Parallel Dynamic Simulation: MPI Implementation

Cong Wang
School of Computing
Clemson University
North Charleston, SC, USA
Email: cong2@clemson.edu

Liwei Wang
School of Computing
Clemson University
North Charleston, SC, USA
Email: liweiw@clemson.edu

Shuangshuang Jin
School of Computing
Clemson University
North Charleston, SC, USA
Email: jin6@clemson.edu

***Abstract*—Large-scale power system modeling and simulation is essential to understand the dynamics of the components in a distribution network. There are several commercial software tools developed by industrial organizations leveraging the serial programs to fulfill the planning and diagnostic requirement of the system. However, in a time-critical situation, a serial run cannot provide real-time solutions due to the intensive computations and the use of limited computing resources. Therefore, to accelerate the program execution time, High-Performance Computing (HPC) technology is introduced. In this report, two Message Passing Interface (MPI) versions of power system dynamic simulation are presented with the details of the proposed method, implementation, and test result discussion. With 16 CPU cores and 30 GB memory, the speedup can reach to more than 5 times faster over the serial version by testing a 3600-bus-1200-generator power grid case.**

***Keywords*—power system, dynamic simulation, HPC, MPI, speedup**

I. Introduction

Large-scale power system modeling and simulation in the computing domain requires efficient time-critical functions to plan and analyze its complexity and dynamics in real-time. High-Performance Computing (HPC) techniques are ways to address these kinds of issues. This report mainly discusses 1) background of dynamic transient stability analysis in power system including the algorithm of matrix manipulation and simulation process; 2) proposed parallel approaches to showcase how the performance boost can be applied within Python and C++ computing environment; 3) performance comparison and evaluation for such implementations by using two different testing cases. We also briefly conclude our strategies and future work in the final section of this report. This introduction needs to be expanded

II. Background

A power system dynamic simulation program generally consists of nodal admittance matrix (**Y** matrix) building and multiple time-step simulation. It is used to evaluate the system's transient trajectories when there is a system disturbance (fault), e.g. a sudden change in generator or load, or a network short circuit followed by protective branch switching operations, etc. Modeling the system dynamics and network requires the computationally intensive time-domain solution of numerous differential and algebraic equations (DAEs) 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 **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 []. In this section, we introduce the fundamentals of how the program is established.

A. *Y Matrix and Reduced Y Matrix*

In power engineering, **Y** matrix is an N x N matrix describing a power system typology with the information about N buses, M generators, and Z branches. In realistic systems that contain thousands of buses, the full **Y** matrix is quite sparse. Each bus in a real power system is usually connected to only a few other buses through the transmission lines [].

The algebraic equations in (1) can be represented by (2).

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

The vector of current injections **I** at each bus is expressed as the production of full **Y** bus and the vector of bus voltages **V**. 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, full **Y** matrix can be reduced to only contain generator internal buses, $\mathbf{Y}'$. According to [**?**] and [**?**], (3) represents the logic of acquiring reduced nodal admittance matrix,

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

$\mathbf{Y}_{MM}$ 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

$$\mathbf{Y}' * \mathbf{V}' = \mathbf{I}', \tag{4}$$

where $\mathbf{I}'$ is the current injection and $\mathbf{V}'$ is the internal voltages of generators.

Parallelizing full **Y** matrix and reduced **Y** matrix manipulations based on pre-fault, on-fault, and post-fault conditions of a given power system serve as the first time-saving task for the program performance.

B. *Simulation*

The intensive computation involved in the injected current calculation, power-angle equations formulation, machine swing equations formation, and numerical integration makes deriving solutions a dominant time-consuming limiting factor

to meet the real-time constraints that characterize the dynamic assessment of system security. The equations of motion for an individual generator i in the complex system could be represented by (5) if a classical generator model is used in the dynamic simulation,

$$\begin{cases} \dfrac{dw_i}{dt} = \dfrac{w_s}{2H_i}(P_{mi} - P_{ei} - D_i(w_i - w_s)) \\ \dfrac{d\theta_i}{dt} = w_i - w_s \end{cases} \tag{5}$$

for generator i, H_i is the inertia constant, w_i is the speed, w_s is the synchronous speed, P_{mi} and P_{ei} are the mechanical power input and active power at the air gap, D_i is the damping coefficient, and θ_i is the angular position of the rotor in the electrical radians with respect to synchronously rotating reference [].

