Tiako P.F. (ed) *Smart Computing and Emerging Applications*. Chronicle of Computing. OkIP. APDC26#6
 https://doi.org/10.55432/978-1-6692-0015-4_3

Single-Source Heterogeneous Programming Approach for Power System Transient Stability Analysis and Alternating Current Power Flow

Xin Yang[a]*, Cong Wang[b], Shuangshuang Jin[b]

[a]*UTSW Medical Center, 5323 Hines Blvd. Dallas, Tx, 75390, USA*
[b]*Clemson University, 1240 Supply St, North Charleston, SC, 29405, USA*

ABSTRACT

The rise of heterogeneous computing requires that parallel applications be executed on various hardware from different vendors. Thus, efficient solutions emerge using compiler directives, communication protocols, high-level libraries, and execution policies. This paper presents a portable heterogeneous programming approach for scientific application development in Python. The implementation realizes the parallel architecture through Message Passing Interface (MPI) with CPU-based and GPU-based array computing at a high level. Two numerical algorithms in the computational power system area, transient stability analysis (TSA) and alternating current power flow (ACPF), are used to test the fitness of the design. The performance result on the largest 36864-bus-12288-generator system indicates that a 30-second TSA program run with pure CPU-based parallelism finishes the computation in 9.89 seconds, and the program's CPU+GPU acceleration (hybrid switch) reaches 2.33 seconds on a cluster node, making it 12.9x faster than real-time simulation. The ACPF solution can be calculated to converge in 9.69 seconds on the CPU and 2.38 seconds in hybrid mode, respectively. The proposed method significantly improves computational performance with high portability and adaptiveness to various hardware environments.

Keywords: Heterogeneous computing, MPI, array computing, TSA, ACPF, performance

I. INTRODUCTION

Heterogeneous computing refers to accelerating the computational performance of algorithms on more than one computing hardware. It is well-known for solving computationally intensive problems using more than one type of parallelism, such as single-instruction multiple-data (SIMD) and multiple-instruction multiple-data (MIMD) [1]. Nonetheless, adapting the scientific models to run on different hardware is not trivial. Offloading the algorithms typically involves rewriting the original code using other approaches. This process is labor-intensive since significant efforts are required to design, optimize, and maintain the implementations for every computing architecture. It would take much more expertise and human power than any research can provide.

One paradigm shift that would help these experiments prepare for upcoming computing challenges is the ability to utilize heterogeneous resources that allow a single-source code to be configured and run on various backends. High-level programming languages, models, and libraries that provide advanced array computing structures and parallel execution options can be adapted to produce optimal performance. Fortunately, several frameworks are pursuing a single codebase running on different accelerators. For example, developers can specify high-level parallelism and shared memory management through traditional compiler directives such as OpenMP [2] and OpenACC [3]. Kokkos [4] and Alpaka [5], which support the executions on CPU and GPU from Intel, NVIDIA, and AMD, are newly introduced models in recent years. Scientific research [6]–[9] programmed using these portable tools has been recognized for transforming traditional CPU or GPU algorithms in various areas [10]–[14]. Another solution is the std::parallel::execution (std::par) interface within the C++17 standard. Its Application Programming Interface (API) allows for a high-level description of concurrent loops. Various compiler options, such as oneapi::dpl [15] and nvc++, support offloading those loops to the device. Furthermore, SYCL [16] is another programming model based on the C++17 standard that includes CPU host and GPU kernel code in the same source file. Despite having plenty of choices to realize heterogeneous computing conveniently, these Fortran or C/C++ extensions still require proficient programming knowledge and massive coding effort, especially for domain experts.

This paper uses Message Passing Interface (MPI) [17] with high-level interfaces in Python to discuss the single-source development of heterogeneous solutions for power system transient stability analysis (TSA) and alternating current power flow (ACPF). The acceleration on the CPU and GPU is switchable without changing the parallel architecture of the code. The contributions are listed below:

1) The unified multiprocessing architecture realizes both pure CPU parallelism and CPU+GPU parallelism.

2) The single-source code with a hybrid switch enables heterogeneity and portability to different accelerators.

3) Compared to the state-of-the-art applications, the proposed approaches significantly enhance computational efficiency and reduce resource usage.

II. BACKGROUND

This section reviews the reduced nodal admittance matrix (reduced Y matrix) method of TSA, the Newton-Raphson (NR) method of ACPF, and the existing strategies and heterogeneous implementations for accelerating the computational performance of these scientific applications.

* Corresponding author E-mail: xin.yang2@utsouthwestern.edu

Presented at the 2026 OkIP International Conference
on Advances in Parallel and Distributed Computing (APDC)
in Oklahoma City, OK, USA, and Online, on April 1, 2026

A. TSA with Reduced Y Algorithm

Power system TSA is critical for online and offline Energy Management System. It has been widely used in dynamic contingency analysis to predict potential physical and electrical transient stability constraints with certain system component failures [18]. For a large power system comprising thousands of buses and branches, the modeling process usually imposes a substantial computational burden for faster than real-time (FTRT) operations. The core computation in TSA generally consists of nodal admittance matrix (full **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 (e.g., 30 seconds) as shown in (1),

