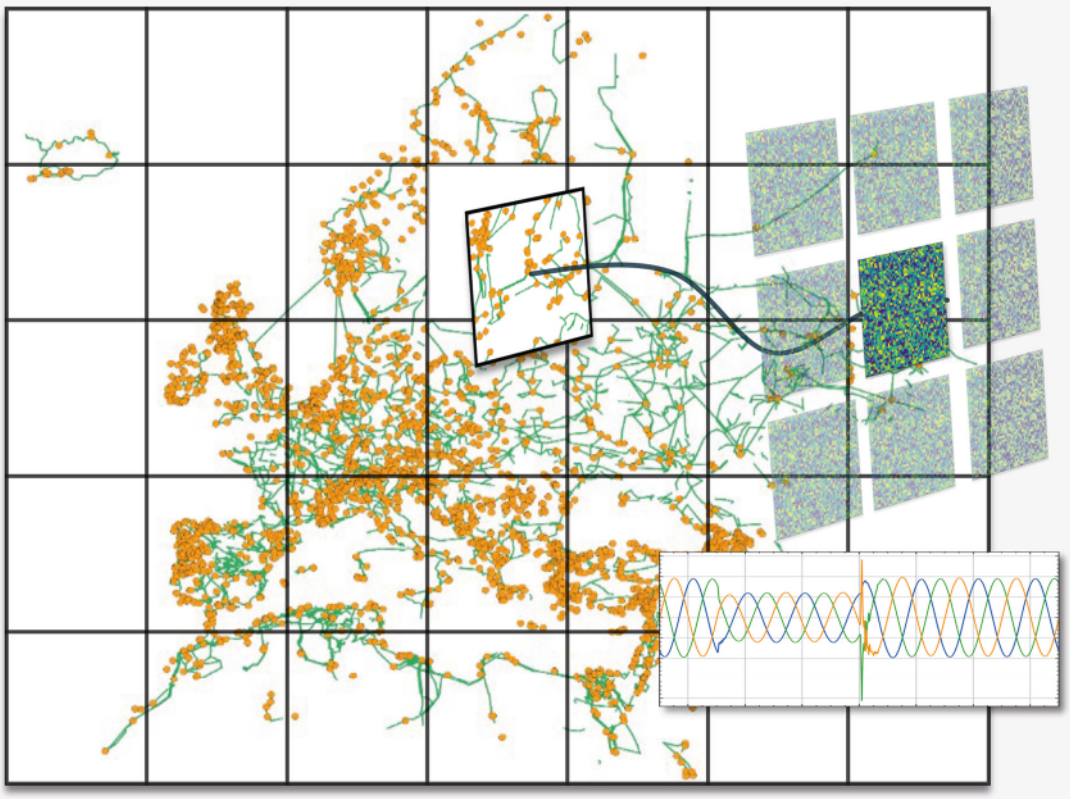


A Parallel-in-Space Simulator for Accelerating Power System Simulation on Graphics Processing Units

Junjie Zhang

Energie & Umwelt / Energy & Environment
Band / Volume 691
ISBN 978-3-95806-882-7

Mitglied der Helmholtz-Gemeinschaft

JÜLICH
Forschungszentrum

Schriften des Forschungszentrums Jülich
Reihe Energie & Umwelt / Energy & Environment Band / Volume 691

Forschungszentrum Jülich GmbH
Institute of Climate and Energy Systems (ICE)
Energiesystemtechnik (ICE-1)

A Parallel-in-Space Simulator for Accelerating Power System Simulation on Graphics Processing Units

Junjie Zhang

Schriften des Forschungszentrums Jülich
Reihe Energie & Umwelt / Energy & Environment Band / Volume 691

ISSN 1866-1793 ISBN 978-3-95806-882-7

Bibliografische Information der Deutschen Nationalbibliothek.
Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der
Deutschen Nationalbibliografie; detaillierte Bibliografische Daten
sind im Internet über http://dnb.d-nb.de abrufbar.

Herausgeber und Vertrieb:	Forschungszentrum Jülich GmbH Zentralbibliothek, Verlag 52425 Jülich Tel.: +49 2461 61-5368 Fax: +49 2461 61-6103 zb-publikation@fz-juelich.de www.fz-juelich.de/zb
Umschlaggestaltung:	Grafische Medien, Forschungszentrum Jülich GmbH
Druck:	Grafische Medien, Forschungszentrum Jülich GmbH
Copyright:	Forschungszentrum Jülich 2026

Schriften des Forschungszentrums Jülich
Reihe Energie & Umwelt / Energy & Environment, Band / Volume 691

ISSN 1866-1793
ISBN 978-3-95806-882-7

Vollständig frei verfügbar über das Publikationsportal des Forschungszentrums Jülich (JuSER)
unter www.fz-juelich.de/zb/openaccess.

ACKNOWLEDGEMENTS

I would like to firstly thank Prof. Andrea Benigni for his guidance and supervision across my journey as doctoral researcher, as well as Prof. Antonello Monti for his valuable feedback and serving as co-examiner in the doctoral examination.

I am grateful to Dr. Robert Speck for many valuable discussions and guidance together with Prof. Matthias Bolten in parallel-in-time algorithms and beyond. In addition to that, I would like to thank the Jülich Supercomputing Centre (JSC) for the access to HPC resources. Some of the benchmarks in this work would not have been possible without the supercomputers.

I greatly enjoyed my time at ICE-1 (formerly IEK-10) and the supportive working environment there. My thanks go to my group leader Dr. Thiemo Pesch for his support and guidance, and to my colleagues Yifei, Sina, Yiwen, Carsten, Marcel, Ariana, Philipp, Chuan and many others, for their company in the office and our numerous interesting discussions. I also appreciate the beautiful Forschungszentrum Jülich campus, surrounded by its natural forest.

And finally, I would like to thank my family for their continuous support.

ABSTRACT

The requirements on power system simulation techniques have been challenged due to growing size and complexity in the modern power systems. In order to address these challenges, we apply parallel-in-space algorithms and utilize GPU for acceleration, and the results show that our prototype achieves orders of magnitude speedups over execution on conventional multi-core processors. We observed speedups up to more than $80\times$ over an optimized sequential simulation and faster than real time execution capabilities.

On the algorithm level, the parallel-in-space approach proposed in this dissertation extracts computations into single-data-multiple-threads form so that it can be mapped and executed on GPUs efficiently. Furthermore, the dependencies among computing tasks are also exploited so that task-level parallelism can be achieved on top of data-parallel. The modeling of power system also took advantage of the shifted frequency analysis to balance between accuracy and computational burden. Moreover, the parallel-in-space algorithm is also combined with the parallel-in-time algorithm to exploit parallelism on temporal level to achieve higher speedup. We analyzed the algorithms for spatial and temporal algorithms with respect to computing efficiency, convergence to achieve better throughput.

On the implementation level, the prototype simulator was developed based on data-oriented design instead of traditional object-oriented design, so that the hardware accelerator as well as the space-time parallel algorithm can be exploited more efficiently. Moreover, we utilized the similarity of heterogeneous computing frameworks and implemented a flexibility layer which abstracts different frameworks based on the host-device model, such that the simulator can utilize GPUs under different computing frameworks as well as other hardware accelerators like FPGA.

The computing performance on GPU for component computation is tuned automatically using our approach based on automatic optimization empirical optimization of software. The approach can generate high performance kernels for numerical integration routines on GPU. Benchmark shows that these optimized integration routines outperform routines implemented using standard GPU-based linear algebra libraries with speedup between $1.3\times$ to $6.7\times$.

KURZFASSUNG

Die Anforderungen an Simulationstechniken für Stromversorgungssysteme sind aufgrund der zunehmenden Größe und Komplexität moderner Stromversorgungssysteme gestiegen. Um diesen Herausforderungen gerecht zu werden, wenden wir „Raum-parallele“-Methoden an und nutzen GPUs zur Beschleunigung. Die Ergebnisse zeigen, dass unser Prototyp gegenüber herkömmlichen Multi-Core-Prozessoren eine um mehrere Größenordnungen höhere Geschwindigkeit erreicht. Wir konnten Geschwindigkeitssteigerungen von mehr als dem 80-fachen gegenüber einer optimierten sequenziellen Simulation beobachten sowie eine Ausführung schneller als in Echtzeit.

Auf Algorithmenebene extrahiert der in dieser Dissertation vorgeschlagene „Raum-parallele“-Ansatz Berechnungen in eine Single-Data-Multiple-Threads-Form, sodass sie effizient auf GPUs ausgeführt werden können. Darüber hinaus werden die Abhängigkeiten zwischen den Rechenaufgaben genutzt, um zusätzlich zur Datenparallelität auch Parallelität auf Aufgabenebene zu erreichen. Bei der Modellierung wurde die verschobene Frequenzanalyse genutzt, um Genauigkeit und Rechenaufwand auszugleichen. Zudem wird der „Raum-parallele"-Algorithmus mit dem „Zeit-parallelen“-Algorithmus kombiniert, um höhere Geschwindigkeiten zu erzielen. Wir haben die Algorithmen hinsichtlich Recheneffizienz und Konvergenz analysiert, um besseren Durchsatz zu erreichen.

Auf Implementierungsebene wurde der Prototyp des Simulators auf Basis eines datenorientierten Designs entwickelt, anstelle eines traditionellen objektorientierten Designs, damit der Hardwarebeschleuniger sowie der „Raum-Zeit-parallele“ Berechnung effizienter genutzt werden können. Zudem haben wir eine Flexibilitätsschicht implementiert, die verschiedene Frameworks auf Basis des „Host-device“-Modells abstrahiert, sodass der Simulator GPUs unter verschiedenen Rechenframeworks sowie andere Hardwarebeschleuniger wie FPGA nutzen kann.

Die Rechenleistung auf der GPU wird mit unserem Ansatz auf Basis automatischer empirischer Optimierung abgestimmt. Der Ansatz generiert hochleistungsfähige Kernel für numerische Integrationsroutinen auf der GPU. Benchmarks zeigen, dass diese optimierten Integrationsroutinen Routinen auf Basis standardmäßiger GPU-basierter Linearer-Algebra-Bibliotheken übertreffen, mit Beschleunigungen zwischen dem 1,3-fachen und dem 6,7-fachen.

NOMENCLATURE

HPC	high performance computing
HIL	hardware-in-the-loop
SFA	shifted-frequency analysis
DAE	system of differential-algebraic equations
ODE	ordinary differential equation
MGRIT	Multi-Grid reduction in time
PinT	parallel-in-time
PinS	parallel-in-space
MNA	modified nodal analysis
FLOP	floating point operation
RK4	Fourth order Runge Kutta
EB	Euler Backword
AEOS	automatic empirical optimization of software
ATLAS	Automatically Tuned Linear Algebra Software
BLAS	basic linear algebra subprograms
API	Application Programming Interface
CSR	compressed sparse row
ELL	ELLPACK format
CSR	Compressed sparse row
COO	Coordinate list
GPU	graphics processing unit
SIMT	single instruction, multiple threads

SIMD	single instruction, multiple data
DG	distributed generator
AC	alternative current
PLL	phase-locked loop
FTRT	faster-than-real-time
RT	real-time
WR	waveform relaxation
TLM	transmission line modeling
SSNA	state-space nodal analysis
LB-LMC	latency-based linear multi-step compound
JIT	just-in-time
CPU	central processing unit
PFASST	parallel full-approximation scheme in space and time
OO	object-oriented
DO	data-oriented
TSA	transient stability analysis
AE	algebraic equation
SDC	spectral deferred correction
MATE	multi-area thevenin equivalent
MPI	message-passing interface
PinTS	parallel-in-time-and-space
FPGA	field programmable gate arrays
UVLS	under voltage load shedding
UFLS	under frequency load shedding
EMT	electro-magnetic transient

TABLE OF CONTENTS

LIST OF ILLUSTRATIONS

LIST OF TABLES

PUBLICATIONS

Journal Publications

- Carsten Hartmann, Junjie Zhang, Carlos D. Gonzalez Calaza, Thiemo Pesch, Kristel Michielsen, and Andrea Benigni. "Quantum Annealing Based Power Grid Partitioning for Parallel Simulation". In: *IEEE Transactions on Power Systems* (2025), pp. 1–13. ISSN: 1558-0679. DOI: 10.1109/TPWRS.2025.3578243. (Visited on 06/13/2025)

- Junjie Zhang, Marcel Mittenbühler, and Andrea Benigni. "Shifted Frequency Analysis Based, Faster-than-Real-Time Simulation of Power Systems on Graphics Processing Unit". In: *International Journal of Electrical Power & Energy Systems* 159 (Aug. 1, 2024), p. 110014. ISSN: 0142-0615. DOI: 10.1016/j.ijepes.2024.110014

- Marcel Mittenbühler, Junjie Zhang, and Andrea Benigni. "Automatically Optimized Component Model Computation for Power System Simulation on GPU". in: *Electric Power Systems Research* 235 (Oct. 1, 2024), p. 110740. ISSN: 0378-7796. DOI: 10.1016/j.epsr.2024.110740

- Junjie Zhang, Lukas Razik, Sigurd Hofsmo Jakobsen, Salvatore D'Arco, and Andrea Benigni. "An Open-Source Many-Scenario Approach for Power System Dynamic Simulation on HPC Clusters". In: *Electronics* 10.11 (11 Jan. 2021), p. 1330. ISSN: 2079-9292. DOI: 10.3390/electronics10111330. https://www.mdpi.com/2079-9292/10/11/1330 (visited on 09/06/2023)

Conference Publications

- Junjie Zhang, Marcel Mittenbuehler, Lukas Razik, and Andrea Benigni. "Parallel Simulation of Power Systems with High Penetration of Distributed Generation Using GPUs and OpenCL". in: *2022 IEEE 13th International Symposium on Power Electronics for Distributed Generation Systems (PEDG)*. June 2022, pp. 1–6. (Visited on 09/29/2023)

- Han Zhang, Yifei Lu, Junjie Zhang, and Andrea Benigni. "Real-Time Simulation of an Electrolyzer with a Diode Rectifier and a Three-Phase Interleaved Buck Converter". In: *2022 IEEE 13th International Symposium on Power Electronics for Distributed Generation Systems (PEDG)*. June 2022, pp. 1–6

- Sigurd Hofsmo Jakobsen, Junjie Zhang, Tor Inge Reigstad, Lukas Razik, Salvatore D'Arco, and Andrea Benigni. "Modelica-Based Parallel Computing Framework for Power System Adaptive Special Protection Schemes". In: *2022 Open Source Modelling and Simulation of Energy Systems (OSMSES)*. Apr. 2022, pp. 1–6. DOI: 10.1109/OSMSES54027.2022.9769162

Chapter 1

INTRODUCTION

1.1 Motivation

Following the settlement of the Paris Agreement with 2050-carbon-neutral goal [15], over 100 countries have committed to achieving net-zero emissions by the middle of the 21st century. In order to achieve this goal, massive integration of renewable resources and upgrade of existing power system infrastructures have been catalyzed around the globe. Since these sources and new devices are mostly using power electronic devices as interfaces, this transition has inevitably brought more complexity to electric power systems and changed its dynamic behavior.

Before deploying new devices to the field, testing is required as the power systems are designed to be highly reliable. This need has promoted the development of *Digital Twins* as new means to test devices before their deployments [MMB20]. A digital twin is defined as a digital model that replicates a real entity with data that can be seamlessly transmitted between the real entity and its digital replica. Digital twin can thus facilitate more rapid and rigorous device testing under various conditions. Moreover, the measurement data of device prototypes can be used to improve the device itself as well as the digital twin model, thus creating a feedback loop to develop more accurate digital twin models.

Several simulation techniques and implementations are investigated along with the developments of digital twin models [MMB20; She+22; AK23]. If the simulation can be executed in real-time, then it is possible to interface the digital twin model with the real world or real devices. This approach is referred to as hardware-in-the-loop (HIL) testing. To achieve that, simulators are required to have real-time simulation capability, which is a challenge both to the simulation algorithm and the computing hardware.

The change of dynamics in the system is also posing new challenges to the simulators. Dynamics in power systems can be divided into different categories based on the time characteristics, such as electromagnetic transients, and electromechanical transients [Mac+20], as shown in Fig. 1.1. Modern power systems have less inertia due to the increasing percentage of converter-interfaced generations. At the same time, the converter-interfaced generations are posing significantly different

dynamics than traditional synchronous-generator-based generations [GG23]. They all brought more importance to study the electromagnetic transients in the system, which gives more challenge to the simulators as they usually require smaller simulation time steps to get accurate results.

In summary, as a result of the transition of power systems into a renewable future. Traditional simulation techniques are facing challenges in both increasing problem size and complexity. The requirement on the computational performance is also increased in order to meet real-time or even faster-than-real-time needs. The main goal of this work is to utilize GPU with a massively parallel-in-space algorithm to accelerate the transient simulation of power systems. Moreover, we also look into the possibility to parallelize the simulation of system under temporal parallelization algorithms.

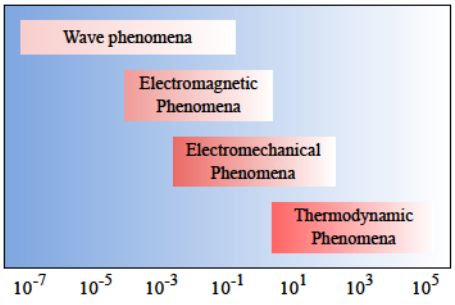


Figure 1.1: Power system dynamics with different time scales [Mac+20]

1.2 Related Work

The evolution of power system simulation methods has always followed the evolution of computer architecture. Before the massive roll-out of commodity parallel hardware, the major advancements of the CPU are higher clock frequency, more complex instruction sets, etc. At the same time, simulation methods were focusing on treating the problem as a whole and applying efficient and reliable numerical methods to improve computational speed [Sto79]. Over the last two decades, with the fast development of parallel computing hardware, such as high-performance multi-core CPUs, faster interconnections in CPU clusters, as well as the rise of GPUs, more studies have been conducted to exploit the parallelization potential of power system simulation methods.

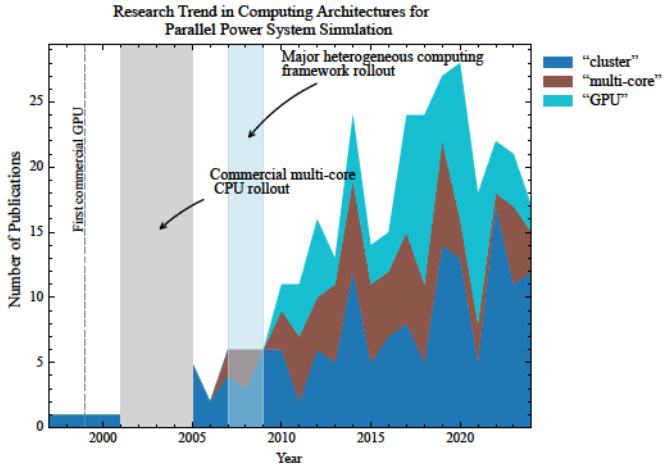


Figure 1.2: Relative research interest in power system simulation with different parallel hardware.

Since the 80s and beyond, there has been research into developing parallel algorithms to execute transient stability simulation on parallel computer [CB93; 92]. At the time, CPU clusters dominated parallel computing infrastructure, and early parallel algorithms focused on decomposing the simulation problem into several loosely dependent tasks, i. e. utilizing *task parallelism*. These approaches faced bottlenecks in scalability due to their high dependency on the latency of interconnects for synchronization and data exchange [CB93]. Moreover, the data-parallel potential is not yet well exploited due to limited parallel execution on a single CPU.

The early 2000s marked a turning point with the commercial rollout of multi-core CPU (from e. g. IBM, AMD, Intel)[1], as shown in Fig. 1.2[2] , which opens the research in optimizing parallel simulation performance on multi-core devices. The multi-core architecture enables more efficient data parallel execution on a single CPU. This has promoted the research into more fine-grained parallelization [XSZ05; JD09].

The GPU was first commercialized in the late 1990s to improve the performance of computer graphics, research on applying GPU to improve the performance of power system simulation was catalyzed by the release of heterogeneous computing frameworks such as OpenCL, CUDA etc. by the late 2000s. In this case, the GPU-

[1] Intel is a registered trademark of Intel Corporation; AMD is a registered trademark of Advanced Micro Devices, Inc.; IBM is a registered trademark of International Business Machines Corporation.

[2] Data taken from IEEE Xplore, considering only journal publications

related research lagged behind the hardware for almost a decade. On the one hand, this is due to the necessary algorithm re-design to match GPU's SIMT or even SIMD structure, on the other hand, the release of heterogeneous computing frameworks eventually facilitates the application of GPU as general-purpose computing device. Nevertheless, early practices still face challenges such as in minimizing host-device communications, etc.

Over the last decade, GPU is gaining more and more attention due to its capability for massively parallel execution and fast memory access [ZD17; JD10; Son+18; Wu+20; LD19]. Among them, [ZD17] uses fine-grained parallelization methods, which decompose the equations of the power system into linear and nonlinear parts and uses GPU to solve the decomposed linear system with direct method, and the non-linear system with the Newton-Raphson method, in a parallel way. This design can efficiently achieve data-parallelism within the linear and non-linear solver kernels. Similarly, in [Wu+20], a fine-grained method is applied to simulate power electronic systems where the integration of each state variable is computed in parallel. However, the algorithm, like many EMT and transient stability simulation approaches, requires updating the LU factorizations at each time step. This is avoided in the parallelization method [BM15] employed in this work so to improve scalability. The approaches shown in [JD10; LD19] design dedicated kernels for the different types of components considered, those approaches share a similar approach to the one presented in this paper. [JD10] is one of the earliest examples of using GPU for transient stability simulation and had highlighted the potential of using GPU to accelerate dynamic simulations. [LD19] utilizes CUDA's dynamic parallelism, thanks to which a kernel can launch new kernels during execution. With this feature, the whole power grid is implemented as a single kernel. During each simulation time step, it launches several component computation kernels as child kernels. Within the child kernels, there are multiple kernels for components with detailed models, e.g. kernels that simulate the identical circuit parts within the MMC model. Therefore, this approach can simulate systems containing very detailed component models efficiently. Overall, the approach proposed in [LD19] is very efficient, however, managing the whole simulation on the device makes applying the vendor's math library sometimes not possible as the device-side API is not always available [CUDAT18]. Launching kernel from the device side is not free, and only limited resources can be dedicated to the child kernels [Tan+17], therefore, it can generate significant overheads when the number of child kernels increases.

Related works regarding faster-than-real-time (FTRT) are mainly focused on using reconfigurable devices [CLD20; CLD23] as well as using high performance computing (HPC) platforms [Hua+18]. Reconfigurable devices usually have very limited memory and computing resources that restrict the application to the simulation of large scale grids. HPC platforms usually do not have these constraints, but the communication between parallel processes as well as the memory bandwidth becomes a bottleneck.

1.3 Contribution

This thesis focuses on applying parallel-in-space method to accelerate power system simulations with highlight in GPU execution. The main contributions of this thesis are summarized as:

- We propose an approach that exploits data- and task-parallelism in an automated way for component computations so to efficiently utilize the GPU and to increase the performance, as well as maintaining scalability when the type of component increases with future developments. Moreover, as we schedule kernel launch from the host side, vendor-optimized math libraries can be used.

  GPU-based approaches are intuitively data-parallel, but there were few studies regarding the application of task-parallel. In [Son+18], task-parallelism was considered to create data-parallel kernels so that load balancing among GPU cores is improved. In the end, a single compute kernel is created via automatic code generation technique. This implementation is improved to real-time execution, but it also limits the customization and extension of the kernel code, for example using a numerical library for part of the computations. A multi-stream approach is applied in [Wu+20] but is not completely task-parallel, because the concurrently executed two LU-solver kernels are the same with different input data, therefore, it is still possible to achieve similar performance via a pure data-parallel implementation. The task-parallelism exploited in this work is different to previous studies, as it affects the execution order and concurrency of different compute kernels.

- We propose a graph-based approach to overcome race condition issues while maintaining high scalability. Paralleled component computations can easily cause race conditions since each bus can have connections to multiple components. Previous studies often use programming tools such as mutex, atomic

operations to avoid this issue. However, these can potentially degrade the performance largely [KH13].

- We demonstrate the use of a flexibility layer that could translate the compute kernel as well as the host program to the target heterogeneous computing framework during compile time. The implementations of almost all previous GPU-oriented approaches had only focused on using CUDA as the programming framework, which can efficiently utilize the performance of Nvidia GPU but does not allow the use of hardware from other vendors. The compatibility of our implementation is extended without compromising on the performance. Moreover, vendor-tuned numerical libraries can be used when executing on GPUs from different vendors, leading to efficient utilization of the GPU hardware.

- The proposed parallel-in-space method is combined with parallel-in-time to achieve higher speed up. We implemented the combined space-time parallel simulation algorithm using a heterogeneous approach where the parallel-in-space (PinS) propagator is executed on GPU and PinT propagator on CPU, the result shows positive speedup when the two algorithms are combined. In order to increase PinT convergence, we proposed the hybrid model and buffered coarsening technique, both techniques have shown improved convergence. Moreover, an analytical model is also proposed to estimate the performance, which can help to determine the applicability of PinT to the candidate problem and estimate the needed resources to gain performance improvement.

- We propose a data-oriented design to facilitate simulation under space-time parallel algorithms, as well as execution on hardware accelerators like GPU. Compared to traditional object-oriented design, the data-oriented design can achieve better efficiency in memory utilization, and easier to be mapped into SIMT model, which is suitable for execution on GPU.

  Moreover, we also show that the C++-based simulator implementation can be interfaced to Python via `pybind`, so that network models as well as simulations can be constructed and executed directly in a Python script, giving better flexibility in real applications.

- The numerical integration performance on GPU can be automatically optimized using the approach we proposed in , the approach searches for optimal settings of hyperparameters such as matrix format, thread work group size etc.,

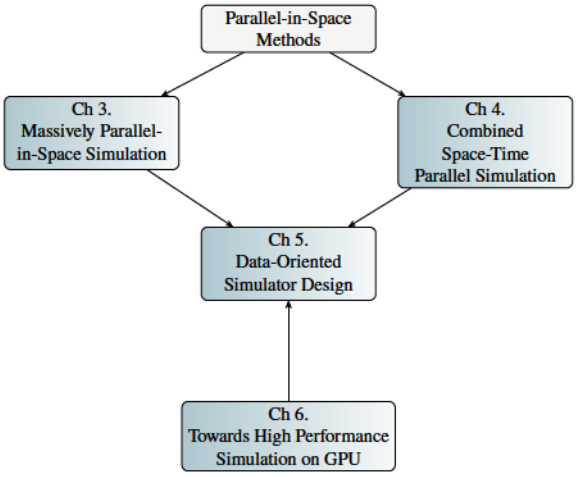


Figure 1.3: Dissertation Outline

as well as exploiting vectorizations. The optimized integration routines show better performance than implementations relying on standard GPU-based linear algebra libraries.

1.4 Outline

This dissertation is organized as illustrated in Fig. 1.3:

Chapter 2 introduces theoretical foundations of this work, including power system modeling and simulation, parallel paradigms of different simulation methods, as well as basics to parallel programming on GPU.

Chapter 3 describes the massively parallel simulation approach using GPU to accelerate the simulation, including extraction SIMT parallelism from computations, as well as the exploitation of data and task parallel.

The fourth chapter combines parallel-in-space method with parallel-in-time method to achieve higher speedup.

The fifth chapter introduces our data-oriented design, which allows efficient computation using space-time parallel method and is naturally suitable for computations on GPU.

Chapter 6 dives more deep into optimizing performance on GPU execution, we analyzed the computation and memory patterns and proposes an automatic opti-

mization technique that improves the numerical integration routines targeting on GPUs automatically.

Chapter 2

FUNDAMENTALS

In this chapter, the background related to this dissertation is introduced, including power system modeling and simulation techniques, parallelization methods, as well as parallel computing techniques with an emphasis on programming hardware accelerators like GPU.

We first describe the basic mathematical formulation of the power system simulation problem, followed by the introduction of the traditional electromagnetic transient simulation program. Then, the concept of shifted frequency analysis is described, as well as the power system component models formulated using the shifted frequency analysis method. Later, different parallel paradigms of power system simulation methods are introduced, which gives an overview on the characteristics of different methods. Finally, we give a short introduction on parallel computing techniques, including the abstraction of computing hardware, execution model, as well as parallel programming techniques.

2.1 Power System Simulation

Without delving deep into specific modeling methods or various models, in general, power systems can be formulated as a set of system of differential-algebraic equations (DAE):

$$\dot{x} = f(x, y), \tag{2.1}$$

$$0 = g(x, y). \tag{2.2}$$

The differential equations, on the one hand, are contributed by the dynamic components, such as inductors, capacitors, and so on. On the other hand, the algebraic equations mostly come from the topological constraints of the network. Numerical simulation of power systems aims to solve these equations as an initial value problem numerically over time. During this process, different integration methods, including explicit, implicit, or multi-step methods, can be applied to compute the approximation of the DAE at the required time point based on the given initial condition. The accuracy as well as the stability of the applied numerical methods are the key criteria to evaluate the computational performance of the simulation. Comprehensive and in-depth discussions and comparisons of these methods in general are beyond the

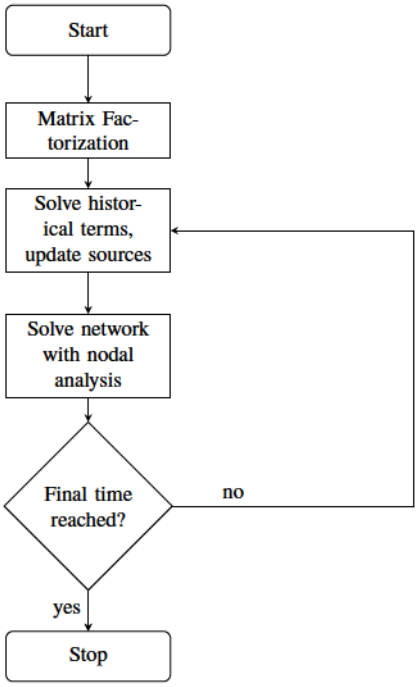


Figure 2.1: Execution flow-chart of a typical EMTP type simulator.

scope of this work. Research on these topics can be found in a variety of publications as well as books on numerical methods [CK06; Atk89].

One of the earliest simulator for electro-magnetic transient simulation is the *EMTP* simulator [Dom69], which was developed in the late 60s and has been one of the most impactful simulators in the power system community. Overall, the simulation flow of the EMTP-type simulator is divided into two steps, as shown in Fig. 2.1, which is developed based on the concept of nodal analysis. Dynamic components are represented using Thevenin or Norton equivalent and inserted to the network, whose equations are discretized to contain historical terms. During each simulation time step, dynamic components update their injections to the network via updating the reference values of the equivalent sources, afterwards, the network is solved using modified nodal analysis (MNA). After the network solution is updated, the simulation time is advanced and dynamic components that relies on the network solution are updated. This process repeats until the final time is reached.

