Tiako P.F. (ed) Automated Systems, Data, and Sustainable Computing. Chronicle of Computing. OkIP.
 https://doi.org/10.55432/978-1-6692-0001-7_4

Adaptive Development of Parallel Power System Dynamic Simulation Application in Python

Cong Wang[a]*, Liwei Wang[a], Shuangshuang Jin[a]

[a]*Clemson University, 1240 Supply St, North Charleston, SC, 29410, USA*

ABSTRACT

Due to its intensive computational demands for real-time operation and diagnosis, large-scale power system dynamic simulation requires high-performance computing technologies to accelerate its computation on advanced computing platforms. In this paper, leveraging high-level Python and its parallel scientific computing libraries, three parallel power system dynamic simulation applications are adaptively developed using native MPI for Python on CPU, PETSc for Python on CPU, and CuPy on GPU with dedicated data manipulation strategies and implementations, respectively. Their computational performance is compared using different sizes of testing systems and indicates that: 1) MPI and PETSc can make a decent performance for small and moderate-size systems on limited CPU resources, and 2) GPU has better potential in speeding up dynamic simulation for larger and more complex systems. The results demonstrate Python's suitability in parallelizing power system modeling and simulation with fast computational performance and easy development.

Keywords: Python, dynamic simulation, HPC, scalability, speedup

I. INTRODUCTION

Power system dynamic simulation is a critical but compute-intensive function to monitor system dynamic security margins in real-time and keep interconnected power systems operating in a security region. The current practice heavily relies on commercial software tools. However, most of the commercial tools are optimized on a single processor without utilizing high-performance computing (HPC) techniques. For example, the PowerWorld Simulator (PowerWorld Corp, 2012) that is based on a sequential package takes over 60 seconds to perform a 20-second dynamic simulation (Huang, et al., 2017). This time difference leaves many uncertainties and security problems for the power system balancing and operation as the system conditions might change when the solution results are obtained, not to mention the challenges of fulfilling the daily demands of electricity with the quick expansion of the power system topology and upgraded component complexity these days. This emergent need drives the research on faster-than-real-time dynamic simulation.

Most of the work that has been done in this area is focused on the dynamic simulation functions that are implemented using C/C++ types of languages. While they are highly efficient, the complexity of code development is relatively high, especially for people who do not have strong programming skills. Comparing to C/C+, Python is one of the high-level programming languages in the computer science domain. It provides thousands of object-oriented interfaces and millions of function calls, making it easy and safe to program, especially for other domain experts such as power engineers. Currently, Python-based power system modeling tools are still functionally and computationally limited to perform dynamic analysis. For example, pandapower (Thurner, et al., 2018), PowerGAMA (Svendsen & Spro, 2016), and PYPOWER (pypower, 2015) only offer serial programs for dynamic modeling. High-Performance Computing (HPC) techniques taking advantage of advanced shared memory or distributed memory computing architecture are ways to go to fit the gap.

In this paper, three parallel approaches are implemented to accelerate power system dynamic simulation. The structure of this paper is 1) the workflow and algorithm of power system dynamic simulation and the state-of-the-art of parallel programming in Python in Section II; 2) two proposed CPU-based parallel implementations using Message Passing Interface (MPI) (The MPI Forum Corp, 1993) and Portable Extensible Toolkit for Scientific Computation (PETSc) (Balay, et al., 2001), and a GPU-based parallel implementation using CuPy (Nishino & Loomis, 2017) in Section III to showcase how the performance boost can be applied within various Python programming environments; and 3) the performance and analysis of each implementation and the recommended one considering the sizes of the power systems and the constraints of the available computing resources in Section IV. Finally, Section V concludes the current research outcomes and proposes the future work for enhancement.

II. BACKGROUND

A. Power System Dynamic Simulation

Power system dynamic simulation program generally consists of nodal admittance matrix (full $\mathbf{Y}$ matrix) manipulation 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., 10 seconds), as shown in Eq. 1,

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

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 (Jin, Huang, Diao, Wu, & Chen, 2013).

Given a power system with N buses, M generators, and Z branches, the algebraic equations in Eq. 1 can be represented by Eq. 2.

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

To simplify the complexity of the matrix and achieve the 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 (Jin, Huang, Diao, Wu, & Chen, 2017) and (Anderson & Fouad, 2008), Eq. 3 represents the logic of acquiring 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 generators' resistance and reactance, $\mathbf{Y}_{MN}$ 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 Eq. 4.

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

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

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

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 (Jin, Huang, Diao, Wu, & Chen, 2013).

To solve the DAEs, the differential equation set in Eq. 1 needs to be first discretized into algebraic equations, which are then lumped with the original algebraic equations. The Modified Euler (ME) (Atkinson, 2008) method are usually used to solve these equations at each time step.

