Effective Parallelism for Equation and Jacobian Evaluation in Large-Scale Power Flow Calculation

Hantao Cui, *Senior Member, IEEE*, Fangxing Li, *Fellow, IEEE*, and Xin Fang, *Senior Member, IEEE*

***Abstract*—This letter investigates parallelism approaches for equation and Jacobian evaluations in large-scale power flow calculation. Two levels of parallelism are proposed and analyzed: inter-model parallelism, which evaluates models in parallel, and intra-model parallelism, which evaluates calculations within each model in parallel. Parallelism techniques such as multi-threading and single instruction multiple data (SIMD) vectorization are discussed, implemented, and benchmarked as six calculation workflows. Case studies on the 70 000-bus synthetic grid show that equation evaluations can be accelerated by ten times, and the overall Newton power flow advances the state of the art by 20%.**

***Index Terms*—Power flow calculation, parallelism, multi-threading, single instruction multiple data (SIMD).**

I. Introduction

POWER flow calculation is a fundamental routine widely used for power system analysis and operation. In power systems with a large contingency set and various renewable scenarios, the calculation speed of power flow is crucial to ensure system security. Recent CPUs have stalled on clock rate and started to pack more cores and provide sophisticated instruction sets. Therefore, power flow workflows need to be fine-tuned to utilize new computing hardware.

Power flow calculation consists of four sequential tasks: (1) updating equation residuals, (2) updating Jacobian elements, (3) solving sparse linear equations, and (4) updating variable values and checking for convergence. Generally, task (4) is light-weight, and task (3) is handled by highly efficient external solvers [1], such as KLU on CPU or CuSparse on GPU [2], [3]. We investigate the parallelization within tasks (1) and (2).

Available parallel computing techniques include instruction-level, data, and task parallelism. Instruction-level parallelism utilizes dedicated instructions available on the platform. Data parallelism distributes the same computing task on large data sets across multiple processors. Task parallelism dispatches each processor core, using *multi-processing* or *multi-threading*, for one computing task on the same or different data. Given the size of power flow problems, instruction-level and task parallelism are more relevant.

Related work has explored instruction-level parallelism using Single Instruction Multiple Data (SIMD) vectorization for Task (3), namely, factorizing sparse matrices and solving sparse linear equations, on GPU [4], [5]. CPU multi-processing is reported in [6], [7] for power flow, both using OpenMP for message passing. A CPU-GPU architecture is proposed in [8] that combines GPU SIMD with multi-processing for power flow runs of many instances of the same power network. To our best knowledge, this letter is the first to quantify the effectiveness of SIMD and multi-threading for parallelizing the equation and Jacobian evaluations on modern CPUs.

In most existing tools, such as PSAT [9] and ANDES [10], the numerical equations and Jacobian elements are obtained by calling power flow models in serial, which is partially due to the limited parallelism supports in MATLAB and Python. In modern programming environments, power flow routines with instruction-level and task parallelism can be prototyped to provide directions for improving existing tools.

This work studies the parallelism for the equation and Jacobian evaluation tasks in power flow calculation. Since these two tasks share the same type of computing demand, namely, arithmetic evaluation of equations for residuals and Jacobian elements, the parallel procedures are the same. The main contributions are: 1) The concepts of inter- and intra-model parallelism are proposed. Implementation, advantages and limitations are discussed. 2) Six workflows, as the combinations of two inter-model and three intra-model workflows, are benchmarked using large-scale synthetic grids with up to 70 000 buses [11].

An an outcome, the Newton power flow package that implements the most effective parallelism is over 20% faster than the state-of-the-art for large-scale systems, as will be discussed in Section IV.