2.2 Power System Modeling

Shifted Frequency Analysis

When contingencies take place in power systems, the frequency will normally still oscillates around its fundamental frequency. This can be interpreted as the electrical signals in the power system have a base frequency which is modulated by slower events. Such phenomenon is analogously to the communications systems where a base signal is modulated by higher frequency components to convey information. In steady-state analysis of alternative current (AC) circuits, electrical signals such as power, voltage, currents are commonly represented by phasor signals, which uses a complex value to represent the magnitude and phase angle of the sinusoidal electrical signal:

$$x(t) = Xe^{j\omega_s t} \tag{2.3}$$

The concept of shifted frequency analysis is similar to the phasor representation, whose goal is to represent the original signal using a signal with much lower frequency, as if the frequency of the original signal was shifted to the left of its frequency spectrum. As a result, a larger time step size can be applied to simulate the shifted signal numerically. This is due to the maximum sampling frequency given by the Nyquist theorem (or Nyquist–Shannon sampling theorem), as the frequency of a signal is decreased, the maximum sampling frequency is also decreased. Therefore, by representing the desired signal using its envelope, a larger time step is also allowed to be applied without violating the sampling theorem.

The procedure of performing SFA can be briefly described as following [Mar+14]. First, the analytical signal of the original signal is produced with the help of Hilbert transform:

$$x_a(t) = x(t) + j\mathcal{H}\left[x(t)\right], \tag{2.4}$$

where $\mathcal{H}[\cdot]$ denotes the Hilbert transform, defined formally as:

$$\mathcal{H}\left[x(t)\right] = \frac{1}{\pi}\int_{-\infty}^{+\infty}\frac{x(\tau)}{t-\tau}d\tau. \tag{2.5}$$

Suppose that a power system signal $x(t) = A(t)cos(\omega_s t + \theta)$ is given, then the analytical signal is obtained by

$$\begin{aligned}\mathcal{H}\left[A(t)cos(\omega_s t + \theta)\right] &= A(t)sin(\omega_s t + \theta),\\ x_a(t) = x(t) + j\mathcal{H}[x(t)] &= A(t)cos(\omega_s t + \theta) + jsin(\omega_s t + \theta)\\ &= A(t)e^{j\theta}e^{j\omega t}.\end{aligned}$$

Afterwards, the signal is shifted by the frequency $-f_s$, using its angular speed we get

$$\begin{aligned} x_{a-shifted}(t) &= (A(t)cos(\omega_s t + \theta) + jsin(\omega_s t + \theta)) \cdot e^{-j\omega_s t}, \\ &= A(t)e^{j\theta}e^{j\omega t} \cdot e^{-j\omega_s t} \\ &= A(t)e^{j\theta}. \end{aligned}$$

Therefore, the original power system signal is shifted to frequency equal to zero. The model of network components e. g. capacitors, inductors can be found in [ZMD10].

Moreover, if the original signal has a spread of different frequency components, the SFA can also be extended by using a set of Fourier coefficients as explained in [YBA13]. The application as well as accuracy analysis regarding SFA and Hilbert transform based method are discussed in [ZMD10; Mar+14; YBA13; DFP20; Pao+20].

Resistive Companion Method

A technique applied frequently in power system modeling or circuit analysis is the resistive companion method. The method uses an equivalent circuit model to represent the numerical solving process for linear dynamic components, here the term dynamic components are referred to the components whose behaviors are described involving differential equations. Take the inductor model as an example, its current is described by

$$\dot{i} = L^{-1}u. \tag{2.6}$$

Numerical integration is needed to calculate the current over it

$$i = L^{-1}\int udt. \tag{2.7}$$

If the Euler backward method is applied to perform discretization and integration, the inductor current is calculated by

$$i_{k+1} = i_k + h \cdot L^{-1}u_{k+1}.$$

Note that since the inductance is a constant, its product with voltage can be seen as calculating the current over an equivalent resistor. Therefore, the dynamic inductor can be replaced by an equivalent current source and a resistor connected in parallel, where the reference current of the source i_{equiv} is updated by the current over the inductor branch from the last time step $i_{\langle k \rangle}$, and the conductance of the equivalent resistor G_{eq} equals L^{-1}:

$$i_{k+1} = i_{eq} + h \cdot G_e u_{k+1}. \tag{2.8}$$

Moreover, the resistive companion method can be applied in combination with other modeling techniques for instance the SFA. When applying SFA to the inductor model, the differential equation Eq. (2.6) becomes:

$$\dot{\mathbf{i}} + j\omega\mathbf{i} = L^{-1}\mathbf{u} \tag{2.9}$$

again applying a numerical method to discretized it, in case of Euler backward:

$$\mathbf{i}_{k+1} = \mathbf{i}_k + h \cdot (L^{-1}\mathbf{u}_{k+1} - j\omega\mathbf{i}_{k+1}). \tag{2.10}$$

Reformulating it so that the right-hand side remains the same form as Eq. (2.8):

$$\mathbf{i}_{k+1} = \frac{1 - j\omega h}{1 + \omega^2 h^2} \cdot \mathbf{i}_k + \frac{h(1 - j\omega h)}{L(1 + \omega^2 h^2)} \cdot \mathbf{u}_{k+1} \tag{2.11}$$

with the new equivalent source and conductance computed by

$$i_{eq} = \frac{1 - j\omega h}{1 + \omega^2 h^2} \cdot \mathbf{i}_k, \quad G_{eq} = \frac{h(1 - j\omega h)}{L(1 + \omega^2 h^2)}. \tag{2.12}$$

Similarly, for capacitors, assuming SFA is applied

$$\mathbf{i} = C\dot{\mathbf{u}} + j\omega\mathbf{u}, \tag{2.13}$$

and discretized with Euler backward method

$$\mathbf{i}_{k+1} = \left(\frac{C}{h} - j\omega\right)\mathbf{u}_{k+1} - \frac{C}{h}\mathbf{u}_k. \tag{2.14}$$

Note that here the equivalent current source is updated by the previous component voltage, this is because the state variable in this case is voltage instead of current.

In general, the resistive companion method can be applied to all linear dynamic components. Let the component be modeled in state space form

$$\dot{x} = Ax + Bu \tag{2.15}$$

and discretized using Euler backward

$$x_{k+1} = x_k + h \cdot (Ax_{k+1} + Bu_{k+1}). \tag{2.16}$$

Reformulating it so that the right-hand side again complies with Eq. (2.8)

$$x_{k+1} = \frac{1}{1 - hA} \cdot x_k + \frac{hB}{1 - hA} \cdot u_{k+1}, \tag{2.17}$$

with the equivalent current source and conductance being calculated by

$$i_{eq} = \frac{1}{1 - hA} \cdot x_k, \quad G_{eq} = \frac{hB}{1 - hA}. \tag{2.18}$$

Reference Frame Theory

Power systems using alternate current are typically configured as three-phase symmetric systems. Therefore, the electric signals, for instance the variating abc phase quantities are naturally represented in a static reference frame, referred to as abc frame. In this case, these signals represent the original physical signals. It is however computationally challenging as these signals usually have high frequencies.

One efficient method to reduce the computational requirement is to use *Park-transform* to project the signals into a rotating $dq0$ reference frame that rotates at the same angular speed as the fundamental frequency of the system. Therefore, the steady-state signals, which variates at fundamental frequency, become constant in this rotating frame. The transform is given by

$$\begin{bmatrix} x_d \\ x_q \\ x_0 \end{bmatrix} = \frac{2}{3} \begin{bmatrix} \cos\theta & \cos\left(\theta - \frac{2\pi}{3}\right) & \cos\left(\theta + \frac{2\pi}{3}\right) \\ -\sin\theta & -\sin\left(\theta - \frac{2\pi}{3}\right) & -\sin\left(\theta + \frac{2\pi}{3}\right) \\ \frac{1}{2} & \frac{1}{2} & \frac{1}{2} \end{bmatrix} \begin{bmatrix} x_a \\ x_b \\ x_c \end{bmatrix}, \tag{2.19}$$

where $\theta = \omega t + \theta_0$ is the instantaneous angle of the rotating frame, and $\omega = \mathrm{d}\theta/\mathrm{d}t$ being its angular velocity. To recover abc quantities, The inverse transform is given by

$$\begin{bmatrix} x_a \\ x_b \\ x_c \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta & 1 \\ \cos\left(\theta - \frac{2\pi}{3}\right) & -\sin\left(\theta - \frac{2\pi}{3}\right) & 1 \\ \cos\left(\theta + \frac{2\pi}{3}\right) & -\sin\left(\theta + \frac{2\pi}{3}\right) & 1 \end{bmatrix} \begin{bmatrix} x_d \\ x_q \\ x_0 \end{bmatrix}. \tag{2.20}$$

In practice, when the three-phase quantities are symmetric, the zero-sequence component x_0 can be ignored because $x_a + x_b + x_c = 0$, the transformation is reduced to

$$\begin{bmatrix} x_d \\ x_q \end{bmatrix} = \frac{2}{3} \begin{bmatrix} \cos\theta & \cos\left(\theta - \frac{2\pi}{3}\right) & \cos\left(\theta + \frac{2\pi}{3}\right) \\ -\sin\theta & -\sin\left(\theta - \frac{2\pi}{3}\right) & -\sin\left(\theta + \frac{2\pi}{3}\right) \end{bmatrix} \begin{bmatrix} x_a \\ x_b \\ x_c \end{bmatrix}. \tag{2.21}$$

In the real model of power systems, multiple rotating reference frames could coexists. For instance, when the network is modeled with SFA, and component with the $dq0$ frame. Therefore, to interface different reference frames, an additional transformation between two reference frames based on the difference of their angular speed need to be applied

$$\begin{bmatrix} x_R \\ x_I \end{bmatrix} = \begin{bmatrix} \sin\delta & \cos\delta \\ -\cos\delta & \sin\delta \end{bmatrix} \begin{bmatrix} x_d \\ x_q \end{bmatrix}. \tag{2.22}$$

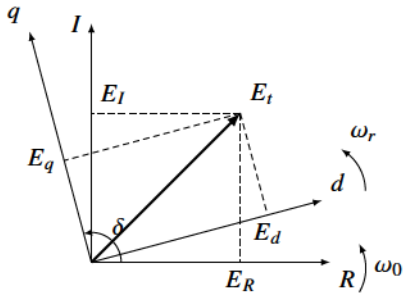


Figure 2.2: SFA frame to dq frame

with δ defined as the angle between the q axis to the real axis of SFA frame. As shown in Fig. 2.2. It needs to be noted that there are alternative definitions of δ in different literature, and the transformation matrices need to be adapted when the definition differs. In this work, we comply with the definitions in [KBL94].

Component Characteristics and Modeling

Synchronous Machine: Most of the power plants use synchronous machines as the generation unit, for instance hydropower plants, thermal power plants, and even nuclear power plants. Therefore, modeling of the synchronous machines is very important to studying power system dynamics of traditional systems. One of the major difficulties in modeling them is due to its rotating nature, which leads to constantly changing flux linkages. Therefore, most of the parameters are also changing according to its rotation. A common practice is to use Park-transformation to project the variables into a rotating frame which rotates with the synchronous frequency, therefore, the variating flux linkages becomes constant. The resulting electromagnetic circuit can be represented by equivalent circuits over $dq0-$axis, which is shown in Fig. 2.3, representing a classical model is introduced in [KBL94]. Formulating the dq-axis stator and rotor voltage equations in *per unit* notation, we

can get

$$\dot{\Psi}_d = e_d - \Psi_q \dot{\theta}_r - R_a i_d \tag{2.23}$$
$$\dot{\Psi}_q = e_q - \Psi_d \dot{\theta}_r - R_a i_q \tag{2.24}$$
$$\dot{\Psi}_0 = e_0 - R_a i_0 \tag{2.25}$$
$$\dot{\Psi}_{fd} = e_{fd} - R_{fd} i_{fd} \tag{2.26}$$
$$\dot{\Psi}_{1d} = -R_{1d} i_{1d} \tag{2.27}$$
$$\dot{\Psi}_{1q} = -R_{1q} i_{1q} \tag{2.28}$$
$$\dot{\Psi}_{2q} = -R_{2q} i_{2q} \tag{2.29}$$

where Eq. (2.23)-(2.25) are the stator voltage equations, and Eq. (2.26)-(2.29) the rotor voltage equations. The variable θ_r describes the angle between the axis of phase A and d-axis, and $\dot{\theta}_r$ the angular velocity ω_r of the rotor. For a 50 Hz network under steady state conditions, ω_r equals to the synchronous angular velocity, i. e. $\dot{\theta}_r = \omega_r = \omega_s = 2\pi \cdot 50$; the term $\Psi_q \dot{\theta}_r$ and $\Psi_d \dot{\theta}_r$ are referred to as *speed voltages* as they are induced by the flux change in space, and $\dot{\Psi}_d$, $\dot{\Psi}_q$ the *transformer voltages*, as they are induced by the flux change in time.

Flux linkages on stator and rotor flux can be described via the following algebraic equations, using again per unit system:

$$\Psi_d = -(L_{ad} + L_l) i_d + L_{ad} i_{fd} + L_{ad} i_{1d} \tag{2.30}$$
$$\Psi_q = -(L_{aq} + L_l) i_q + L_{aq} i_{1q} + L_{aq} i_{2q} \tag{2.31}$$
$$\Psi_0 = -L_0 i_0 \tag{2.32}$$
$$\Psi_{fd} = L_{ffd} i_{fd} + L_{f1d} i_{1d} - L_{ad} i_d \tag{2.33}$$
$$\Psi_{1d} = L_{f1d} i_{fd} + L_{11d} i_{1d} - L_{ad} i_d \tag{2.34}$$
$$\Psi_{1q} = L_{11q} i_{1q} + L_{aq} i_{2q} - L_{aq} i_q \tag{2.35}$$
$$\Psi_{2q} = L_{aq} i_{1q} + L_{22q} i_{2q} - L_{aq} i_q \tag{2.36}$$

And finally, the three-phase electric power output from the stator terminal is

$$P_t = e_a i_a + e_b i_b + e_c i_c = \frac{3}{2}(e_d i_d + e_q i_q), \tag{2.37}$$

assuming balanced condition where $e_0 i_0 = 0$. Using stator voltage equations Eq. (2.23)-Eq. (2.25), we can rewrite the equation of electric power in terms of flux linkage

$$P_t = \frac{3}{2}(\Psi_d i_q - \Psi_q i_d)\omega_r. \tag{2.38}$$

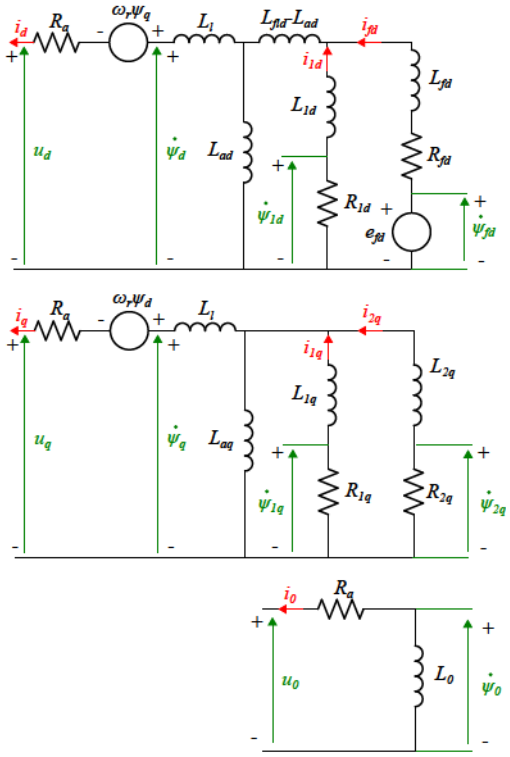


Figure 2.3: Equivalent circuit on *dq0*-axis of synchronous machine. Figure reproduced from [MZB24] under the CC BY 4.0 license.

The per-unit air-gap torque can be calculated based on the stator output power by

$$T_e = \frac{P_t}{\omega_{mech}} = \frac{3}{2}(\Psi_d i_q - \Psi_q i_d)\frac{\omega_r}{\omega_{mech}}. \tag{2.39}$$

The prime mover of the synchronous generator accelerates or decelerates by the unbalance of the applied torques. The equation of motion around the prime mover is

$$J\dot{\omega}_m = T_m - T_e, \tag{2.40}$$

where J is the combined moment of inertia of generator and turbine in $\mathrm{kg} \cdot \mathrm{m}^2$. A more common approach is to use per unit *inertia constant* H, defined as

$$H = \frac{1}{2}\frac{J\omega_{0m}^2}{S_{base}}, \tag{2.41}$$

where ω_{0m} is the rated angular velocity in mechanical rad/s, and S_{base} the rated apparent power. Since the Park-transform needs the mechanical rotor angle θ,

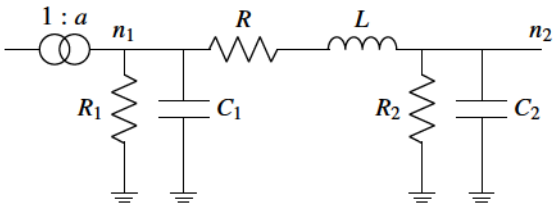


Figure 2.4: Pi-Model to represent transmission line with lumped parameters or transformers, the grounding branches can be removed depending on the need.

which is the second derivative of the mechanical ω. To stay an ordinary differential equation (ODE), the equation of motion for synchronous machines is usually written as

$$\Delta\dot{\delta} = \omega_0 \Delta\omega \tag{2.42}$$

$$\Delta\dot{\omega}_r = \frac{1}{2H}(T_m - T_e - k_D \Delta\omega_r), \tag{2.43}$$

where $k_D \Delta\omega_r$ represents the damping torque, which is not accounted for in the computation of T_e.

Branches: Transformer and Transmission Line with Pi-Model: The Pi-Model has been applied widely in modeling transformers as well as transmission lines with lumped parameters [KBL94]. As illustrated in Fig. 2.4, the Pi-model is consists of an RL-branch representing resistive and inductive losses, and two shunts at two terminals representing grounding admittance. The transformer turns ratio is defined as the ratio between the nominal voltage of the primary and secondary sides $n = n_p/n_s$. In the case of transmission lines, the turn ratio is simply set to $n = 1$, and the additional matrix stamping for turn ratio is ignored. The matrix stamping scheme for the pi-model branches is the same as the simulator DPsim [Mir+19].

2.3 Parallel Paradigms of Power System Simulation Methods

There are different parallel paradigms where different simulation methods can be categorized based on their characteristics. Firstly, based on the granularity of the decomposed problem, methods can be categorized into *fine-grained* or *coarse-grained* parallelization methods. However, there are no clear distinction as coarse-grained methods can always be formulated into a fine-grained problem [Ari15]. Secondly, based on the expression of parallelism, the methods can be divided into *explicit* or *automatic* parallelization methods. In general, an explicit method can be, for instance, expressed in the parallel constructs in the programming language

```
1    #pragma omp parallel
2    {
3        for(int i=0; i<size; i++){
4            data[i]+=1;
5        }
6    }
```

Listing 2.1: OpenMP parallel construct, which creates a team of OpenMP threads that execute the region.

of the mathematical model itself, for example, the parallel region using OpenMP as shown in List. 2.3.

Another type of explicit parallelization is achieved by structuring the problem into computational components with strongly typed communication interfaces [Aro06], an example is using transmission line modeling to distribute simulations in Modelica [Sjö+10]. Methods of this type are usually categorized as explicit coarse-grained methods.

As for automatic methods, in [Aro06], Aronsson et al. gave a detailed classification with the automatic fined-grained methods:

- Parallelism over the method: Under certain conditions, some ODE or DAE solvers can be parallelized, e. g., multiple steps can be computed in parallel. For instance, in [Spe18], when using collocation method to solve ODEs with spectral deferred correction (SDC) method, multiple collocation nodes can be updated simultaneously. For most of the solvers, at least for ODE solvers, there are only very limited parallelism can be exploited. By altering the solver algorithm, may change the convergence behavior, which may in terms decrease the simulation performance.

- Parallelism over time: In time domain simulations, the solution of the next step is usually always dependent on the previous time step. However, there still exists PinT algorithms, for instance *parareal* [Gur+16], which accelerates these simulations by solving dynamic systems at multiple time points simultaneously. The core idea of the PinT algorithm is to use a coarse step to solve the system to give an estimation, then update this coarse solution using fine steps performed in parallel from multiple coarse time points. Another parallel over time possibility is in discrete event simulations, where multiple events at

different time points can be evaluated in parallel as long as it obeys causality constraints.

- Parallelism of the system: The modeled system, i. e. the equations of the system, is parallelized. For ODE or DAE systems, this means the evaluation of the right-hand side is parallelized. Researches in this direction are focused mostly on the equation level parallelism, such as in Modelica [Aro06; Lun+09; Wal+14].

In addition to using fine-grained methods to achieve automatic parallelization, coarse-grained methods can also be realized. It is also possible to exploit parallelism among strongly connected computational components by, for instance in the simulator DPsim [Mir+19], task parallelism can be utilized to schedule execution of different computational tasks so that parallelism is exploited at a higher level. This form of parallelism is also elaborated in the following sections of this work.

Applications of explicit parallelism can be found in parallelization methods like transmission line modeling (TLM) [Sjö+10], diakoptics [Hap74], waveform relaxation (WR) [LRS82], etc.; and for automatic parallelism in fine-grained methods like [ZD17], in coarse-grained methods such as the state-space nodal analysis (SSNA) [DMB11] and LB-LMC [BM15]. A systematic comparison of different parallel methods can be found in [Geb19; Lun+09].

2.4 Parallel Programming and Architectures

Heterogeneous computing frameworks such as OpenCL, CUDA or HIP share a similar concept in their hardware abstractions, which contains three levels: *device*, *compute unit* (CU), and *processing element* (PE). Illustration of the three levels as well as the memory hierarchy are shown in Fig. 2.5. Such framework can be described as a *host-device* model, where the host prepares the computing tasks and schedules their execution on a single or multiple devices.

To map the hardware abstraction with the actual hardware, take Nvidia GPU as an example, where the Nvidia *streaming multiprocessors* (SMs) are the CUs, which is composed of *CUDA cores*, being their PEs; and the whole GPU is represented as a device [KH13].

During parallel execution, the finest processing granularity is a *work-item* (or *thread* in CUDA/HIP), which executes the code written in *kernel* in parallel on the device; a collection of work-items that are scheduled together and executed on the same

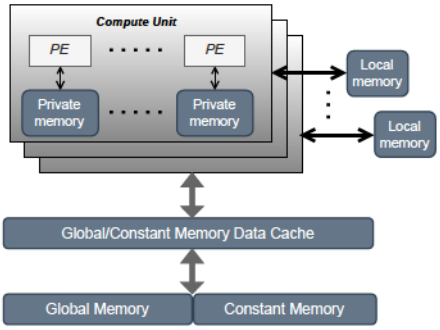


Figure 2.5: Hardware abstraction of heterogeneous computing frameworks like OpenCL, CUDA or HIP. Illustration using terminology of OpenCL. Figure reproduced based on [MZB24] under the CC BY 4.0 license.

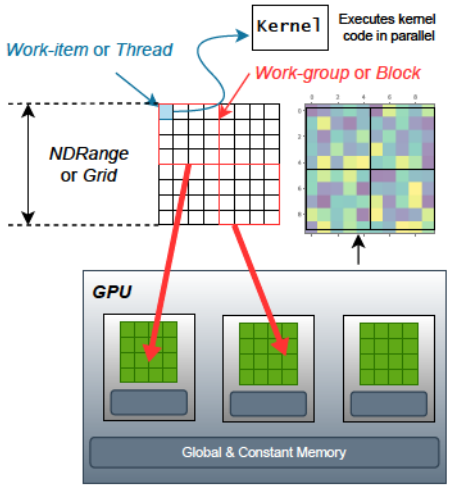


Figure 2.6: Illustration of the parallel execution model [KH13], based on terminology from OpenCL and CUDA/HIP. Figure reproduced from [MZB24] under the CC BY 4.0 license.

CU is a *work-group* (or *block* in CUDA/HIP), work-items in the same work-group can share data via the local memory if needed. Finally, a collection of work-groups forms *NDRange* (or *grid* in CUDA/HIP) which represents all work-items that is spawned during execution of a *kernel*. An illustration of the execution model is shown in Fig. 2.6 [MZB24; KH13].

Chapter 3

MASSIVELY PARALLEL-IN-SPACE SIMULATION

Early in the 60s, there has been practices of parallel-in-space methods [Bra77]. Over the past decades, along with the advancements in computing hardware, different high performance computing techniques as well as various simulation methods have been investigated to improve the simulation performance.

In this chapter, we introduce a massively parallel-in-space method for power system simulations that executes natively on GPUs across different platforms. We first introduce the parallelization method to decompose the original simulation problem, as well as the application of shifted-frequency analysis to alleviate computational burden while maintaining the needed accuracy. Afterwards, the single-instruction-multiple-threads mapping of the decomposed problem into parallel hardware is illustrated. Other than exploiting data parallel execution on GPU devices, the method also exploits task parallelism among various computing tasks, achieving higher level of parallelism. Moreover, in order to extend the compatibility over different heterogeneous frameworks, the code is not simply based on a certain framework but using an abstraction over frameworks, so that it can be executed with more flexibility and not restricted to a certain framework. Finally, benchmark results show that our approach achieves faster-than-real-time capability for systems with hundreds to thousands of nodes.

In the end, we also introduce a generalized formulation for the parallelization approach applied, which unifies the component-level decomposition and network partitioning. This formulation can be further employed in simulating larger network across multiple GPUs.

The work in this chapter has been partially presented in [ZMB24].

3.1 Parallelization method

In Chap. 2 we reviewed that power systems can be represented by a DAE, where the ODEs represent the dynamic components in the system: for example, in traditional transient stability analysis (TSA) programs, these differential equations come from the machine equations and the corresponding controllers; the algebraic equations (AEs) represent the topological constraints of the network. Such formulation

can also be extended to model modern power systems with suitable modeling of distributed generation components e. g. photovoltaic fields, wind turbines, etc.

The main idea of the LB-LMC method [BM15] employed in this work is to separate the component computations from the global network solution. Explicit integration is used typically for the non-linear components, and implicit integration is used for the linear network representation. Once component equations are discretized, components can be clustered in voltage or current type based on their characteristics at the terminal, i. e.

$$I^n(k+1) = \mathbf{f}(v(k), i(k), x(k), u(k), k), \tag{3.1}$$

$$V^n(k+1) = \mathbf{f}(v(k), i(k), x(k), u(k), k). \tag{3.2}$$

Since the decoupled components are represented by controlled source in the original network, the new network solution is obtained by applying the MNA method. Combining Eq. (3.1) - Eq. (3.2), we can represent this step as

$$YX(k+1) = b(v(k), i(k), I^n(k+1), V^n(k+1), k+1), \tag{3.3}$$

where Y is the admittance matrix, b is the source vector, and $X(k+1)$ is the new network solution vector.

By using explicit integrator for nonlinear components we achieve high parallelizability of the solution while by implicitly integrate the linear components of the network we improve the stability of the overall numerical solution. The accuracy concern about using explicit methods has already been clarified via stability analyses in [BM15].

The whole solution flow can be roughly divided into two steps, namely *component step*, where Eq. (3.1)- Eq. (3.2) are evaluated in parallel, and *network step*, where Eq. (3.3) is solved with the help of GPU-based basic linear algebra subprograms (BLAS) library. An additional step named *reduction* is introduced between these two steps to solve the race condition among threads and is discussed in Sect. 3.2.

3.2 Computational Approach

In order to execute and coordinate the execution on GPU devices, we implemented the program using the OpenCL framework [Gro23]. Like most of the other frameworks targeting heterogeneous platform, the programming model of the OpenCL framework abstracts the hardware into two types, namely the *host* and the *device*. The host is used to coordinate the execution of computing tasks as well as the data

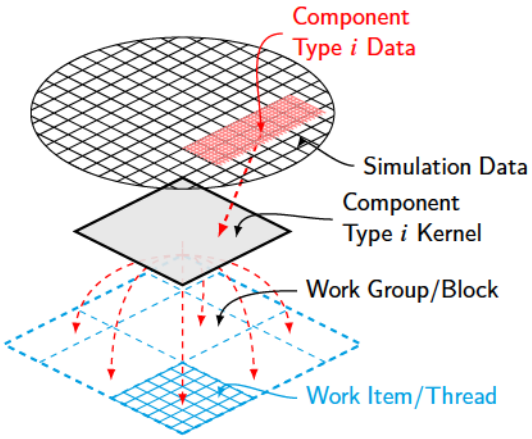


Figure 3.1: SIMT-based compute kernel design for different components. Figure reproduced from [ZMB24] under the CC BY 4.0 license.

movement among different memory objects, whereas the device is responsible for the actual computation. The compute kernels are the code that will be executed in parallel on the device. In this section, we will introduce the design of our kernels so to achieve data- and task-parallelism.

Data parallelism

A compute kernel is by its nature data-parallel as the same code is executed by multiple threads, and each thread is usually programmed to take different data by its thread-ID. In order to utilize the GPU efficiently, many previous works have managed to maintain the computations fully on GPU so that the communication overhead between GPU and CPU can be avoided. This is also the case with this work. Finally, since we have a *component step* and *network step* during each simulation step, our GPU kernels are designed for the two steps, respectively. In this work, the component kernels are designed following the SIMT model, i. e. a compute kernel is designed for each type of component, and each instance of such component is computed by a thread, as shown in Fig. 3.1. Therefore, the complete routine of one type of component—including the numerical integration as well as auxiliary transformations e. g. park transform—is performed inside the corresponding component kernel. As a result, different threads for the same compute kernel follows an identical execution path. Therefore, thread homogeneity is maintained for each type of component.

The network step is formulated as a set of algebraic equations; therefore, it can be solved via standard BLAS libraries or linear solver libraries, which automatically generate compute kernels and exploits data-parallelism during execution. These libraries have been extensively studied and developed for different platforms, hardware vendors also provide their own implementations, e. g. cuBLAS, cuSOLVER by Nvidia, and rocBLAS, rocSOLVER by AMD, etc.

Graph-based thread safety design

When component computations are parallelized, collecting all component outputs into the source vector b will likely cause *race conditions*. This is because most of the nodes are connected with multiple dynamic components, e. g. two generators, to the same bus or at any bus with more than a single line connected to it. Apart from modeling approaches such as aggregated component models, static network models, different programming techniques can be applied as well to solve this issue, e. g. mutex, atomic operations, etc. However, since these techniques basically serialize the concurrent operations, when the network structure becomes more complicated, or the number of multi-terminal components is large, performance may degrade severely [KH13]. To maintain scalability when simulating complex networks, we propose a graph-based approach to overcome this issue. Let's assume the power network is represented using a directed graph $G = (V, E)$, where all buses are mapped into vertices V and all components are mapped into edges E. Selecting all edges mapped from dynamic components $E^{'} \subseteq E$, and associated vertices $V^{'} \subseteq V$. Let D be the incidence matrix of $G' = (V', E')$, e be the vector of component outputs, then the new source vector b for MNA can be calculated using this incidence matrix D and the e vector as

$$b = De(V^n(k+1), I^n(k+1)). \tag{3.4}$$

Task parallelism

Operations like the launch of compute kernels or read-write actions to/from the device's memory are normally referred to as *tasks*. In addition to the data parallelism within the compute kernels, a high level of parallelism is exploited via the task graph which enables concurrent kernel execution. The task graph is constructed by analyzing the data dependency in each kernel's input data; kernels without data races can be labeled as independent tasks. For instance, when different kernels are taking the same data as input, no data race exists if there are only read-after-read operation.