B. Parallel Programming Libraries in Python

There are many portable CPU or GPU-based parallel computing modules widely used in Python, e.g. MPI, Multiprocessing (Palach, 2014), PETSc, Numba (Lam, Petrou, & Seibert, 2015), and CuPy, etc. Three libraries that are leveraged in this work are listed below:

a) mpi4py: As a standardized message-passing library, MPI offers process communications via messages through a communication network. *mpi4py* (Dalcin, Paz, & Storti, 2005), which supports point-to-point and collective communications of Python buffer objects (NumPy (Walt, Colbert, & Varoquaux, 2011) arrays, builtin bytes, string, and Python array, etc.) provides the capabilities to code MPI programs in Python. In this work, it serves as a baseline implementation for parallel dynamic simulation development on multi-core CPUs through fine-tuned data distribution and explicit inter-processor communications from scratch.

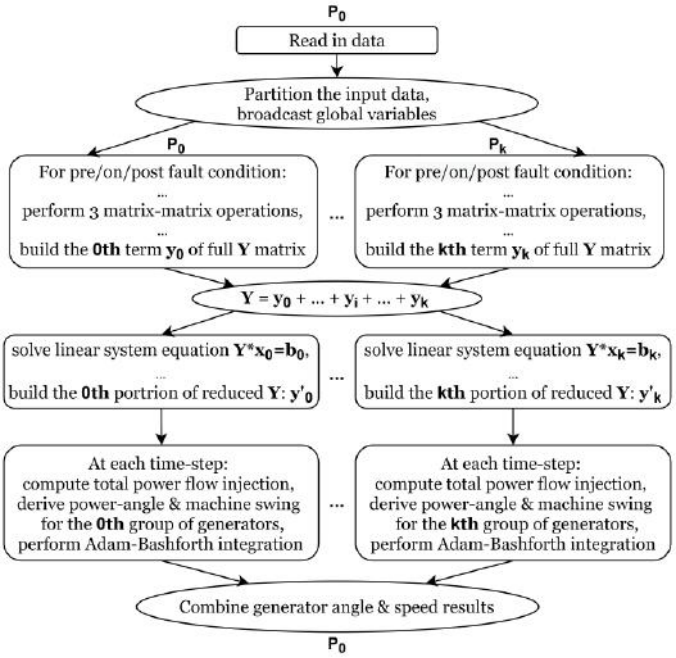

Fig. 1. CPU-based parallel strategy with MPI and PETSc.

b) petsc4py: PETSc is a C or Fortran-based suite of algorithms and data structures for the solution of large-scale scientific and engineering problems on high-performance parallel computing environments. *petsc4py* (Dalcin, Paz, Kler, & Cosimo, 2011) is an open-source software project that provides bindings to PETSc libraries in Python. It offers high-level interfaces with collective semantics so that users rarely have to make explicit message-passing calls to support inter-processor data communication (Balay, et al., 2001). In this work, it serves as an adaptive implementation for parallel dynamic simulation development on multi-core CPUs in a semi-automated way as it averts the message passing calls and leaves the data organization behind.

c) CuPy: CuPy is a high-level NumPy compatible data structure library accelerated with NVIDIA CUDA (Sanders & Kandrot, 2010) on the backend. It makes the full use of the GPU architecture by directly leveraging CUDA computing libraries such as cuSolver (Buck, 2007), cuBLAS (Buck, 2007), cuSPARSE (Naumov, Chien, Vandermersch, & Kapasi, 2010), and cuDNN (Chetlur, et al., 2014) to support the array functionalities and computations. It also provides an API for writing customized CUDA C/C++ kernels (element-wise, reduction, and raw kernels) with enhanced flexibility. In this work, it serves as a portable and resources affordable GPU-based implementation by making extensive use of CuPy multi-dimensional array data structure to alleviate the heavy-duty computations from CPU to GPU.

* Corresponding author E-mail: cong2@clemson.edu

III. Proposed Approach

The adaptive development of parallel dynamic simulation on different computing platforms allows for extensive evaluations of each implementation's computational capabilities and its feasibility to the specific engineering problem.