II. Power Flow Models and Workflows

A. Power Flow Formulation and Models

This work employs an extended power flow formulation that models the voltage control of PV generators and the voltage and phase angle controls of the slack generator. In addition the bus voltages and phase angles solved from the traditional formulation, this formulation allows checking reactive power limits in iterations, and same formulation can be used before and after a PV generator switches to a PQ load. The formulation

Manuscript received November 22, 2020; revised March 12, 2021 and April 10, 2021; accepted April 11, 2021. Date of publication April 15, 2021; date of current version August 19, 2021. This work was supported primarily by the Engineering Research Center Program of the National Science Foundation (NSF) and the Department of Energy under NSF Award Number EEC-1041877 and the CURENT Industry Partnership Program. Paper no. PESL-00332-2020. *(Corresponding author: Fangxing Li.)*

Hantao Cui and Fangxing Li are with the Department of Electrical Engineering and Computer Science, The University of Tennessee, Knoxville, TN 37996 USA (e-mail: hcui7@utk.edu; fli6@utk.edu).

Xin Fang was with the Department of EECS, The University of Tennessee and CURENT research center, Knoxville, TN 37996 USA (e-mail: allen.fangxin@gmail.com).

Color versions of one or more figures in this article are available at https://doi.org/10.1109/TPWRS.2021.3073591.

Digital Object Identifier 10.1109/TPWRS.2021.3073591

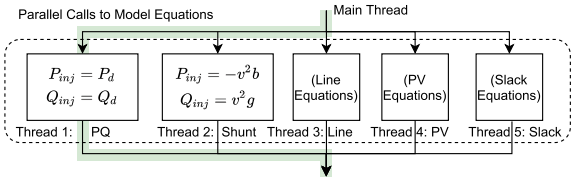

Fig. 1. Detailed view of inter-model parallelism for equations evaluations.

is given by the compact notation that

$$\boldsymbol{g}(\boldsymbol{y}) = \mathbf{0}\,, \tag{1}$$

where algebraic equations $\boldsymbol{g}$ in (2) are organized by grouping active power mismatches, reactive power mismatches, voltage control errors, and angle control error in order. Variables $\boldsymbol{y}$ in (3) are grouped by bus voltage angles, bus voltage magnitude, PV reactive power outputs, and Slack active power output in order.

$$\boldsymbol{g}(\boldsymbol{y}) = \left[\boldsymbol{g_p}, \boldsymbol{g_q}, \boldsymbol{g_V}, g_\theta\right]^T\,, \tag{2}$$

$$\boldsymbol{y} = \left[\boldsymbol{\theta}, \boldsymbol{V}, \boldsymbol{Q_g}, P_s\right]^T\,. \tag{3}$$

This formulation can be extended, for example, to include control modes of converters in power flow. We also note that this formulation has more equations and variables than the well-known one. Like most power flow programs, this work considers models including bus, PQ load, PV generator, slack generator, lines (including the π-model transmission line and two-winding transformer) and shunt capacitor. In particular, line injections are calculated on a per-line basis and summed at the connected buses. Since the power injections at each terminal are calculated independently, parallelization can be possible, as will be discussed in the following.

B. Parallel Workflows: Inter-Model and Intra-Model Parallelism

Two levels of parallelism are proposed for tasks (1) and (2). The first level is the *inter-model parallelism*, which aims to evaluate the equations and Jacobian elements of multiple models in parallel. A high-level view of inter-model parallelism is shown in Fig. 2(b), where each solid dot represents a model. A detailed view of the parallel part is shown in Fig. 1, where PQ, Shunt, and other models compute their power injection equations in parallel. Results from the parallel evaluations need to be joined as the whole system. For example, injections from lines, loads, and generators on the same bus will be joined through summation.

The inter-model parallelism is a coarse-grained approach typically assigns one processor core for each function of each model. However, this workflow has a limitation relevant for power flow: all jobs must have been completed before results can be joined. This limitation is known as Cannikin's Law, which suggests that, although threads run in parallel, the total time is bottlenecked by the slowest thread. For large systems, the line model is the slowest due to the instance number and calculation complexity.