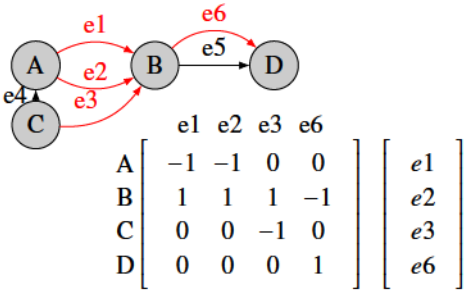


Figure 3.2: Simple connected directed graph and its incidence matrix. The dynamic edges are marked with red. Figure reproduced from [ZMB24] under the CC BY 4.0 license.

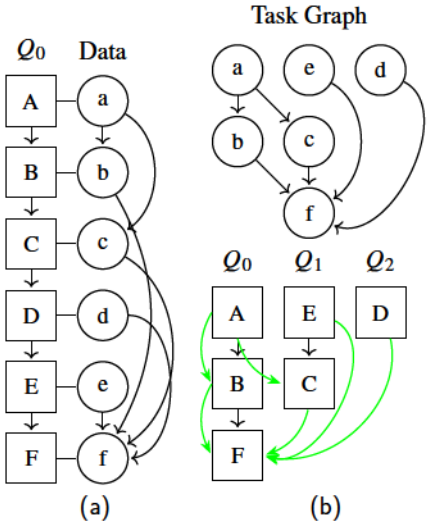


Figure 3.3: An example execution task graph during one time step. Q_i represents command queue or stream. Squares represent the task and round circles represent the associated data, (a) shows an in-order execution of tasks, data dependency is illustrated to the right but usually has no effect to the execution. (b) shows the task graph based on the data dependency, and the concurrent execution of the original tasks based on the task graph. Green edges indicate the synchronization among tasks. Figure reproduced from [ZMB24] under the CC BY 4.0 license.

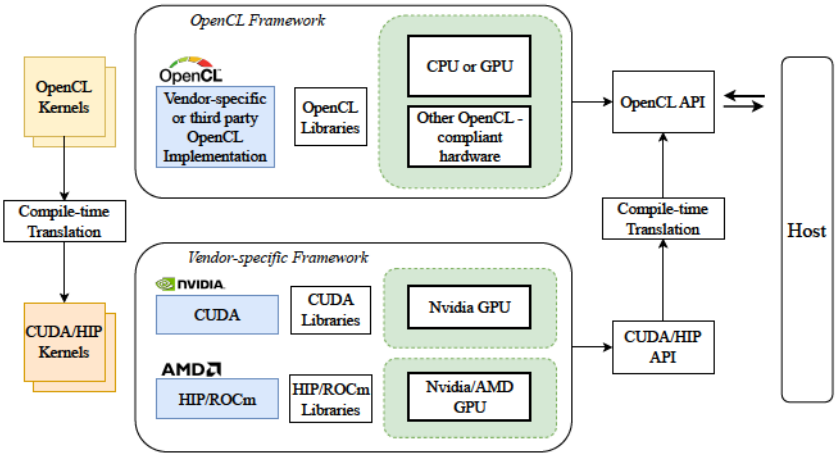


Figure 3.4: Implementation of the fully parallel program. Figure reproduced from [ZMB24] under the CC BY 4.0 license.

The implementation of a task-parallel runtime mostly relies on the corresponding OpenCL implementation. In OpenCL, the execution order of enqueued tasks is determined by the implementation itself, even the tasks (or commands) in an in-order command queue can be reordered during execution [Gro23]. In out-of-order command queues, a task graph can be dynamically constructed according to the explicitly indicated data dependencies, therefore, independent tasks can be executed concurrently. In contrast, CUDA requires the user to launch kernels on different streams or to manually construct a CUDA graph so that concurrent kernel execution can take place. To overcome this difference, we designed our kernel execution similar to the CUDA way. Since the tasks to be executed mostly remains the same during the simulation, we construct the task graph offline and launch independent tasks for different command queues or streams, and use events to synchronize between them. An example can be found in Fig. 3.3, which is constructed from the data dependency of the originally in-order executed kernels as shown in Fig. 3.3a.

3.3 Implementation

Single-core implementation for CPU: Optimized sequential implementation

To evaluate the performance of the proposed approach, we need the best possible sequential implementation of the same problem. Sequential simulation algorithms executed on the modern computer architecture might not actually be executed se-

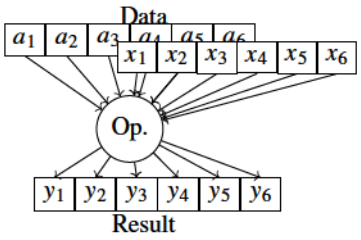


Figure 3.5: Illustration of SIMD execution on single CPU core.

quentially, especially for compiled languages such as C/C++. Automatic parallelization techniques of the compilers has been studied for many years [FO05] and are already available on most of the compilers, e. g. the automatic vectorization by GNU GCC [GNUGCC]. With these techniques, instructions executing on a single CPU core can follow a SIMD style as shown in Fig. 3.5, where a vectorized $y \leftarrow ax$ operation is executed. This instruction level parallelism can be applied to almost all of the modern CPUs since the SIMD instruction sets e. g. SSE, `AVX`, are supported by almost all of the modern CPUs, provided that the code implementation fulfils certain requirements, e. g. memory alignment, etc. Since numerical simulations are often dominated by linear algebra operations, the compiler optimization like auto-vectorization has a significant impact on the overall performance. Therefore, these features should be enabled in most cases for C/C++ based programs to achieve the best performance on CPU. To ensure a fair comparison, we use the linear algebra library Eigen [GJ+10] to implement the matrix operations and enabled the highest compiler optimization. As a result, the sequential implementation we will use as reference for performance evaluation already has a certain level of parallelization.

Fully Parallel Implementation

The fully parallel program adapts the LB-LMC method and the compute kernel design introduced in Sect. 3.1 and Sect. 3.2 to exploit data- and task-level parallelisms. The architecture of the fully parallel program is reported in Fig. 3.4. Using the OpenCL framework leads to high portability because of the wide range of OpenCL-conformant products [23]. Since the framework is only an open standard and a corresponding implementation needs to be provided, either a vendor-supplied OpenCL implementation or an open-source implementation [Jää+15] can be used. To increase the performance while keeping the portability, a translation layer is added so that the OpenCL Application Programming Interfaces (APIs) are internally trans-

lated to use the vendor-specific programming framework's API, and the compute kernels are translated into the target framework's kernel as well, as shown in the lower part of Fig. 3.4. This translation is implemented for Nvidia and AMD GPUs via our own extension of CLCudaAPI [CLCUDA] for the host program, and for the compute kernels, a directive-based tool is implemented. These translations take place during the compilation, therefore, they will not produce extra overhead during the execution. For instance, when the program is executed on an Nvidia GPU and CUDA is selected as the back-end, the program can automatically translate the original OpenCL API calls to CUDA and Nvidia's just-in-time (JIT) compiler NVRTC to compile the compute kernels. For AMD GPU, the program uses its HIP framework and selects the ROCm framework as the computing back-end.

Another advantage for translating the OpenCL API to vendor-specific framework's API is that the program is allowed to use vendor-optimized numerical libraries, e. g. cuBLAS on CUDA device and rocBLAS on ROCm device, providing more efficient utilization of the hardware.

3.4 Study Cases

System response during disturbance

Before evaluating the computational performance of the proposed approach and its implementation for FTRT execution, we first evaluate its accuracy using as reference the conventional EMT simulator DIgSILENT PowerFactory [PFACTORY], which is widely used for electrical power system analysis.

For this purpose we used the IEEE14 network (Fig. 3.6). An example provided by DIgSILENT. The machines are modeled with the standard model in PowerFactory. At the time $t = 0.1\,\text{s}$, Line_0001_0005 is opened, creating a disturbance into the system. To observe the system responses, the EMT simulation with PowerFactory is executed with $50\,\mu\text{s}$ time step for $20\,\text{s}$. We then compared the result obtained from this simulation with our SFA-based approach with the same time step, as shown in Fig. 3.7. Even if slight mismatch can be observed from the results, it is clear the developed tool based on SFA provided equivalent results to the EMT one. We imagine that the slight mismatch can be attributed to the use of different numerical integration schemes, as well as to the event handling methods.

Accuracy analysis

To compare the SFA simulation result with the EMT simulation result as well as to compare SFA simulation with different time steps, we use the original signals

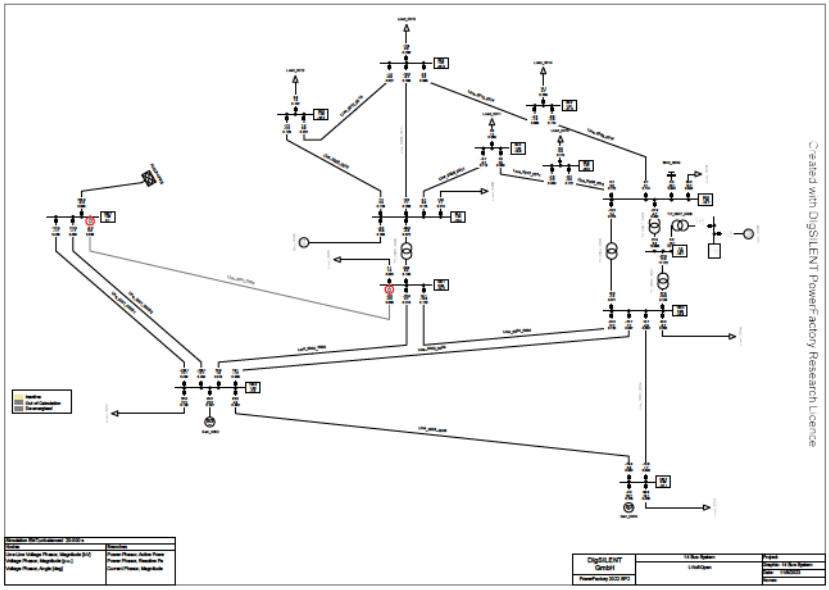


Figure 3.6: IEEE14 Network in DIgSILENT PowerFactory, with Line_0001_0005 open at $t = 0.1$ s. Figure reproduced from [ZMB24] under the CC BY 4.0 license.

reconstructed from the SFA signal. This is done by first interpolating the signals with a larger time step into the time step of the reference via linear interpolation. Afterwards, the original signal is reconstructed by shifting the SFA signal with its central angular speed ω_c

$$\begin{aligned} a(t) &= \mathbf{Re}\{\langle a \rangle (t) e^{j\omega_c t}\} \\ &= A(t)\cos(\omega_c t + \theta(t)), \end{aligned} \tag{3.5}$$

where $A(t)$, and $\theta(t)$ is the amplitude and phase angle of the SFA signal, respectively. Finally, the absolute relative error over time $e(t)$ is calculated via

$$e(t) = \left| \frac{a(t) - a(t)_{ref}}{a(t)_{ref}} \right| \tag{3.6}$$

where the reference $a(t)_{ref}$ is chosen as the reconstructed signal from SFA simulation with 50 µs time step. To evaluate the capability of our SFA-based approach with larger time steps, the SFA simulation is executed with 100 µs, 500 µs and 1 ms time

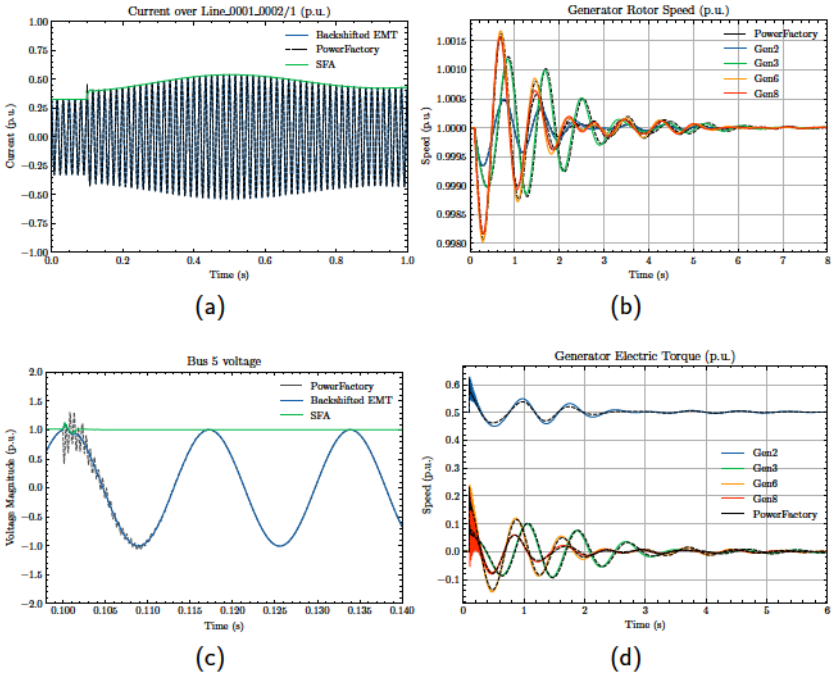


Figure 3.7: System responses after opening Line_0001_0005, (a) Current over Line_0001_0002/1, (b) Generator rotor speed, (c) Voltage at Bus 5, (d) Generator electric torque. Figure reproduced from [ZMB24] under the CC BY 4.0 license.

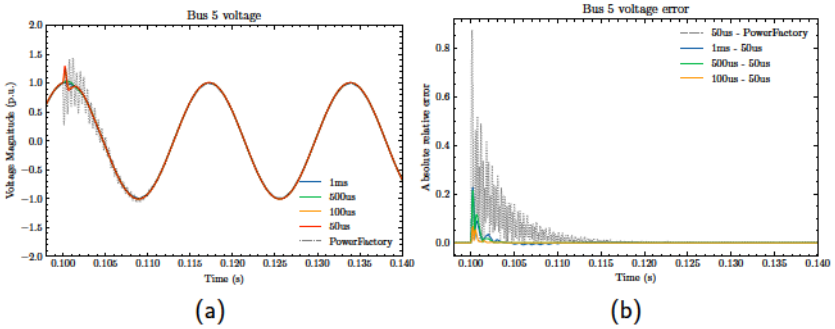


Figure 3.8: (a) Voltage magnitude of Bus5 with different simulation time step, (b) Evolution of absolute relative error during simulation, compared with SFA at 50us time step. Figure reproduced from [ZMB24] under the CC BY 4.0 license.

Table 3.1: Mean and Median Relative absolute error during first cycle after event with different time step. If without specification, listed cases are compared with 50us time step simulation with SFA

	100us	500us	1ms	50us SFA - EMT (PowerFactory)
Mean	2.46e-03	9.06e-03	1.26e-02	1.38e-01
Median	1.04e-04	3.11e-04	3.75e-03	8.84e-02

Table 3.2: Mean and Median Relative absolute error after 10 cycles after event with different time step. If without specification, listed cases are compared with 50us time step simulation with SFA

	100us	500us	1ms	50us SFA - EMT (PowerFactory)
Mean	1.72e-05	7.11e-05	1.50e-04	9.73e-04
Median	1.94e-05	7.87e-05	1.54e-04	1.08e-03

steps. Results are shown in Fig. 3.8, including the evolution of voltage magnitude as well as the absolute relative error. We also calculated the error of SFA with 50 µs compared to electro-magnetic transient (EMT) result in PowerFactory.

Tab. 3.1 and Tab. 3.2 lists the mean and median value of the absolute relative error $e(t)$ during the first cycle, i. e. $t = [0.1,\ 0.116]$ s, as well as after 10 cycles, i. e. $t = [0.26,\ 0.276]$ s. It can be noticed that the error is small throughout the period and grows slowly with increasing time steps.

3.5 Performance Analysis

To evaluate the performance of our simulator with different sizes of networks and especially large networks, we create synthesized networks using copies of the IEEE-118 test system [Ill] connected via transmission lines, as shown in Fig. 3.9, where the Bus2 of the current copy and the Bus117 of the next copy are connected. The benchmark is executed for 2× up to 64× system copies. As a single IEEE-118 system contains 52 generators, the 64× system contains 3328 generators, and in total 43456 dynamic components.

Simulations in this section are compiled and executed on a server with two AMD EPYC 7H12 CPUs (2.6 GHz base clock frequency, 64 cores each, hyper-threading disabled); 256 GB DDR4 3200 MHz main memory; one Nvidia A100-40 GB GPU, and one AMD MI100 GPU. The operating system installed is `Ubuntu 20.04.5 LTS`, kernel version `5.15.0-67-generic`; CUDA Toolkit version `V11.8.89`; AMD HIP version `5.1.20532-f592a741`, ROCm version `5.1.2.50102-55`; programs with AMD ROCm enabled requires clang compiler, and thus they are compiled

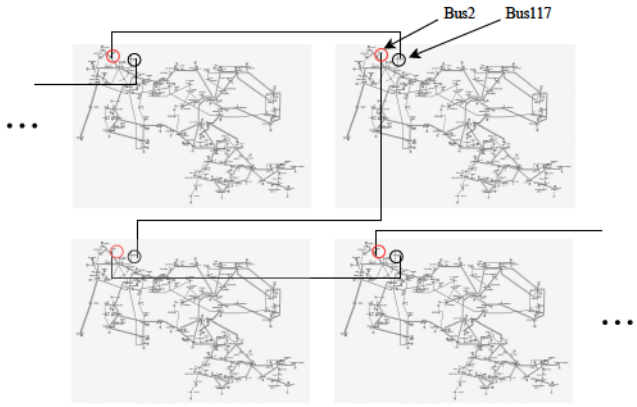


Figure 3.9: Synthesized large-scale network using connected copies of the IEEE-118 test system. Figure reproduced from [ZMB24] under the CC BY 4.0 license.

using clang with version `10.0.0-4ubuntu1`, the rest are compiled with GNU gcc version `9.4.0`.

The host program was implemented in C++ using the C++17 standard. The optimized sequential program uses the Eigen library [GJ+10] with version `3.3.7-2` to implement linear algebraic operations on the host side, which is also configured to use OpenBLAS [Wan+13] as its back-end with maximum threading set to one.

Optimized Sequential

The simulation algorithm that the sequential program uses is the same as the parallel one except that the reduction step is discarded, since it can directly write to the source vector b sequentially without data races. Therefore, the computational load for the sequential program is lighter than the parallel program. Results show that the optimized sequential implementation can already meet FTRT for 2× to 8× IEEE-118 systems, assuming a time step of 1ms, as illustrated later in Sect. 3.5 in Fig. 3.10a.

Effect of task-parallel execution on data-parallel program

The performance increase with task-parallelism could vary greatly to different systems and scenarios. Using the 2× connected IEEE-118 system copies test case, the task-parallel execution has contributed around 20% speedup on top of pure data-parallel execution, as shown in Tab. 3.3. We need to point out that our test case is nearly the worst-case scenario for the task-parallel execution. Because the tasks

Table 3.3: Speedup with task parallel execution

Computing framework	Average execution time per step (ms)		Speedup
	Sequential task execution	Parallel task execution	
HIP(ROCm)	0.576	0.493	1.17
CUDA	0.203	0.169	1.2

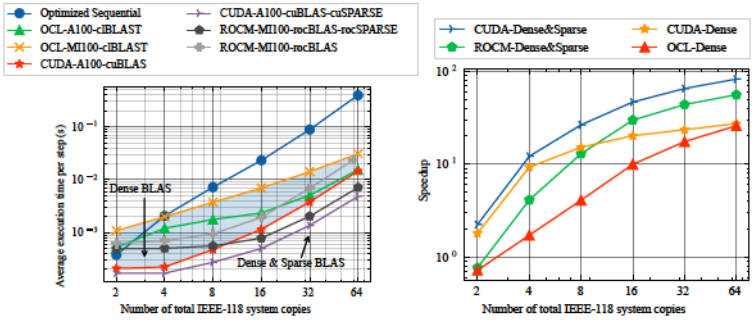


Figure 3.10: (a) Average execution time per simulation step, (b) Speedup of parallel simulation on GPU over optimized sequential simulation on CPU. Figure reproduced from [ZMB24] under the CC BY 4.0 license.

in our IEEE-118 system are associated with synchronous generators, inductors and capacitors, which share a large difference in their computational load.

Fully Parallel

The performance of the fully parallel program on GPU is benchmarked with different number of IEEE-118 system copies as well as on different hardware with different implementations. Three groups of benchmarks were performed on AMD and Nvidia GPU listed as following:

- OpenCL back-end: compute kernels are compiled using OpenCL, the reduction and network steps are executed using dense BLAS library clblast,
- CUDA/ROCm back-end: compute kernels are compiled using the vendor-specific framework, namely CUDA or ROCm, reduction and network step are also executed using the dense BLAS library of each framework, namely cuBLAS or rocBLAS,

Table 3.4: Minimum time step allowed for faster-than-real-time execution dt_{min}, k_{frt} at 1ms step, and overall speedup over sequential simulation with different systems sizes (as IEEE118 system copies)

System size (×118)	dt_{min}	k_{frt} at 1ms step	Speedup over sequential
2	0.17 ms	5.91	2.24
4	0.17 ms	5.86	12.12
8	0.27 ms	3.66	26.44
16	0.50 ms	2.00	46.53
32	1.37 ms	0.73	65.03
64	4.82 ms	0.21	82.16

- CUDA/ROCm back-end with dense & sparse BLAS: based on the previous benchmark, the reduction step is executed using a sparse BLAS library of each framework, namely cuSPARSE or rocSPARSE, instead of a dense one.

The benchmark results are shown in Fig. 3.10a. It can be noticed that the execution on GPU with vendor-optimized libraries has demonstrated the best performance among all implementations. The MI100 GPU has shown slightly lower performance than the A100 GPU, which can be attributed to the lower memory bandwidth of the former GPU. Nevertheless, albeit lower performance than vendor-libraries, simulations using the OpenCL framework and clBlast library still demonstrated FTRT capability up to around 32× IEEE-118 system. The speedup of different implementations over sequential execution is calculated by

$$k_{speedup} = \frac{t_{parallel}}{t_{sequential}}$$

and illustrated in Fig. 3.10b. The most performant implementation achieves over 82× speedup compared to the optimized sequential program. To see the FTRT capability clearly, we define the FTRT factor k_{ft} as

$$k_{ft} = \frac{h}{\bar{t}_{step}},$$

where h is the simulation time step, and $\bar{t}_{step}$ is the average execution time per step taken from the benchmark. By assuming a simulation time step of 1 ms, we can calculate the factor k_{ft} for the CUDA case with dense and sparse BLAS, this is shown together with minimum time step allowed for FTRT execution, and speedup over sequential execution as shown in Tab. 3.4.

3.6 GLB-LMC: An Generalized formulation for LB-LMC

In Sect. 3.1 we introduced the LB-LMC method. We now discuss a generalized representation of the LB-LMC method to fully exploit its potential. A similar idea has also been discussed in [MBM19] to allow the simulation being executed on multiple devices. In this section, we give a more systematic formulation of such idea and analyze its computational efficiency.

Recall that the multi-area thevenin equivalent (MATE) method [Arm+06] can be applied to parallelize simulation via partitioning the network. In that case, the MNA-based network step can be rearranged into a block diagonal form as

$$\begin{bmatrix} \begin{bmatrix} Y_{N1} & & \\ & \ddots & \\ & & Y_{Nk} \end{bmatrix} & \widetilde{C} \\ \widetilde{C}^\top & -Z_L \end{bmatrix} \begin{bmatrix} x_{N1} \\ \vdots \\ x_{Nk} \\ i_L \end{bmatrix} = \begin{bmatrix} h_{N1} \\ \vdots \\ h_{Nk} \\ -V_L \end{bmatrix}, \tag{3.7}$$

where h_{Nk}, Y_{Nk}, x_{Nk} for $k \in \{1, \ldots, k\}$ represents independent state variables, admittance matrix, nodal states, in each sub-network, respectively. $\widetilde{C}$ represents the connectivity matrix of the link between partitions (referred to as the *cut* in the following), and Z_L, i_L, V_L are the impedance matrix, current and voltage vectors over the cut, respectively.

Consider the simplified system of equations

$$\begin{bmatrix} Y & C \\ C^\top & -Z_L \end{bmatrix} \begin{bmatrix} X \\ I_L \end{bmatrix} = \begin{bmatrix} H \\ -V_L \end{bmatrix}. \quad \begin{matrix} ① \\ ② \end{matrix}$$

Adding $-C^\top Y^{-1} \cdot ①$ to ② and multiplying ① with Y^{-1} from the left yields

$$\begin{bmatrix} 1 & Y^{-1}C \\ 0 & -Z_L - C^\top Y^{-1} C \end{bmatrix} \begin{bmatrix} X \\ I_L \end{bmatrix} = \begin{bmatrix} Y^{-1}H \\ -V_L - C^\top Y^{-1} H \end{bmatrix}.$$

Hence, Eq. (3.7) can be reformulated as

$$\begin{bmatrix} I & \cdots & 0 & Y_{N1}^{-1} C_{N1} \\ \vdots & \ddots & \vdots & \vdots \\ 0 & \cdots & I & Y_{Nk}^{-1} C_{Nk} \\ 0 & \cdots & 0 & \sum_{i=1}^{k} C_{Ni}^\top Y_{Ni}^{-1} C_{Ni} + Z_L \end{bmatrix} \begin{bmatrix} x_{N1} \\ \vdots \\ x_{Nk} \\ i_L \end{bmatrix} = \begin{bmatrix} Y_{N1}^{-1} h_{N1} \\ \vdots \\ Y_{Nk}^{-1} h_{Nk} \\ \sum_{i=1}^{k} C_{Ni}^\top Y_{Ni}^{-1} h_{Ni} + V_L \end{bmatrix} \tag{3.8}$$

We can notice that the solution of each sub-networks depends on the cut, therefore, once the cut is solved, the two sub-networks can be solved independently. Now if we consider the solution of each subnetwork

$$v_{Ni} + \left(Y_{Ni}^{-1} C_{Ni}\right) i_L = Y_{N1}^{-1} h_{Ni} \tag{3.9}$$

as the new *component* equations, and the cut

$$\left(\sum_{i=1}^{k} C_{Ni}^{\top} Y_{Ni}^{-1} C_{Ni} + Z_L\right) i_L = \sum_{i=1}^{k} C_{Ni}^{\top} Y_{Ni}^{-1} h_{Ni} + V_L \tag{3.10}$$

as the new *network* step, we arrive at an extended form of the original LB-LMC. And the original formulation of LB-LMC is a special case where C_{Ni} is $\mathbf{0}$, i. e. a zero matrix.

The advantage of the new formulation is twofold: firstly, we can adjust the size of each component as well as the network without changing the model of individual components, so that we can balance the computational load between component and network step if needed; secondly, we could reuse code for existing models by considering them as a subnetwork, which facilitates the development of new components in simulator programs.

Computational Efficiency Analysis

We follow the discussion by analyzing the computational efficiency of this generalized form. Let the cost c_i for solving the equations of the i^{th} component, e.g. a generator, which depends on the numerical method used to integrate the model numerically. Let $\sigma_{i,j}$ be the number of floating point operations (FLOPs) for evaluating the equation governing the j^{th} state variable in the i^{th} component model. We have

$$c_i = k \sum_j \sigma_{i,j} + \Delta_i,$$

where k represents the number of times the derivatives or the Jacobian is evaluated. That is, k depends on the number of stages or iterations of the integration method. Here, Δ_i denotes the additional overhead such as vector addition, e.g. in $x \leftarrow x + dx$. For simplicity, we assume the use of Euler forward and ignore these additional overhead, which yields

$$c_i \approx \sum_j \sigma_{i,j}.$$

As for the network step, since it is a linear algebraic system, conventional LU-decomposition based triangular solves can be applied, in many cases, it is also practical to use the pre-computed inverse of the admittance matrix and apply matrix-vector multiplication. Nevertheless, most of these methods have $O(n^2)$ complexity [GV13]. Hence, we take simply the FLOPs for matrix-vector multiplication, i. e. when pre-computed admittance matrix is used, to estimate the cost. Therefore, the cost for network step is calculated by

$$c_{net} = (2n - 1)n. \tag{3.11}$$

Take a system which consists of m components, for simplicity, assume that we have homogeneity in the type of components, i. e. m identical components. The total computational cost per time step can be estimated by:

$$C_{\text{total}} = \underbrace{m(\sum_j \sigma_{i,j})}_{\substack{\text{Component} \\ \text{Cost}}} + \underbrace{(2n-1)n}_{\substack{\text{Global Network} \\ \text{Solve Cost}}}, \tag{3.12}$$

Now consider we move q network nodes to each component, i. e. each component therefore needs to solve q additional algebraic equations locally, while the global network solves the remaining $n - mq$ algebraic equations.

$$C_{\text{total}}(q) = m\left[\sum_j \sigma_{i,j} + q(2q-1)\right] + \left[2(n-mq)^2 - (n-mq)\right]. \tag{3.13}$$

What interest us is to see how we can reduce the cost by moving which quantity of network nodes into component computations, i. e. adjusting q so that we can find a range where $C_{total}(q) < C_{total}(0)$, here the term $C_{total}(0)$ denotes the original C_{total} in Eq. (3.12). Since the number of components as well as network equations are limited, the range of q is bounded by

$$0 < q < \min\left\{\frac{2n}{(m+1)}, \frac{n}{m}\right\}. \tag{3.14}$$

The optimal $C_{total}(q)$ can be calculated by analyzing Eq. (3.13), for which we could find the point q^* where Eq. (3.13) has its minimum

$$q^* = \arg\min_q C_{\text{total}}(q) = \frac{n}{m+1}. \tag{3.15}$$

It suggests that reducing the nodes in network step as much as possible, and partition the network corresponding to the number of dynamic components, is beneficial to the overall computation efficiency.

Interoperability

The generalized formulation offers more flexibility as it allows a component model, or even an entire grid from an external simulator, to be treated as a *component*. This capability enables seamless interoperability with third-party software, as well as to extend the application of the PinS approach to accelerate simulations. Here we give an example of modeling a small inverter-based system using the simulator ORTIS [MB21], as shown in Fig. 3.11b. The integration method applied for different components are marked with different color, green denotes the use of explicit method and blue the implicit method. In Fig. 3.11a, the modeled inverter-based system is treated as a component and interfaces to the network as a Norton equivalent.

With the inverter-based system model, we present a realistic study case using the German transmission grid model taken from the SciGrid project [Med+17], referred hereafter as the *SciGrid-DE* model. Since there are increasing interest in the inverter-based zero-inertia systems, whose dynamic behaviors differ significantly from that of conventional synchronous generators. To show this, all conventional generators replaced by the inverter-based system created using ORTIS solver, as shown in Fig. 3.12. For simplicity, the primary energy resources (e. g. solar, wind) are assumed to be infinite, i. e. modeled as constant DC source, and the controllers are neglected.