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 to be solved. The Modified Euler method [**?**] presented in (6),

$$\begin{cases} Predict: y_{i+1} = y_i + hf(t_i, y_i) \\ Update: y_{i+1} = y_i + \frac{h}{2}[f(t_i, y_i) + f(t_{i+1}, y_{i+1})] \end{cases} \tag{6}$$

as the most commonly used integration approach in power system analytics, needs the network equations to be solved twice (predictions and updates) at each time step, whereas the Adams-Bashforth method [] only requires once. The simplified formula of the two-step Adams-Bashforth integration can be summarized in equation (7).

$$y_{i+1} = y_i + \frac{3h}{2}f(t_i, y_i) - \frac{h}{2}f(t_{i-1}, y_{i-1}) \tag{7}$$

Given the input variables, h is the step size, $f(t, y) = y'$ and $y(t_0) = \alpha$, where α is the initial value of y at time step t_0, $t_{i+1} = t_i + h$. In this work, the successful substitution of the Modified Euler method by exploiting the parallel Adams-Bashforth method has more accelerating potentials in both mathematical and computational perspectives.

III. Proposed Approach

This section profiles the logics for the developed algorithm and the parallel strategy of dynamic simulation in Python and C++, respectively. Python is one of the high-level programming languages in the computer science domain, but since Python programs are interpreted, they are inherently slower than equivalent code written in compiled languages. Even so, Python provides thousands of object-oriented interfaces and millions of function calls which make it easy and save to program. Moreover, a wide variety of packages, tools, and strategies have been developed over the years to overcome the limitation. To compare with the Python version of parallel code, C++ programming language is chosen along with the use of PETSc [**?**], which is a portable, extensible toolkit for scientific computation.

A. Parallel Implementation in Python

In our development, to take advantages of distributed computing architecture, mpi4py [] is used to support point-to-point (sends, receives) and collective (broadcasts, scatters, gathers) communications of Python object exposing the single-segment buffer interface (NumPy arrays, built-in bytes, string, array objects, etc.). Figure 2 summarizes the overall algorithm and implementations.

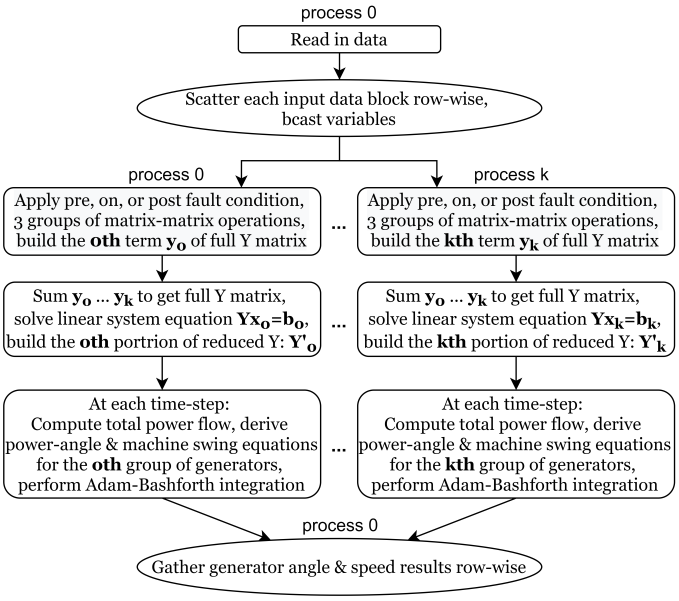

Fig. 1. Dynamic simulation parallel strategy in Python

The working program splits the input data blocks row-wise, which means each process should own a portion of information of buses, branches, and generators. For each fault condition, in a full $\mathbf{Y}$ matrix formation, the three groups of matrix-matrix operations involved can be boosted by downsizing the configured matrices. The first group of them is shown in (8) as an example,