The second level is the *intra-model parallelism*, a fine-grained approach that parallelizes independent calculations within models. Consider the active power injections at transmission line terminals:

$$P_h = v_h^2(g_L + g_{L,h}) - v_h v_k(g_L cos\theta_{hk} + b_L sin\theta_{hk})$$

$$P_k = v_k^2(g_L + g_{L,k}) - v_h v_k(g_L cos\theta_{hk} - b_L sin\theta_{hk})\,, \tag{4}$$

where v_h and v_k are the bus voltages at terminals h and k, and θ_{hk} is the voltage angle difference, g_L and b_L are the conductance and susceptance of the series component, and $g_{L,h}$ and $g_{L,k}$ are the conductance of the shunt-component. The intra-model parallelism aims to compute P_h and P_k in parallel due to independence. Two approaches are available for implementing intra-model parallelism: task parallelism or SIMD vectorization, which will be discussed in Section II-D. Intra-model parallelism can efficiently utilize processors since independent calculations of the same model usually have similar complexity, as are P_h and P_k in (4). However, intra-model parallelism is more difficult to implement, as one needs to carefully eliminate data racing within each model.

Further, one can create *nested parallelism* by combining the inter-model and intra-model parallelism to assign one core for each model and assign one core (or uses SIMD) for each equation-level calculation. At first glance, the idea may be appealing, given that all equation-level computations are executed in parallel. However, if task parallelism is used for both inter- and intra-model parallelism, processor resources may run out quickly.

C. Task Parallelism: Multi-Processing or Multi-Threading

As mentioned, task parallelism can be implemented with multi-processing or multi-threading to utilize multiple cores. By design, a process is a program in execution composed of code, current activity, data, heap, and stack. Processes can only share resources through techniques such as message passing or shared memory, which require explicit arrangements by the programmer [12]. Therefore, Multi-processing is ideal for computations where data exchanges between individual runs are not required, such as contingency screening and stochastic time-domain simulation [13].

A thread is a basic unit of CPU utilization within a process. By default, a thread shares the resources (such as code and data) of the process to which it belongs. Empirically, threads consume significantly less time to create and manage [12]. Such characteristics make multi-threading suitable for parallelizing equations and Jacobian update functions, which need to be called multiple times and require the passing of inputs (from the previous iteration) and outputs.

D. Single Instruction Multiple Data (SIMD) Vectorization

SIMD is a mechanism that vectorizes arithmetic operations at the processor level. This vectorization is completely different from MATLAB's vectorization, which expresses loop operations using vectors. For example, on Intel processors that supports AVX512, the 512-bit Advanced Vector Extension instructions, programs can pack 32 double-precision floating point calculations per clock cycle. To compute a power system with 32 000 line devices, for example, each equation in (4) only needs 1000 evaluations with AVX512.

However, SIMD is not automatic and requires meticulous implementation. Specifically, the following requirements must be satisfied: 1) Use vectors to store numericals in contiguous memory. 2) Eliminate bound checking when accessing vectors. 3) Use proper directives to suggest SIMD to the compiler. 4) Inspect the generated machine code to verify vectorizations.

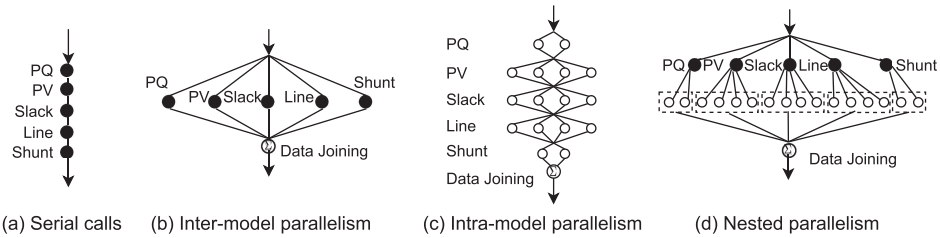

Fig. 2. Power flow workflows. (Solid dots) Aggregation of all equation/Jacobian evaluations in a model. (Circles) Each equation or Jacobian element evaluation.