When studying the transient behaviors, typically a fault or load step event is included. Fig. 3.13b, Fig. 3.13c shows the transient voltage behavior at two arbitrarily picked buses when a fault is applied at another bus. Such simulations are needed for studies involving dynamic characteristics, such as the low voltage ride through capabilities, system stability, etc., under a zero-inertia grid scenario.

The Benchmark was performed on the AMD MI100 GPU and AMD EPYC 7H12 CPU. Performance results are presented in Tab. 3.5. In both cases, simulation was performed for 4000 steps, and the average time per step is calculated accordingly, and the simulation result in both cases is retrieved once per 1000 steps. We can notice that the PinS approach on GPU provides nearly 46 $\times$ speedup than the sequential case.

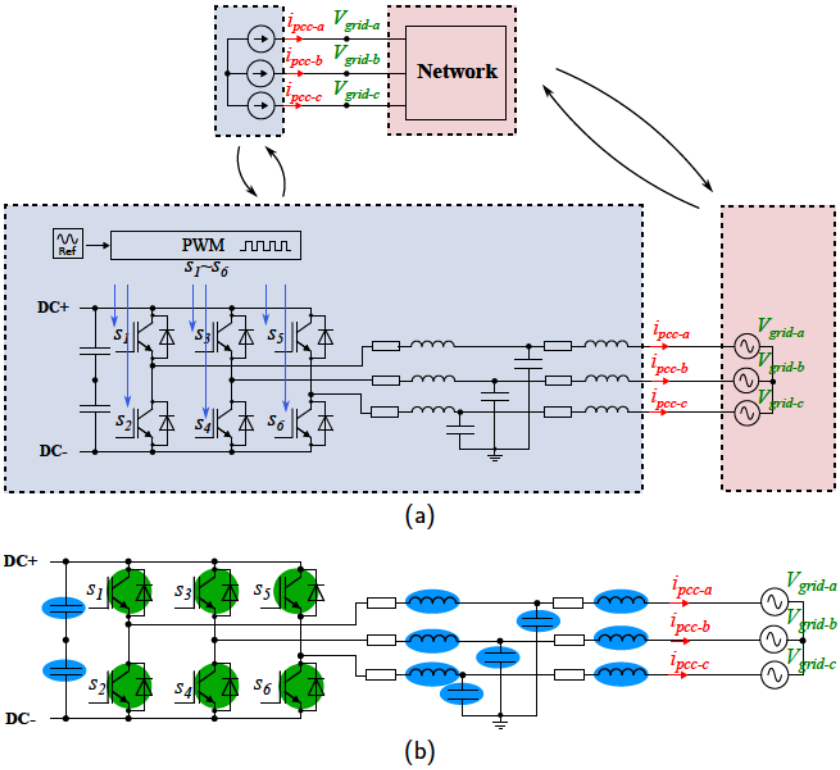


Figure 3.11: (a) The inverter subsystem as a component and its representation in the network (b) Decoupling of component computations within the inverter subsystem

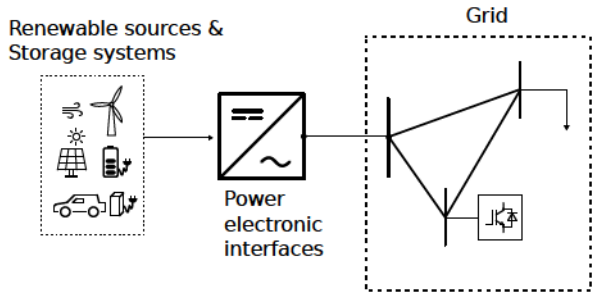


Figure 3.12: All the generators are replaced by inverter-based resources, the three-phase inverter model is taken from [Vyg+21].

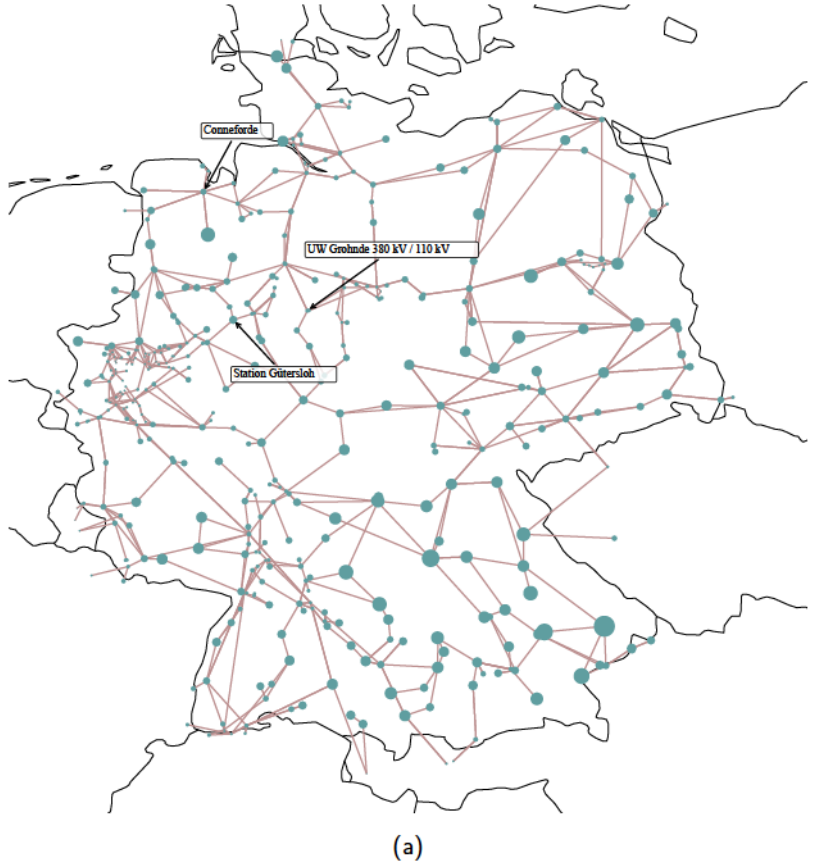


Figure 3.13: Transient voltage responses after a fault (cleared after 0.05 s) on bus "UW Grohnde" under a zero-inertia scenario. (a) grid topology with generation distribution; (b) voltage response at bus "Station Gütersloh", (c) voltage response at bus "Conneforde".

Table 3.5: Average execution time per step for the parallel and sequential version.

	Average execution time per step [ms]	*Speedup*
Parallel (GPU)	0.489	45.9x
Sequential (CPU)	22.449	-

3.7 Summary

In this chapter, we demonstrated our approach to accelerate power system simulation using GPU and achieves FTRT capability for small and large networks. The original simulation problem is parallelized using the LB-LMC method and separated into two steps, namely the *component step* and the *network step*. The component step is mapped into massively parallel execution following the SIMT programming model, whereas the network step is executed using a high performance linear algebra library. Our concept exhibits two levels of parallelism, namely data-parallel and task-parallel, where the former is brought by the SIMT-based design of compute kernels for components, and the latter by the concurrent task execution based on a task graph. Moreover, we also demonstrated the importance of HPC implementations on real-time (RT) or FTRT simulations: the optimized sequential simulation can also achieve FTRT for small networks with the help of SIMD execution; the fully parallel program gains further performance increase by exploiting sparse matrix structure in implementing linear algebraic operations. In the end, our implementation shows FTRT capability for electromagnetic transients with a dynamic phasor representation for networks with more than six thousand buses.

Limitations of our approach mainly exist in the fact that we achieve data-parallelism based on component types. When simulating a system where the components are highly divergent in types, the massively parallel thread capability will be limited and the overall performance relies on the task-parallel execution, i. e. concurrent execution of different compute kernels. Future work could be conducted to study the performance and improvements under such scenario, which also needs the implementation of more complex components. Moreover, a more detailed study of HPC implementations for GPUs is also important to performance improvements, e. g. a detailed benchmark of different BLAS libraries on GPU as well as different numerical methods for linear systems. Furthermore, the possibility of multi-GPU execution should be studied in the future to enable fast simulation for even larger networks.

Chapter 4

COMBINED SPACE-TIME PARALLEL SIMULATION

When the spatial parallelization potential has been exhausted, it becomes challenging to achieve further improvements in performance. Therefore, the application of parallel-in-time has gained increased interest in recent years, as it operates on the temporal dimension and therefore enables its integration with existing algorithms where spatial parallelization is already applied.

The earliest research into parallel-in-time methods for differential equations can be traced back to the 1960s [Nie64]. After that, a variety of methods have been developed, including the *parareal* method introduced in 2001 [LMT01], together with the parallel full-approximation scheme in space and time (PFASST) [EM12] method. Nevertheless, these two methods can be seen as special cases of the MGRIT method [EM12]. The continuous development of PinT methods has also sparked a variety of applications [Fal+14; Gur+16], and especially the recent applications in power system simulations [FLW19; Gun+20; SDB22; CLD22].

In this chapter, we introduce the combined space-time parallel simulation approach, which combines the proposed PinS algorithm with the parallel-in-time method MGRIT to exploit spatial and temporal parallelization together, aiming to achieve higher speedup than applying PinS individually.

4.1 Multi-Grid Reduction in time

Methodology

We start by briefly introduce the methodology of MGRIT following [SDB22] and [FLW19]. Consider again the power system simulation problem, it can be formulated in a general DAE formulation

$$\mathcal{F}(\dot{x}, x, u, t) = 0, \quad \text{for } t \in [t_0, t_f],$$

which we assume is continuous and differentiable in the interval $[t_0, t_f]$. Discretizing the function with a time step

$$\delta = \frac{t_0 - t_f}{h}$$

for some value of h, we then obtain the fine grid with h points

$$\Theta_\delta = \{t_i \in \mathbb{R} \mid t_i = t_0 + i\delta, \text{for } i = 0, \dots, h\}.$$

Let ϕ to be the PinS propagator that applies on x

$$x(t+\delta) \approx \phi(x(t), t) \quad i = 0, \dots, k-1$$

which computes the approximation to the solution at next time point $x_{i+1} \approx x(t_i + \delta)$. Therefore, by applying ϕ iteratively on the fine grid Θ_δ, we get a sequence of approximate solutions

$$\phi(x_i) = x_{i+1}, \text{for } i = 0, \dots, k,$$

Which is equivalent to performing numerical integration on $\mathcal{F}$ with the initial condition x_0. To illustrate this more clear, assume a trivial case where the propagator is a linear operator, we can rewrite the numerical integration procedure as

$$\boldsymbol{A}\boldsymbol{x} = \begin{pmatrix} I & & & \\ -\phi & I & & \\ & \ddots & \ddots & \\ & & -\phi & I \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \\ \vdots \\ x_k \end{pmatrix} \stackrel{!}{=} \begin{pmatrix} x_0 \\ 0 \\ \vdots \\ 0 \end{pmatrix} =: \boldsymbol{g}. \tag{4.1}$$

In this form, sequential time stepping corresponds to performing a forward triangular solve to this equation, which is usually not directly parallelizable.

We now introduce a coarse grid Θ_Δ which is obtained similarly to the fine grid but with a coarse step

$$\Delta = \frac{t_0 - t_f}{hk_c},$$

where the coefficient k_{cf} is referred to as *coarsening factor*. Then the coarse grid can be defined as

$$\Theta_\Delta = \{t_i \in \mathbb{R} \mid t_i = t_0 + i\Delta, \text{for } i = 0, \dots, hk_{cf}\}.$$

Analogously to the fine propagator in fine grid, the propagator employed in coarse grid is referred to as *coarse propagator* and can be different from the one in fine grid. To introduce the solution of coarse grid, we denote k_c times of successive application of fine propagator ϕ as ϕ^{k_c}, we can rewrite Eq. (4.1) as

$$\boldsymbol{A}_\Delta \boldsymbol{X} = \begin{pmatrix} 1 & & & \\ -\phi^{k_c} & 1 & & \\ & \ddots & \ddots & \\ & & -\phi^{k_c} & 1 \end{pmatrix} \begin{pmatrix} X_0 \\ X_1 \\ \vdots \\ X_K \end{pmatrix} \stackrel{!}{=} \begin{pmatrix} X_0 \\ 0 \\ \vdots \\ 0 \end{pmatrix} =: \boldsymbol{g}_\Delta. \tag{4.2}$$

It is easy to verify that the solution vector $\boldsymbol{x}$ of Eq. (4.1) at time points x_i, for $i = 0, \ldots, hk_{cf}$, satisfies the solution vector $\boldsymbol{X}$ in Eq. (4.2). Now use the coarse propagator Ψ to replace ϕ, we can get

$$\boldsymbol{B}\boldsymbol{X} = \begin{pmatrix} 1 & & & \\ -\Psi & 1 & & \\ & \ddots & \ddots & \\ & & -\Psi & 1 \end{pmatrix} \begin{pmatrix} X_0 \\ X_1 \\ \vdots \\ X_K \end{pmatrix} \overset{!}{=} \begin{pmatrix} X_0 \\ 0 \\ \vdots \\ 0 \end{pmatrix} = \boldsymbol{g}_\Delta, \tag{4.3}$$

which represents the approximation to the coarse grid. The approximation is performed with the help of classical residual-correction methods [BKK17], which is the basis of the multigrid algorithm. We give an example from [SDB22] of how such residual-correction method is applied. The residual at C-points by l-th iteration $\boldsymbol{R}^{(l)}$ can be defined as

$$\boldsymbol{R}^{(l)} \coloneqq \boldsymbol{g}_\Delta - \boldsymbol{A}_\Delta \boldsymbol{X}^{(l)}, \tag{4.4}$$

and the coarse grid correction $\boldsymbol{C}^{(l)}$ is defined as

$$\boldsymbol{C}^{(l)} \coloneqq \boldsymbol{B}^{-1}\boldsymbol{R}^{(l)}. \tag{4.5}$$

The coarse grid correction $\boldsymbol{X}^{(l)}$ is used to update the states in the coarse grid, i. e. the C-points, iteratively:

$$\boldsymbol{X}^{(l+1)} = \boldsymbol{X}^{(l)} + \boldsymbol{C}^{(l)}. \tag{4.6}$$

Plugging Eq. (4.4) and (4.5) into (4.6), the update rule becomes

$$\boldsymbol{X}^{(l+1)} = \boldsymbol{X}^{(l)} + \boldsymbol{B}^{-1}\left(\boldsymbol{g}_\Delta - \boldsymbol{A}_\Delta \boldsymbol{X}^{(l)}\right), \tag{4.7}$$

which can be viewed as a pre-conditioned stationary iteration. The term $\boldsymbol{A}_\Delta \boldsymbol{X}^{(l)}$ represents the collection of fine-solutions, where each time point can be computed in parallel, using the results from the preceding coarse solve as initial values. The $(j+1)$-th row of Eq. (4.7), with $j = 0, \ldots, K-1$, corresponds to a given C-point, and may be reformulated by multiplying Eq. (4.7) with $\boldsymbol{B}$ from the left

$$X_{j+1}^{(l+1)} = \Psi X_j^{(l+1)} - \Psi X_j^{(l)} + \phi^{\mathrm{cf}} X_j^{(l)}.$$

It is easy to see that the fine-grid approximation is recovered at the C-points if the terms $\Psi X_j^{(l+1)}$ and $\Psi X_j^{(l)}$ converge towards each other as l grows.

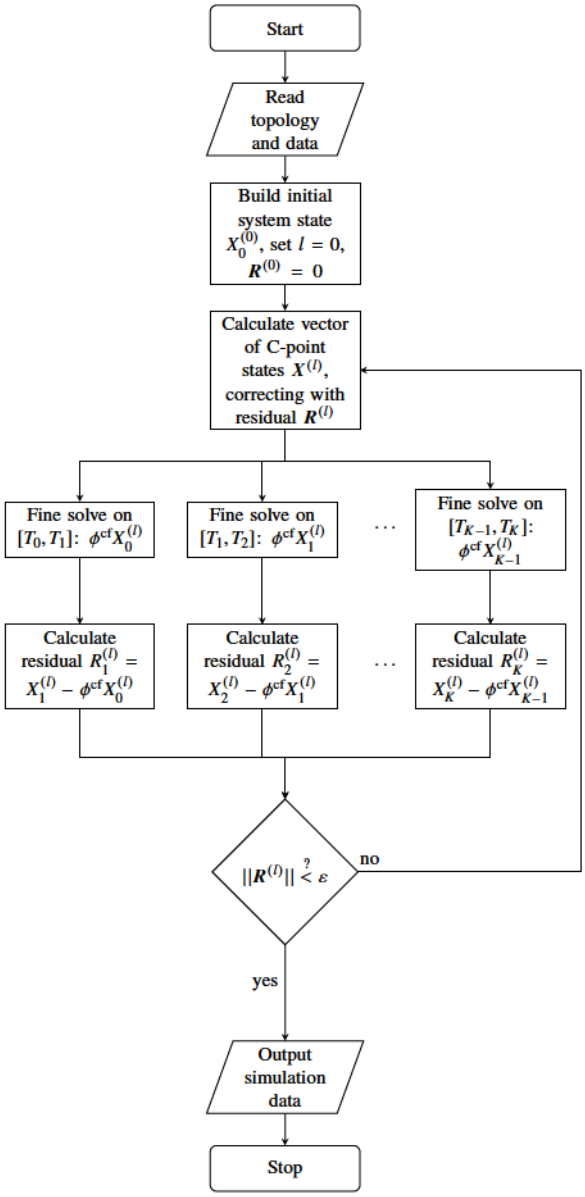


Figure 4.1: Execution flow-chart of the implemented PinT algorithm in two-levels. Figure reproduced based on [SDB22] under the CC BY 4.0 license.

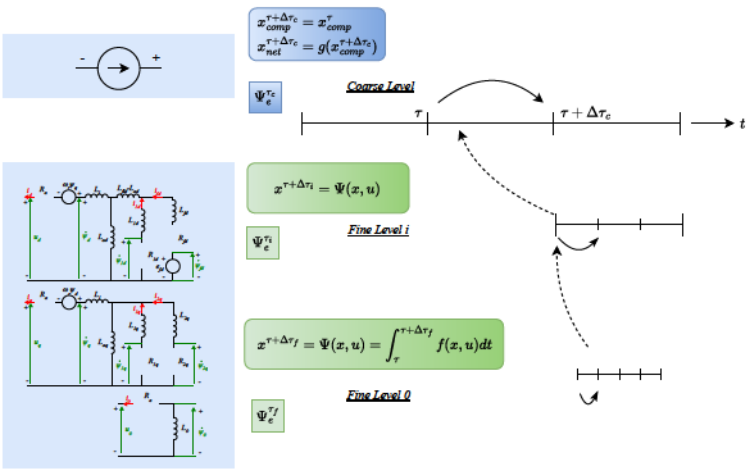


Figure 4.2: Illustration of hybrid models, different models are applied at different PinT level.

4.2 Improving the Convergence with Parallel-in-Time

Parallel-in-Time with Hybrid Models

Typical selection of time propagator for PinT scheme are the L-stable methods, for instance implicit Euler or implicit midpoint rule [FS21; SDB22], so that MGRIT method perform better. It is not difficult to notice this choice in studies of power system simulation under PinT, where an L-stable methods, or at least an A-stable methods, are used. For instance, in [Fal+14; FLW19], Implicit Runge Kutta is used, in [SDB22], Euler backward, and in [CLD22], implicit trapezoidal rule is used.

In the PinS algorithm introduced in Chap. 3, nonlinear components are explicitly integrated. Therefore, in the case where coarsening factors are large, the coarse grid could diverge already from the beginning. To overcome this convergence issue, we proposed a hybrid modeling technique, so that when we were performing integration on the coarse grids, a different model, and preferably a reduced-order model that is easier to converge, is applied. We then investigate three different model combinations for performing stable integration on the coarse grids:

1. Full-Oder-Full-Order: This is the most straightforward approach, added as

reference, where the same model and integration method is used across all levels.

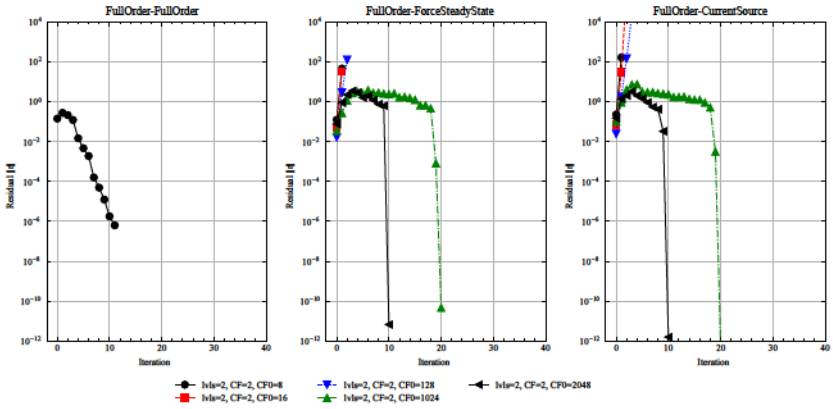
2. Full-Oder-Force-Steady-State: In the coarse grid, we assume the dynamic model is already in a steady state. Therefore, we can solve the differential equations as algebraic equations by setting the derivatives of all differentiated variables to zero. Meanwhile, due to the steady-state assumption, some models can be further simplified, e. g., for the synchronous generators, the coupling of rotor speed to dq-axis flux linkage is decoupled since the steady state speed is 1 (p.u.).

3. Full-Oder-Reduced-Order: Models are replaced by a reduced-order model in the coarse grids. Take synchronous generators again as an example, there are models with different orders [Mil10] by neglecting transient dynamics with different orders. In our case, we reduced the dynamic model into a static current source model in the coarse grid.

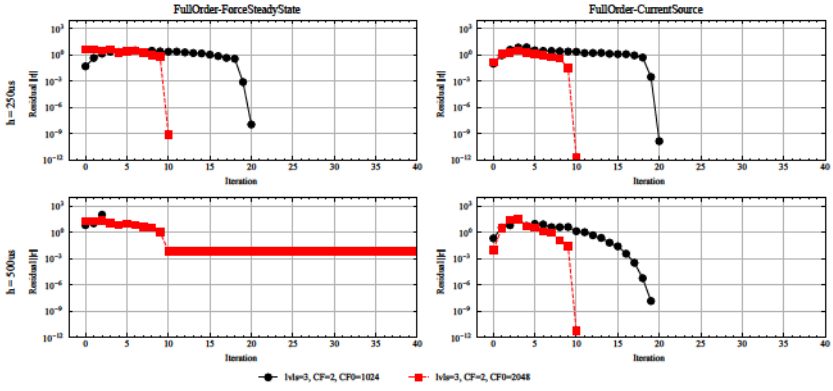
The result of these three approaches is shown in Fig. 4.3. In general, both hybrid model approaches have shown better convergence than the straightforward Full-Oder-Full-Order, i. e. using the full-order model at all levels. In Fig. 4.3a, the Full-Order-Full-Order case already diverged from beginning when the first level coarsening factor, i. e. CF0, gets large, due to the poor convergence on the coarse level. However, both hybrid approaches start to show better convergence when the coarse grid is further away from the fine grid. This might due to the fact that the steady state models used in the coarse grid are only more accurate after the dynamics from the fine grid are damped out, therefore, the interval between two C-points needs to be large enough. Fig. 4.3b gives a more detailed comparison of two hybrid approaches, it can be notice that the Full-Order-Force-SteadyState model would go into stagnation when the fine grid step size getting larger, such effect has also been observed [De +23] in other applications by the parallel-in-time community. This might due to the fact that the coarse solution is too erroneous so that fine grid approximation cannot improve convergence anymore.

Buffered Coarsening

In order to increase convergence with multi-level PinT algorithm under aggressive coarsening, we proposed a technique called *buffered coarsening*, the trick is to first use a smaller coarsening factor for the first level of coarse grid, then for the coarser



(a)



(b)

Figure 4.3: Comparison of different hybrid model approaches on the coarse level to improve convergence.

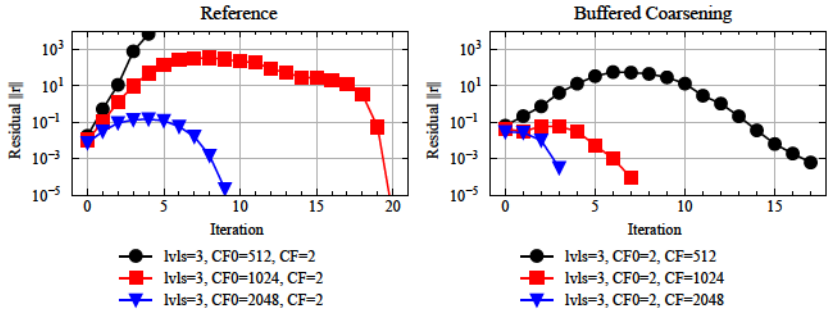


Figure 4.4: Convergence performance using buffered coarsening, where the coarse grid first uses a small coarsening factor, and then applying more aggressive coarsening

grids more aggressive coarsening can be applied. A benchmark is performed based on the SciGrid-DE case and shown in Fig. 4.4.

When using buffered coarsening, we could observe that the convergence is improved compared to simple monotonic coarsening while keeping time points in the coarsest grid the same. The reason is that when we perform buffered coarsening, the resulting coarse grids are in fact more close to the fine grid, whereas in the case with increasing coarsening factor monotonically, especially in the case with aggressive coarsening, all coarse grids are far away from the fine grid, which deteriorates the quality in the coarse grids and therefore posing higher challenge to the global convergence.

4.3 Computational Performance Analysis

Theoretical Speedup

Let w be the work needed for computing single time step, total work for the sequential time stepping is

$$W_s = wn_s, \tag{4.8}$$

where n_s is the total number of steps, similarly, we can formulate the total work for performing PinT, take a two-level algorithm for example:

$$\begin{aligned} W_p(1) &= \underbrace{w \cdot \frac{n_s}{c_f}}_{\text{sequential}} + \underbrace{w \cdot k \cdot \frac{n_s}{c_f} \cdot c_f + kD(1)}_{\text{parallelizable}} \\ &= \frac{wn_s}{c_f} + wkn_s + kD(1), \end{aligned} \tag{4.9}$$

with c_f be the coarsening factor, n_s the total number of steps at finest level, and k the number of iterations until converge, and $D(p)$ the additional work brought by PinT algorithm in each iteration, e. g. computing residual, communication, etc. Let p to be the number of processors for computing the parallel part with $p \leq \frac{n_s}{c_f}$ as the intervals between consecutive C-points are parallelized, we can formulate the *estimated parallel speedup* $\tilde{S}_p$ by

$$\begin{aligned} \tilde{S}_p &= \frac{W_s}{W_p(p)} = \frac{wn_s}{\frac{wn_s}{c_f} + \frac{wkn_s}{p} + \frac{kD(p)}{p}} \\ &= \frac{c_f}{1 + k \cdot c_f/p + D'(p)}, \end{aligned} \tag{4.10}$$

where

$$D'(p) = \frac{kD(p) \cdot c_f}{wn_s}.$$

Normally we could expect that $kD(p) \cdot c_f \ll wn_s$, therefore, by ignoring $D'(p)$ and assuming p takes its maximum, we can find an upper-bound

$$\tilde{S}_p < \frac{c_f}{1 + c_f^2/n_s}. \tag{4.11}$$

In a multi-level PinT algorithm like MGRIT, the computation for coarse grid is also parallelized. Denote the work and speedup for a two-level algorithm as $W_p^{\langle 0 \rangle}$ and $S_p^{\langle 0 \rangle}$, we can formulate the work and speedup for an $l+2$ level algorithm as

$$W_p^{\langle l \rangle}(p) = W_p^{\langle l-1 \rangle}(p) + \frac{wkn_s}{p} + kD(p). \tag{4.12}$$

This formulation can also be expanded as

$$W_p^{\langle l \rangle}(p) = \frac{wn_s}{\prod_{j=0}^{l} c_f^{\langle j \rangle}} + \left(\frac{wkn_s}{\prod_{j=0}^{l-1} c_f^{\langle j \rangle}} + \cdots + \frac{wkn_s}{c_f^{\langle 0 \rangle}} + wkn_s \right) \cdot \frac{1}{p} + kD(p), \tag{4.13}$$

where $c_f^{\langle j \rangle}$ denotes the coarsening factor at j-th level. In Eq. (4.13), the sequential part is the first term, we can notice that it is significantly reduced compared to the sequential part in Eq. (4.9). Assume each level was parallelized by the number of corresponding C-points, we can estimate the max parallel speedup for multi-level case by

$$\tilde{S}_p = \frac{W_s}{W_p^{\langle l \rangle}(p)}. \tag{4.14}$$

Plugging Eq. (4.13) into Eq. (4.14), simplifying it, and assume that we have the same number of processors as the parallelizable intervals:

$$\tilde{S}_p = \frac{\prod_{j=0}^{l} c_f^{\langle j \rangle}}{1 + k \cdot \sum_{i=0}^{l} \left(\prod_{j=i}^{l} c_f^{\langle i \rangle} \right) \cdot (n_s \cdot p)^{-1} + kD(p) \prod_{j=0}^{l} c_f^{\langle j \rangle} \cdot (wn_s)^{-1}}, \quad (4.15)$$

with p less than the number of parallelizable intervals. If the $D(p)$ term can still be ignored, it gives

$$\tilde{S}_p = \frac{\prod_{j=0}^{l} c_f^{\langle j \rangle}}{1 + k \cdot \sum_{i=0}^{l} \left(\prod_{j=i}^{l} c_f^{\langle i \rangle} \right) \cdot (n_s \cdot p)^{-1}}, \quad (4.16)$$

The maximum theoretical parallel speedup for parallel-in-time application can also be derived using another approach, for which we refer to as PinT margin. For a two-level algorithm, the PinT margin m_t is defined as

$$m_t = 1 - \frac{1}{c_f} - \frac{k \cdot c_f^{\langle 0 \rangle}}{n_s}. \quad (4.17)$$

Recall that the C-grid and F-grid starts from the same initial values, i. e.

$$\Psi X_0 = \psi_c^k x_0,$$

where $X_0 = x_0$. The worst case for PinT is where the number of iterations equals to the number of C-points, in this case, parallel-in-time execution collapses to sequential-in-time execution. We could view the process of PinT algorithm as if the PinT progress is "chasing" the sequential-in-time progress, and the margin m_t shows when the PinT algorithm converges, how much time, in terms of C-points, is still unfinished in sequential-in-time simulation. Therefore, m_t indicates the portion of simulation result which was gained "for free" compared to conventional sequential-in-time simulation. The larger value of the margin then indicates higher effectiveness in applying the PinT over sequential-in-time stepping, and implies potentially higher gain from optimizing the parallel execution performance than the cases with lower margins.

We tested the convergence with various coarsening factor $c_f^{\langle 0 \rangle}$, Fig. 4.5a shows the change of residual during the iterations, and Fig. 4.5b shows the theoretical maximum speedup $\tilde{S}_p$ calculated based on the convergence. We can observe that the cases with smaller coarsening factor yields relatively larger $\tilde{S}_p$. This may imply

that smaller coarsening factors lead to better acceleration potential, however, this implication has two main flaws. First, since C-points and F-points are relatively close to each other, the accuracy of coarse grid may already suffice, and performing PinT could only be over-complicating the problem; secondly, there are not much F-points to be computed between the C-points, therefore, computational efficiency may be compromised since the solver need to switch between residual computation and fine propagation frequently. In fact, this is one of the way to fool the mass when giving parallel-in-time result [Göt+21]. Moreover, in Fig. 4.5b we also illustrated the effect of the parallel overhead term $D(p)$ affects the theoretical speedup limit, as the overhead is scaled up by the number of iterations k, we can notice the penalties for poor convergence on the performance in the case where it took more iterations to converge.