A. Parallel Implementation on CPU

Figure 1 summarizes the overall parallel design and implementations of dynamic simulation. At the start, the working program splits the input data blocks row-wise, meaning that each process should own a portion of information of buses with *n*, branches with *z*, and generators with *m* in a typical power system. Hence, the full arrays used in a serial code can be initialized to partial arrays and perform partial computations on each process as needed. For example, in a full **Y** matrix formation at each fault condition, the three groups of matrix-matrix operations involved can be boosted by downsizing the configured matrices. The first group in a serial program is shown in Eq. 6,

$$\begin{cases}\mathbf{D}_{ZN} = \alpha * \mathbf{CH}_{ZZ} * \mathbf{F}_{NZ}^{T} + \beta * \mathbf{D}_{ZN} \\ \mathbf{Y}_{NN} = \alpha * \mathbf{F}_{NZ} * \mathbf{D}_{ZN} + \beta * \mathbf{Y}_{NN}\end{cases} \quad (6)$$

where α and β are scalars. In our parallel approach, since the original left side **CH** matrix is a diagonal matrix with size *Z* x *Z*, sequentially, it can be initialized and value-assigned on each process to **ch** with the size *z* x *z* based on the distributed data size *z* the process holds. In addition, the right side matrix **F** (*N* x *Z*) can go with **f** (*N* x *z*) and the additional term **D** can be changed to **d** with *z* x *N*. Consequently, on each process, the operations are converted into Eq. 7. By leveraging the outcomes from their former group, the other two groups can make their own operation like Eq. 7. Finally, each process still obtains a resulting *N* x *N* **y** matrix, however, the summation of all **y** matrices from all processes is expected to be equal to the original full **Y** matrix, Y_{NN}.

$$\begin{cases}\mathbf{d}_{zN} = \alpha * \mathbf{ch}_{zz} * \mathbf{f}_{Nz}^{T} + \beta * \mathbf{d}_{zN} \\ \mathbf{y}_{NN} = \alpha * \mathbf{f}_{Nz} * \mathbf{d}_{zN} + \beta * \mathbf{y}_{NN}\end{cases} \quad (7)$$

After forming the full **Y** matrix and returning it to all processes, the linear system in Eq. 8 and the subsequent matrix-matrix multiplications in the reduced **Y** matrix operations, are parallelized easily as the large right-hand side matrix $\mathbf{Y}_{NM}$ can be built to smaller $\mathbf{Y}_{Nm}$ on each process.

$$\mathbf{Y}_{NN} * \mathbf{X}_{NM} = \mathbf{Y}_{NM} \quad (8)$$

Similarly, each process should be able to make its own $\mathbf{y}_{mM}$ in Eq. 9 based on the split input data and other calculated matrices. Therefore, it only solves the work assigned to it and finally outputs a partial reduced **Y** matrix ($\mathbf{y}'_{mM}$).

$$\mathbf{y}'_{mM} = \mathbf{y}_{mM} - [\mathbf{Y}_{MN} * \mathbf{Y}_{NN}^{-1} * \mathbf{y}_{Nm}]^{T} \quad (9)$$

The remaining work flow of the time-series simulation wraps up several computational intensive calculations (Jin, Chen, Wu, Diao, Huang, 2015) including:

- Fault determination and injected power flow solution.
- Constant impedance conversion.
- Equations of generator dynamics formation in Eq. 5.
- Numerical integration.

The design of the parallel simulation function is that each process directly intakes the outcome of partial reduced **Y** from themselves without any additional communications. In another word, by taking advantage of $\mathbf{y}'_{mM}$ and the previously partitioned information about system and generators, each process has the capability to compute the dynamic state variables of the distributed number of generators independently at a single time step. For instance, the current injection in Eq. 4 can be expressed as Eq. 10.

$$\mathbf{y}'_{mM} * \mathbf{V}_{M} = \mathbf{i}'_{m} \quad (10)$$

Eventually, after all the processes complete the iterations, the master process merges the results together and writes the outputs of the program.

a) MPI: The development of the native MPI program is straightforward. A conda environment with Python 3.8.2 and the package of mpi4py 3.0.3 is established to take advantage of the distributed computing architecture. In terms of the sparse nature of a realistic power system topology, instead of only using NumPy dense arrays to represent the matrices, SciPy (Virtanen, et al., 2020) sparse data type *csr_matrix* is considered to save memory space and computational time, especially in the phases of full **Y** and reduced **Y** formation. The linear algebraic object *spsolve()* for sparse matrices can also be utilized to solve the system equation for each fault condition. But as coding MPI program heavily relies on the explicit decomposition and distribution of data across processes, this approach has a significant limitation in its level of implementation difficulty and requires a relatively higher parallel programming understanding and skills from the developer.