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

where the vector **x** represents dynamic state variables such as generator rotor angles and speeds, and the vector **u** represents algebraic variables such as the network bus voltage magnitudes and phase angles, and real and imaginary parts of the bus voltage [19]. Given a power system with **N** buses, **M** generators, and **Z** branches, the algebraic equations in (1) can be represented by (2),

$$\boldsymbol{Y_{NN}} * \boldsymbol{V_N} = \boldsymbol{I_N} \quad (2)$$

where the vector of current injections at each bus $\boldsymbol{I_N}$ is the production of full **Y** matrix $\boldsymbol{Y_{NN}}$ and vector of bus voltages $\boldsymbol{V_N}$. To simplify the complexity of the matrix and achieve the best matrix operation performance, for a power system with the classical model and constant impedance load, $\boldsymbol{Y_{NN}}$ can be reduced to only contain generator internal buses, $\boldsymbol{Y'_{MM}}$. According to [20] and [21], (3) represents the logic of acquired reduced **Y** matrix,

$$\boldsymbol{Y'_{MM}} = \boldsymbol{Y_{MM}} - \boldsymbol{Y_{MN}} * \boldsymbol{Y_{NN}}^{-1} * \boldsymbol{Y_{NM}} \quad (3)$$

Thus, the algebraic equations can be modified to (4).

$$\boldsymbol{Y'_{MM}} * \boldsymbol{V'_M} = \boldsymbol{I'_M} \quad (4)$$

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

$$\begin{cases} \dfrac{d\omega_a}{dt} = \dfrac{\omega_{sa}}{2H_a}\left(P_{ma} - P_{ea} - D_a(\omega_a - \omega_{sa})\right) \\ \dfrac{d\theta_a}{dt} = \omega_a - \omega_{sa} \end{cases} \quad (5)$$

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 for synchronously rotating reference [22]. The explicit Adams-Bashforth (AB) integration executes a one-time approximation in each iteration by utilizing the solutions of the current and the last two time steps. A predefined fault is applied to perturb the system balance during the simulation, resulting in pre, on, and post fault situations.

B. ACPF with NR Method

ACPF study is an important numerical analysis for the future expansion and deployment of the power system. The basic problem can be described as shown in (6), which contains a set of nonlinear equations to be solved.

$$\boldsymbol{f}(\boldsymbol{u}) = \boldsymbol{0} \quad (6)$$

In the NR iterations, the Jacobian matrix **J** with the dimension of 2(**N**-1) by 2(**N**-1) is formed to find the incremental changes Δu for the bus phase angles Δϕ and voltage magnitudes ΔV as shown in (7) to (10),

$$\boldsymbol{J\Delta u} = -\boldsymbol{f}(\boldsymbol{u}) \quad (7)$$

$$\begin{cases} P_i = \Sigma_{k=1}\,|V_i||V_k|(G_{ik}cos\phi_{ik} + B_{ik}sin\phi_{ik}) \\ Q_i = \Sigma_{k=1}\,|V_i||V_k|(G_{ik}sin\phi_{ik} - B_{ik}cos\phi_{ik}) \end{cases} \quad (8)$$

$$\begin{cases} \Delta P(u) = P_i - P_i(u) \\ \Delta Q(u) = Q_i - Q_i(u) \end{cases} \quad (9)$$

$$J\begin{bmatrix} \Delta\phi \\ \Delta|V| \end{bmatrix}\begin{bmatrix} \Delta P(u) \\ \Delta Q(u) \end{bmatrix} \quad (10)$$

where P and Q represent real and imaginary parts for the quantity values in the full **Y** matrix. i indicates the target bus to be calculated, and bus k is responsible for generating the phase angle between the two. The parameters G_{ik} are conductance and the B_{ik} are susceptance. Consequently, in the iteration n, the updated expression of ϕ and V can be

$$\begin{bmatrix} \phi^{n+1} \\ |V|^{n+1} \end{bmatrix} = \begin{bmatrix} \phi^{n} \\ |V|^{n} \end{bmatrix} - \left(J(u^n)\right)^{-1}\begin{bmatrix} \Delta P(u^n) \\ \Delta Q(u^n) \end{bmatrix} \quad (11)$$

As the Jacobian matrix **J** itself also has the form

$$J = \begin{bmatrix} J_1 & J_2 \\ J_3 & J_4 \end{bmatrix}\begin{bmatrix} \dfrac{\partial P}{\partial \phi} & \dfrac{\partial P}{\partial |V|} \\ \dfrac{\partial Q}{\partial \phi} & \dfrac{\partial Q}{\partial |V|} \end{bmatrix} \quad (12)$$

(11) can be substituted and the new values of ϕ and V can be obtained. The iteration continues until the residuals of ΔP and ΔQ are less than the pre-set tolerance (E^{-8}).

III. Implementation Methodology

This section presents a single-source heterogeneous programming approach to accelerate the computational performance of TSA and NR ACPF on the pure CPU and CPU+GPU architecture. The configured programs enable straightforward algorithm adaptations with minimal modifications to the hardware settings.

A. High-Performance Array Programming

In this work, both CPU and GPU-based high-level interfaces can be coupled with MPI protocol using mpi4py [23], which supports the communications of Python buffer-like and generic objects. NVIDIA MPS [24] on a Volta GPU offers an MPI+CUDA parallelism to realize multiprocessing on a single device. The CPU-based implementation adopts NumPy and SciPy for dense and sparse array manipulations. They are powered by high-performance math libraries such as Intel MKL

[25]. Similarly, CuPy provides nearly drop-in programming interfaces and functionalities using CUDA libraries [26], [27] as acceleration engines.