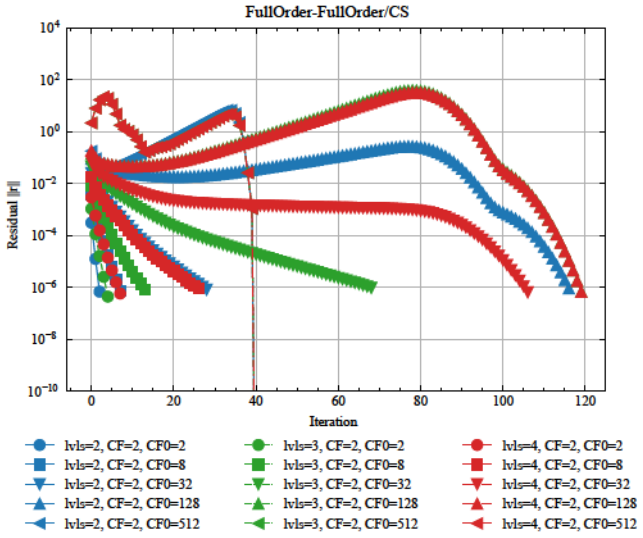
Performance Estimation

Using the performance model developed in the previous section, we can estimate the simulation performance under PinT acceleration. The performance model is based on parameters such as coarsening factor at each level $c_f^{\langle j \rangle}$, number of iterations to converge k, and number of processors p. In addition, we also need a rough estimation of $D(p)/w$. Since $D(p)$ represents the communication overhead as well as the additional computation overhead induced by the PinT algorithm, therefore, it depends on both problem size and number of processors employed. The modeling of $D(p)$ can be found in previous studies on PinT performance such as [Fal+14] or in general on parallel computing [Qui08], since the scaling overhead due to communication usually scales $O(\log(n))$. For simplicity, we treat the fraction between $D(p)$ and w as

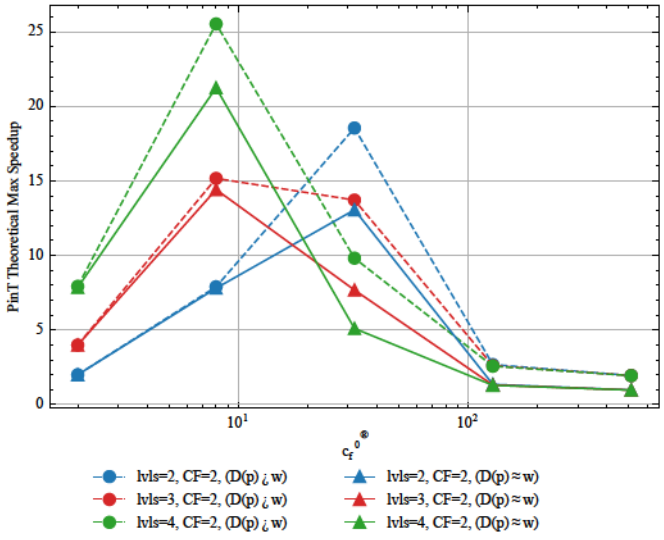
$$D_w(p) = \frac{D(p)}{w} \approx d_w \cdot log(p). \tag{4.18}$$

The value of d_w can be further approximated by performing a test simulation and comparing the measured performance with the estimated performance. Therefore, we can obtain an empirical estimation of additional parallelization overhead on the target platform and problem.

We compare the estimated performance with measurements for the IEEE14 and the SciGrid-DE case, results are shown in Fig. 4.6. In general, the estimated performance fits with the actual performance. However, there are still a gap between estimated and actual speedup, which is due to the fact that we simplified the additional costs $D(p)$. Better estimations can be made by modeling $D(p)$ in a more



(a)



(b)

Figure 4.5: Comparison of theoretical maximal PinT speedup for different MGRIT levels and coarsening factor $c_f^{\langle 0 \rangle}$.

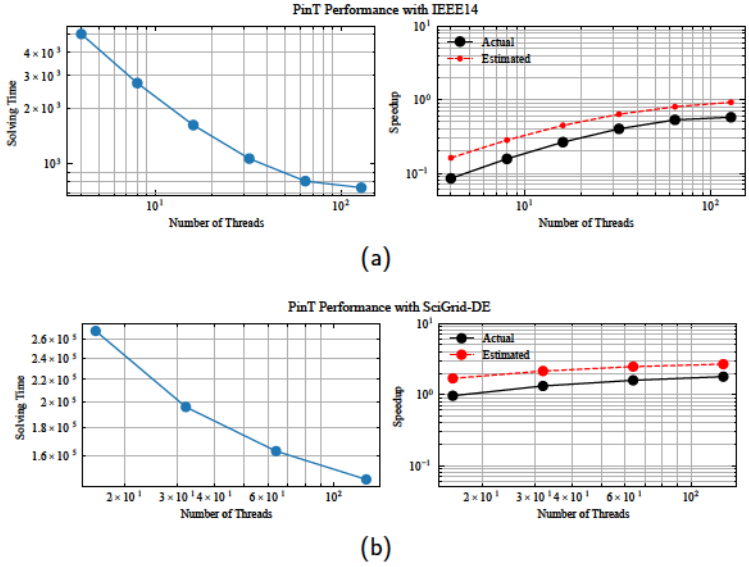


Figure 4.6: Estimated parallel speedup using the performance model compared with actual measurement

realistic way, for instance modeling it in the form of $\alpha + \beta \cdot \log(p)$ [Fal+14], with some assumptions and measurements to infer the value of α and β, which represents the computation related and communication related overhead, respectively.

4.4 Implementation of the Combined Parallel-in-Space-Time Solver

In this section, we present a non-intrusive implementation to combine PinT and PinS algorithms. The PinT solver is taken from the open-source software package XBraid [XBRAID], which provides an implementation for MGRIT; the PinS solver uses the implementation from Chap. 3. Special considerations in implementing the PinS including a data-oriented design, which is not common in traditional power system simulators, will be discussed in more detail in later chapters. The key reason for such data-oriented design is that both heterogeneous computing oriented PinS implementation and PinT algorithm need a flattened architecture for the system *states*, so that these states can be easily accessed and propagated. In traditional object-oriented program design, these data would have been encapsulated in different objects, for instance the objects that represent the entities owning these data. The difference of these two program designs is beyond the scope of this section and are discussed later.

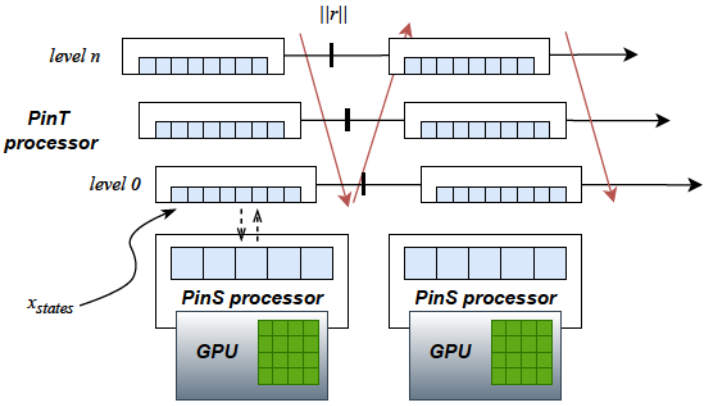


Figure 4.7: PinTS Implementation

The implementation of the combined PinTS solver is illustrated in Fig. 4.7. Where PinT processors are executed on CPUs which manages the allocated PinS solvers on GPUs. Each PinT processor communicates with other processors viamessage-passing interface (MPI) in order to coordinate the parallel execution.

In addition, to increase the through-put from a single compute node, we also enabled dynamic scheduling of multiple PinS solvers on the same GPU. This is because each PinS solver will create its own CUDA context, and the execution is scheduled by the GPU itself, which usually turns out to be inefficient. To overcome this, one solution is to apply a dynamic scheduler. In this case, we use the scheduler from NVIDIA by enabling multi-processing service (NVIDIA-MPS) on the computing nodes. And we kept this configuration in all later benchmarks.

Combined Parallel-in-Space-Time Performance

Finally, we benchmark the simulation performance under combined PinTS algorithm. The simulations are executed on the supercomputer JURECA at the Jülich Supercomuting Center. The supercomputer JURECA has 480 standard compute nodes, where each node has: 2 AMD EPYC 7742 CPUs, in total 2×64 cores per node with a base frequency of 2.25 GHz; 512 GB memory with base frequency of 3200 MHz, and the network connection uses InfiniBand HDR100 (NVIDIA Mellanox Connect-X6). In addition, it has 192 accelerated compute nodes, which

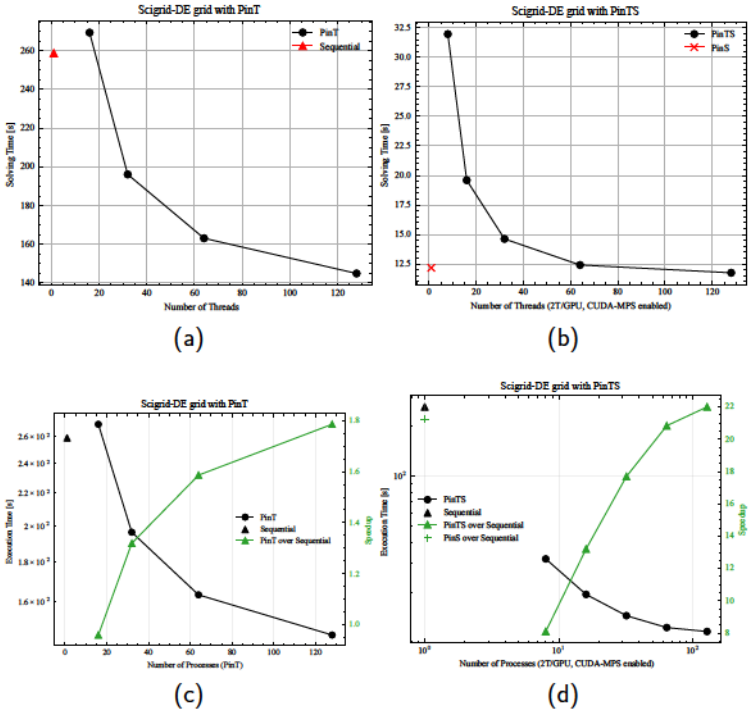


Figure 4.8: Benchmark result on dynamic SciGrid-DE system. (a)(c) Solving time and speedup with PinT, (b)(d) Solving time and speedup with PinTS

has additionally 4 NVIDIA A100 GPU per node with in total 4× 40 GB HBM2e memory. The compute nodes are diskless, with a network connection of 350 GB/s to the Jülich Storage Cluster (JUST). The simulator is compiled with Intel MKL `2024.2.0`, OpenMPI `5.0.5`, and NVHPC `24.9`.

The result can be found in Fig. 4.8, we can gain additional speedup over applying PinT or PinS alone after around 128 processors. We can notice that this break-even point needs much more processors compared to applying PinT alone, which was around 32 processors. This is because the PinS algorithm is executed on GPU whereas the PinT algorithm is executed on the CPU, it yields higher communication overhead within each iteration due to slower data transfers between chips.

4.5 Summary

In this chapter, we introduced the PinT algorithm and the combined PinTS algorithm to accelerate power system simulation with transient dynamics. In order to increase convergence under PinT, we employed several approaches. Firstly, we employed a hybrid modeling technique that dynamically changes the order of the model at different PinT grids. Secondly, we use buffered coarsening to reduce the number of iterations needed with the same total coarsening factors. A performance model is derived to estimate the actual PinT performance, which provides an insight on potential computing resources needed to achieve desired speedup. Finally, we demonstrated the performance using the combined PinTS approach, showing additional speedup compared to applying PinT or PinS independently.

Several improvements on methodology and implementations could still be further investigated. For instance, one could consider combining L-stable method on the coarse level and still employ cheap integrators on the fine level; moreover, different the combined space-time parallel approach suffers from communication delays since the PinT algorithm is executed on CPU and the PinS algorithm on GPU. An implementation that extracts SIMD from the PinT algorithm is also possible and could be further studied.

Chapter 5

DATA-ORIENTED SIMULATOR DESIGN FOR FLEXIBLE HARDWARE ACCELERATION AND CO-SIMULATION

In this chapter, we introduce a data-oriented design for power system simulator which can efficiently combine parallel-in-space and parallel-in-time algorithms, supporting multi-domain simulations as well as efficient utilization of hardware accelerators.

From Chap. 3 and Chap. 4, we already introduced the methodology of the PinS as well as combined PinT and PinS solver, and especially the concept of mapping power system simulation into heterogeneous computing framework. In this chapter, we go more into detail about how the design of the simulator could perform execution with hardware accelerator and cooperate with PinT algorithm efficiently.

Traditional simulation software usually use object-oriented (OO) designs, as a result of the trend of OO programming languages like Java and C++. Features in object-oriented design like polymorphism and inheritance has improved modularity and code reusability, therefore promoting the wide adoption of these languages. Examples can be found in open-source power system simulation tools, for instance GridDyn [Kel+15], DPsim [Mir+19], etc.

Besides the well-known OO design pattern, the data-oriented (DO) design pattern has also arisen discussions. In the following sections, we discuss the advantages of data-oriented design compared to the traditional object-oriented design and explain why it is suitable for implementing high performance simulators. Recent studies have also explored the potential of data-oriented simulators for power system simulation. For instance, in a very recent work [CDD23], Cheng et al. proposed data-oriented simulator design for cyber-physical systems, focusing the interaction of power system and communications systems, as well as the simulation of cyber-attacks.

5.1 Data-Oriented Design

Data-oriented design has been widely adopted in game engines, for instance in popular engines like Unity [UNI25], Unreal Engine [UNR25]. Additionally, there have been ongoing discussions concerning the application of data-oriented design patterns in general software development in order to improve runtime efficiencies.

Nevertheless, most of the opinions reached consensus that data structure is the core to a program [KP10]. In this work, we will not engage in debates such as whether which pattern is better [CPP18], instead, we will focus on the advantages of applying data-oriented design in our specific application case.

In general, there are several main advantages of object-oriented design pattern:

1. Inheritance: The concept of having data classes that represents objects, and allows definition of sub-classes which can inherit members and methods of parent classes. This feature could reduce development time and allows more reuse with the existing code.

2. Polymorphism: Allows different data types or classes to be accessed using the same interface. This again facilitates code reuse and reduces development time.

3. Encapsulation: Data and related behaviors are bundled and hidden into a single class, therefore the access to an object's data is restricted, providing higher system security and avoids accidental data corruptions.

These features are generally favorable for the maintenance and development of the software. However, in computing-intensive programs, they can become obstacles to performance, a main issue is the allocation of data in the memory.

The memory space available for program is divided into two segments, namely the heap and the stack memory. The stack allows data to be implemented in a last-in, first out order, and the allocation is performed linearly in the memory space; whereas the heap implements a tree-based structure that allows data to be dynamically allocated and accessed during the runtime, and results in a non-sequential allocation in memory. Therefore, when a program needs to allocate data dynamically during the runtime, it will look for free spaces in heap and allocate the data, which will lead to the data being allocated non-sequentially even though they are sequentially instantiated. This will cause two main issues: first, if we only need to apply certain behavior on a small set of data of an object, the processor still needs to fetch the complete object from the memory, which leads to inefficient usage of cache; second, if we need to operate on the data from multiple objects, as they are spread in the memory, it will result in high cache misses, which, in turn, eventually lead to lower performance.

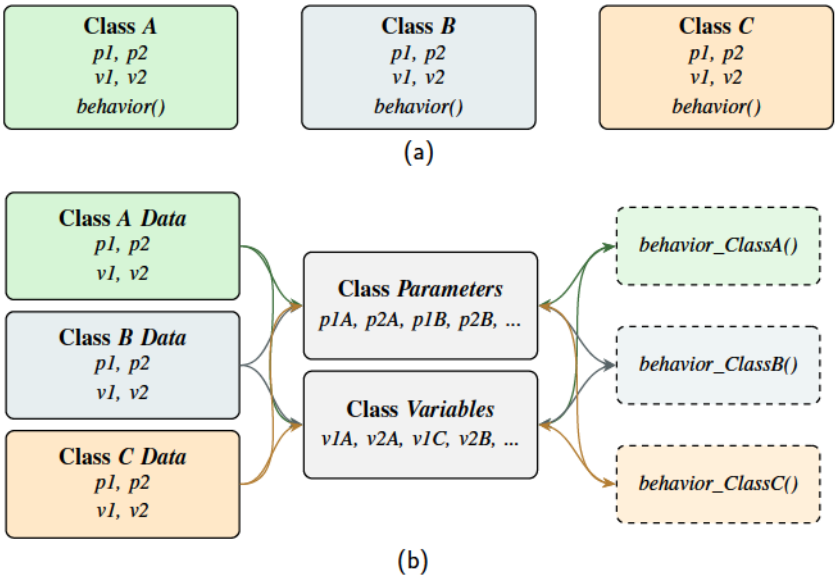


Figure 5.1: Example of (a) object-oriented design: objects has its data and own behavior internally implemented, explicit interaction among objects is not needed; (b) data-oriented design: data and behavior of the objects are implemented separately.

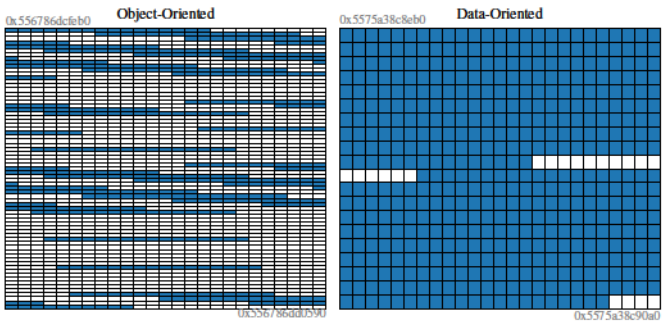


Figure 5.2: Comparison of memory layout in data-oriented and object-oriented design, each cell represents a memory address. Addresses with data allocated are marked.

We demonstrate the difference of the OO and DO design pattern via an example shown in Fig. 5.1. We define three object classes with each of them containing two parameters and two variables, and own behavior as a member function. In the OO design, data and behavior are encapsulated in each object definitions. The DO design defines two central classes, namely *Parameters* and *Variables* to summarize all the data from the objects, and implement the behavior functions for each object separately. The difference of their data allocation in memory is illustrated in Fig. 5.2. We can notice that the data layout in DO design is more aligned, whereas with OO design it is more scattered in the memory space. This has demonstrated the core idea of using a DO design pattern, that is to ensure more efficient data access.

The DO design is also beneficial to parallel execution. As illustrated in the example Fig. 5.1, since all data are exposed and stored centrally, operations can be easily parallelized by the compiler or via implementing parallel constructs explicitly.

Overall, the DO design pattern gives more emphasize on the data rather than on the code. For computing-intensive tasks, DO design provides potentially higher performance and easier for optimizations.

5.2 Data-Oriented Design for Heterogeneous Computing Framework

We proposed a DO design so that we can utilize its advantages to increase simulation performances, moreover, the traditional OO design pattern is still maintained on the initialization phase, for instance for setting up the data structures, etc. The complete simulator implementation follows a hybrid design pattern, whereas DO is more respected in terms of simulation.

Overall, the simulator execution is fit into three steps:

- Initialization: circuit setup and initializing data structure on host and device; matrix factorization or calculation of matrix inverse
- Solve: main computation loop, components and networks are solved
- Post-Solve: ending phase, write result and release memory

Since the network can be seen as interconnection of different components such as lines, generators, the implementation of component objects is the critical part of the overall structure. Two classes have been implemented to describe each type of component. The first of these is a *component* object to represent a single instance

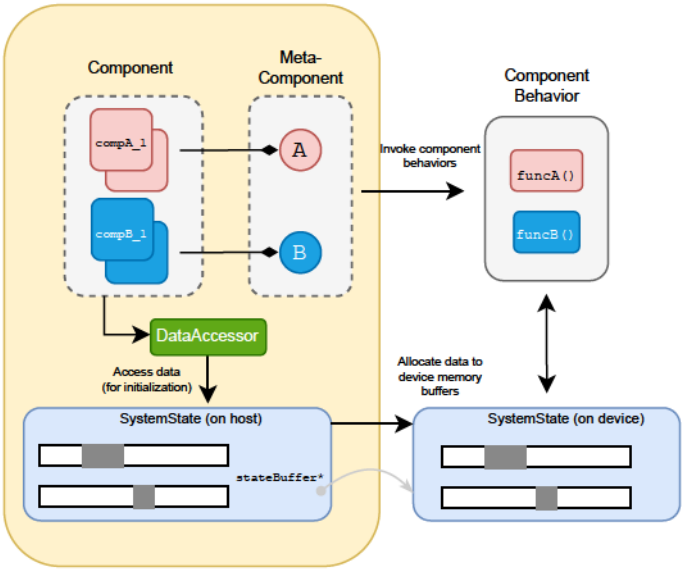


Figure 5.3: Data-oriented design for execution on heterogeneous architecture

of a component of certain type. The second of these is a *meta-component* object representing components of certain type.

The component objects implement methods that can initialize the *SystemState* object which includes simulation-related matrices, parameters, variables, etc., this is similar to the classes in Fig. 5.1b. Each instance of the component object represents a physical component in the system, such as a transmission line, a generator, or even an inductor. Each component object has a link to the meta-component of its type. The meta-component contains the method for accessing the data in *SystemState* as well as the corresponding *behavior*. The behaviors are the simulation-related functions of different types of components, such as numerical integration, and are implemented separately. Moreover, each behavior can be consists of different *basic behaviors*, such as transformation (e. g. Park Transform), vector addition, etc. The overall aim of this approach is twofold: firstly, to increase the reusability of the implemented

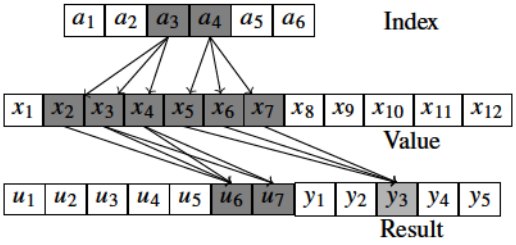


Figure 5.4: Illustration of the gather-scatter approach for parallel execution on device.

code, and secondly, to reduce the difficulties in optimizing the performance during parallel execution.

In addition, a *DataAccessor* object is implemented to help component objects to access component-related data stored in *SystemState*. The relationship of these objects are illustrated in Fig. 5.3.

System States

The simulator is designed around the data structure, which in our case is the *SystemState* class. In order to separate component behaviors from the objects, the data is structured into two layers: an index layer and a value layer, both using a generic data type, for example a vector `std::vector` from the standard containers of C++. Therefore, the behaviors are executed following a gather-scatter approach: all necessary indices are loaded from memory, including the location of parameters, variables in the data, and the matrix/vector location in MNA where each component instance should read/write into. The behavior function reads data from these locations, perform computations, and writes to the relevant locations after computations, as shown in Fig. 5.4.

The host-side class `SystemState` consists of several standard containers such as `std::vector` and `std::map`, as shown in Fig. 5.5, which are filled by all the component objects during the initialization phase. On the device side, the `SystemState` data are stored in memory *buffers*, it is initialized before the simulation and resides on the device throughout its lifetime. The host-side `SystemState` contains a pointer to the memory buffer on GPU where the GPU-side states are allocated, so that it can operate with this pointer to access the data stored on the GPU and perform read operations, for example to update the results on the host-side.

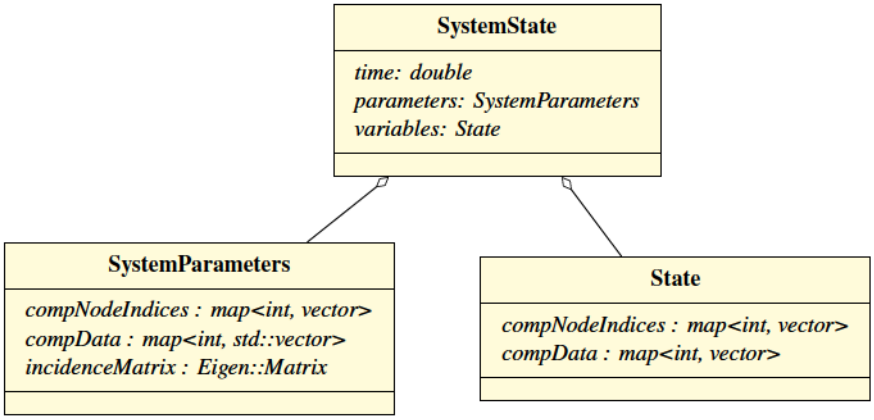


Figure 5.5: Class diagram of `SystemState`

Behaviors

Simulation-related behaviors are for example numerical integration routines, which are implemented as separate functions to the components, and whose execution is controlled by the *meta-components*.

On the device, these behaviors are implemented as kernel functions that take pointers to the memory buffers where the *SystemState* data is allocated. An example of device-side behavior functions can be found in List. 5.1, which shows a *pre-step* that stamps the results into the right-side vector in the network step (MNA), a *post-step* that performs integration, as well as helper functions that can only be called by the device-side kernels.

The OpenCL framework and the flexibility layer were introduced in Chap. 3, where we exploit the similarity between vendor-specific frameworks and OpenCL so that the code can be translated into each other. On the host side, the implementation of flexibility layer relies on abstracting the API of different frameworks; on the device side, it relies on a set of predefined macros as we want to keep the structure more flat and the fact that the kernel code is compiled at the runtime.

5.3 Integration with Parallel-in-Time Algorithm

In Chap. 4 we introduced the PinTS method, which computes the solution of a coarse grid and computes finer grids based on it to accelerate the computation of global fine grid solution. Implementing PinT in a power system simulator using an

```
#include <clcuUtils.h>

// macros defined from host to determine the target framework
// # USE_CUDA
// # USE_HIP
// # USE_OPENCL

HELPER void transform(REAL* vars_in, Real* vars_out) {
// helper function that only called on the device
}

// kernel function
KERNEL void preStep( const GLOBAL REAL* params, GLOBAL REAL* variables, int* indicies) {
    int id = GET_GLOBAL_ID;
    REAL params_local[NUM_PARAMS], variables_local[NUM_VARS];
    arrloadn(params, params_local, NUM_PARAMS);
    arrloadn(variables, variables_local, NUM_VARS);
    writeToComponentOutputVector(variables_local, paramas_local);
}

    // kernel function
KERNEL void postStep( const GLOBAL REAL* params, GLOBAL REAL* variables, int* indicies) {
    int id = GET_GLOBAL_ID;
    REAL params_local[NUM_PARAMS], variables_local[NUM_VARS];
    arrloadn(params, params_local, NUM_PARAMS);
    arrloadn(variables, variables_local, NUM_VARS);
    integrate(variables_local, paramas_local);
    arrstoren(variables_local, variables, NUM_VARS);
}
```

Listing 5.1: Illustration of behavior code for kernels

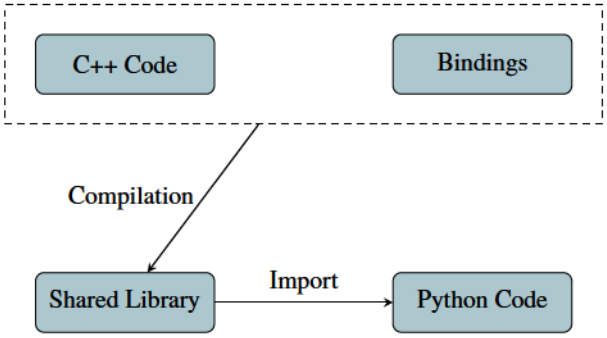


Figure 5.6: Concept of the Python Interface to the Implemented C++ Simulator

object-oriented design is challenging, because it requires a set of system *states* to be selected and exposed to the PinT solver to operate, which conflicts with the design of object abstractions and encapsulated variables. However, data-oriented design is naturally suited to this requirement, as the vector of states is naturally exposed and therefore is directly manipulable. During execution, the state vector can be accessed and distributed by the parallel-in-time processors.

5.4 Python Interface

Python has been one of the most popular programming languages for years. In the recent rankings, it is still at the top of the languages listed [Cas24]. This scripting language features clear syntax, and automatic memory management, which leads to reduced program development time. In addition, Python programs are usually *interpreted* without the need for compilation, and the python implementation (or Python interpreter) is also very portable, so that a python program can be easily deployed on different platform. As a result, Python is easy to learn and favored in many fields, including the engineering field.

To increase the usability of the implemented simulator, we implemented a Python interface using Pybind [25], so that the simulations can be created and executed from a Python script. The concept is illustrated in Fig. 5.6, after implementing bindings for related objects and functions, the original program can be compiled with the bindings together and create a shared library, which can then be imported into Python scripts and the bindings can be invoked.

Chapter 6

TOWARDS HIGH PERFORMANCE SIMULATION ON GPU

In Chapter 3, 4, we introduced the parallel algorithms and analyzed their theoretical speedup, as well as the data- and task-parallel design for efficient execution on GPU, in Chapter 5 we introduced the data-oriented design to increase the overall performance of the simulator. In this chapter, we look in more detail at how to optimize the GPU-based computations in our simulation.

Optimizing the performance of software, and in particular scientific computing programs, has been studied for decades. Typical tuning techniques such as loop unrolling, vectorization, etc. can usually be applied to improve code performance. In addition, to tune the performance over different environments or platforms, the configuration of these techniques, e. g. loop unroll factor, tile sizes [KKO03], needs also to be analyzed and adapted. In order to see how the applied techniques and its parameters impact the performance, the simplest way is to implement them in a separate program and run benchmarks for comparison. Of course, this approach can be very time-consuming and challenging: technical insight, expert knowledge of the algorithm applied, hardware architecture, etc. are also required in order to tune the performance more effectively. In the end, this leads to a time-consuming process and needs to be performed repeatedly for evolving hardware. Obviously, the ability to tune the performance automatically across diverse platforms would be highly advantageous.

Historically, there have been two separate pathways to tune the code automatically. The first way is to improve the compilers so that they can generate high performance machine code for different platforms [HPP09] with minimal adjustments in the code, this is the path the compiler community is constantly working. The second way is to extract the part with the dominant cost, and then eventually produce packaged tools with highly optimized code by groups of experts. Many linear algebra libraries or numerical functions are of examples of the second way [CPD01], e. g. BLAS, FFT, etc. Since our objective is not to build a compiler, in this work, we aim to see the extent to which the overall performance can be improved via the second approach, where we extract and package the computation-intensive routines, and eventually optimize those routines specifically.

There exist approaches to build auto-tuners that can perform such optimizations automatically. Famous examples can be found in BLAS libraries, such as in the Automatically Tuned Linear Algebra Software (ATLAS) project [CPD01], where the automatic empirical optimization of software (AEOS) approach was employed to automatically tune the performance.