b) PETSc: Unlike the native MPI-based approach which establishes the parallel program by manually keeping all the partitioned data and partial matrix operations on each process throughout the program to save memory usage and avoid inter-processor communications, the PETSc-based implementation automatically has all the data taking matrix and vector as the basic unit to manipulate. These built-in datatypes are highly optimized for running on parallel architectures. Once defined and allocated in parallel, each processor stores a part of the matrix or vector and computes the work locally. As a semi-automated implementation, PETSc allows developers to employ at a high level of abstraction. For example, to reduce unnecessary computation workload caused by zero elements, we simply set the matrices as *mpiaij* type, which represents a parallel sparse matrix. Appendix B gives a snapshot of some representative vector and matrix operations in this implementation to demonstrate the simplicity in the semantics of PETSc. In addition, PETSc provides a built-in linear iterative solver (ksp (Balay, et al., 2019)) and several direct solvers (e.g., mumps (Amestoy, Patrick, & Duff, 2001), superlu (Demmel, et al., 1999), etc.). In our case, the direct solver with Lower and Upper (LU) factorization and approximate minimum degree ordering is selected considering the size and sparsity of the linear system. Moreover, it also offers other lower-level APIs to facilitate any customized operations if the built-in functions

Fig. 2. GPU accelerated approach for dynamic simulation.

cannot meet specific implementation needs. Thus, the variety of options for scientific computing brings in better coding flexibility and computational efficiency.

B. Parallel Accelaration on GPU

In most cases, GPU-based package in Python such as CuPy provides a nearly drop-in replacement interface, and meanwhile, outperforms NumPy and SciPy on data-level parallelism if the problem size of numerical analysis is large enough. As a result, we adapt the original serial code by offloading intensive matrix and vector operations to GPU. Figure 2 depicts the overall implementation strategy of the CuPy program. Except for the initial data parsing and a few unavoidable conditional and control flow statements, most of the array tasks are substituted by the objects and related kernel functions supported by CuPy 9.0.0 with cudatoolkit 11.2 in the entire program.

Appendix C reveals the CuPy semantics using the sparse matrix-matrix multiplications as an example. In full **Y** matrix computations, *cupyx.scipy.sparse._matrix.dot*() kernel function for sparse matrix allows the array manipulations executed using hundreds of GPU tensor cores, which are much faster than NumPy dense and SciPy sparse counterparts on CPU. To solve the linear system equations and obtain reduced **Y** matrix, sparse function *cupyx.scipy.sparse.linalg.spsolve()* is able to achieve the desired solutions. For the simulation, in one time step, all the dynamic parameters of all generators are calculated based on the GPU array operations such as dot product, Hadamard product, and element-wise addition.

The CuPy-based approach is very user-friendly and cost-effective. It is the most concise implementation among the three. Given the superb multi-processing capability of GPU, we envision this implementation exhibits comparable or even better computational performance of parallel dynamic simulation than the two CPU-based implementations on large system cases.

C. Optimization of the Data Structure and Algorithm

a) Sparse Matrix Operations: The power system dynamic simulation program is originally developed with dense matrices. However, as dense datatype requires each element to be involved in the computation, extra computing efforts and resources are needed. As aforementioned, we turn all necessary matrix formulations into sparse matrix operations to minimize memory usage and speed up data processing.

b) Adam-Bashforth Integration: Although the ME method is the most commonly used integration method in power system dynamic simulation, it needs the network equations to be solved twice (predictions and updates) at each time step, leading to doubled performance cost. Alternatively, the Adams-Bashforth (AB) method (Atkinson, 2008) only requires a one-time approximation in a loop by utilizing the solutions of the current and the last time steps. Unlike our previous work in (Jin, Huang, Diao, Wu, & Chen, 2013) and (Jin, Chen, Wu, Diao, Huang, 2015), in this work, we switch from the ME integration method to AB, which theoretically provides a two times speedup regardless of any hardware constraints due to one less approximation step in each iteration.

TABLE I. POWER SYSTEM TEST CASES

Test Case	Bus	Branch	Generator	Source
Polish3120b	3120	3693	93	MATPOWER
3600b	3600	3602	1200	400 x 3g9b
8100b	8100	8102	2700	900 x 3g9b

IV. RESULTS AND ANALYSIS

To validate the proposed approaches and implementations, three test cases with different sizes are selected to evaluate the computational performance. Each case is run multiple times to take an averaged execution time for the evaluation of its computational performance with the least bias.

A. Test Cases

The smallest-size test case in this study is the realistic Polish3120b system. The medium-size 3600b and the largest-size 8100b are artificial cases derived from a 3g9b system by duplicating itself 400 and 900 times, respectively. All the parallel dynamic simulations developed in Section III run on all testing cases for a 30-second simulation with a time step of 0.005 seconds. A fault is applied at a selected bus at 3 seconds and cleared at 3.05 seconds of the simulation to mimic a system disturbance and the relevant generators' dynamics. The dimension of each test case can be found in Tab. 1.