$$\begin{cases} \mathbf{D}_{ZN} = \alpha * \mathbf{CH}_{ZZ} * \mathbf{F}_{NZ}^T + \beta * \mathbf{D}_{ZN} \\ \mathbf{Y}_{NN} = \alpha * \mathbf{F}_{NZ} * \mathbf{D}_{ZN} + \beta * \mathbf{Y}_{NN} \end{cases} \tag{8}$$

where α and β are scalars. Since the original left side **CH** matrix is a diagonal matrix with size Z x Z, it can be initialized and value-assigned on each process to **ch** with the size z x z based on the split data size the process holds. In addition, the right side matrix **F** (N x Z) can go with **f** (N x z) and the addition term **D** can be changed to **d** with z x N. Consequently, on each process, the operations are converted into (9). By leveraging the outcomes from their former group, the other two groups can make their own similar operation. Finally, each process still obtains a resulting N x N **y** matrix, however, the summation of all **y** matrices from all processes is expected equal to the original full **Y** matrix.

$$\begin{cases} \mathbf{d}_{zN} = \alpha * \mathbf{ch}_{zz} * \mathbf{f}_{Nz}^T + \beta * \mathbf{d}_{zN} \\ \mathbf{y}_{NN} = \alpha * \mathbf{f}_{Nz} * \mathbf{d}_{zN} + \beta * \mathbf{y}_{NN} \end{cases} \tag{9}$$

For the serial reduced **Y** matrix operation, the lower-level linear algebraic solver ZGESV[] is utilized to solve the system equation $\mathbf{Y}_{NN}$*$\mathbf{X}_{NM}$ = $\mathbf{Y}_{NM}$ for each fault condition. In our version, since the system and its following matrix-matrix operations are easy to be parallelized, for each process, it only solves the work assigned to it and finally outputs a partial

```
For each timestep "ts":
  // Compute internal voltage "eprime" based on q-axis voltage "eqprime" and generator angle "genAngle";
  VecCopy(genAngle_s0, vecTemp);
  VecScale(vecTemp, PETSC_i);
  VecExp(vecTemp);
  VecPointwiseMult(eprime_s0, eqprime, vecTemp);

  // Compute current "current" based on reduced Y-Bus "prefy/fy/posfy" and "eprime";
  if (flag1 == 0) MatMultTranspose(prefy, eprime_s0, current);
  else if (flag1 == 1) MatMultTranspose(fy, eprime_s0, current);
  else if (flag1 == 2) MatMultTranspose(posfy, eprime_s0, current);

  // Compute generator's electric real power output "pElect";
  VecConjugate(current);
  VecPointwiseMult(pElect, eprime_s0, current);
  VecCopy(pElect, vecTemp);
  VecConjugate(vecTemp);
  VecAXPY(pElect, 1.0, vecTemp);
  VecScale(pElect, 0.5);

  // Compute off-change generator angle "dgenAngle" and speed "dgenSpeed";
  VecCopy(genSpeed_s0, vecTemp);
  VecShift(vecTemp, -1.0);
  VecScale(vecTemp, basrad);
  VecCopy(vecTemp, dgenAngle_s0);
  VecPointwiseMult(vecTemp, pElect, MVA);
  VecCopy(pMech, dgenSpeed_s0);
  VecAXPY(dgenSpeed_s0, -1.0, vecTemp);
  VecCopy(genSpeed_s0, vecTemp);
  VecShift(vecTemp, -1.0);
  VecPointwiseMult(vecTemp1, d0, vecTemp);
  VecAXPY(dgenSpeed_s0, -1.0, vecTemp1);
  VecCopy(h, vecTemp);
  VecScale(vecTemp, 2);
  VecPointwiseDivide(dgenSpeed_s0, dgenSpeed_s0, vecTemp);

  // Compute next-timestep generator angle "genAngle" and speed "genSpeed" with Euler step;
  VecCopy(genAngle_s0, genAngle_s1);
  VecAXPY(genAngle_s1, h1, dgenAngle_s0);
  VecCopy(genSpeed_s0, genSpeed_s1);
  VecAXPY(genSpeed_s1, h1, dgenSpeed_s0);

  // Compute next-timestep "eprime" based on the next timestep "genAngle";
  VecCopy(genAngle_s1, vecTemp);
  VecScale(vecTemp, PETSC_i);
  VecExp(vecTemp);
  VecPointwiseMult(eprime_s1, eqprime, vecTemp);
End of loop
```