In order to improve the performance of our power system simulation kernels on GPU, and ideally across multiple platforms, the autotuning framework with the AEOS approach is employed. As proposed in [MZB24], the fundamental concept of the tool is to generate the entire numerical integration kernel for specific components, instead of implementing kernels that invokes BLAS libraries, with the objective of achieving better performance.

In this chapter, we analyze the techniques to tune our code for GPU execution, and introduce an automated optimization technique to improve the performance of our solver on different hardware.

The work in this chapter has been partially presented in [ZMB24] and [MZB24].

6.1 Bottlenecks

Taking the PinS formulation from Chapter 3, we apply the Roofline model [WWP09a] to identify the performance constraints with respect to different hardware. The model uses theoretical peak performance to visualize the "ceiling" of the computing hardware, and therefore evaluate the performance of the program execution on them based on the *operational intensity* I and achieved performance P. The operational intensity I is defined by

$$I = \frac{W}{Q} \quad \left[\frac{\text{FLOP}}{\text{byte}}\right], \tag{6.1}$$

where W represents the number of total floating-point operations performed, and Q the total bytes from read and write operations during the execution. Assuming explicit Fourth order Runge Kutta (RK4) method is used to integrate nonlinear components, and Euler Backword (EB) for the linear components, the Work W needed by each simulation step can be easily calculated by

$$W = W_{component} + W_{parallelization} + W_{network} \tag{6.2}$$

$$= \sum_{i}^{M} k_i W_{Rk4,i} + \sum_{j}^{N} k_j W_{EB,j} + W_{parallelization} + W_{network}, \tag{6.3}$$

where $W_{component}$ represents the computation cost of components, $W_{parallelization}$ additional overhead induced by parallel execution, and $W_{network}$ by solving the linear

Table 6.1: Operational intensity analysis for basic numerical operations with FP64

Operation	W	Q_r	Q_w
Complex multiplication	6	$4 \cdot 8$	$2 \cdot 8$
Complex add/sub	2	$4 \cdot 8$	$2 \cdot 8$
Sine[1]	≥ 10	8	8
Cosine[1]	≥ 10	8	8

Table 6.2: Operational intensity analysis for main simulation steps

Main simulation functions	internal stages	$W(n)$	$Q_r(n)$	$Q_w(n)$
Capacitor / Inductor	-	$\geq 16n_k$	$\geq 8 \cdot 6n_k \cdot 2$	$\geq 8 \cdot 6n_k \cdot 2$
Synchronous Machine	rotatingFrameTransform	$\geq 44n_k$	0	0
	evaluateDirevative	$\geq 518n_k$	0	0
	calculateOutput	$\geq 101n_k$	0	0
	memoryRW	-	$\geq 8 \cdot 180n_k$	$\geq 8 \cdot 17n_k$
Reduction (sparseBLAS)	-	$\approx 2 \cdot nnz \cdot 2$	$\approx 8 \cdot 3 \cdot nnz + 2 \cdot 8 \cdot n_c$	$\approx 2 \cdot 8 \cdot n_n$
Network	-	$8n_n^2 + 8n_n$	$\geq 2 \cdot 8 \cdot (n_n^2 + 2n_n)$	$\geq 2 \cdot 8 \cdot n_n$

algebraic equations of the network. Similarly, memory traffic Q can be represented by

$$Q_t = \sum_i^M k_i Q_{component} + Q_{parallelization} + Q_{network}, \tag{6.4}$$

where each $Q_{...}$ term includes read and write traffic

$$Q = Q_{read} + Q_{write}.$$

Tab. 6.1 calculates the W and Q for basic numerical operations with FP64, i. e. double-precision floating-point. The operational intensity of the overall simulation is then analyzed and listed in Tab. 6.2, where n_k denotes the number of corresponding component, nnz denotes the number of non-zeros in the incidence matrix, n_c the total number of dynamic branches, and finally n_n the number of total simulation nodes.

For simulating the components, actual W and Q will vary based on different numerical integration methods and implementations, as well as for the reduction step, actual W and Q also varies for different sparse matrix formats. Therefore, they are both difficult to estimate, and here we give only rough estimates. In addition, in sequential simulation the reduction step is not needed since there will be no race conditions.

[1] Implementation of basic nonlinear math functions such as trigonometric functions are highly system-specific, and could eventually be built-in in compiler, for instance in GNU GCC [GMA25]. Here is only a rough estimation based on its Taylor series.

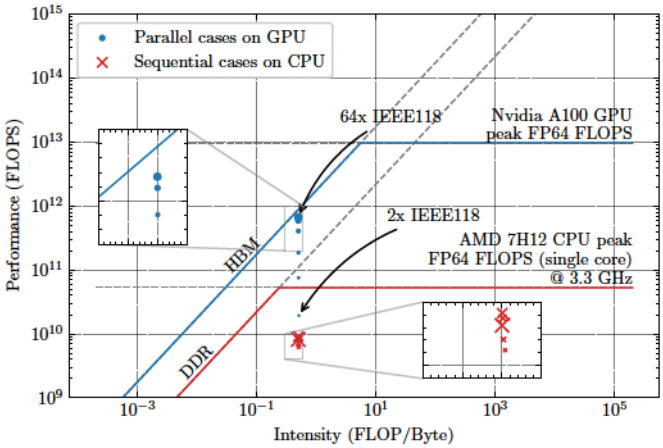


Figure 6.1: Roofline model of the parallel and sequential cases, larger marker represent larger network sizes. Figure reproduced from [ZMB24] under the CC BY 4.0 license.

The performance P is calculated by

$$P = \frac{W}{T} \quad \left[\frac{\text{FLOP}}{\text{second}}\right],$$

where T is the execution time of the program. Finally, the roofline model is built based on the theoretical analysis of numerical intensity and execution benchmarks as shown in Fig. 6.1. It can be noticed that the FLOPS achieved via parallel simulation on GPU already exceeds the theoretical peak FLOPS of the CPU with 4x IEEE118 cases. Moreover, the performance observed with our approach on the A100 GPU is converging to its memory bandwidth bound when simulating larger networks and achieving nearly 1 TFLOPS. This has demonstrated the performance and efficiency of our parallel simulation approach.

We can also notice from Fig. 6.1 that the operational intensity of the simulation indicates that our problem is memory-bounded. Because the large portion of the computation involved are linear algebraic operations, which is the outcome when we want to map computations on GPU efficiently. The memory-bounded characteristic suggests that if the same simulation algorithm is maintained, improving memory access efficiencies could lead to a substantial enhancement in performance. In the following sections, we introduce the approaches that were applied to improve the overall simulation performance.

6.2 Resource Contentions and Concurrency Model

In Sect. 3.2 we proposed to use incidence matrix to perform the *reduction* step. As introduced in Chapter 3, this is needed to solve the resource contention during concurrent execution of component computations. Traditional approaches usually rely on explicit synchronizations or linearization of operations. Early discussions can be traced back to the 60s where E. W. Dijkstra proposed an algorithm to solve resource contentions [Dij65], modern programming languages and parallel execution libraries also provides APIs for mutual exclusion lock and other synchronization techniques to resolve resource contentions. A typical lock-based approach is illustrated in Fig. 6.2.

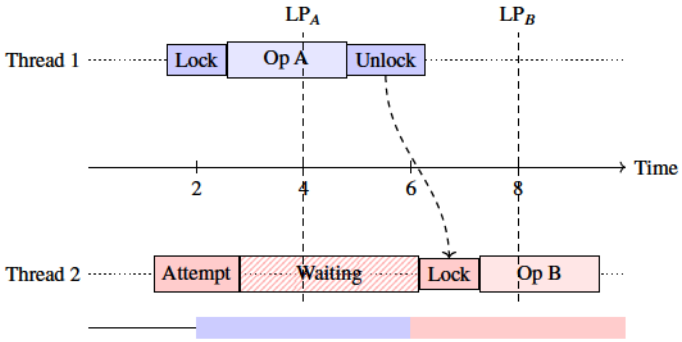


Figure 6.2: Traditional lock-based approach for resource contention. Synchronization timeline with critical sections (colored regions) and linearization points. The dashed arrow shows lock handoff.

In recent years, research has shifted to combine lock-free approaches with hardware support to achieve efficient handling of resource contentions, for instance the studies in transactional memories [HS12] as well as hardware support by vendors [Chr+10; Nak+15] which enables hardware-level realization of atomic operations to resolve resource contentions.

A Lock-Free Approach and Its Algebraic Concurrency Model

Using algebraic models to study the concurrency has also gained interests [WN95]. Besides solving resource contention, it is also applied to other practical domains, such as in [NT12], where an algebraic model for concurrency is used to routing of concurrent processes. In this section, we give here a more general formulation regarding the graph-based thread safety design introduced in Chapter 3.

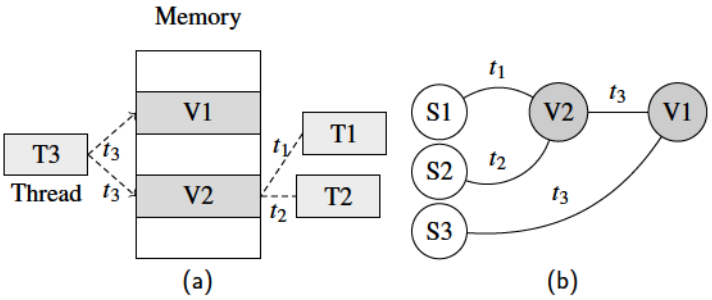


Figure 6.3: (a) Threads accessing memory concurrently; (b) Its representation via graph

Assuming n threads are accessing the elements in a vector with a random manner, i. e., different threads can access the same elements concurrently, as shown in Fig. 6.3.

As introduced in Sect. 3.2, the graph based approach treats the threads as edges and memory locations as vertices, the operation can be formulated as

$$V' = V + DT \tag{6.5}$$

where v is the original vector, D is the incidence matrix, T is the vector of data contained in all threads, and V' the vector holding new data after all modifications. The operation between V and DT is addition, meaning that every thread is performing an addition at the accessed location. Each row in v' is

$$v'_j = v_j + \sum d_i \cdot t_i, \quad d_i \in \{0, 1\} \tag{6.6}$$

corresponds to the operations performed by the threads at one address.

Since we try to generalize this approach, we now examine the cases for different operations in general. Consider a group $(S, \oplus)$ where $\oplus : V \times V \rightarrow V$ is a binary operation representing modifications performed by threads. The modification by threads on the data can be defined as

$$v'_j = v_j \oplus \bigoplus_{i=1}^{n} \mathcal{B}(t_i), \quad d_i \in \{0, 1\}, \tag{6.7}$$

where $\bigoplus$ represents a series of $\oplus$ operations: $(\mathcal{B}(t_1)) \oplus (\mathcal{B}(t_2)) \oplus \cdots \oplus (\mathcal{B}(t_n))$, and $\mathcal{B}(t_i)$ is a "state machine" defined as

$$\mathcal{B}(t_i) = \begin{cases} t_i & \text{if } i\text{-th thread contributes} \\ e & \text{otherwise} \end{cases} \tag{6.8}$$

where e is the identity element in $(S, \oplus)$. It selects the threads that are contributing to the j-th address. Alternatively, the state machine $\mathcal{B}(t_i)$ can be represented by

$$d_i \odot t_i = \begin{cases} t_i & \text{if } d_i = 1 \\ e & \text{if } d_i = 0 \end{cases} \tag{6.9}$$

To determine whether thread's contribution in the form of $\oplus$ operation can be transformed into the graph representation, the operators must satisfy algebraic properties to allow the system to be expressed as a linear combination in an additive group. There are following properties:

Property 1. $(S, \oplus)$ must form an Abelian group, ensuring commutativity, i. e.:

$$x \oplus y = y \oplus a \quad \forall x, y \in S, \tag{6.10}$$

Property 2. There exists a map $f : S \to A$ to an additive Abelian group $(A, +)$ satisfying:

$$f(x \oplus y) = f(x) + f(y) \quad \forall x, y \in S, \tag{6.11}$$

$$f(d \odot x) = d \cdot f(x) \quad \forall d \in \{0, 1\}. \tag{6.12}$$

We now explain these properties. First, $(S, \oplus)$ must form an Abelian group, because modifications performed by threads have to be commutative, otherwise concurrent modifications will lead to non-deterministic result.

The second property uses homomorphism f to ensure algebraic consistency between S and A. It translates the $\oplus$ operation in S to the additive operation $+$ in A, and the scalar operation $\odot$ in S to scalar multiplication in A. In this way, we can make sure that the $\oplus$ operation can be translated to satisfy the algebraic properties in graph representations where edge contributions are summed up at the adjacent vertices. It also ensures that the aggregated thread updates are mapped to a linear combination in A, enabling matrix-vector computations like $D \cdot f(T)$. Considering these properties, we can notice that the original formulation Eq. (6.7) can be transformed into

$$f\left(\bigoplus_{i=1}^{n} (d_i \odot T_i)\right) = \sum_{i=1}^{n} d_i \cdot f(T_i) \tag{6.13}$$

by applying its homomorphism map f.

We can now analyze the applicability of our approach for conventional operators:

1. Multiplication: When $\oplus$ is a multiplication, since multiplicative groups are Abelian groups, we only need to find a homomorphism to translate $(S, \times)$ into $(A, +)$. For this purpose, one widely used homomorphism is logarithm, which can be found in various applications such as homomorphism encryption [Aca+18]. Thread update is formulated as in the case of multiplication

$$v' = v \cdot \prod_{i=1}^{n} t_i^{d_i}. \tag{6.14}$$

Applying $f = \log$ we obtain

$$\begin{aligned} \log v' &= \log(v \cdot \prod_{i=1}^{n} t_i^{d_i}) \\ &= \log v + \sum_{i=1}^{n} d_i \log(t_i). \end{aligned} \tag{6.15}$$

Which naturally satisfies property 2 and 3. Therefore, when the threads are multiplying the data concurrently, we can apply log as homomorphism and apply our lock-free approach.

2. Division: When threads are performing divisions, it will be problematic. Because $(S, \div)$ is not an Abelian group since division is not commutative:

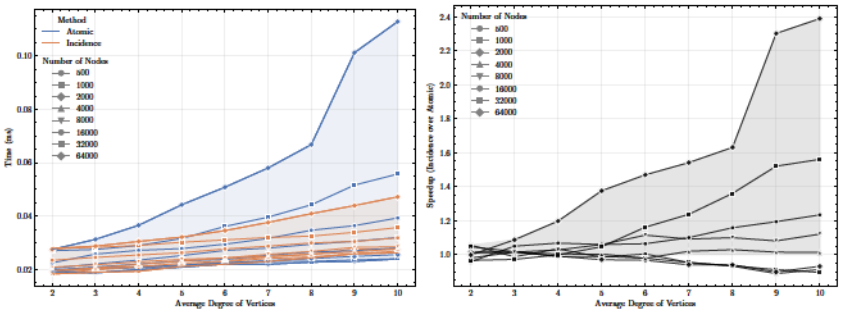
$$a \div b \neq b \div a. \tag{6.16}$$

Therefore, it does not satisfy property 1 and our approach cannot be applied. However, in a special case where all threads are simply performing division where their contributions are the denominators, then the operations is equivalent to multiplying a t_i^{-1}. Therefore, it can be treated as multiplication case and our approach can apply.

3. Subtraction: Similar to division, $(S, -)$ is also not an Abelian group due to lack of commutativity:

$$a - b \neq b - a. \tag{6.17}$$

And our approach could only be applied in the case where the subtraction can be treated as addition, i. e. contribution of all threads can be rewritten as $-t_i$ and added to the vector.



(a) (b)

Figure 6.4: Comparison of execution time for using atomic operation and incidence matrix with respect to different network size and different degrees

Benchmark

As analyzed before, the lock-free method should be more suitable when the network is more large and complex, but it is not yet clear with which size or complexity this method is more preferred over the `mutex` (mutual exclusion) or atomic operation [HW90]. Therefore, we benchmark the performance between the two methods with different graph sizes and complexity. Here, complexity is described using the average degree of vertices. The trick here to use the average degree is because of the degree sum formula, where the sum of degree in an undirected graph equals to $2E$, i. e., $\sum_{i=1}^{n} d_i = 2|E|$, therefore, the average degree d_{av} can be calculated by

$$d_{av} = \frac{2|E|}{|V|}, \tag{6.18}$$

as a result, we can easily determine the number of edges of a graph once d_{av} and $|V|$ are selected. This simplifies the effort to create our cases.

The benchmark is performed with AMD MI100 GPU, using HIP version `5.1.2053`, AMD `clang` version `14.0.0`. The result is shown in Fig. 6.4, we can notice that when the graph size and degree are relatively small, atomic operations are more efficient, and the lock-free approach is beneficial when the graph is getting larger and more complex. This fits our expectations, as the lock-free approach introduces new kernels, thus bringing additional kernel launch overhead.

Limitations and Remarks

The lock-free approach in general gives better scalability when the problem size is large and more complex operations. The goal is to transform the concurrent

operation into a linear algebra problem such that it can be parallelized conveniently and efficiently using conventional BLAS libraries. For smaller problem scales, the lock-based or atomic-operation-based solution can offer better performance, since the needed kernels are reduced. Moreover, recent advances in processor architecture have increasingly incorporated support for simple atomic operations at the instruction level [Nak+15], alleviating the overhead typically associated with these operations.

6.3 Automatic Optimization of Numerical Integration Routines

Take the DAE formulation of arbitrary power system component:

$$F(x, \dot{x}, u, t) = 0, \tag{6.19}$$

since there exists transformations [KM98] to convert that linear or nonlinear DAE into *strangeness-free* form:

$$\dot{x} = F'(x, u, t), \tag{6.20}$$

therefore, we can use a general representation for arbitrary nonlinear DAE of dynamic components as

$$\dot{x}(t) = A(t)x(t) + B(t)u(t) + \chi_{\dot{x}}(t, x(t), u(t)), \tag{6.21}$$

$$y(t) = C(t)x(t) + D(t)u(t) + \chi_y(t, x(t), u(t)), \tag{6.22}$$

with time-dependant input $u \in \mathbb{R}^{N_u}$, state $x \in \mathbb{R}^{N_x}$, output $y \in \mathbb{R}^{N_y}$, and potentially also time-dependent coefficient matrices $A \in \mathbb{R}^{N_x \times N_x}$, $B \in \mathbb{R}^{N_x \times N_u}$, $C \in \mathbb{R}^{N_y \times N_x}$, $D \in \mathbb{R}^{N_y \times N_u}$ formulated using a state-space formulation, leaving the nonlinearities $\chi_{\dot{x}}, \chi_y$ untouched. Assuming explicit integrator is used, the integration can be seen as a linear operator applied on Eq. (6.20), the explicit integration routine can be formulated as

$$x_{k+1} = x_k + \Gamma(\dot{x_k}). \tag{6.23}$$

obviously, the performance in evaluating $\Gamma(\dot{x_k})$ is critical to overall performance.

In practice, we can implement a function for evaluating the derivatives $\dot{x}$ which will be invoked by the integration routine during execution.

Overall Approach

The overall optimization process consists of a two-levels: the first level considers the possibility of vectorization inside component kernels and applies mixed matrix

Algorithm 1 Pseudocode for linear part of component kernels. Nonlinear part can be added as additional callbacks anywhere.

1: PreCallback$(t, x, y, u, ...args)$
2: $\dot{x} \leftarrow Ax + Bu$
3: DerivativeCallback$(\dot{x}, t, x, y, u, ...args)$
4: $x \leftarrow$ Integrate$(x, \dot{x}, t, step)$
5: NextStateCallback$(t, x, y, u, ...args)$
6: $y \leftarrow Cx + Du$
7: OutputCallback$(t, x, y, u, ...args)$

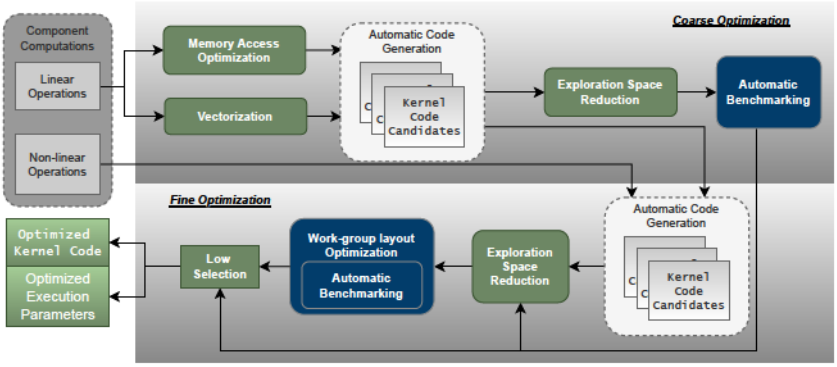


Figure 6.5: Illustration of the overall optimization process. Figure reproduced from [MZB24] under the CC BY 4.0 license.

formats with different matrix storage strategies to increase the computing throughput; the second level treats the combination of linear and nonlinear parts as a black box and searches for the optimal configuration of parameters such as the degree of vectorization, the combination of matrix formats, and the *group* sizes during parallel execution on the GPU.

To perform vectorization inside component kernels, we take the discretized ODE of the components and separate them into linear and nonlinear contributions. The computation of the model's linear part is based on matrix-vector multiplication (`mv`) operations. A first optimization level can already be achieved by applying vectorization on the `mv` operations. Moreover, since the `mv` operation is limited by memory transfer speed, applying sparse matrix formats boosts the performance when the associated matrices are sparse [BG08]. The nonlinear part includes any other operations that are not or difficult to be treated as linear algebraic. By combining the

two optimization levels, our implementation provides a framework that automatically benchmarks different parameters and selects the best-performing one. Furthermore, we also introduce an algorithm in Sect. 6.3 to reduce the exploration space during automatic benchmarking by constraining the possible parameters.

The nonlinear part includes any other operations that are not or difficult to be treated as linear algebraic. By combining the two optimization levels, our implementation provides a framework that automatically benchmarks different parameters and selects the best-performing one. Furthermore, we also introduce an algorithm in Sect. 6.3 to reduce the exploration space during automatic benchmarking by constraining the possible parameters.

Vectorization

Efficient vectorization requires a regular structure in computations to be efficient. A typical example of such a regular structure is linear algebra operations. In fact, GPUs are mostly designed and optimized for these operations, and automatic vectorization can be easily achieved by processing each dimension with a different thread.

Automatic vectorization is difficult to apply to power system components, as they mostly contain nonlinearities. Usually, after discretization, a large portion of the required computations can be formulated into linear algebra operations with a small nonlinear part remaining. Therefore, processing a vast part of the nonlinear component in parallel is possible, whilst a residual part is computed sequentially.

To perform vectorization inside component kernels, the ODE of the components are reformulated similarly to a state-space representation as

$$\dot{x}(t) = A(t)x(t) + B(t)u(t) + \chi_{\dot{x}}(t, x(t), u(t)) \tag{6.24}$$

$$y(t) = C(t)x(t) + D(t)u(t) + \chi_y(t, x(t), u(t)). \tag{6.25}$$

with time-dependant input $u \in \mathbb{R}^{N_u}$, state $x \in \mathbb{R}^{N_x}$, output $y \in \mathbb{R}^{N_y}$, component matrices $A \in \mathbb{R}^{N_x \times N_x}$, $B \in \mathbb{R}^{N_x \times N_u}$, $C \in \mathbb{R}^{N_y \times N_x}$, $D \in \mathbb{R}^{N_y \times N_u}$, and nonlinearities $\chi_{\dot{x}}$ and χ_y. Fig. 6.6 visualizes the interactions of the linear contributions in a block diagram. Effectively, this splits the component computation into linear and nonlinear contributions, where the linear contribution can be seen as a state space model.

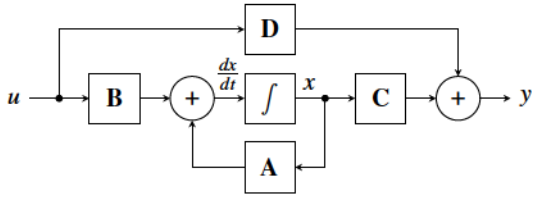


Figure 6.6: Block diagram of the model described by Eq. (6.24) and Eq. (6.25) when setting the nonlinearities $\chi_{\dot{x}} = \chi_y = 0$. Figure reproduced from [MZB24] under the CC BY 4.0 license.

Memory Access Optimization

The main computation of the linear part of components is matrix-vector multiplication which is largely limited by the bandwidth of the device [Gou+09]. By reducing the amount of data that has to be transferred, the overall computation can be accelerated. Moreover, reducing memory requirements allows storing more components in memory, thereby enabling the simulation of larger systems.

A multitude of different sparse formats have been explored in the past with various kernel implementations [SK12; DFM05; BG08; Gre+16; Nau+10]. All formats have in common that they try to minimize the number of stored zero elements. The main difference is the method used to store the positions of the remaining non-zero elements. Each format performs differently depending on the sparsity pattern of the matrix and the architecture of the GPU.

Of course, the nonzero structure of the component matrices can vary vastly, although they are for the same component. Thus, each matrix can have its own format in our implemented kernels. Moreover, we also convert matrices that are zero or the identity matrix to a scalar, so that they do not necessarily to be stored as matrices.

When considering matrix storage, except utilizing different sparse formats like ELLPACK format (ELL) [KOY89], Compressed sparse row (CSR) [LV15], Coordinate list (COO), etc., we proposed following strategies to store the matrices of multiple component instances. Nevertheless, new sparse formats and strategies can be easily integrated.

Block-Diagonal storage: The state space equations are decoupled; therefore, the equations for all component instances can be combined into one. This results in block diagonal coefficient matrices that can be efficiently stored using sparse formats. Each thread can then compute the required rows of the matrix-vector multiplication.

Concatenated storage: Not all formats can be stored efficiently in a block diagonal form. For example, storing a block diagonal matrix in a dense format is inefficient. Moreover, some other formats also perform better, or even require (e. g., COO), to store instance individually. In this case, the matrices for each instance are encoded into certain matrix format (dense or sparse format) and then concatenated into one buffer, an additional buffer is used to locate the individual instances.

Pattern storage: The nonzero pattern of the component matrices is usually determined by the algebraic description of the component's behavior, implying that it suffices to store this pattern once and reuse it for all instances of that component. By just storing the pattern, and allowing instances to have different values, this reduces memory transfers significantly as only the values of the component matrices are transferred, not the pattern. The downside of this approach is that the component must have a known and fixed nonzero pattern that can be determined during optimization and remains fixed in simulation, but it is usually the case.

Exploration space reduction and optimization

The linear part is vectorized, allowing multiple threads executing the same `mv` operation. Normally, one thread is assigned to process each row, however, when there are still resource available, i. e., if there are more threads available than the number of rows, then multiple threads process each row simultaneously. In this case, the element-wise products in each row are summed using a binary reduction.

To quantify the constraint for exploration space reduction, we first limit the number of threads per row to be a power of two. Then we introduce the number of rows per thread N_r and the number of threads per row during vectorized `mv` execution N_v, as shown in Fig. 6.7 as

$$N_r = \left\lceil \frac{N_m N_j}{N_g} \right\rceil,$$
$$N_v = 2^{\left\lfloor \log_2 \frac{N_g}{N_m N_j} \right\rfloor},$$

with N_m rows per matrix, a group size of N_g, and N_j components per group. Furthermore, the maximal number of threads per row $\hat{N}_v$ for a matrix with N_n columns is

$$\hat{N}_v = 2^{\lceil N_n \rceil}.$$

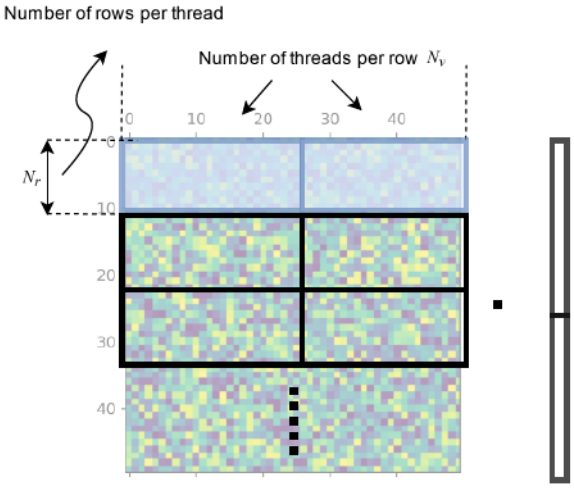


Figure 6.7: Illustration of two of the vectorization parameters N_v and N_r on the matrix-vector multiplication (mv). When more than one thread work on the same row, a parallel binary reduction is used to collect the sum for each row. Figure reproduced from [MZB24] under the CC BY 4.0 license.

The optimization for component kernels have six degrees of freedom. The matrix format choice for the component matrices introduces four parameters. In addition, a kernel should be built for a certain group size N_g on the GPU, i. e., the number of SIMT threads working in sync. Finally, the number of instances processed by one group N_j must be selected.

By design, the device limits the group size N_g to $1 \leq N_g \leq \hat{N}_g$ with the upper limit group size $\hat{N}_g$. Furthermore, the group size should be a multiple of two for efficient scheduling. For example, the NVIDIA A100 supports group sizes of up to 1024 threads, resulting in just 10 efficiently usable group sizes. Also, we assume that $1 \leq N_j \leq N_g$ – i.e., each thread computes at most one component – to stay within the regime of thin threads for better work distribution and less memory consumption [Kli+11].

Our goal is to minimize idle time among threads, as this can increase computation times. Thus, we maximize the number of components per group N_j so that the work performed per thread stays constant. Effectively, this should remove inefficient parameter sets. In practice, we loop over all N_j for a given group size N_g and only

benchmark those where $N_j + 1$ results in a different N_r or N_v.

With these prerequisites, we can define a function f

$$f : (N_m, N_n, \hat{N}_g) \rightarrow \{(N_g, N_j)\} \tag{6.26}$$

with N_m rows and N_n columns of the considered matrix that maps the external parameters given by the matrix dimensions and the GPU to a set of parameters fulfilling the above-mentioned constraints. As four matrices describe each component, the set of suitable parameters $\mathcal{P}$ is given by

$$\mathcal{P} = \bigcup_{x \in \{A,B,C,D\}} f(N_m^{(x)}, N_n^{(x)}, \hat{N}_g). \tag{6.27}$$

For the parameter selection, we split the optimization into a coarse optimization that benchmarks one part at a time and a fine-grained optimization to select the best combination of parameters as shown in Fig. 6.8. This two-step process significantly lowers the required optimization time.

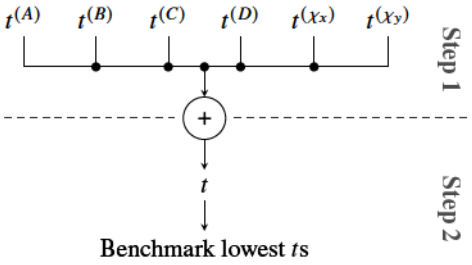


Figure 6.8: Visualization of the optimization flow. The first phase benchmarks each part isolated. The second part then combines the results to predict runtimes for combinations and benchmarks the ones with the lowest predicted runtimes. Figure reproduced from [MZB24] under the CC BY 4.0 license.