B. System and Hardware Configuration

All three working codes are implemented on Clemson University's supercomputing facilities Palmetto Cluster. For the two parallel implementations on CPU, a computing node consisting of 16 CPU cores (Intel Xeon(R) Gold 6148@2.40 GHz) with 64 GB memory is requested to imitate the limit of hardware resources and perform the tasks. For the GPU accelerated program, with a powerful Tesla V100 16 GB GPU, only 1 CPU core with 4 GB memory is enough as there are few operations in the program that need CPU.

C. Performance Improvement

Remarkable improvement in computational performance has been observed due to the following two design and development strategies applied in all three implementations.

a) Sparse Matrix Operations: Table 2 lists the performance of Full **Y** manipulations for Polish3120b (**Y** Sparsity = 0.9989) in MPI version using dense and sparse matrix respectively as an example. By converting matrix data type, the computational performance for Full **Y** formation is

TABLE II. THE EXECUTION TIME (SEC) OF FULL **Y** MATRIX FORMATION IN NATIVE MPI USING DIFFERENT (DENSE AND SPARSE) MATRIX TYPES

Process	1	2	4	8	16
Dense	125.55	46.14	18.96	8.97	4.71
Sparse	1.71	0.72	0.34	0.17	0.11

TABLE III. PERFORMANCE COMPARISON (SECOND) IN THE SIMULATION FUNCTION BETWEEN TWO METHODS IN MPI WITH 4 PROCESSES

Test Case / Method	Modified Euler	Adam-Bashforth	Speedup
Polish3120	0.81	0.49	1.66
3600b	1.04	0.66	1.65
8100b	1.61	0.95	1.69

largely improved and furthermore greatly contributed to the reduction of the total dynamic simulation time.

b) Adam-Bashforth Integration: Table 3 displays the execution time of the integration methods by running the MPI-based program with 4 MPI processes as an example, which shows a 1.6+ times speedup on all test cases. Likewise, the PETSc and CuPy-based implementations also exhibit a similar boost after the switch. Figure 3 gives a comparison of integration output from PETSc on a generator's machine speed value using the ME method vs. the AB method. The slight differences between each other demonstrate the accuracy of applying the Adam-Bashforth method in dynamic simulation besides its faster computational capability.

D. Scalability Analysis

For our two CPU-based implementations, as the number of computing processes increases, the computation times are expected to decrease until the capability is limited by the problem size once a certain number of computing processes is reached. The speedup of each program is obtained by dividing the computation time of single-process serial run over the multi-process parallel run under specific numbers of processes. For CuPy-based implementation, since the GPU's computing power is fixed, CuPy makes extensive use of its well-optimized CUDA kernel resources to allow for maximum performance against non-GPU approaches. The entire testing results are summarized in Appendix A and the performance curves are plotted in Fig. 4 for better illustration. A horizontal straight line is used in each plot to represent CuPy's performance since it does not require multiple CPU processes to run in parallel.

a) Polish3120b Case:

- MPI: peak performance (1.34 seconds) is reached at 8 processors with a 2.66-time speedup comparing to its serial run. Consistent performance gains are achieved until 16 processors are used when the communication overhead begins to counteract the profit of parallelization.
- PETSc: it takes 0.87 seconds when running in serial. It reaches the best computational performance of 0.74 seconds with a speedup of 1.18 times when utilizing 8 CPU processors. The contribution to the scalability in this implementation mainly comes from the parallelism of the Reduced **Y** matrix potion. The contribution from the Full **Y** matrix formulation is quite limited due to its high sparsity and low computational intensity.
- CuPy: it has a 2.86-second total execution time to complete the computation. Even though such a GPU-based program has the best performance in matrix and vector operations and linear algebraic solutions which significantly reduce the computing time in Full **Y** matrix and Reduce **Y** matrix formation, an enormous kernel launch overhead in each time step at the Simulation phase ruins the overall performance when the matrix size is not large enough to guarantee a high weight in the entire computation.

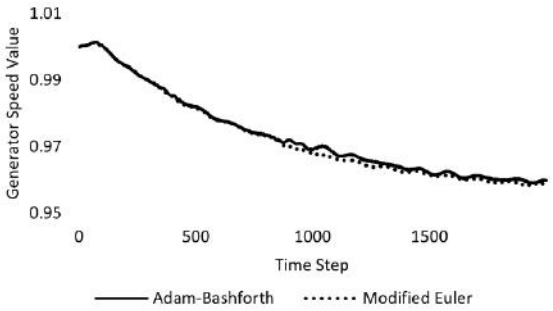

Fig. 3. A comparison of simultion results between the Adam-Bashforth and the Modified Euler methods.

