Hawaii Int. Conf. Syst. Sci.*, Jan. 2014, pp. 2536–2545.

[18] P. M. Anderson and A. A. Fouad, *Power System Control and Stability*. Hoboken, NJ, USA: Wiley, 2008.

[19] L. L. Grigsby, *Power System Stability and Control*. Boca Raton, FL, USA: CRC Press, 2007.

[20] K. E. Atkinson, *An Introduction to Numerical Analysis*. Hoboken, NJ, USA: Wiley, 2008.

[21] *NVIDIA Tesla V100 GPU Architecture Whitepaper*, NVIDIA, Santa Clara, CA, USA, 2017.

[22] J. Sanders and E. Kandrot, *CUDA by Example: An Introduction to General-Purpose GPU Programming*. Reading, MA, USA: Addison-Wesley, 2010.

[23] J. Palach, *Parallel Programming With Python*. Birmingham, U.K.: Packt Publishing, 2014.

[24] R. Nishino and S. H. C. Loomis, "CuPy: A NumPy-compatible library for NVIDIA GPU calculations," in *Proc. 31st Conf. Neural Inf. Process. Syst.*, 2017, p. 151.

[25] S. K. Lam, A. Pitrou, and S. Seibert, "Numba: A LLVM-based Python JIT compiler," in *Proc. 2nd Workshop LLVM Compiler Infrastruct. HPC*, 2015, pp. 1–6.

[26] J. Choquette, W. Gandhi, O. Giroux, N. Stam, and R. Krashinsky, "NVIDIA A100 tensor core GPU: Performance and innovation," *IEEE Micro*, vol. 41, no. 2, pp. 29–35, Mar. 2021.

[27] M. P. I. Forum, "MPI: A message passing interface," in *Proc. ACM/IEEE Conf. Supercomput.*, Nov. 1993, pp. 878–883.

[28] NVIDIA. (Oct. 2021). *Multi-Process Service*. [Online]. Available: https://docs.nvidia.com/deploy/mps/index.html

[29] R. D. Zimmerman, C. E. Murillo-Sánchez, and R. J. Thomas, "MATPOWER: Steady-state operations, planning, and analysis tools for power systems research and education," *IEEE Trans. Power Syst.*, vol. 26, no. 1, pp. 12–19, Feb. 2011.

[30] M. Naumov, L. Chien, P. Vandermersch, and U. Kapasi, "Cusparse library," in *Proc. GPU Technol. Conf.*, 2010, pp. 1–15.

[31] J. W. Demmel, S. C. Eisenstat, J. R. Gilbert, X. S. Li, and J. W. H. Liu, "A supernodal approach to sparse partial pivoting," *SIAM J. Matrix Anal. Appl.*, vol. 20, no. 3, pp. 720–755, 1999.

[32] I. Buck, "GPU computing with NVIDIA CUDA," in *Proc. ACM SIGGRAPH Courses*, Aug. 2007, pp. 1–6.

[33] Clemson Universty. *Palmetto Documentation*. Accessed: May 12, 2022. [Online]. Available: https://www.palmetto.clemson.edu/palmetto/

[34] V. Jalili-Marandi and V. Dinavahi, "SIMD-based large-scale transient stability simulation on the graphics processing unit," *IEEE Trans. Power Syst.*, vol. 25, no. 3, pp. 1589–1599, Aug. 2010.

[35] C. Wang, S. Jin, and Y. Chen, "Directive-based hybrid parallel power system dynamic simulation on multi-core CPU and many-core GPU architecture," in *Advances in Parallel & Distributed Processing, and Applications*. Cham, Switzerland: Springer, 2021, p. 405.