Coarse Optimization: For the coarse optimization, we benchmark each component of Alg. 1 individually to try out each format for each matrix. Assume F is the matrix format, and x identifies the matrix, then the runtime for the matrix-vector multiplication is $t_F^{(x)}$ and depends on the total number of instances N_i, as well as the group size N_g, and the number of components per group N_j. We determine this by running a benchmark, as runtime prediction is generally a hard problem [CSV10; GWC13; Lee+04]. The same benchmark is performed only with the nonlinearities to get $t^{(\chi_x)}$ and $t^{(\chi_y)}$.

Fine Optimization: In general, the optimization goal is to minimize the kernel's runtime. This can be estimated by assuming the runtime t of the combined kernel consists of the runtimes of the parts benchmarked in the coarse optimization as

$$t \equiv t^{(A)} + t^{(B)} + t^{(C)} + t^{(D)} + t^{(\chi_x)} + t^{(\chi_y)}. \tag{6.28}$$

Using this, we find the parameter configurations expected to perform best. Nevertheless, this is only an approximation, as some other influences from the combination may change the runtime slightly. Moreover, predicting runtimes on GPU is generally difficult, imprecise, or computationally expensive. Therefore, we benchmark some of the predicted configurations and select the one with the lowest runtime.

Performance Evaluation

We evaluate our approach using three representative components with different models – in particular, we consider a distributed generator (DG) inverter, an electrolyzer, and a synchronous machine. The time integration of Eq. (6.24) uses the explicit Euler method; however, an implementation for RK-4 also exists. The correctness of our kernels for all matrix formats, group sizes, and components per group was verified with negligible computation errors.

Simulations in this section are compiled and executed on a server with two AMD EPYC 7H12 CPUs (2.6 GHz base clock frequency, 64 cores each, hyper-threading disabled); 256 GB DDR4 main memory; one Nvidia A100-40 GB GPU with 40 GB HBM2 global memory, and one AMD MI100 GPU with 32 GB global memory.

To evaluate the performance of the optimized kernels, we prepare two alternative implementations to compare:

Library implementation: Previous works on power system simulation with GPU show that component computations can be transformed and aggregated into a unified kernel. Moreover, general purpose numerical solver libraries [Abh+18; Bal+21] will usually also aggregate all differential equations and solve simultaneously, e. g., by calculating the Jacobian, and computed with the help of a BLAS library. An example of this is the SUNDIALS library with GPU acceleration [Bal+21]. The benefit of using general solvers is that the user, or modeller, needs less concern regarding computational performance, since the computations are relying on other linear algebra libraries. However, this poses difficulty in implementing new models or power system specific models that are essentially an algorithm, e. g., specific controllers, automation systems, etc. Therefore, to represent the computation resulted from this approach, the linear part of the considered component models are

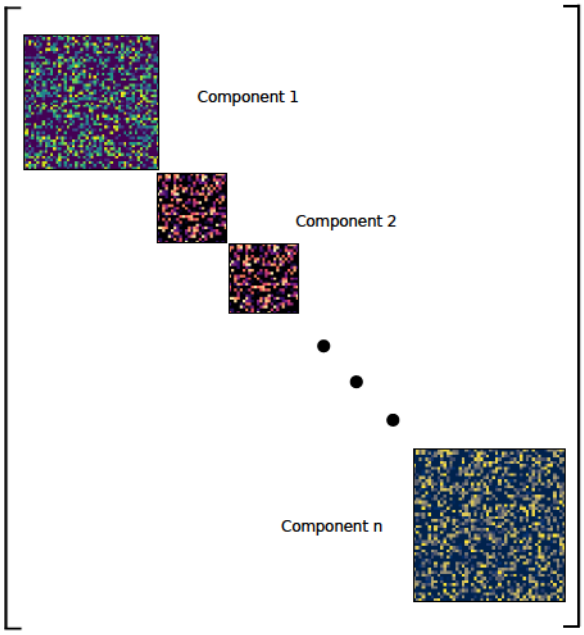


Figure 6.9: Illustration of the library implementation to represent computations in generic solvers, e. g. in [Bal+21]. Figure reproduced from [MZB24] under the CC BY 4.0 license.

aggregated into a single state-space representation, where the coefficient matrices of each component are placed along the diagonal of the aggregated coefficient matrix, as shown in Fig. 6.9. The numerical integration will then be processed by a vendor-supplied sparse BLAS library – so that the many off-diagonal zeros do not affect the computation – such as cuSparse or hipSparse. These libraries are highly optimized as they are utilized in many simulation environments. It needs to be pointed out that for simplicity, the nonlinear part of each model is ignored for this implementation.

In addition, to ensure a fair comparison, since the tested sparse libraries only support fixed sparse matrix format throughout the computation, prior to each benchmark, we tested among different matrix formats to find the most performant format for each library and use it in that benchmark.

Baseline implementation: We take the kernel implementations with the formulation in Sect. 6.3, i. e., the reformulation into a linear and nonlinear part, without applying any optimization. The group size and components per group are set to $N_g = 32$ and

$N_j = 32$, respectively. Such group size matches the recommended default group size on the considered devices, as it matches the SIMT width of one compute unit. This is a versatile configuration that should work well in many cases.

The kernels for the baseline implementation as well as the optimized code uses OpenCL C; library implementations only need to invoke the related API calls from the host, which is implemented in C++ in our case.

Benchmarked Component Models

Distributed generation inverter: We take the DG inverter introduced in [PPG07]; it is modeled by an averaged inverter model with a grid following control and neglects the switching dynamics. The kernel is also implemented in our previous work [Zha+22b]. Such model depth is already sufficient for a majority of test cases [De +22] involving power electronics. The controller and circuit representation of the inverter model is shown in Fig. 6.10. The input three-phase AC signal to the controllers is first transformed into the dq domain via a Park transform. The controller can be divided into three main parts: a phase-locked loop (PLL) tracks the system's angular frequency; an average power calculation block that calculates the current output power, and two PI controllers that track reference power set points by controlling the output voltage of the converter.

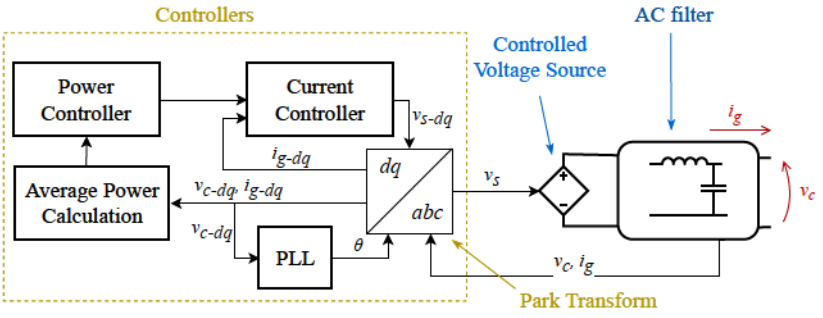


Figure 6.10: Distributed generation inverter model [PPG07; Zha+22b]. Figure reproduced from [MZB24] under the CC BY 4.0 license.

The converter controllers, excluding the Park transform block, can be formulated

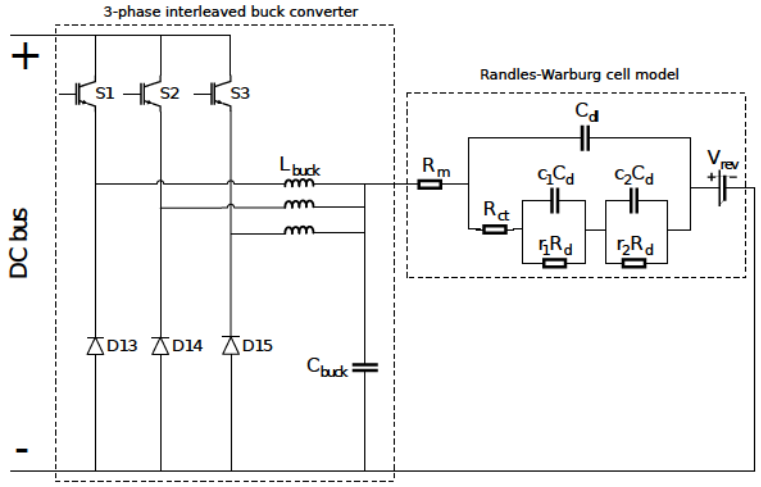


Figure 6.11: Electrolyzer with three-phase interleaved buck converter [Zha+22a; AE20]. Figure reproduced from [MZB24] under the CC BY 4.0 license.

via the following state-space formulation:

$$\dot{x} = Ax + B(x)u, \tag{6.29}$$

$$y = Cx + Du, \tag{6.30}$$

where the B matrix is dependent on the state x. This is due to the average power calculation block that performs multiplications over states (v_{c-dq} and i_{g-dq}) to calculate the power, thus introducing non-linearity in the state-space representation.

Electrolyzer: We use the electrolyzer model in [Zha+22a], where the electrolyzer is considered to be connected with a three-phase interleaved buck converter. The electrolyzer is modeled with Randles-Warburg (RW) cell model, and the buck converter uses the generalized state-space average model introduced in [AE20]. The reason for selecting an electrolyzer model for the benchmark is twofold: first, electrolyzers are gaining more attention due to the increasing interest in hydrogen; second, the main computational load in this model we selected is the interleaved buck converter, therefore, it could be used to partly represent the computational tasks when simulating a power-electronics-based system with switching dynamics.

Synchronous Machine: We take the machine model in [KBL94] with saturation ignored, and its dq0-axis equivalent circuits are shown in Fig. 2.3. Similar to the previous inverter model, the machine is modeled in the dq frame as well. The

machine model also introduces non-linearity due to the coupling of $d-$ and $q-$axis and between electromagnetic and mechanical equations. The overall equation set can be represented by:

$$\dot{\Psi} = f(\Psi, \omega_r, U), \tag{6.31}$$

$$\dot{\delta}_r = f(\omega_r), \tag{6.32}$$

$$\dot{\omega}_r = f(\Psi, I, \omega_r), \tag{6.33}$$

$$0 = g(\Psi, I). \tag{6.34}$$

Where Ψ is a 7×1 vector of flux linkage; U, I are vectors of stator and rotor voltages and currents with the same length, respectively. ω_r is the mechanical angular frequency of the rotor.

We benchmarked our optimized component kernels against the library and the baseline implementation by performing numerical integration with a simple Euler forward method. The measured execution time for different component types and different component counts are shown in Fig. 6.12, including a benchmark of simulating a combination with three types of components together with an equal number of each type. The speedup of our optimized kernels to the library implementation is between 1.3 and 6.7 times and to the baselines by up to 10.2 times, which shows that the optimized kernels outperform the compared implementations by some margin.

Compared with the library implementation, the customized component kernels, i. e. the baseline and optimized, have a reduced number of kernel launches, as our kernels perform the whole computation in a single kernel launch although with part of computations being computed sequentially, i. e. nonlinear contributions, whereas the library implementation requires separated launches e. g. for updating the states and outputs. This results in better cache coherence and less overhead in our implementations. Nevertheless, the library implementation still outperforms the baselines when the problem size is large enough.

In all test cases of the library implementation, the NVIDIA A100 outperformed the AMD MI100. It can be attributed to the better optimization with the cuSparse library than the hipSparse library. However, it needs to be noted that the A100 GPU has higher memory bandwidth, therefore, it gains more advantage in the linear algebra related benchmarks which are mainly memory-bounded operations.

Finally, we performed a roofline analysis [WWP09b] with the optimized kernels. This relates the computational intensity, given by the FLOP per byte transferred,

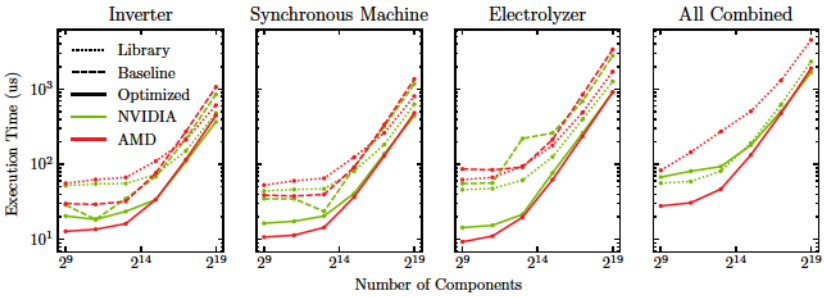


Figure 6.12: Execution time for performing numerical integration with different component models and component counts, with simulating each type of component along or with all types combined. Figure reproduced from [MZB24] under the CC BY 4.0 license.

to the achievable performance. The plot is shown in Fig. 6.13 and shows that especially high component counts lead to efficient utilization of the hardware and close to the peak performance. The devices are likely not fully utilized for lower component counts. This is because runtime overheads, such as scheduling time, become a significant factor compared to light computational load, leading to a large gap between peak and actual performance.

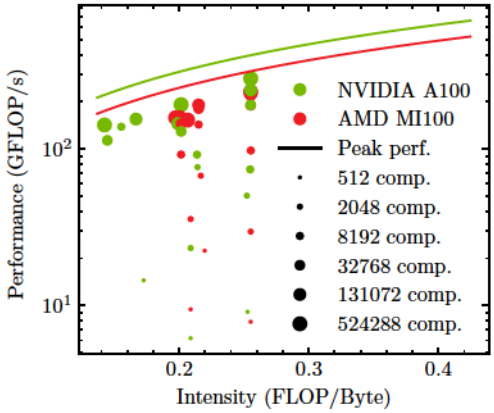


Figure 6.13: Performance of the optimized kernels in a roofline plot to relate kernel performance to peak performance. The size of the dots relates to the number of components. Figure reproduced from [MZB24] under the CC BY 4.0 license.

Memory Consumption

Reformulation of the components inevitably leads to increased memory consumption, which could be a limiting factor for very large systems. We tracked the required buffer sizes needed for each component to perform numerical integration and compared them between the optimized and library implementation, as shown in Fig. 6.14. The electrolyzer model consumes the most memory per component since it has a more detailed converter model considering switching dynamics; the DG inverter considers only controller dynamics and therefore needs less memory but is still larger than the synchronous machine. Results show that our optimizer finds different matrix format combinations for different problem sizes: when the component count is small, the GPU memory bandwidth is usually not saturated, hence leading our optimization to choose more performant formats, including dense, regardless of memory usage, and the increasing memory usage clearly shows that dense format was used in some cases. With the growing component count, the GPU memory bandwidth is eventually saturated, therefore, the optimizer tends to find format combinations that provide better compression on the matrices.

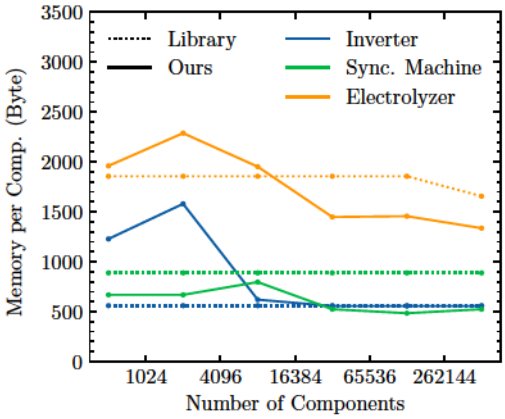


Figure 6.14: Memory consumption per component on the NVIDIA A100 GPU of different component models, for our and the library implementation. The lower limit is calculated by considering the minimum number of parameters required for each component. Figure reproduced from [MZB24] under the CC BY 4.0 license.

6.4 Summary

In this section, we analyzed the potential performance bottleneck of the parallel-in-space approach and proposed methods to further optimize its performance. Moreover, a more general formulation with an algebraic concurrency model is given to the graph-based thread safety design introduced previously in Chapter 3.

This work provides an approach to automatically accelerate the numeric integration of component models for power system simulations on GPU. Our approach automatically exploits the data parallelism of the GPU by introducing vectorization and different matrix storage strategy with mixed matrix formats into the component computation. The approach can be flexibly applied to kernels for any new components, or be applied to existing compatible implementations to improve performance.

We demonstrated that our approach outperforms the aggregated model approach based on sparse linear algebra libraries with a speedup between 1.3 and 6.7 times, and up to 10 times compared to the unoptimized baseline implementation, demonstrating the effectiveness of our optimizations.

With the growing size and complexity of the power system, or when specific study cases require detail modelling, component models can be more complex. Therefore, memory bandwidth and, in some cases, memory volume will become a limiting factor. The memory footprint shown in Sect. 6.3 suggests that our approach could increase the efficiency in memory utilization for component computations.

Nevertheless, it needs to be pointed out that the optimization procedure takes a few minutes per component since it depends on many automatically executed benchmarks, and need to be re-executed for different problem sizes, indicating further improvements needed, e. g., on the optimization algorithm.

We plan to consider different explicit and implicit integration schemes other than the explicit Euler and RK-4 schemes. Moreover, one may consider the vectorization potential of the nonlinearities to achieve even better performance. Including more specialized matrix formats for the component matrices may increase performance further.

Chapter 7

CONCLUSIONS

This thesis presents new methods to apply parallel-in-space to accelerate power system simulations. We show that using GPU with our PinS method could achieve orders of magnitude speedup compared to a highly efficient simulator based on C++, and the combination of PinT and PinS opens up further performance potential after the spatial parallelization is saturated. The main findings of this dissertation are summarized as follows:

1. *Can we map power system simulation onto hardware accelerators efficiently?* In this work, we exploited parallelisms at different phases and different levels during the simulation, such that the simulation could be executed efficiently on the GPU. The performance achieved on GPU has also demonstrated real-time and faster-than-real-time capability for large test cases.

2. *Will GPU outperform other hardware in terms of power system simulation?* With the analyses based on the roofline model presented in Chap. 6, and results shown in Chap. 3, we show that with the state-of-the-art commercial hardware, it should be theoretically impossible for a multi-core CPU to achieve the same performance using the same parallel-in-space algorithm. However, we need to point out that the advantage of GPU relies not only on the large number of processing elements, but also heavily on its high memory bandwidth. Current trend in high performance computing architecture also sees the merging of different processor, for instance CPU and GPU into single integrated chip [Tek+21], and there are already commercial processors that closely combining both, for instance the NVIDIA GH superchip [NVI2424].

3. In Chap. 4 we proposed the combined space-time parallel approached based on the PinS algorithm in Chap. 3 and the parallel-in-time method MGRIT. Result shows additional speedup by applying a non-intrusive MGRIT implementation XBraid and execute with the PinS implementation heterogeneously, i. e. executed both on CPU and GPU.

4. In addition to the research in parallel simulation methods for power system simulation, this work also investigated on the realization of such algorithms

with a suitable simulator design. In Chap. 5, we demonstrated the implementation of our parallel-in-space simulator using data-oriented design. The simulator shows flexibility in integrating different parallelization methods as well as different execution hardware. However, it is difficult to compare directly the performance of two design patterns, and such comparison would be beyond the scope of this work. Nevertheless, as the hardware accelerators like GPU is becoming more dominant in computing hardware, we believe that such design would be more popular in the future simulator implementations.

5. The performance of simulations executed on the hardware accelerators like GPU can be tuned automatically with our approach proposed in Chap. 6. The approach automatically tunes the kernels that are going to be executed on the GPU with the help of AEOS. The tuner demonstrated effective performance improvements on component kernels compared to implementation based on standard GPU libraries.

Chapter 8

OUTLOOK

In this work, the hardware accelerator employed for the parallel-in-space approach was GPU. The rising of field programmable gate arrays (FPGA) processors is also attracting interests and there are recent literature investigating these reconfigurable devices. The benefit of FPGA is that it can reconfigure hardware circuits on chip to realize the computations programmed, therefore potentially achieve low latency with massively parallel execution. In our paper [Zha+22b], we already demonstrated that the implemented kernels can also be synthesized on the FPGAs, however, the result is not comparable to GPUs as these kernels did not utilize the device efficiently. More research could be conducted in synthesizing high-performance kernels or even the entire simulation engines so that it can be executed on FPGAs efficiently. As we see the trend where CPUs and GPUs are integrating, future trend also sees the FPGA chips being integrated into other processors such as CPU [PAT25]. Therefore, algorithms that utilize different heterogenous architecture could be further investigated, especially on the emerging processors such as CPU with integrated FPGA, super-chip based on CPU and GPU.

Moreover, on the methodology perspective, the PinS method relies on explicit integration of nonlinear components. More investigation could be conducted on applying explicit stiff-stable methods to minimize divergence caused by nonlinear components. The possibility of extracting parallelism under fully implicit schemes, for instance with the SDC method [Spe18], can also be studied.

Combining parallel-in-space with parallel-in-time needs specific implementation of parallel-in-time. The approach introduced in Chap. 4 employs heterogeneous computing design. Due to the large overhead, it is only suitable for large cases with long simulation duration. Future work could also be focused on implementing PinT algorithm that can be executed with PinS on the same hardware natively. Moreover, more investigation could be conducted on other space-time parallel algorithm such as PFASST and perform comparisons with their performances.

In Chap. 6, we proposed automatic optimization of explicit numerical integration routines. In the future, it is worth studying the effect on implicit or hybrid numerical routines. Moreover, we could further benchmark the approach for other physical

systems, for instance applying it in a general purpose ODE solver.

Beyond researches in accelerating a single simulation, we could also investigate methods to simulate multiple scenarios efficiently. For instance, extending the simulation framework for many-scenario simulations proposed in [Zha+21]. Future works could see potentials in the interaction with power system components that causes discrete events, for instance protection systems, as well as system behaviors during interaction with these systems, for example under voltage load shedding (UVLS) and under frequency load shedding (UFLS).

BIBLIOGRAPHY

[15] "Paris Agreement". In: *Report of the Conference of the Parties to the United Nations Framework Convention on Climate Change (21st Session, 2015: Paris). Retrived December*. Vol. 4. Paris, 2015, p. 2.

[23] *The Khronos Group*. The Khronos Group, Mar. 8, 2023. https://www.khronos.org/adopters/conformant-products/opencl (visited on 03/08/2023).

[25] *Pybind/Pybind11*. pybind. Feb. 2025. (Visited on 02/18/2025).

[92] "Parallel Processing in Power Systems Computation". In: *IEEE Transactions on Power Systems* 7.2 (May 1992), pp. 629–638. ISSN: 1558-0679. DOI: 10.1109/59.141768. (Visited on 01/27/2025).

[Abh+18] Shrirang Abhyankar, Jed Brown, Emil M. Constantinescu, et al. *PETSc/TS: A Modern Scalable ODE/DAE Solver Library*. June 4, 2018. arXiv: 1806.01437 [math]. (Visited on 03/13/2023). preprint.

[Aca+18] Abbas Acar, Hidayet Aksu, A. Selcuk Uluagac, et al. "A Survey on Homomorphic Encryption Schemes: Theory and Implementation". In: *ACM Comput. Surv.* 51.4 (July 2018), 79:1–79:35. ISSN: 0360-0300. DOI: 10.1145/3214303. (Visited on 02/23/2025).

[AE20] Peter Azer and Ali Emadi. "Generalized State Space Average Model for Multi-Phase Interleaved Buck, Boost and Buck-Boost DC-DC Converters: Transient, Steady-State and Switching Dynamics". In: *IEEE Access* 8 (2020), pp. 77735–77745. ISSN: 2169-3536.

[AK23] Felipe Arraño-Vargas and Georgios Konstantinou. "Modular Design and Real-Time Simulators Toward Power System Digital Twins Implementation". In: *IEEE Transactions on Industrial Informatics* 19.1 (Jan. 2023), pp. 52–61. ISSN: 1941-0050. DOI: 10.1109/TII.2022.3178713. (Visited on 03/06/2025).

[Ari15] Petros Aristidou. "Time-Domain Simulation of Large Electric Power Systems Using Domain-Decomposition and Parallel Processing Methods". PhD thesis. Université de Liège, June 2015. DOI: 10.13140/RG.2.1.1119.1527.

[Arm+06] M. Armstrong, J.R. Marti, L.R. Linares, et al. "Multilevel MATE for Efficient Simultaneous Solution of Control Systems and Nonlinearities in the OVNI Simulator". In: *IEEE Transactions on Power Systems* 21.3 (Aug. 2006), pp. 1250–1259. ISSN: 1558-0679. DOI: 10.1109/TPWRS.2006.879254.

[Aro06] Peter Aronsson. "Automatic Parallelization of Equation-Based Simulation Programs". PhD thesis. Linköping University, 2006. (Visited on 01/06/2025).

[Atk89] Kendall E. Atkinson. *An Introduction to Numerical Analysis*. 2nd ed. New York: Wiley, 1989. ISBN: 978-0-471-62489-9.

[Bal+21] Cody J. Balos, David J. Gardner, Carol S. Woodward, et al. "Enabling GPU Accelerated Computing in the SUNDIALS Time Integration Library". In: *Parallel Computing* 108 (Dec. 2021), p. 102836. ISSN: 0167-8191. DOI: 10.1016/j.parco.2021.102836. (Visited on 02/21/2024).

[BG08] Nathan Bell and Michael Garland. *Efficient sparse matrix-vector multiplication on CUDA*. Tech. rep. Nvidia Technical Report NVR-2008-004, Nvidia Corporation, 2008.

[BKK17] Larisa Beilina, Evgenii Karchevskii, and Mikhail Karchevskii. *Numerical Linear Algebra: Theory and Applications*. Cham: Springer International Publishing, 2017. ISBN: 978-3-319-57302-1 978-3-319-57304-5. DOI: 10.1007/978-3-319-57304-5. (Visited on 02/05/2025).

[BM15] Andrea Benigni and Antonello Monti. "A Parallel Approach to Real-Time Simulation of Power Electronics Systems". In: *IEEE Transactions on Power Electronics* 30.9 (2015), pp. 5192–5206. DOI: 10.1109/TPEL.2014.2361868.

[Bra77] Vladimir Brandwajn. "Synchronous Generator Models for the Simulation of Electromagnetic Transients". PhD thesis. University of British Columbia, 1977.

[Cas24] Stephen Cass. *Top Programming Languages 2024 - IEEE Spectrum*. https://spectrum.ieee.org/top-programming-languages-2024. Aug. 2024. (Visited on 02/18/2025).

[CB93] J.S. Chai and A. Bose. "Bottlenecks in Parallel Algorithms for Power System Stability Analysis". In: *IEEE Transactions on Power Systems* 8.1 (Feb. 1993), pp. 9–15. ISSN: 1558-0679. DOI: 10.1109/59.221242. (Visited on 01/27/2025).

[CDD23] Tianshi Cheng, Tong Duan, and Venkata Dinavahi. "ECS-Grid: Data-Oriented Real-Time Simulation Platform for Cyber-Physical Power Systems". In: *IEEE Transactions on Industrial Informatics* 19.11 (Nov. 2023), pp. 11128–11138. ISSN: 1941-0050. DOI: 10.1109/TII.2023.3244329. (Visited on 12/09/2024).

[Chr+10] Dave Christie, Jae-Woong Chung, Stephan Diestelhorst, et al. "Evaluation of AMD's Advanced Synchronization Facility within a Complete Transactional Memory Stack". In: *Proceedings of the 5th*

European Conference on Computer Systems. EuroSys '10. New York, NY, USA: Association for Computing Machinery, Apr. 2010, pp. 27–40. ISBN: 978-1-60558-577-2. DOI: 10.1145/1755913.1755918. (Visited on 03/08/2025).

[CK06] François E. Cellier and Ernesto Kofman. *Continuous System Simulation*. New York: Springer, 2006. ISBN: 978-0-387-26102-7 978-0-387-30260-7.

[CLCUDA] *CNugteren/CLCudaAPI at 9.0*. https://github.com/CNugteren/CLCudaAPI. (Visited on 03/08/2023).

[CLD20] Shiqi Cao, Ning Lin, and Venkata Dinavahi. "Faster-Than-Real-Time Dynamic Simulation of AC/DC Grids on Reconfigurable Hardware". In: *IEEE Transactions on Power Systems* 35.2 (Mar. 2020), pp. 1539–1548. ISSN: 1558-0679. DOI: 10.1109/TPWRS.2019.2944920.

[CLD22] Tianshi Cheng, Ning Lin, and Venkata Dinavahi. "Hybrid Parallel-in-Time-and-Space Transient Stability Simulation of Large-Scale AC/DC Grids". In: *IEEE Transactions on Power Systems* (2022), pp. 1–1. ISSN: 0885-8950, 1558-0679. DOI: 10.1109/TPWRS.2022.3153450.

[CLD23] Shiqi Cao, Ning Lin, and Venkata Dinavahi. "Faster-Than-Real-Time Hardware Emulation of Extensive Contingencies for Dynamic Security Analysis of Large-Scale Integrated AC/DC Grid". In: *IEEE Transactions on Power Systems* 38.1 (Jan. 2023), pp. 861–871. ISSN: 1558-0679. DOI: 10.1109/TPWRS.2022.3161561.

[CPD01] R. Clint Whaley, Antoine Petitet, and Jack J. Dongarra. "Automated Empirical Optimizations of Software and the ATLAS Project". In: *Parallel Computing*. New Trends in High Performance Computing 27.1 (Jan. 2001), pp. 3–35. ISSN: 0167-8191. DOI: 10.1016/S0167-8191(00)00087-9.

[CPP18] *CppCon 2018: OOP Is Dead, Long Live Data-oriented Design– Stoyan Nikolov : Standard C++*. https://isocpp.org/blog/2019/08/cppcon-2018-oop-is-dead-long-live-data-oriented-design-stoyan-nikolov. (Visited on 02/16/2025).

[CSV10] Jee W Choi, Amik Singh, and Richard W Vuduc. "Model-driven autotuning of sparse matrix-vector multiply on GPUs". In: *ACM sigplan notices* 45.5 (2010), pp. 115–126.

[CUDAT18] *NVIDIA CUDA Toolkit Release Notes*. Oct. 30, 2018. https://docs.nvidia.com/cuda/archive/10.0/cuda-toolkit-release-notes/index.html#deprecated-features (visited on 02/20/2024).

[De +22] Giovanni De Carne, Georg Lauss, Mazheruddin H Syed, et al. "On modeling depths of power electronic circuits for real-time simulation–a comparative analysis for power systems". In: *IEEE Open Access Journal of Power and Energy* 9 (2022), pp. 76–87.

[De +23] H. De Sterck, R. D. Falgout, O. A. Krzysik, et al. "Efficient Multigrid Reduction-in-Time for Method-of-Lines Discretizations of Linear Advection". In: *Journal of Scientific Computing* 96.1 (May 2023), p. 1. ISSN: 1573-7691. DOI: 10.1007/s10915-023-02223-4. (Visited on 02/07/2025).

[DFM05] Eduardo F D'Azevedo, Mark R Fahey, and Richard T Mills. "Vectorized sparse matrix multiply for compressed row storage format". In: *International Conference on Computational Science*. Springer. 2005, pp. 99–106.

[DFP20] Asja Derviskadic, Guglielmo Frigo, and Mario Paolone. "Beyond Phasors: Modeling of Power System Signals Using the Hilbert Transform". In: *IEEE Transactions on Power Systems* 35.4 (July 2020), pp. 2971–2980. ISSN: 0885-8950, 1558-0679. DOI: 10.1109/TPWRS.2019.2958487. (Visited on 02/20/2024).

[Dij65] E. W. Dijkstra. "Solution of a Problem in Concurrent Programming Control". In: *Commun. ACM* 8.9 (Sept. 1965), p. 569. ISSN: 0001-0782. DOI: 10.1145/365559.365617. (Visited on 02/20/2025).