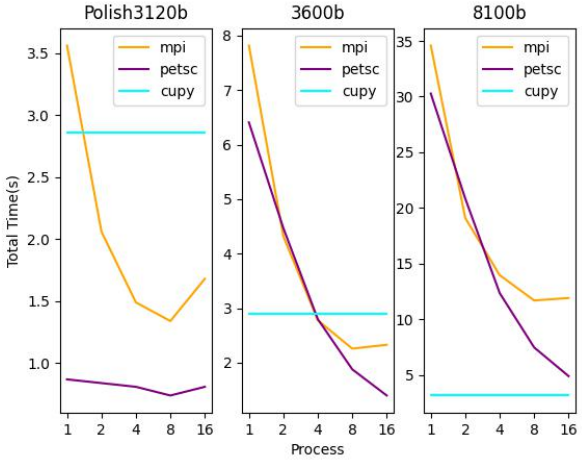

Fig. 4. The total performance (time) to finish each test case using three approaches (CuPy is fixed).

b) 3600b Case: In the moderate-size 3600b case, the computational intensity increases dramatically due to the significantly increased number of generators. It results in larger matrix-matrix operations and linear equations solving in terms of the Full **Y** matrix and Reduce **Y** matrix formations, and incurs increased complexity to compute generator state variables in the Simulation phase.

- MPI: program reaches its peak performance (2.26 seconds) at 8 processors. Impressive scalability is observed in all phases of the executions. Compared to the Polish3120b case, the program run time does not bounce up when the number of processors is increased from 8 to 16. Since the computation intensity is greatly larger than the Polish3120b case, at least 5 times longer Reduced **Y** operation can be observed at each process number.
- PETSc: it runs in 6.41 seconds in sequential. The execution time drops to 1.40 seconds with 16 processors, a speedup of 4.6. The scalability from the Full **Y** matrix formation remains limited due to the small change of bus and branch sizes. However, better scalability in Reduced **Y** matrix formation and Simulation are consistently observed as the big increase of generator size (from 93 to 1200) significantly contributes to the density of the matrices being manipulated in these two phases.
- CuPy: it is again marked as the best one in Full **Y** and Reduced **Y**, however, it takes time to finish the Simulation. The problem size is still not large enough to occupy a larger proportion of the overall execution.

c) 8100b Case: The 8100b system is purposely made 2.25 times larger than the 3600b case for further evaluation on the potential impact of increased system size on the performance of parallel implementations.

- MPI: peak execution performance (11.68 seconds) occurs at 8 processors with a 2.96-time speedup comparing to its serial run. Although consistent close-to-linear scalability is achieved at each phase of the execution, the most time-consuming part changes from the Simulation to the Reduced **Y** matrix formation, which implies a higher computational burden of linear system solving in this phase as the problem size continues to increase.
- PETSc: the scalability of this case is on the same track as the 3600b one. With 16 cores, the total execution time is reduced to 4.88 seconds, which is only 16% of a sequential run. Similar to the native MPI program, the solution time of linear systems at the Reduced **Y** matrix formation phase begins to dominate the computation in the entire program.
- CuPy: it finally starts to show powerful GPU-based computing efficiency as the problem size becomes larger. The cumbersome linear system solving in Reduced **Y** matrix formation remains solvable in a considerably short time (0.06 seconds). The entire program can be run within 3.14 seconds in this GPU-based implementation, which already outperforms the other two CPU-based implementations.

E. Discussion

From the comparison above, following observations are identified:

a) The native MPI-based implementation has the advantage of explicitly decoupling the problem with fine-tuned algorithms but at a cost of high programming effort. The strategies to split the data from the beginning have significant impacts on the alleviation of Full **Y** matrix and Reduced **Y** matrix operation burdens. Best scalability can be achieved across the three cases as a result of this effort. However, the parallel communication routine *MPI.Allreduce(**y**)* becomes more significant with the increased processor number and problem size because multiple large matrix additions and the collective communication between processors are required. Furthermore, since the MPI routines only take buffer-like Python objects for the communications, sparse arrays must be switched back to dense type at their full dimensions, resulting in an unavoidable large data transfer and relatively low total performance gain. Therefore, the MPI-based approach is not recommended for modeling large-scale power systems..

b) The semi-automated PETSc implementation reduced the complexity of parallel programming. Like MPI, it requires several collective communication calls (*VecScatter()*, *toAll()*, and *toZero()*, etc.) for the Full **Y** matrix and Reduced **Y** matrix formations to guarantee every single parallel step can be built successfully. Nonetheless, its highly abstracted and optimized routines support sparse and compressed data operations and communications throughout the entire program. PETSc also makes a huge gain to perform vectorized matrix operations if the process number increases. As a result, both the communication and computation are more efficient as compared to the MPI-based implementation. It's recommended as a good candidate to solve dynamic simulation for small to medium-size systems.

c) The GPU-based implementation simply changes the original NumPy methods in serial code to CuPy related functions. Thus, the parallel program is highly element-wise operation-based and easy to implement. From the observations, it has the best capability to resolve **Y** and Reduced **Y**, but costs a lot in the Simulation phase due to the overhead of kernel launch in each iteration. Based on the overall testing results, it is highly recommended for solving large-scale dynamic simulation in parallel.