TABLE I
COMPUTATION TIME FOR LINE EQUATIONS WITH AND WITHOUT SIMD

	Julia (SIMD)	Julia (Array)	Python (Array)	MATLAB
Time (ms)	0.54	5.2	12.2	1.2

Although SIMD is intricate to implement, the speedup is huge compared with scalar calculations, as will be shown in Section IV.

III. VALID IMPLEMENTATION AND BENCHMARK

To utilize multi-core processors, the power flow program must be implemented in a language that (1) compiles into bare-metal machine code (as opposed to virtual machine byte-code), and (2) is capable of dispatching multiple cores. Unfortunately, none of the two popular languages for scientific computing, namely, MATLAB and Python, meets the requirement. Due to the global interpreter lock (GIL), Python threads are executed one after another, defeating the purpose of simultaneous multi-threading. Python's multiprocessing does allow side-stepping GIL to achieve process-based parallelism. However, as discussed, processes are computationally costly to start and require careful arrangements by the programmer for data sharing. MATLAB provides a multi-processing-based parallel for-loop (parfor) solution, which occurs only if the parfor-loops are compiled into MEX functions using an OpenMP-compliant compiler. The restriction renders MATLAB unfriendly to structure parallel programs.

This work implements the multi-threaded workflows in the Julia language [14], which is high-level, just-in-time compiled, and multi-threaded. Julia's high-level syntax allows quick prototyping of the power flow models and equations. Under the hood, Julia uses the Low-Level Virtual Machine (LLVM) compiler to translate the code into platform-specific, highly optimized machine code. Most importantly, Julia can readily dispatch processor cores for multi-threading.

Run-time memory allocation is another factor that invalidates multi-threading efforts. One of the core challenges in high-performance computing is to minimize memory access time. Run-time allocations will incur access time and trigger garbage collection, which is another performance deal-breaker. In this work, equation calls are allocation-free, and Jacobian calls only allocate while assembling triplets into sparse matrices.

IV. CASE STUDIES

A. Test Systems and Environments

The effectiveness of parallelism is best validated using large test systems on recent-generation hardware. The study utilizes the Synthetic 70 000-bus system, which contains 88 207 branches and transformers, 160 780 variables, and 998 495 non-zero Jacobian elements. Simulations are performed on a workstation with the six-core Intel Xeon W-2133 and 32 GiB of memory running Ubuntu 16.04, Julia 1.5.2, Python 3.8.5, NumPy 1.19.1, and MATLAB 2020a. Computation time is measured using the Newton power flow.

TABLE II
DESCRIPTIONS OF THE SIX PROPOSED WORKFLOWS

#	Descriptions
(1)	Serial inter-model and intra-model execution
(2)	Serial inter-model execution, multi-threaded intra-model execution
(3)	Multi-threaded inter-model execution, serial intra-model execution
(4)	Multi-threaded inter-model and intra-model execution
(5)	Serial inter-model execution with intra-model SIMD
(6)	Multi-threaded inter-model execution with intra-model SIMD

B. The Effectiveness of SIMD Vectorization

SIMD is first evaluated for effectiveness by comparing implementations in Julia, Python, and MATLAB. The times to calculate the four equations of Line (active and reactive power injections at two terminals) are reported. SIMD is verified in the generated assembly code with instructions such as `vaddpd` and `vmulpd`, where `v` is vectorized, `add` and `mul` are addition and multiplication, and `pd` indicates packed data (as opposed to scalar data `sd`).

As reported in Table I, turning on SIMD for the same Julia code makes the computation is an order of magnitude faster. MATLAB records satisfactory performance compared with the SIMD version in Julia, because MATLAB utilizes SIMD under the hood for array operations. On the other hand, NumPy is relatively slow due to its limited SIMD support, which is an ongoing work.