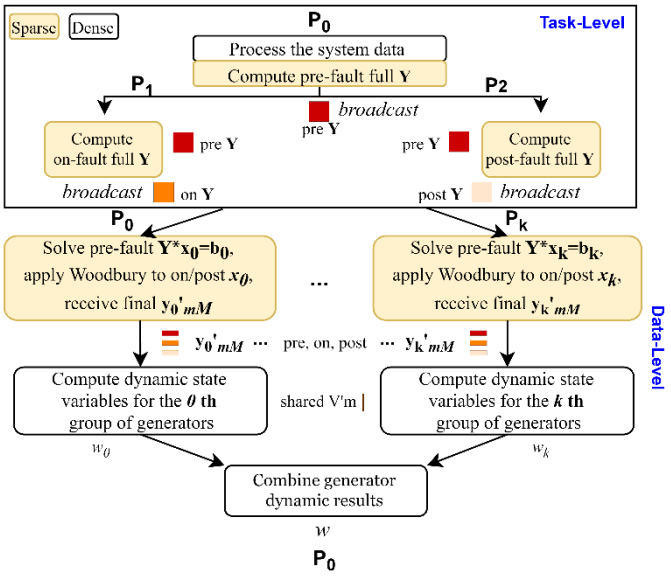

Fig. 1. The improved parallelism for TSA reduced **Y** model.

B. *Unified MPI Parallelism for TSA*

[28] proposes an improved approach to eliminating the loop for on and post-phases by effectively collecting the reusable outcomes from the pre-fault phase. Thus, only the pre-fault steps must be solved once to obtain the final equivalent $\boldsymbol{Y'_{MM}s}$. In this paper, the Woodbury lemma method [29] is also applied to avoid the linear system solving in (3) for the other two fault conditions. The multiprocessing implementations through MPI in [22], [30], [31] have been proven efficient in parallel TSA. The parallel strategy can be optimized to obtain more speedup by tuning the workload distribution and data communications based on the reduced Y model as shown in (13),

$$\begin{cases} \boldsymbol{Y}_{NN}^{\text{on,post}} = \boldsymbol{Y}_{NN}^{pre}[B_b, rx]^{\text{on,post}} \\ \boldsymbol{y'}_{mM}^{\text{on,post}} = \boldsymbol{Y}_{NN}^{\text{on,post}}[\boldsymbol{y}_{mM}, \boldsymbol{Y}_{MN}, \boldsymbol{y}_{nM}]^{pre} \end{cases} \quad (13)$$

where **y** is a portion or term of **Y**. m and n are the sizes of distributed data size on each process.

Fig. 1 shows the designed MPI parallelism. The main process P_0 takes pre-fault and dispatches the other two individual $\boldsymbol{Y}_{NN}$ tasks to P_1 and P_2 if k≥2, where k is the total number of processes used. P_0 broadcasts the results as left-side matrix $\boldsymbol{Y}_{NN}^{pre}$ for the network equation **YX = B**. One process's computational cost of this linear system can be reduced since the right-hand side matrix **B** is constructed locally as b_i ($0 \le i \le k$) with a portion of input data on P_i. The computed x_i then generates part of $\boldsymbol{Y'_{MM}}$ ($\boldsymbol{y'_{mM}}$), which is used directly in the time-domain simulation for partial DAE solving on each process. On and post-fault can take advantage of x_i and the Woodbury method for the other two $\boldsymbol{y'_{mM}}$s in (13). The task-data level combination optimizes the overall scalability by creating minimal communications between processes. Thus, an evenly distributed workload provides a nearly synchronized execution across all processes after full **Y** formation.

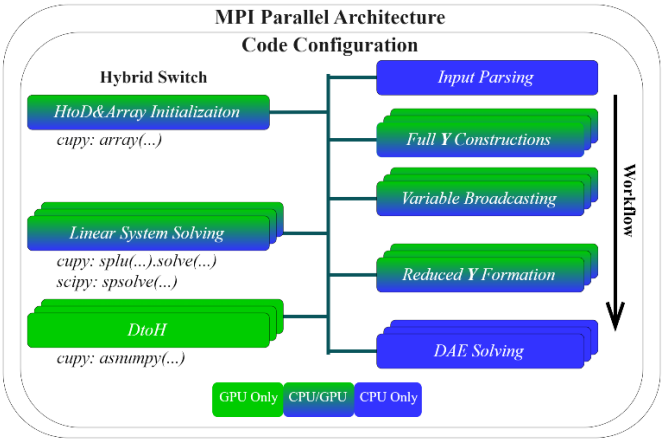

Fig. 2. The code with CPU or GPU-specific definitions requires a switch in the single-source TSA application.

Furthermore, for DAE solving with thousands of iterations, each time step contains two coupled array functionalities to compute the generators' state variables:

- Matrix and vector dot product for algebraic equation solving in (4)
- Vector and vector element-wise operations for differential equation solving in (5) using AB approximation

The MPI program has the capability to avoid data racing since the generator's dynamic states are independent of each other. To optimize the consecutive dense matrix-vector multiplication and vector-vector element-wise operations in each time step, initializing the global shared object $\boldsymbol{V'_M}$ accessible to all processes is essential to reduce memory copies and communication overhead.