V. Conclusion And Future Work

This paper adaptively presents three parallel Python-based implementation approaches to speed up power system dynamic simulation application on multi-core CPUs and many-core GPU. Sparse matrix operations and a fast integration method are applied to improve the computational performance of all implementations. Benchmarking tests are made to evaluate the feasibility and capability of each implementation in terms of matrix manipulation, linear system solving, and simulation integration for power systems at different size levels. For small and medium cases, the PETSc version is the best option using limited CPU cores and memory. CuPy GPU computing is portable, cost-effective and suitable to run more complex systems. Future work involves the further optimized CPU and GPU versions, and real-time data analytics and visualization, which are also considered to be incorporated into this work to build a seamless data processing, computation, and analysis pipeline to facilitate integrated comprehensive studies on fast electric power system dynamic simulation in one unified HPC environment.

APPENDIX

A. COMPARISON OF THE EXECUTION TIME (SEC) OF PYTHON-BASED IMPLEMENTATIONS ON DIFFERENT CASES

		POLISH3120B				3600B				8100B			
METHOD	PROCESS	FULL Y	REDUCED Y	SIMULATION	TOTAL	FULL Y	REDUCED Y	SIMULATION	TOTAL	FULL Y	REDUCED Y	SIMULATION	TOTAL
MPI	1	1.71	0.33	0.58	3.56	1.83	2.79	1.19	7.82	9.54	13.14	2.27	34.59
	2	0.72	0.17	0.51	2.06	0.75	1.42	0.84	4.32	3.82	6.48	1.35	19.09
	4	0.34	0.10	0.49	1.49	0.36	0.72	0.66	2.78	1.67	3.25	0.95	13.95
	8	0.17	0.06	0.55	1.34	0.18	0.39	0.60	2.26	0.80	1.63	0.79	11.68
	16	0.11	0.05	0.78	1.68	0.10	0.27	0.58	2.33	0.41	0.96	0.76	11.90
PETSC	1	0.03	0.42	0.27	0.87	0.03	5.53	0.75	6.41	0.06	28.38	1.60	30.27
	2	0.06	0.40	0.27	0.84	0.05	3.82	0.49	4.47	0.07	19.64	0.91	20.84
	4	0.05	0.35	0.31	0.81	0.04	2.26	0.40	2.80	0.05	11.48	0.61	12.34
	8	0.05	0.27	0.33	0.74	0.04	1.39	0.37	1.88	0.05	6.75	0.47	7.46
	16	0.05	0.26	0.40	0.81	0.04	0.89	0.38	1.40	0.05	4.18	0.46	4.88
CUPY		0.026	0.016	2.49	2.86	0.023	0.021	2.49	2.89	0.023	0.06	2.58	3.14

B. PETSC SEMANTICS EXAMPLE

```
from petsc4py import PETSc

#Update the main diagonal: Y_d[i,i] = Y_d[i,i] + yl
Y_d.setDiagonal(yl,addv=2)

#Perform matrix multiplication
Y_mod=diagy.matMult(permPV)

#Setup the solver
pcr=PETSc.PC().create(comm=PETSc.COMM_WORLD)
pcr.setType('lu')
pcr.setFactorSolverType('superlu_dist')
pcr.setFactorShift(shift_type=1, amount=1e-10)
pcr.setFactorOrdering(ord_type='amd')
pcr.setFromOptions()
pcr.setOperators(Y_d)
pcr.setUp()
F=pcr.getFactorMatrix()

#Solve linear system: AX=B
F.matSolve(Y_c,X)
X.assemblyBegin()
X.assemblyEnd()

#Perform Y_a = 1*Y_a + frevec
Y_a.axpy(1, frevec)
...
```

C. SPARSE MATRIX-MATRIX MULTIPLICATION IN CUPY

```
import cupy as cp
from cupyx.scipy.sparse import csr_matrix

#Convert matrices to sparse csr matrix
chrgfull=csr_matrix(chrgfull)
c_from=csr_matrix(c_from)
...

#Perform sparse matrix multiplications
Y_dummy=chrgfull.dot(c_from.transpose())
Y=c_from.dot(Y_dummy)
...
```

REFERENCES

Amestoy, P. R., Duff, I. S., L'Excellent, J. Y., & Koster, J. (2001). A fully asynchronous multifrontal solver using distributed dynamic scheduling. *SIAM Journal on Matrix Analysis and Applications*, *23*(1), 15-41.