Fig. 2. This is ME, replace this by AB implementation

reduced $\mathbf{Y}$ matrix ($\mathbf{y}'_{mM}$) as shown in (10), which can be used directly to compute the current injection $\mathbf{I}'$ in the simulation step.

$$\mathbf{y}'_{mM} = \mathbf{y}_{mM} - [\mathbf{Y}_{MN} * \mathbf{Y}_{NN}^{-1} * \mathbf{y}_{Nm}]^T \tag{10}$$

This part also needs some clarification

B. Parallel Implementation in C++

There are some strategies the C++ program follow differing to Python's, so please state these variances. Also, what I'm thinking here is inserting a part of C++ pseudo code such as Y matrix formation, reduced Y formation or parallel Adam Bashforth integration method to better illustrate how it works. It may not only make up the code effort contribution the CS paper should have, also it enumerates the variances so that the reviewers can compare this with Python strategies mentioned above.

Unlike Python implementation, which establishes the parallel program on each process by almost keeping the data separated, C++ version

C. System and Hardware Configuration

The working code is implemented on Clemson University's supercomputing facilities Palmetto Cluster [**?**], which is a Linux based environment comprised of 2,021 compute nodes (totaling 23,072 CPU cores) including 386 nodes equipped with NVIDIA Tesla GPUs. A computing node consisting of 16 CPU cores with 30 GB memory is requested to perform the

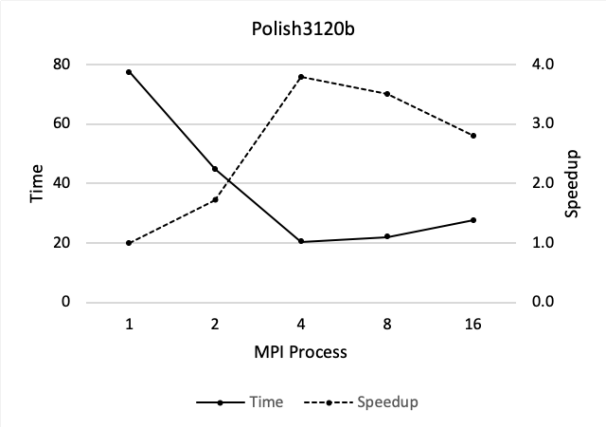

Fig. 3. The computational performance of parallel dynamic simulation on Polish3120b system in Python

tasks. To ensure the code was running on the same platform, GNU compiler collection (GCC), which has the ability to enable parallel interfaces such as OpenMP, OpenACC, and MPI, is selected.

IV. Results and Analysis

For testing the performance of the proposed algorithm, two widely acceptable power system test cases are selected to check the accuracy and scalability. The standard Siemens PTI format [**?**] representing power grid systems are adapted as the input to our two different dynamic simulation packages.

A. Testing Cases

The Polish 3120-bus-93-generator system (Polish3120b), which is converted from MATPOWER [**?**], an open-source tool for electric power system simulation and optimization, and a 3600-bus-1200-generator (IEEE3600b) system, which derives from the duplications of a small 9-bus-3-generator case, are selected as the test cases. For both cases, the simulation time is 30 seconds with a time step of 0.01 second. A fault is applied at bus 6 at the 3rd second and cleared at the 3.05 second to mimic a system disturbance and the relevant generators' dynamics (e.g., mechanical angles and speeds, power angles and speeds) as a response to the disturbance.