C. *Single-source Heterogeneous TSA*

According to the previous research in [32] and [33], GPU computing for producing reduced **Y** matrices is superior to the CPU once the problem sizes become large. However, CPU-based parallelism is better for solving DAEs because the GPU kernel launches for related computations in each time step, which prevents efficient execution. Fig. 2 demonstrates the details of the program design and configurations. The default CPU setting executes every sequential and MPI parallel computation on the CPU. As a result, five code blocks with two colors turn blue. Once we opt for the switch to trigger the MPI CPU+GPU hybrid parallelism, these blocks will go green for the **Y** related formations on the GPU. To achieve this flexible control, the specific command is only added to three portions of the code: 1) explicit CPU host and GPU device data movement (HtoD, DtoH), 2) CPU or GPU-based array initialization, and 3) the variation of calling the linear system solver on CPU and GPU. The other code areas remain consistent without structural modifications by leveraging the shared interfaces for array programming supported across CuPy, NumPy, and SciPy. Therefore, the configured design enables adaptivity to different hardware backends with the same source file and parallel architecture.

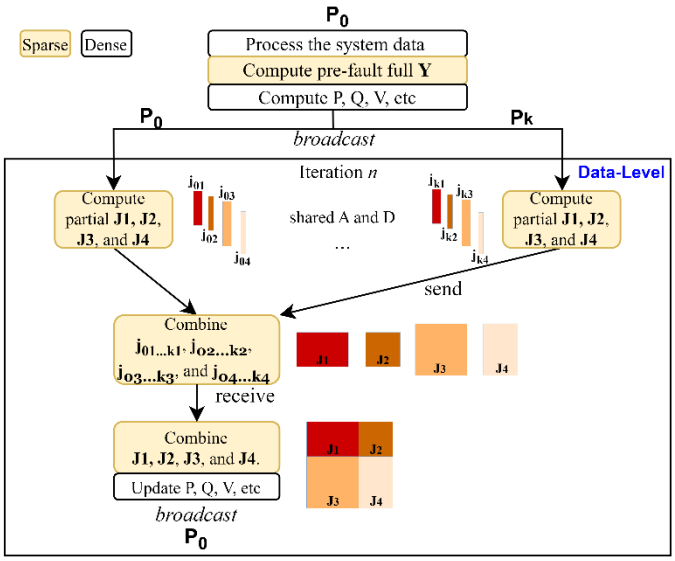

Fig. 3. The designed MPI parallelism for the NR ACPF model.

D. Unified MPI Parallelism for NR ACPF

According to Section II. B, the main computational burden for the NR ACPF algorithm is inside the $\mathbf{J}_1$, $\mathbf{J}_2$, $\mathbf{J}_3$, and $\mathbf{J}_4$ formations. The intensive work is ideally distributed to each process using partial right-hand side matrices for data-level parallelism. As in (14), each process will have partial values of the full results. Thus, the computations can be accelerated.

$$\begin{cases} \boldsymbol{j_1 = -AEa^T} \\ \boldsymbol{j_2 = ABC^{-1}d^T} \\ \boldsymbol{j_3 = DEa^T} \\ \boldsymbol{j_4 = DBC^{-1}d^T} \end{cases} \quad (14)$$

Similarly, creating shared objects for matrices A and D is critical to avoid extra data copies on each process. Figure 3 demonstrates the MPI parallel logic of the NR ACPF algorithm. After getting partial values, the processes return the results to the main process P_0. P_0 is responsible for organizing them into a full J matrix and solving the updates of system variables. It is worth mentioning that compared to the j formations in (14), the other parts of the computation are considered trivial. Therefore, these tasks can keep their original serial steps on P_0.

E. Single-source Heterogeneous NR ACPF

Compared to TSA, implementing single-source heterogeneous NR ACPF is easier since there are fewer computations than solving DAEs. As shown in Fig. 4, the GPU acceleration for matrix multiplications and linear system solving in **j** formation is integrated into the MPI multiprocessing architecture. The sparse matrix operations in SciPy and CuPy are the most commonly used techniques as the power flow problem owns its sparse property. Following the above design, a switch command can be passed to the program without changing the code. The program will stop once converged, unless the iterations exceed the maximum number.

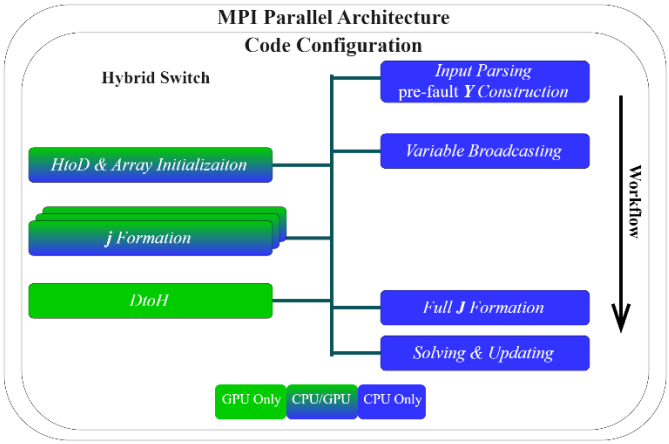

Fig. 4. The code only has one significant block (**j** formation) to be boosted with CPU or GPU in the single-source NR ACPF program.