Anderson, P. M., & Fouad, A. A. (2008). *Power system control and stability*. John Wiley & Sons.

Atkinson, K. E. (2008). *An introduction to numerical analysis*. John wiley & sons.

Balay, S., Abhyankar, S., Adams, M., Brown, J., Brune, P., Buschelman, K., ... & Zhang, H. (2019). PETSc users manual.

Balay, S., Buschelman, K., Gropp, W. D., Kaushik, D., Knepley, M. G., McInnes, L. C., ... & Zhang, H. (2001). PETSc. *See http://www. mcs. anl. gov/petsc*.

Buck, I. (2007). Gpu computing with nvidia cuda. In *ACM SIGGRAPH 2007 courses* (pp. 6-es).

Dalcin, L. D., Paz, R. R., Kler, P. A., & Cosimo, A. (2011). Parallel distributed computing using Python. *Advances in Water Resources*, *34*(9), 1124-1139.

Dalcín, L., Paz, R., & Storti, M. (2005). MPI for Python. *Journal of Parallel and Distributed Computing*, *65*(9), 1108-1115.

Demmel, J. W., Eisenstat, S. C., Gilbert, J. R., Li, X. S., & Liu, J. W. (1999). A supernodal approach to sparse partial pivoting. *SIAM Journal on Matrix Analysis and Applications*, *20*(3), 720-755.

Huang, R., Jin, S., Chen, Y., Diao, R., Palmer, B., Huang, Q., & Huang, Z. (2017, July). Faster than real-time dynamic simulation for large-size power system with detailed dynamic models using high-performance computing platform. In *2017 IEEE Power & Energy Society General Meeting* (pp. 1-5). IEEE.

Jin, S., Chen, Y., Wu, D., Diao, R., & Huang, Z. H. (2015). Implementation of parallel dynamic simulation on shared-Memory vs. distributed-Memory environments. *IFAC-PapersOnLine*, *48*(30), 221-226.

Jin, S., Huang, Z., Diao, R., Wu, D., & Chen, Y. (2013, July). Parallel implementation of power system dynamic simulation. In *2013 IEEE Power & Energy Society General Meeting* (pp. 1-5). IEEE.

Jin, S., Huang, Z., Diao, R., Wu, D., & Chen, Y. (2017). Comparative implementation of high performance computing for power system dynamic simulations. *IEEE Transactions on Smart Grid*, *8*(3), 1387-1395.

Lam, S. K., Pitrou, A., & Seibert, S. (2015, November). Numba: A llvm-based python jit compiler. In *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC* (pp. 1-6).

Naumov, M., Chien, L., Vandermersch, P., & Kapasi, U. (2010, September). Cusparse library. In *GPU Technology Conference*.

Nishino, R. O. Y. U. D., & Loomis, S. H. C. (2017). CuPy: A NumPy-compatible library for NVIDIA GPU calculations. *31st confernce on neural information processing systems*, *151*.

Palach, J. (2014). *Parallel programming with Python*. Packt Publishing Ltd.

pypower-dynamics. PyPI. (2015, May 31). https://pypi.org/project/pypower-dynamics/.

Sanders, J., Kandrot, E., & by Example, C. U. D. A. (2010). An Introduction to General-Purpose GPU Programming.

Svendsen, H. G., & Spro, O. C. (2016). PowerGAMA: A new simplified modelling approach for analyses of large interconnected power systems, applied to a 2030 Western Mediterranean case study. *Journal of Renewable and Sustainable Energy*, *8*(5), 055501.

The MPI Forum, C. O. R. P. O. R. A. T. E. (1993, December). MPI: a message passing interface. In *Proceedings of the 1993 ACM/IEEE conference on Supercomputing* (pp. 878-883).

Thurner, L., Scheidler, A., Schäfer, F., Menke, J. H., Dollichon, J., Meier, F., ... & Braun, M. (2018). pandapower—an open-source python tool for convenient modeling, analysis, and optimization of electric power systems. *IEEE Transactions on Power Systems*, *33*(6), 6510-6521.

Van Der Walt, S., Colbert, S. C., & Varoquaux, G. (2011). The NumPy array: a structure for efficient numerical computation. *Computing in science & engineering*, *13*(2), 22-30.

Virtanen, P., Gommers, R., Oliphant, T. E., Haberland, M., Reddy, T., Cournapeau, D., ... & Van Mulbregt, P. (2020). SciPy 1.0: fundamental algorithms for scientific computing in Python. *Nature methods*, *17*(3), 261-272.