B. Performance of Scalability

The scalability of a parallel application 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. As the number of computing cores increases, the computation time is expected to decrease until the capability is limited by the problem size once a certain number of computing cores is reached. Therefore, the larger IEEE3600b system is expected to have better scalability over the medium Polish3120b system. The speedup performance of the two programs is obtained by comparing computational times against the single-core CPU serial run and multi-core CPUs parallel run with various number of MPI processes.

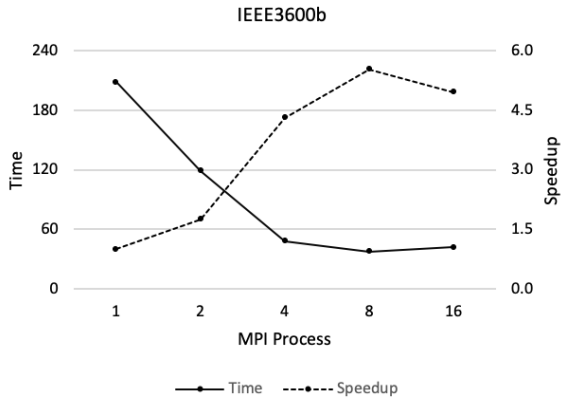

Fig. 4. The computational performance of parallel dynamic simulation on IEEE3600b system in Python

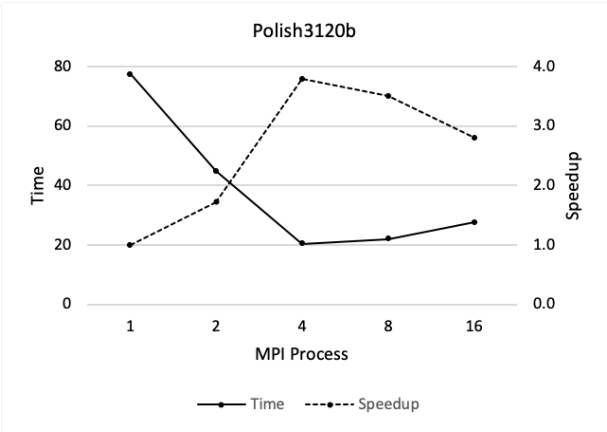

Fig. 5. The computational performance of parallel dynamic simulation on Polish3120b system in C++

1) Polish3120b: as Fig. 3 and Fig. 5 demonstrate, the CPU serial run takes 77.36 seconds in Python while the C++ run takes only 65.34 seconds. Both versions achieve their best performance using 4 CPU cores (20.41 seconds for Python, 15.17 seconds for C++) where the consumptions of **Y** matrix, reduced **Y** matrix building, and simulation are all the least throughout the tested processes. Python reaches a speedup of 3.79 times over the CPU serial run and C++ goes even faster with 4.21 times. No significant speedup gains can be achieved beyond 4 MPI processes due to the aforementioned saturation constrained by the problem size.

2) IEEE3600b: the computational intensity increases dramatically due to the growth of buses and generators. It results in larger matrix operations and linear equations solving in terms of the proposed algorithm. Figure 4 and Figure 6 depicts the trends of both implementations. The CPU serial run takes over 200 seconds to complete a 30-second simulation after testing our Python program. When 8 MPI processes are used, the best performance is achieved with a speedup of 5.53 and a run time of 37.69 seconds, offering the fastest run time that much close to the real-time response. At 16 MPI processes, even though simulation time continues decreasing, since reduced **Y** matrix building bounds back more, the total performance is slightly behind. The C++ version...

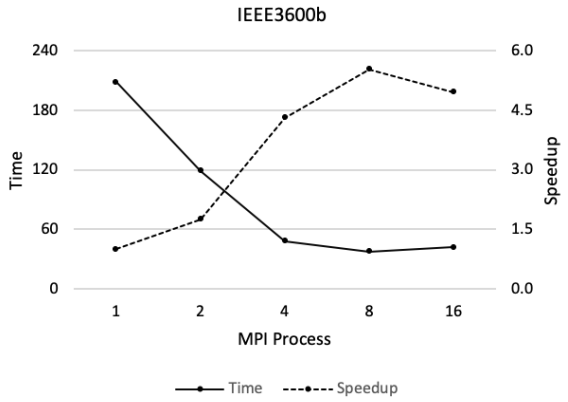

Fig. 6. The computational performance of parallel dynamic simulation on IEEE3600b system in C++