IV. Case Studies

In this section, the performance and efficiency are analyzed with a detailed benchmark. According to the test results and coding effort, we also conclude the portability of heterogeneous development using the proposed approach for TSA and NR ACPF. A line-tripping fault is applied during a 30-second simulation period and cleared after 0.03 seconds. The step width is 0.005 seconds, which forms 6000 iterations in total. All the working programs are run on four settings, including two separate nodes (Node 1 and Node 2) on a supercomputing cluster, one lab server, and one personal computer (PC). Table I demonstrates the hardware configurations. Six artificially expanded power systems with different scales are used to evaluate the computational performance. They are all generated from a 9-bus-9-branch-3-generator system (IEEE 9-bus) to test the influence of problem sizes.

TABLE I TEST SYSTEMS AND PLATFORMS.

Hardware	Node 1	Node 2	Lab Server	PC
CPUs	16 Xeon(R) Gold 6248	16 Xeon(R) Plat. 8358	10 Xeon(R) W-2255	12 Core i7-12700
GPU	Tesla V100	Tesla A100	Tesla A4000	GTX 3070

A. Scalability Analysis for Single Case

1) TSA: The computational performance and scalability are essential to evaluate the parallel approaches and relevant development. We retrieve the previous MPI CPU implementation in [31] for benchmarking purposes. It has balanced data partitions and communication approaches. As discussed in Section III, the improved model, unified data and task-level parallel strategy, and shared memory programming are embedded into the single-source heterogeneous solution. As illustrated in Fig. 5, we profile the performance scalability at each number of processes used on Node 1 and the cumulative communication cost at 16 processes using a 9216-bus system. The labeled numbers next to the best performance in the left chart indicate the maximum speedup of each serial implementation. Compared to the implementation in [31], our program outperforms the previous research outcome as it optimizes the overall computations and lowers data

communication overhead. The result shows that the pure CPU can finish the task in 1.99 seconds. As expected, the GPU for large matrix operations dominates the computational performance among the three, as shown in the hybrid switch line. It achieves a minimum of 1.84 seconds run at eight processes. However, the major contribution of the scalability of our hybrid parallelism comes from the DAE solving accelerated on the CPU. MPI-based large distributed linear algebra on a single GPU makes the computing resource and memory space tight for completing intensive jobs concurrently.

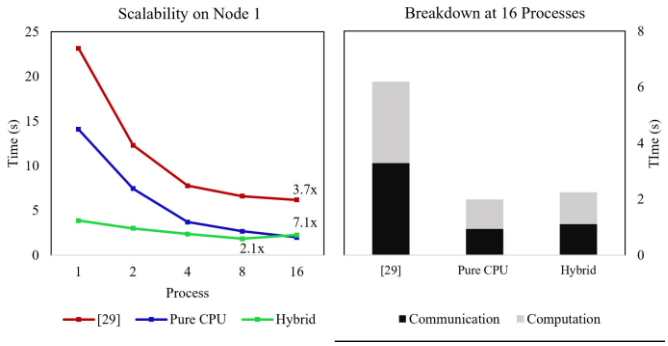

Fig. 5. The scalability and the percentage of communications (16 processes) of multiprocessing-based TSA programs.

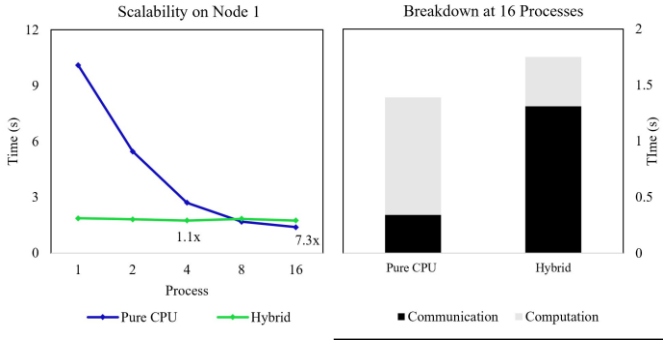

Fig. 6. The scalability and the percentage of communications (16 processes) of multiprocessing-based NR ACPF programs.

2) NR ACPF: In this work, the power flow is calculated to converge within five iterations. Thus, the Jacobian matrix is formed five times for each execution. We use the same 9216-bus system to illustrate the program's scalability, as shown in Fig. 6. The pure CPU parallelism (1.39 seconds with 16 processes) wins as it generates much less communication overhead than the hybrid version (1.75 seconds at 4 and 16 processes). Another reason is that the problem size is not enough for the GPU to beat the parallel CPU version. As the process increases, the same observation can be found. A single GPU for handling multiple processes of concurrency is limited in its computing capability. It also accumulates the host-device waste for large data offloading and downloading.

B. Performance across Cases

1) TSA: Another multiprocessing method, the Portable, Extensible Toolkit for Scientific Computation (PETSc [34]) CPU applications in [35] and [31] have been proven efficient by adopting the built-in parallel data structure and automated information exchange methods. It helps save hard programming effort when developing scientific software. In this paper, the tested PETSc program follows the optimized reduced **Y** workflow in Section III. A to speed up the overall computational performance. Regarding each problem size, we benchmark the best FTRT performance of it with our solution on Node 1. Fig. 7 shows the execution time comparison. For the 18432-bus system, the CPU-based parallelism significantly boosts performance with only 5.28 seconds (3.9x speedup over PETSc). Although PETSc reduces explicit parallel processing for matrix manipulations, the required data type conversions for certain operations cost a lot. Thus, it needs more cores to share the workload. The hybrid switch has the best results on the three largest system tests, demonstrating that GPU is the better handler for full **Y** and reduced **Y** formation once the power system expands the topology beyond 9216-bus. Overall, our solution gives considerable performance improvements.