C. Parallel Workflow Comparisons

The combination of two inter-model execution modes (serial or multi-threaded) and three intra-model execution modes (serial, multi-threaded, or SIMD) yields six possible workflows. Table II describes the six workflows, where Workflow (1) is serial and the other five involve parallelism. All the workflows are implemented in Julia with the same codebase, ruling out performance discrepancies of programming languages. The computation times are reported in Table III, where the circled number is the workflow number, and each cell contains the equation time and Jacobian time separated by a slash. The observations are: (a) The traditional serial workflow (1) is the slowest, and (2) the inter-model multi-threading alone has limited effect. (b) Workflow (3) - multi-threading within models - **speeds up by 5x**, but nested parallelism (4) is as slow as the serial workflow. (c) Workflow (5) - SIMD within each model with serial model

TABLE III
EQUATION / JACOBIAN TIME (MS) FOR SIX WORKFLOWS

Intra-Model \ Inter-Model	Serial	Multi-threaded
Serial	① 5.5 / 21.5	② 5.3 / 20.9
Multi-threaded	③ 1.2 / 4.2	④ 5.7 / 20.9
SIMD	⑤ 0.6 / 2.8	⑥ 0.5 / 2.7

TABLE IV
EQUATION TIME (μs) BY MODEL IN THREE INTRA-MODEL WORKFLOWS

Workflows \ Model	PQ	PV	Slack	Line	Shunt
Serial	13.46	18.53	0.01	5172.90	1.66
Threaded	10.81	11.10	6.50	1206.60	7.53
SIMD	17.13	7.60	0.01	520.6	1.28

TABLE V
SYNTHETIC GRID COMPUTATION TIME IN MILLISECONDS

	2000	10k	25k	70k
Workflow (1) – Serial	41.1	245.2	632.7	2608.9
Workflow (5) – Serial SIMD	37.5	225.8	624.4	2470.5
MATPOWER	100.0	238.0	624.0	2994.0
Speed up over Serial	8.8%	6.6%	1.3%	5.3%
Speed up over MATPOWER	152%	3%	0.1%	20.1%

executions - **has a 10x speedup**. Workflow (6) with inter-model multi-threading brings slight improvements to workflow (5).

To explain why coarse-grained inter-model multi-threading has limited effects, such as in workflow (2) and (6), we break down the equation computation time by power flow model. As explained in Section II-B, the total CPU time of a multi-threaded parallel program is determined by the bottleneck, which is the Line model in power flow. This can be verified in Table IV that workflows (2)'s and (6)'s time is roughly equal to the time for Line equations. Therefore inter-model parallelism is ineffective due to the slow execution of the Line model.

Also, the nested parallelism in workflow (4) has an adverse effect compared with workflow (2) due to thread limits. The processor ends up executing one model in each thread and serially run threads within each model, which is similar to workflow (2). Even worse, workflow (4) has more overhead than (2) due to the creation and termination of threads, making it slower than the less-optimized workflow (2).

Therefore, workflow (5) is recommended considering the implementation complexity and effectiveness. Compared with workflow (6), workflow (5) is almost as fast and is single-threaded, which is far more maintainable. Multiple power flow calculation jobs can run in parallel processes efficiently, as single-threaded jobs will not incur thread switching overhead.

Finally, Table V compares workflow (5) with workflow (1) and the state-of-the-art MATPOWER package [15] to solve Newton power flow for four synthetic systems. MATPOWER's time is the average of five consecutive runs starting from the second run (to allow for data caching and pre-compilation). Since this work accelerates the equation and Jacobian evaluations, the linear equation solver time is the same for workflows (1) and (5). Still, workflow (5) is 8.8% faster for the 2000-bus system and 5.3% faster for the 70k-bus system. Workflow (5) has similar performance to MATLAB for the 10k-bus and the 25k-bus systems, due to MATPOWER's faster linear equation solver but slower equation and Jacobian routines. For large systems, equation and Jacobian time become more influential; the proposed workflow (5) thus outperforms MATPOWER by 20.1%.

V. CONCLUSION

This letter identifies the most effective parallel workflow for power flow calculation by investigates six workflows using multi-threading and SIMD and vectorization. The concepts of inter-model and intra-model parallelism are proposed. Benchmarks using the Synthetic 70k-bus system shows that (1) coarse-grained inter-model parallelism is ineffective because it cannot eliminate the Line model computation bottleneck, and (2) serially executed models with SIMD is the most effective approach owning to its 10x speed up and single-threadedness, which avoids threading overhead.