TABLE I. COMPARISON OF BOTH PYTHON AND C++ PERFORMANCE ACROSS ALL DIFFERENT NUMBER OF PROCESSES

Test Case / Process	1	2	4	8	16
Polish3120b	1.02	1.22	1.33	1,44	1.52
IEEE3600b	1.65	1.74	1.82	1.98	2.06

According to the testing results, it is observed C++ has better performance and more acceleration comparing to Python, Table II demonstrates how faster C++ can make over Python for two testing cases. We simply divide C++ run time by Python run time on each same tested MPI process number. At most, for the IEEE3600b system, the C++ version outperforms 2 times over Python. Considering the nature of these two programming languages and the distinctions of the proposed approaches, the performance differences between the two implementations are in an acceptable range for the parallel dynamic simulation. The above content in this section has some assumptions, charts, and arbitrary numbers, please correct them after completing the code and testing stuff

V. CONCLUSION AND FUTURE WORK

We have not reach our conclusion part yet, besides the marks, we still need to organize sentences, polish the words, and manange all citations

ACKNOWLEDGMENT

The research is funded by the Clemson University Restoration Institute. Clemson University is acknowledged for the generous allotment of computing time on Palmetto Cluster.

TABLE II. PERFORMANCE COMPARISON BETWEEN TWO INTEGRATION METHODS

Test Case / Method	Modified Euler	Adam-Bashforth	Speedup
Polish3120b	1.16	0.67	1.74
IEEE3600b	0.74	0.45	1.66

REFERENCES

[1] P.Faruki, A.Bharmal, V. Laxmi, V. Ganmoor, M.Gaur, M. Conti, and M. Ragarajan, "Android Security: A Survey of Issues, Malware Penetration, and Defenses," IEEE Communication Surveys and Tutorials, 2015, pp. 998-1022.

[2] Mcafee, "Threats Prediction", [Online]. Available: http://www.mcafee.com/es/resources/misc/infographic-threats-predictions-2015.pdf

[3] Android Malware Genome Project, (Online; Last Accessed Feb. 11, 2014). [Online]. Available: http://www.malgenomeproject.org/

[4] C. A. Castillo, "Android malware past, present, future," Mobile Working Security Group McAfee, Santa Clara, CA, USA, Tech. Rep., 2012.

[5] W. Zhou, Y. Zhou, X. Jiang, and P. Ning, "Detecting repackaged smartphone applications in third-party android marketplaces," in Proc. 2nd ACM CODASPY, New York, NY, USA, 2012, pp. 317–326. [Online]. Available: http://doi.acm.org/10.1145/2133601.2133640

[6] Android Security Overview, (Online; Last Accesed Dec. 25, 2013). [Online]. Available: http://source.android.com/devices/tech/security

[7] Google Bouncer: Protecting the Google Play market, (Online; Last Accessed Oct. 15, 2013). [Online]. Available: http://blog.trendmicro.com/trendlabs-security-intelligence/a-lookat-google-bouncer/

[8] J. Oberhide, Dissecting the Android Bouncer, (Online; Last Accessed Jun. 1, 2012). [Online]. Available: http://jon.oberheide.org/blog/2012/06/21/dissecting-the-android-bouncer/

[9] Exercising Our Remote Application Removal Feature, [Online]. Available: http://androiddevelopers.blogspot.in/2010/06/exercising-our-remote-application.html

[10] Kaspersky Security Bulletin 2013, Overall statistics for 2013, (Online; Last Accessed Feb. 11). [Online]. Available: https://www.securelist.com/en/analysis/204792318

[11] McAfee Labs Threats Report: Third Quarter 2013, (Online; Last Accessed Feb. 11). [Online]. Available: http://www.mcafee.com/uk/resources/reports/rp-quarterly-threatq3-2013.pdf