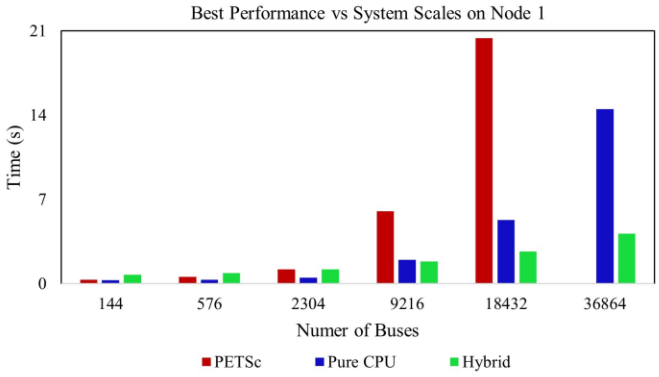

Fig. 7. The benchmark of the best FTRT performance regarding different system scales using a PETSc implementation and the developed single-source solution of TSA.

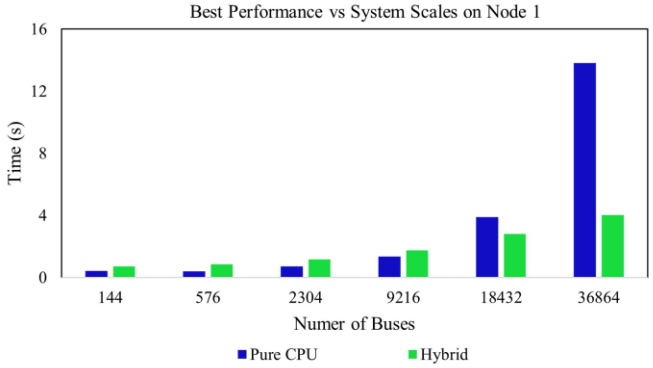

Fig. 8. The benchmark of the best performance regarding different system scales using the developed single-source solution of NR ACPF.

2) NR ACPF: As demonstrated in Fig. 8, the execution cost of NR ACPF has similar trends to the Fig. 7. As the problem becomes more intensive, the hybrid mode gives a much better result beyond the 9216-bus system. Its computations overcome the cost of bulk data movement and MPI communications. In the largest case, the hybrid mode can achieve 4.03 seconds of execution with one process, whereas the CPU only finishes in 13.8 seconds with 16 processes.

TABLE II PURE CPU PARALLELISM (LEFT) AND HYBRID SWITCH (RIGHT) EXECUTION TIME (SEC) ON DIFFERENT TEST PLATFORMS.

System Bus	TSA Node 1	TSA Node 2	TSA Lab Server	TSA PC	ACPF Node 1	ACPF Node 2	ACPF Lab Server	ACPF PC
144	0.29 | 0.73	0.29 | 0.66	0.31 | 0.51	0.20 | 0.64	0.42 | 0.73	0.40 | 0.58	0.46 | 0.56	0.26 | 0.62
576	0.33 | 0.86	0.34 | 0.74	0.35 | 0.70	0.28 | 0.79	0.40 | 0.89	0.40 | 0.77	0.48 | 0.75	0.35 | 0.77
2304	0.49 | 1.17	0.50 | 1.01	0.52 | 1.11	0.41 | 1.46	0.73 | 1.25	0.53 | 1.02	0.58 | 1.18	0.44 | 1.51
9216	1.99 | 1.84	1.45 | 1.27	1.67 | 1.84	1.27 | 2.25	1.39 | 1.75	1.32 | 1.30	1.80 | 1.91	1.92 | 2.34
18432	5.28 | 2.66	3.58 | 1.45	4.59 | 2.74	3.78 | 3.23	3.89 | 2.81	3.36 | 1.52	4.88 | 2.93	3.97 | 3.33
36864	14.5 | 4.17	9.89 | 2.33	14.3 | 5.52	14.6 | 6.84	13.8 | 4.03	9.69 | 2.38	16.0 | 6.11	16.5 | 6.79

C. Comparison between Different Test Platforms

In addition to the test on Node 1, we allocate three other settings to validate our program efficiency. Table II lists the TSA and NR ACPF test results. The numbers in bold indicate the best performance of each case. Surprisingly, the PC makes the overall fastest executions on CPU for the systems under 9216 buses for TSA. This is mainly because of the short interconnection between cores. Both CPU and hybrid parallelism of our program tested on Node 2 have exceptional performance for cases beyond 9216 buses. For the largest 36864-bus system, a recorded 2.33-second run is 12.9x faster than real-time. Compared to the research in [30], which allocates over 100 cores to pursue FTRT, the facts of our test make processing complex TSA jobs on large supercomputing facilities unnecessary. An office workstation attached to a discrete GPU is ready to fulfill the FTRT criteria on either hardware configuration. NR ACPF also runs more efficiently for smaller cases on the CPU. GPU should be considered for further acceleration if a complex model and time-critical computation are required.