Future work involves benchmarking the proposed parallelism workflows with the Fast Decoupled method on heterogeneous computing devices, including CPUs and GPUs.

REFERENCES

[1] F. Milano, "A python-based software tool for power system analysis," in *Proc. IEEE Power Energy Soc. Gen. Meeting*, 2013, pp. 1–5.
[2] G. Zhou, R. Bo, L. Chien, X. Zhang, S. Yang, and D. Su, "Gpu-accelerated algorithm for online probabilistic power flow," *IEEE Trans. Power Syst.*, vol. 33, no. 1, pp. 1132–1135, Jan. 2018.
[3] X. Li, F. Li, H. Yuan, H. Cui, and Q. Hu, "GPU-based fast decoupled power flow with preconditioned iterative solver and inexact newton method," *IEEE Trans. Power Syst.*, vol. 32, no. 4, pp. 2695–2703, Jul. 2017.
[4] V. Jalili-Marandi and V. Dinavahi, "SIMD-based large-scale transient stability simulation on the graphics processing unit," *IEEE Trans. Power Syst.*, vol. 25, no. 3, pp. 1589–1599, Aug. 2010.
[5] C. Vilachá, J. C. Moreira, E. Míguez, and A. F. Otero, "Massive Jacobi power flow based on SIMD-processor," in *Proc. 10th Int. Conf. Environ. Elect. Eng.*, May 2011, pp. 1–4.
[6] I. Dzafic and H. Neisius, "Real-time power flow algorithm for shared memory multiprocessors for European distribution network types," in *Proc. Conf. Proc. IPEC*, Oct. 2010, pp. 152–158.
[7] A. Ahmadi, S. Jin, M. C. Smith, E. R. Collins, and A. Goudarzi, "Parallel power flow based on OpenMP," in *Proc. North Amer. Power Symp.*, Sep. 2018, pp. 1–6.
[8] V. Roberge, M. Tarbouchi, and F. Okou, "Parallel power flow on graphics processing units for concurrent evaluation of many networks," *IEEE Trans. Smart Grid*, vol. 8, no. 4, pp. 1639–1648, Jul. 2017.
[9] F. Milano, "An open source power system analysis toolbox," *IEEE Trans. Power Syst.*, vol. 20, no. 3, pp. 1199–1206, Aug. 2005.
[10] H. Cui, F. Li, and K. Tomsovic, "Hybrid symbolic-numeric framework for power system modeling and analysis," *IEEE Trans. Power Syst.*, vol. 36, no. 2, pp. 1373–1384, Mar. 2021.
[11] A. B. Birchfield, T. Xu, K. M. Gegner, K. S. Shetye, and T. J. Overbye, "Grid structural characteristics as validation criteria for synthetic networks," *IEEE Trans. Power Syst.*, vol. 32, no. 4, pp. 3258–3265, Jul. 2017.
[12] A. Silberschatz, P. B. Galvin, and G. Gagne, *Operating System Concepts*. Hoboken, NJ, USA: Wiley, 2012.
[13] F. Milano and R. Zárate-Miñano, "A systematic method to model power systems as stochastic differential algebraic equations," *IEEE Trans. Power Syst.*, vol. 28, no. 4, pp. 4537–4544, Nov. 2013.
[14] J. Bezanson, A. Edelman, S. Karpinski, and V. B. Shah, "Julia: A fresh approach to numerical computing," *SIAM Rev.*, vol. 59, no. 1, pp. 65–98, 2017.
[15] R. D. Zimmerman, C. E. Murillo-Sánchez, and R. J. Thomas, "Matpower: Steady-state operations, planning, and analysis tools for power systems research and education," *IEEE Trans. Power Syst.*, vol. 26, no. 1, pp. 12–19, Feb. 2011.