D. Discussion

The performance portability addresses the applications that can be executed on different hardware platforms with minimal structural modifications. Current single-source heterogeneous computing prefers high-performance portable implementations due to the ease of parallel realization. However, the investigations on the advantages of using Python over traditional C++/CUDA-based approaches are still limited. In this work, developing Python-based TSA and NR ACPF applications with straightforward array computing saves coding complexity and minimizes further tuning steps. The mature array interfaces efficiently overcome the hardware variation barrier by combining MPI. The comparison for the programming effort of existing parallel TSA implementations is listed in Table III. Our multiprocessing heterogeneous solution beats the others using the fewest lines of code. More importantly, the accelerator-switchable program achieves decent speedup with much fewer resources. Table IV provides evidence that the NR ACPF's hybrid setting examines the largest test case with heterogeneous accelerations. The research so far has tried the comparable test sizes on only the CPU.

TABLE III CODING EFFORT BENCHMARK FOR TSA.

Approach	OpenMP [22]	PETSc [35]	PETSc [31]	This Paper
Language	Fortran	C/C++	Python	Python
Code Line	1126	1978	911	487

TABLE IV CASE SIZE AND PERFORMANCE COMPARISON OF HETEROGENEOUS NR-BASED POWER FLOW STUDY.

Reference	[36]	[37]	[38]	[39]	This Paper
Case Size (bus)	8503	2869	9241	2383	36864
Execution Time (s)	0.73	0.22	8.65	10.24	2.38

V. CONCLUSION

Heterogeneous computing is well-known to speed up scientific software with various computing accelerators. A Python-based single-source programming approach with detailed parallel design and implementations is proposed. The performance of TSA and NR ACPF has been boosted through pure CPU and CPU+GPU. The test results indicate that the heterogeneous configurations to accelerate the program executions are robust regarding different system scales by selecting any accelerator type. The significant improvements indicate that scientific applications in high-level languages have the potential to be applied to achieve promising computational speedup and portability based on hardware settings. A powerful framework will be built with 1) a generalized heterogeneous computing environment for automatically detecting the most adaptive program configuration regarding the problem sizes, 2) a comprehensive study for the suitability of the proposed approach regarding other types of accelerators, and 3) an extension design to merge the other numerical models under the proposed scheme.

VI. REFERENCES

[1] A. A. Khokhar, V. K. Prasanna, M. E. Shaaban, and C.-L. Wang, "Heterogeneous computing: Challenges and opportunities," Computer, vol. 26, no. 6, pp. 18–27, 1993.

[2] R. Chandra, L. Dagum, D. Kohr, R. Menon, D. Maydan, and J. McDonald, Parallel programming in OpenMP. Morgan kaufmann, 2001.

[3] S. Wienke, P. Springer, C. Terboven et al., "Openacc—first experiences with real-world applications," in European Conference on Parallel Processing. Springer, 2012, pp. 859–870.

[4] H. C. Edwards, C. R. Trott, and D. Sunderland, "Kokkos: Enabling manycore performance portability through polymorphic memory access patterns," Journal of parallel and distributed computing, vol. 74, no. 12, pp. 3202–3216, 2014.

[5] E. Zenker, B. Worpitz, R. Widera, A. Huebl, G. Juckeland, A. Knüpfer, W. E. Nagel, and M. Bussmann, "Alpaka–an abstraction library for parallel kernel acceleration," in 2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW). IEEE, 2016, pp. 631–640.

[6] X. Yang, E. Sliheet, R. Iriye, D. Reynolds, and W. Geng, "Optimized parallelization of boundary integral poisson-boltzmann solvers," Computer Physics Communications, vol. 299, p. 109125, 2024.

[7] L. Wang, R. Thiagarajan, S. Jin, and Z. Zhang, "Accelerating simulation for high-fidelity pv inverter system reliability assessment with high-performance computing," in 2022 IEEE 49th Photovoltaics Specialists Conference (PVSC). IEEE, 2022, pp. 0178–0182.
[8] X. Yang, "Tools for biomolecular modeling and simulation," 2024.
[9] K. H. M. Kwok, M. Kortelainen, G. Cerati, A. Strelchenko, O. Gutsche, A. R. Hall, S. Lantz, M. Reid, D. Riley, S. Berkman et al., "Application of performance portability solutions for gpus and many-core cpus to track reconstruction kernels," in EPJ Web of Conferences, vol. 295. EDP Sciences, 2024, p. 11003.
[10] S. A. Ullah, X. Yang, B. Jones, S. Zhao, W. Geng, and G.-W. Wei, "Bridging eulerian and lagrangian poisson–boltzmann solvers by eses," Journal of computational chemistry, vol. 45, no. 6, pp. 306–320, 2024.
[11] J. Chen, Y. Xu, X. Yang, Z. Cang, W. Geng, and G.-W. Wei, "Poisson-boltzmann-based machine learning model for electrostatic analysis," Biophysical Journal, vol. 123, no. 17, pp. 2807–2814, 2024.
[12] S. Lantz, K. McDermott, M. Reid, D. Riley, P. Wittich, S. Berkman, G. Cerati, M. Kortelainen, A. R. Hall, P. Elmer et al., "Speeding up particle track reconstruction using a parallel kalman filter algorithm," Journal of Instrumentation, vol. 15, no. 09, p. P09030, 2020.
[13] C. Wang, L. Wang, and S. Jin, "A single-source multiprocessing parallelism for heterogeneous acceleration of power system dynamic simulation," in 2023 North American Power Symposium (NAPS). IEEE, 2023, pp. 1–6.
[14] X. Yang, S. Zhao, and W. Geng, "A regularized matched interface and boundary method (mib) for solving polarizable multipole poisson-boltzmann model," arXiv preprint arXiv:2511.02714, 2025.
[15] J. Reinders, M. Voss, P. Reble, R. Asenjo-Plaza et al., "C++ for heterogeneous programming: oneapi (dpc++ and onetbb)," 2020.
[16] R. Keryell, R. Reyes, and L. Howes, "Khronos sycl for opencl: a tutorial," in Proceedings of the 3rd International Workshop on OpenCL, 2015, pp. 1–1.
[17] The MPI Forum, Corporate, "MPI: A message passing interface," in Proceedings of the 1993 ACM/IEEE conference on Supercomputing, 1993, pp. 878–883.
[18] H. Bulat, D. Frankovic, and S. Vlahinic, "Enhanced contingency analysis—a power system operator tool," Energies, vol. 14, no. 4, p. 923, 2021.
[19] 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 2014 47th Hawaii International Conference on System Sciences. IEEE, 2014, pp. 2536–2545.
[20] P. M. Anderson and A. A. Fouad, Power system control and stability. John Wiley & Sons, 2008.
[21] L. L. Grigsby, Power system stability and control. CRC press, 2007.
[22] S. Jin, Y. Chen, D. Wu, R. Diao, and Z. H. Huang, "Implementation of parallel dynamic simulation on shared-memory vs. distributed-memory environments," IFAC-PapersOnLine, vol. 48, no. 30, pp. 221–226, 2015.
[23] L. Dalcín, R. Paz, and M. Storti, "MPI for Python," Journal of Parallel and Distributed Computing, vol. 65, no. 9, pp. 1108–1115, 2005.
[24] NVIDIA, "Multi-process service," Oct 2021. [Online]. Available: https://docs.nvidia.com/deploy/mps/index.html
[25] E. Wang, Q. Zhang, B. Shen, G. Zhang, X. Lu, Q. Wu, and Y. Wang, "Intel math kernel library," in High-Performance Computing on the Intel Xeon Phi. Springer, 2014, pp. 167–188.
[26] M. Naumov, L. Chien, P. Vandermersch, and U. Kapasi, "Cusparse library," in GPU Technology Conference, 2010.
[27] S. Chetlur, C. Woolley, P. Vandermersch, J. Cohen, J. Tran, B. Catanzaro, and E. Shelhamer, "cudnn: Efficient primitives for deep learning," arXiv preprint arXiv:1410.0759, 2014.
[28] C. Wang, S. Jin, R. Huang, Q. Huang, and Y. Chen, "A configurable hierarchical architecture for parallel dynamic contingency analysis on gpus," IEEE Open Access Journal of Power and Energy, vol. 10, pp. 187–194, 2023.
[29] A. W. Max, "Inverting modified matrices," in Memorandum Rept. 42, Statistical Research Group. Princeton Univ., 1950, p. 4.
[30] B. Sullivan, J. Shi, M. Mazzola, and B. Saravi, "Faster-than-real-time power system transient stability simulation using parallel general norton with multiport equivalent (pgnme)," in 2017 IEEE Power Energy Society General Meeting, 2017, pp. 1–5.
[31] C. Wang, L. Wang, and S. Jin, "Adaptive development of parallel power system dynamic simulation application in python," Automated Systems, Data, and Sustainable Computing, OKIP, 2022.
[32] V. Jalili-Marandi and V. Dinavahi, "Simd-based large-scale transient stability simulation on the graphics processing unit," IEEE Transactions on Power Systems, vol. 25, no. 3, pp. 1589–1599, 2010.
[33] Z. Wang, A. Chakrabortty, C. Kelley, X. Feng, and P. Franzon, "Improved numerical methodologies on power system dynamic simulation using gpu implementation," in 2019 IEEE Power Energy Society Innovative Smart Grid Technologies Conference (ISGT), 2019, pp. 1–5.
[34] S. Balay, S. Abhyankar, M. Adams, J. Brown, P. Brune, K. Buschelman, L. Dalcin, A. Dener, V. Eijkhout, W. Gropp et al., "Petsc users manual," 2019.
[35] S. Jin, Z. Huang, R. Diao, D. Wu, and Y. Chen, "Comparative implementation of high performance computing for power system dynamic simulations," IEEE Transactions on Smart Grid, vol. 8, no. 3, pp. 1387–1395, 2017.
[36] G. Zhou, X. Zhang, Y. Lang, R. Bo, Y. Jia, J. Lin, and Y. Feng, "A novel gpu-accelerated strategy for contingency screening of static security analysis," International Journal of Electrical Power & Energy Systems, vol. 83, pp. 33–39, 2016.
[37] X. Su, C. He, T. Liu, and L. Wu, "Full parallel power flow solution: A gpu-cpu-based vectorization parallelization and sparse techniques for newton–raphson implementation," IEEE Transactions on Smart Grid, vol. 11, no. 3, pp. 1833–1844, 2019.
[38] I. Araujo, V. Tadaiesky, D. Cardoso, Y. Fukuyama, and A. Santana, "Simultaneous parallel power flow calculations using hybrid cpu-gpu approach," International Journal of Electrical Power & Energy Systems, vol. 105, pp. 229–236, 2019.
[39] J. Singh and I. Aruni, "Accelerating power flow studies on graphics processing unit," in 2010 Annual IEEE India Conference (INDICON). IEEE, 2010, pp. 1–5.