[DMB11] Christian Dufour, Jean Mahseredjian, and Jean Bélanger. "A Combined State-Space Nodal Method for the Simulation of Power System Transients". In: *IEEE Transactions on Power Delivery* 26.2 (Apr. 2011), pp. 928–935. ISSN: 08858977. DOI: 10.1109/TPWRD.2010.2090364. (Visited on 05/17/2019).

[Dom69] Hermann W. Dommel. "Digital Computer Solution of Electromagnetic Transients in Single-and Multiphase Networks". In: *IEEE Transactions on Power Apparatus and Systems* PAS-88.4 (Apr. 1969), pp. 388–399. ISSN: 0018-9510. DOI: 10.1109/TPAS.1969.292459. (Visited on 12/08/2024).

[EM12] Matthew Emmett and Michael Minion. "Toward an Efficient Parallel in Time Method for Partial Differential Equations". In: *Communications in Applied Mathematics and Computational Science* 7.1 (Jan. 2012), pp. 105–132. ISSN: 1559-3940, 2157-5452. DOI: 10.2140/camcos.2012.7.105. https://projecteuclid.org/journals/communications-in-applied-mathematics-and-computational-science/volume-7/issue-1/Toward-an-efficient-parallel-in-time-method-for-partial-differential/10.2140/camcos.2012.7.105.full (visited on 11/19/2024).

[Fal+14] R. D. Falgout, S. Friedhoff, Tz. V. Kolev, et al. "Parallel Time Integration with Multigrid". In: *SIAM Journal on Scientific Computing* 36.6 (Jan. 2014), pp. C635–C661. ISSN: 1064-8275. DOI: 10.1137/130944230.

[FLW19] Robert D. Falgout, Matthieu Lecouvez, and Carol S. Woodward. "A Parallel-in-Time Algorithm for Variable Step Multistep Methods". In: *Journal of Computational Science* 37 (Oct. 2019), p. 101029. ISSN: 18777503. DOI: 10.1016/j.jocs.2019.101029. https://linkinghub.elsevier.com/retrieve/pii/S1877750319304351 (visited on 11/21/2024).

[FO05] B. Franke and M.F.P. O'Boyle. "A Complete Compiler Approach to Auto-Parallelizing C Programs for Multi-DSP Systems". In: *IEEE Transactions on Parallel and Distributed Systems* 16.3 (Mar. 2005), pp. 234–245. ISSN: 1558-2183. DOI: 10.1109/TPDS.2005.26.

[FS21] Stephanie Friedhoff and Ben S. Southworth. "On "Optimal" h-Independent Convergence of Parareal and Multigrid-Reduction-in-Time Using Runge-Kutta Time Integration". In: *Numerical Linear Algebra with Applications* 28.3 (2021), e2301. ISSN: 1099-1506. DOI: 10.1002/nla.2301. (Visited on 11/27/2024).

[Geb19] Mahder Gebremedhin. *Automatic and Explicit Parallelization Approaches for Equation Based Mathematical Modeling and Simulation*. Vol. 1967. Linköping Studies in Science and Technology. Dissertations. Linköping: Linköping University Electronic Press, Jan. 2019. ISBN: 978-91-7685-163-0. DOI: 10.3384/diss.diva-152789. (Visited on 09/16/2022).

[GG23] Yunjie Gu and Timothy C. Green. "Power System Stability With a High Penetration of Inverter-Based Resources". In: *Proceedings of the IEEE* 111.7 (July 2023), pp. 832–853. ISSN: 1558-2256. DOI: 10.1109/JPROC.2022.3179826. (Visited on 03/06/2025).

[GJ+10] Gaël Guennebaud, Benoît Jacob, et al. *Eigen v3*. http://eigen.tuxfamily.org. 2010.

[GMA25] *Gcc/Gcc/M2/Gm2-Libs/MathLib0.Mod at Master · Gcc-Mirror/Gcc*. https://github.com/gcc-mirror/gcc/blob/master/gcc/m2/gm2-libs/MathLib0.mod. (Visited on 02/19/2025).

[GNUGCC] *Auto-Vectorization in GCC - GNU Project*. https://gcc.gnu.org/projects/tree-ssa/vectorization.html. (Visited on 03/07/2023).

[Göt+21] Sebastian Götschel, Michael Minion, Daniel Ruprecht, et al. "Twelve Ways to Fool the Masses When Giving Parallel-in-Time Results". In: *Parallel-in-Time Integration Methods*. Ed. by Benjamin Ong,

Jacob Schroder, Jemma Shipton, et al. Cham: Springer International Publishing, 2021, pp. 81–94. ISBN: 978-3-030-75933-9. DOI: 10.1007/978-3-030-75933-9_4.

[Gou+09] Georgios I. Goumas, Kornilios Kourtis, Nikos Anastopoulos, et al. "Performance evaluation of the sparse matrix-vector multiplication on modern architectures". In: *The Journal of Supercomputing* 50 (2009), pp. 36–77.

[Gre+16] Joseph L Greathouse, Kent Knox, Jakub Poła, et al. "clsparse: A vendor-optimized open-source sparse blas library". In: *Proceedings of the 4th International Workshop on OpenCL*. 2016, pp. 1–4.

[Gro23] Khronos® OpenCL Working Group. *The OpenCL™ Specification*. https://registry.khronos.org/OpenCL/specs/3.0-unified/pdf/OpenCL_API.pdf. Apr. 2023.

[Gun+20] Stefanie Gunther, Robert D. Falgout, Philip Top, et al. "Parallel-in-Time Solution of Power Systems with Unscheduled Events". In: *2020 IEEE Power & Energy Society General Meeting (PESGM)*. Montreal, QC, Canada: IEEE, Aug. 2020, pp. 1–5. ISBN: 978-1-72815-508-1. DOI: 10.1109/PESGM41954.2020.9281595. (Visited on 04/18/2024).

[Gur+16] Gurunath Gurrala, Aleksandar Dimitrovski, Sreekanth Pannala, et al. "Parareal in Time for Fast Power System Dynamic Simulations". In: *IEEE Transactions on Power Systems* 31.3 (May 2016), pp. 1820–1830. ISSN: 1558-0679. DOI: 10.1109/TPWRS.2015.2434833.

[GV13] Gene H. Golub and Charles F. Van Loan. *Matrix Computations*. Fourth edition. Johns Hopkins Studies in the Mathematical Sciences. Baltimore: The Johns Hopkins University Press, 2013. ISBN: 978-1-4214-0794-4.

[GWC13] Ping Guo, Liqiang Wang, and Po Chen. "A performance modeling and optimization analysis tool for sparse matrix-vector multiplication on GPUs". In: *IEEE Transactions on Parallel and Distributed Systems* 25.5 (2013), pp. 1112–1123.

[Hap74] H.H. Happ. "Diakoptics—The Solution of System Problems by Tearing". In: *Proceedings of the IEEE* 62.7 (July 1974), pp. 930–940. ISSN: 1558-2256. DOI: 10.1109/PROC.1974.9545.

[Har+25] Carsten Hartmann, Junjie Zhang, Carlos D. Gonzalez Calaza, et al. "Quantum Annealing Based Power Grid Partitioning for Parallel Simulation". In: *IEEE Transactions on Power Systems* (2025), pp. 1–13. ISSN: 1558-0679. DOI: 10.1109/TPWRS.2025.3578243. (Visited on 06/13/2025).

[HPP09] Mary Hall, David Padua, and Keshav Pingali. "Compiler Research: The next 50 Years". In: *Commun. ACM* 52.2 (Feb. 2009), pp. 60–67. ISSN: 0001-0782. DOI: 10.1145/1461928.1461946.

[HS12] Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming, Revised Reprint*. 1st. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., May 2012. ISBN: 978-0-12-397337-5.

[Hua+18] Renke Huang, Shuangshuang Jin, Yousu Chen, et al. "Faster than Real-Time Dynamic Simulation for Large-Size Power System with Detailed Dynamic Models Using High-Performance Computing Platform". In: *IEEE Power and Energy Society General Meeting* 2018-Janua (Jan. 29, 2018), pp. 1–5. DOI: 10.1109/PESGM.2017.8274505.

[HW90] Maurice P. Herlihy and Jeannette M. Wing. "Linearizability: A Correctness Condition for Concurrent Objects". In: *ACM Trans. Program. Lang. Syst.* 12.3 (July 1990), pp. 463–492. ISSN: 0164-0925. DOI: 10.1145/78969.78972. (Visited on 02/20/2025).

[Ill] Electrical and Computer Engineering Department Illinois Institute of Technology. *IEEE118bus*. http://motor.ece.iit.edu/data/IEEE118bus_inf/. (Visited on 03/23/2023).

[Jää+15] Pekka Jääskeläinen, Carlos Sánchez de La Lama, Erik Schnetter, et al. "Pocl: A Performance-Portable OpenCL Implementation". In: *International Journal of Parallel Programming* 43.5 (Oct. 2015), pp. 752–785. ISSN: 1573-7640. DOI: 10.1007/s10766-014-0320-y. (Visited on 03/26/2023).

[Jak+22] Sigurd Hofsmo Jakobsen, Junjie Zhang, Tor Inge Reigstad, et al. "Modelica-Based Parallel Computing Framework for Power System Adaptive Special Protection Schemes". In: *2022 Open Source Modelling and Simulation of Energy Systems (OSMSES)*. Apr. 2022, pp. 1–6. DOI: 10.1109/OSMSES54027.2022.9769162.

[JD09] Vahid Jalili-Marandi and Venkata Dinavahi. "Instantaneous Relaxation-Based Real-Time Transient Stability Simulation". In: *IEEE Transactions on Power Systems* 24.3 (Aug. 2009), pp. 1327–1336. ISSN: 1558-0679. DOI: 10.1109/TPWRS.2009.2021210.

[JD10] Vahid Jalili-Marandi and Venkata Dinavahi. "SIMD-Based Large-Scale Transient Stability Simulation on the Graphics Processing Unit". In: *IEEE Transactions on Power Systems* 25.3 (Aug. 2010), pp. 1589–1599. ISSN: 1558-0679. DOI: 10.1109/TPWRS.2010.2042084.

[KBL94] P. Kundur, Neal J. Balu, and Mark G. Lauby. *Power System Stability and Control*. EPRI Power System Engineering Series. New

York: McGraw-Hill, 1994. ISBN: 978-0-07-035958-1. (Visited on 10/25/2022).

[Kel+15] Brian M. Kelley, Philip Top, Steven G. Smith, et al. "A Federated Simulation Toolkit for Electric Power Grid and Communication Network Co-Simulation". In: *2015 Workshop on Modeling and Simulation of Cyber-Physical Energy Systems (MSCPES)*. Apr. 2015, pp. 1–6. DOI: 10.1109/MSCPES.2015.7115406.

[KH13] David Kirk and Wen-mei W. Hwu. *Programming Massively Parallel Processors: A Hands-on Approach*. 2. ed. Amsterdam: Elsevier, Morgan Kaufmann, 2013. ISBN: 978-0-12-415992-1.

[KKO03] P. M. W. Knijnenburg, T. Kisuki, and M. F. P. O'Boyle. "Combined Selection of Tile Sizes and Unroll Factors Using Iterative Compilation". In: *The Journal of Supercomputing* 24.1 (Jan. 2003), pp. 43–67. ISSN: 1573-0484. DOI: 10.1023/A:1020989410030. (Visited on 03/10/2025).

[Kli+11] Guido Klingbeil, Radek Erban, Mike Giles, et al. "Fat versus thin threading approach on gpus: Application to stochastic simulation of chemical reactions". In: *IEEE Transactions on Parallel and Distributed Systems* 23.2 (2011), pp. 280–287.

[KM98] Peter Kunkel and Volker Mehrmann. "Regular Solutions of Nonlinear Differential-Algebraic Equations and Their Numerical Determination". In: *Numerische Mathematik* 79.4 (June 1998), pp. 581–600. ISSN: 0945-3245. DOI: 10.1007/s002110050353.

[KOY89] David R Kincaid, Thomas C Oppe, and David M Young. *ITPACKV 2D user's guide*. Tech. rep. Texas Univ., Austin, TX (USA). Center for Numerical Analysis, 1989.

[KP10] Brian W. Kernighan and Rob Pike. *The Practice of Programming*. 19. print. Addison-Wesley Professional Computing Series. Boston Munich: Addison-Wesley, 2010. ISBN: 978-0-201-61586-9.

[LD19] Ning Lin and Venkata Dinavahi. "Exact Nonlinear Micromodeling for Fine-Grained Parallel EMT Simulation of MTDC Grid Interaction With Wind Farm". In: *IEEE Transactions on Industrial Electronics* 66.8 (Aug. 2019), pp. 6427–6436. ISSN: 1557-9948. DOI: 10.1109/TIE.2018.2860566.

[Lee+04] Benjamin C Lee, Richard W Vuduc, James W Demmel, et al. "Performance models for evaluation and automatic tuning of symmetric sparse matrix-vector multiply". In: *International Conference on Parallel Processing, 2004. ICPP 2004.* IEEE. 2004, pp. 169–176.

[LMT01] Jacques-Louis Lions, Yvon Maday, and Gabriel Turinici. "Résolution d'EDP Par Un Schéma En Temps «pararéel »". In: *Comptes Rendus de l'Académie des Sciences - Series I - Mathematics* 332.7 (Apr. 1, 2001), pp. 661–668. ISSN: 0764-4442. DOI: 10.1016/S0764-4442(00)01793-6. https://www.sciencedirect.com/science/article/pii/S0764444200017936 (visited on 11/19/2024).

[LRS82] E. Lelarasmee, A.E. Ruehli, and A.L. Sangiovanni-Vincentelli. "The Waveform Relaxation Method for Time-Domain Analysis of Large Scale Integrated Circuits". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 1.3 (July 1982), pp. 131–145. ISSN: 1937-4151. DOI: 10.1109/TCAD.1982.1270004.

[Lun+09] Håkan Lundvall, Kristian Stavåker, Peter Fritzson, et al. "Automatic Parallelization of Simulation Code for Equation-Based Models with Software Pipelining and Measurements on Three Platforms". In: *ACM SIGARCH Computer Architecture News* 36.5 (June 2009), pp. 46–55. ISSN: 0163-5964. DOI: 10.1145/1556444.1556451. (Visited on 07/13/2023).

[LV15] Weifeng Liu and Brian Vinter. "CSR5: An efficient storage format for cross-platform sparse matrix-vector multiplication". In: *Proceedings of the 29th ACM on International Conference on Supercomputing*. 2015, pp. 339–350.

[Mac+20] Jan Machowski, Zbigniew Lubosny, Janusz W. Bialek, et al. *Power System Dynamics: Stability and Control*. John Wiley & Sons, June 2020. ISBN: 978-1-119-52634-6.

[Mar+14] Jose R. Marti, Hermann W. Dommel, Benedito D. Bonatto, et al. "Shifted Frequency Analysis (SFA) Concepts for EMTP Modelling and Simulation of Power System Dynamics". In: *2014 Power Systems Computation Conference*. 2014 Power Systems Computation Conference (PSCC). Wrocław, Poland: IEEE, Aug. 2014, pp. 1–8. ISBN: 978-83-935801-3-2. DOI: 10.1109/PSCC.2014.7038487. http://ieeexplore.ieee.org/document/7038487/ (visited on 09/25/2023).

[MB21] Matthew Milton and Andrea Benigni. "ORTiS Solver Codegen: C++ Code Generation Tools for High Performance, FPGA-based, Real-Time Simulation of Power Electronic Systems". In: *SoftwareX* 13 (Jan. 2021), p. 100660. ISSN: 2352-7110. DOI: 10.1016/j.softx.2021.100660. (Visited on 01/31/2025).

[MBM19] Matthew Milton, Andrea Benigni, and Antonello Monti. "Real-Time Multi-FPGA Simulation of Energy Conversion Systems". In: *IEEE Transactions on Energy Conversion* (Sept. 2019), pp. 1–1.

ISSN: 0885-8969. DOI: 10.1109/tec.2019.2938811. (Visited on 12/12/2019).

[Med+17] Wided Medjroubi, Ulf Philipp Müller, Malte Scharf, et al. "Open Data in Power Grid Modelling: New Approaches Towards Transparent Grid Models". In: *Energy Reports* 3 (Nov. 2017), pp. 14–21. ISSN: 23524847. DOI: 10.1016/j.egyr.2016.12.001. (Visited on 03/10/2025).

[Mil10] Federico Milano. *Power System Modelling and Scripting*. 2010. ISBN: 978-3-642-13668-9.

[Mir+19] Markus Mirz, Steffen Vogel, Georg Reinke, et al. "DPsim—A Dynamic Phasor Real-Time Simulator for Power Systems". In: *SoftwareX* 10 (July 2019). ISSN: 23527110. DOI: 10.1016/j.softx.2019.100253. (Visited on 09/19/2019).

[MMB20] Markus Mirz, Antonello Monti, and Andrea Benigni. "A Dynamic Phasor Real-Time Simulation Based Digital Twin for Power Systems". PhD thesis. Aachen: E.ON Energy Research Center, RWTH Aachen University, 2020.

[MZB24] Marcel Mittenbühler, Junjie Zhang, and Andrea Benigni. "Automatically Optimized Component Model Computation for Power System Simulation on GPU". In: *Electric Power Systems Research* 235 (Oct. 1, 2024), p. 110740. ISSN: 0378-7796. DOI: 10.1016/j.epsr.2024.110740.

[Nak+15] Takuya Nakaike, Rei Odaira, Matthew Gaudet, et al. "Quantitative Comparison of Hardware Transactional Memory for Blue Gene/Q, zEnterprise EC12, Intel Core, and POWER8". In: *SIGARCH Comput. Archit. News* 43.3S (June 2015), pp. 144–157. ISSN: 0163-5964. DOI: 10.1145/2872887.2750403. (Visited on 03/08/2025).

[Nau+10] Maxim Naumov, L Chien, Philippe Vandermersch, et al. "Cusparse library". In: *GPU Technology Conference*. 2010.

[Nie64] J. Nievergelt. "Parallel Methods for Integrating Ordinary Differential Equations". In: *Commun. ACM* 7.12 (Dec. 1, 1964), pp. 731–733. ISSN: 0001-0782. DOI: 10.1145/355588.365137. https://dl.acm.org/doi/10.1145/355588.365137 (visited on 11/21/2024).

[NT12] Tatsushi Nishi and Yuki Tanaka. "Petri Net Decomposition Approach for Dispatching and Conflict-Free Routing of Bidirectional Automated Guided Vehicle Systems". In: *IEEE Transactions on Systems, Man, and Cybernetics - Part A: Systems and Humans* 42.5 (Sept. 2012), pp. 1230–1243. ISSN: 1558-2426. DOI: 10.1109/TSMCA.2012.2183353. (Visited on 02/22/2025).

[NVI2424] *NVIDIA Grace Hopper Superchip Architecture Whitepaper*. Tech. rep. NVIDIA, 2024. (Visited on 03/10/2025).

[Pao+20] Mario Paolone, Trevor Gaunt, Xavier Guillaud, et al. "Fundamentals of Power Systems Modelling in the Presence of Converter-Interfaced Generation". In: *Electric Power Systems Research* 189 (2020). ISSN: 03787796. DOI: 10.1016/j.epsr.2020.106811.

[PAT25] *US Patent 12008371 Method and Apparatus for Efficient Programmable Instructions in Computer Systems*. (Visited on 03/10/2025).

[PFACTORY] *PowerFactory - DIgSILENT*. https://www.digsilent.de/en/powerfactory.html (visited on 11/17/2023).

[PPG07] Nagaraju Pogaku, Milan Prodanovic, and Timothy C Green. "Modeling, Analysis and Testing of Autonomous Operation of an Inverter-Based Microgrid". In: *IEEE Transactions on Power Electronics* 22.2 (2007), pp. 613–625.

[Qui08] Michael J. Quinn. *Parallel Programming in C with MPI and OpenMP*. Internat. ed. 2003, [Nachdr.] Boston, Mass. [u.a]: McGraw-Hill, 2008. ISBN: 978-0-07-123265-4.

[SDB22] Julius Strake, Daniel Döhring, and Andrea Benigni. "MGRIT-Based Multi-Level Parallel-in-Time Electromagnetic Transient Simulation". In: *Energies* 15.21 (Jan. 2022), p. 7874. ISSN: 1996-1073. DOI: 10.3390/en15217874.

[She+22] Chen Shen, Qianni Cao, Mengshuo Jia, et al. "Concepts, Characteristics and Prospects of Application of Digital Twin in Power System". In: *Proceedings of the CSEE* 42.2 (2022), pp. 487–498. ISSN: 0258-8013. DOI: 10.13334/j.0258-8013.pcsee.211594. (Visited on 03/06/2025).

[Sjö+10] Martin Sjölund, Robert Braun, Peter Fritzson, et al. "Towards Efficient Distributed Simulation in Modelica Using Transmission Line Modeling". In: *MODELS 2010*. 2010. (Visited on 07/11/2023).

[SK12] Bor-Yiing Su and Kurt Keutzer. "clSpMV: A cross-platform OpenCL SpMV framework on GPUs". In: *Proceedings of the 26th ACM international conference on Supercomputing*. 2012, pp. 353–364.

[Son+18] Yankan Song, Ying Chen, Shaowei Huang, et al. "Efficient GPU-Based Electromagnetic Transient Simulation for Power Systems With Thread-Oriented Transformation and Automatic Code Generation". In: *IEEE Access* 6 (2018), pp. 25724–25736. ISSN: 2169-3536. DOI: 10.1109/ACCESS.2018.2833506.

[Spe18] Robert Speck. "Parallelizing Spectral Deferred Corrections across the Method". In: *Computing and Visualization in Science* 19.3 (July 2018), pp. 75–83. ISSN: 1433-0369. DOI: 10.1007/s00791-018-0298-x. (Visited on 09/14/2023).

[Sto79] B. Stott. "Power System Dynamic Response Calculations". In: *Proceedings of the IEEE* 67.2 (Feb. 1979), pp. 219–241. ISSN: 1558-2256. DOI: 10.1109/PROC.1979.11233. https://ieeexplore.ieee.org/document/1455502 (visited on 04/14/2024).

[Tan+17] Xulong Tang, Ashutosh Pattnaik, Huaipan Jiang, et al. "Controlled Kernel Launch for Dynamic Parallelism in GPUs". In: *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. Austin, TX: IEEE, Feb. 2017, pp. 649–660. ISBN: 978-1-5090-4985-1. DOI: 10.1109/HPCA.2017.14. (Visited on 02/15/2024).

[Tek+21] A Tekin, A Tuncer Durak, C Piechurski, et al. *State-of-the-Art and Trends for Computing and Interconnect Network Solutions for HPC and AI*. Tech. rep. PRACE, 2021, p. 38.

[UNI25] *Introduction to the Data-Oriented Technology Stack*. https://unity.com/resources/introduction-to-dots-ebook. (Visited on 02/14/2025).

[UNR25] *Overview of Mass Entity in Unreal Engine | Unreal Engine 5.0 Documentation | Epic Developer Community*. https://dev.epicgames.com/documentation/en-us/unreal-engine/overview-of-mass-entity-in-unreal-engine?application_version=5.0. (Visited on 02/14/2025).

[Vyg+21] Mark Vygoder, Matthew Milton, Jacob D. Gudex, et al. "A Hardware-in-the-Loop Platform for DC Protection". In: *IEEE Journal of Emerging and Selected Topics in Power Electronics* 9.3 (June 2021), pp. 2605–2619. ISSN: 2168-6785. DOI: 10.1109/JESTPE.2020.3017769.

[Wal+14] Marcus Walther, Volker Waurich, Christian Schubert, et al. "Equation Based Parallelization of Modelica Models". In: (Mar. 2014). (Visited on 01/06/2025).

[Wan+13] Qian Wang, Xianyi Zhang, Yunquan Zhang, et al. "AUGEM: Automatically Generate High Performance Dense Linear Algebra Kernels on X86 CPUs". In: *SC '13: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. Nov. 2013, pp. 1–12. DOI: 10.1145/2503210.2503219.

[WN95] Glynn Winskel and Mogens Nielsen. "Models for Concurrency". In: *Handbook of Logic in Computer Science (Vol. 4): Semantic Modelling*. USA: Oxford University Press, Inc., June 1995, pp. 1–148. ISBN: 978-0-19-853780-9. (Visited on 02/22/2025).

[Wu+20] Wei Wu, Peng Li, Xiaopeng Fu, et al. "GPU-based Power Converter Transient Simulation with Matrix Exponential Integration and Memory Management". In: *International Journal of Electrical Power & Energy Systems* 122 (Nov. 2020), p. 106186. ISSN: 0142-0615. DOI: 10.1016/j.ijepes.2020.106186. (Visited on 07/14/2023).

[WWP09a] Samuel Williams, Andrew Waterman, and David Patterson. "Roofline: An Insightful Visual Performance Model for Multicore Architectures". In: *Communications of the ACM* 52.4 (Apr. 2009), pp. 65–76. ISSN: 0001-0782. DOI: 10.1145/1498765.1498785.

[WWP09b] Samuel Williams, Andrew Waterman, and David Patterson. "Roofline: An Insightful Visual Performance Model for Multicore Architectures". In: *Commun. ACM* 52.4 (Apr. 2009), pp. 65–76. ISSN: 0001-0782.

[XBRAID] *XBraid: Parallel multigrid in time*. http://llnl.gov/casc/xbraid.

[XSZ05] Wei Xue, Jiwu Shu, and Weimin Zheng. "Parallel Transient Stability Simulation for National Power Grid of China". In: *Parallel and Distributed Processing and Applications*. Ed. by Jiannong Cao, Laurence T. Yang, Minyi Guo, et al. Berlin, Heidelberg: Springer, 2005, pp. 765–776. DOI: 10.1007/978-3-540-30566-8_89.

[YBA13] Tao Yang, Serhiy Bozhko, and Greg Asher. "Multi-Generator System Modelling Based on Dynamic Phasor Concept". In: *2013 15th European Conference on Power Electronics and Applications (EPE)*. Lille, France: IEEE, Sept. 2013, pp. 1–10. ISBN: 978-1-4799-0116-6. DOI: 10.1109/EPE.2013.6631919. (Visited on 11/18/2023).

[ZD17] Zhiyin Zhou and Venkata Dinavahi. "Fine-Grained Network Decomposition for Massively Parallel Electromagnetic Transient Simulation of Large Power Systems". In: *IEEE Power and Energy Technology Systems Journal* 4.3 (Sept. 2017), pp. 51–64. ISSN: 2332-7707. DOI: 10.1109/JPETS.2017.2732360.

[Zha+21] Junjie Zhang, Lukas Razik, Sigurd Hofsmo Jakobsen, et al. "An Open-Source Many-Scenario Approach for Power System Dynamic Simulation on HPC Clusters". In: *Electronics* 10.11 (11 Jan. 2021), p. 1330. ISSN: 2079-9292. DOI: 10.3390/electronics10111330.

https://www.mdpi.com/2079-9292/10/11/1330 (visited on 09/06/2023).

[Zha+22a] Han Zhang, Yifei Lu, Junjie Zhang, et al. "Real-Time Simulation of an Electrolyzer with a Diode Rectifier and a Three-Phase Interleaved Buck Converter". In: *2022 IEEE 13th International Symposium on Power Electronics for Distributed Generation Systems (PEDG)*. June 2022, pp. 1–6.

[Zha+22b] Junjie Zhang, Marcel Mittenbuehler, Lukas Razik, et al. "Parallel Simulation of Power Systems with High Penetration of Distributed Generation Using GPUs and OpenCL". In: *2022 IEEE 13th International Symposium on Power Electronics for Distributed Generation Systems (PEDG)*. June 2022, pp. 1–6. (Visited on 09/29/2023).

[ZMB24] Junjie Zhang, Marcel Mittenbühler, and Andrea Benigni. "Shifted Frequency Analysis Based, Faster-than-Real-Time Simulation of Power Systems on Graphics Processing Unit". In: *International Journal of Electrical Power & Energy Systems* 159 (Aug. 1, 2024), p. 110014. ISSN: 0142-0615. DOI: 10.1016/j.ijepes.2024.110014.

[ZMD10] Peng Zhang, José R. Marti, and Hermann W. Dommel. "Shifted-Frequency Analysis for EMTP Simulation of Power-System Dynamics". In: *IEEE Transactions on Circuits and Systems I: Regular Papers* 57.9 (Sept. 2010), pp. 2564–2574. ISSN: 1549-8328, 1558-0806. DOI: 10.1109/TCSI.2010.2043992. http://ieeexplore.ieee.org/document/5438913/ (visited on 11/18/2023).

Appendix A

A.1 Optimization Results

Table A.1: Optimization result for Nvidia A100-40GB GPU

Component	#Components	Group size	#Components per group	Matrix format with storage strategy			
Inverter	512	16	2	CDia*	CDia*	Dense-cat	ELL*
	2048	32	32	Dense-cat	Dense-cat	Dense-cat	Dense-cat
	8192	128	32	ELL*	ELL*	CDia*	ELL*
	32768	128	64	ELL*	ELL*	ELL*	ELL*
	131072	64	40	ELL*	ELL*	ELL*	ELL*
	524288	32	20	ELL*	ELL*	ELL*	ELL*
SyncMachine	512	128	14	CDia*	Dia*	ELL*	-
	2048	256	28	CDia*	CDia*	ELL*	-
	8192	256	28	CDia*	Dia*	CDia*	-
	32768	128	64	ELL*	ELL*	ELL*	-
	131072	64	32	ELL*	Dia-BD	CSR*	-
	524288	128	64	ELL*	CDia*	ELL*	-
Electrolyzer	512	256	12	ELL-BD	Dense-cat	CSR*	-
	2048	128	6	Dia*	ELL-BD	CSR-BD	-
	8192	128	12	ELL-BD	ELL-BD	CSR-BD	-
	32768	256	48	CSR-BD	ELL-BD	CSR-BD	-
	131072	64	6	CSR-BD	CDia*	CSR*	-
	524288	256	48	CSR-BD	CSR-BD	CSR*	-

*: pattern storage
-cat: concatenated storage
-BD: block diagonal storage

Table A.2: Optimization result for AMD MI100 GPU

Component	#Components	Group size	#Components per group	Matrix format with storage strategy			
				A	B	C	D
Inverter	512	32	4	ELL*	ELL*	ELL*	ELL*
	2048	256	32	ELL*	ELL*	ELL*	ELL*
	8192	256	32	ELL*	ELL*	CSR*	ELL*
	32768	256	96	ELL*	ELL*	ELL*	ELL*
	131072	128	48	ELL*	ELL*	ELL*	ELL*
	524288	128	32	ELL*	ELL*	ELL*	CSR*
SyncMachine	512	64	7	CDia*	Dia*	ELL*	-
	2048	64	7	CDia*	ELL*	ELL*	-
	8192	256	28	ELL*	Dia*	ELL*	-
	32768	256	96	ELL*	Dia*	ELL*	-
	131072	128	28	ELL*	CSR*	ELL*	-
	524288	256	28	CSR*	CDia*	ELL*	-
Electrolyzer	512	64	3	ELL*	Dense-cat	CSR*	-
	2048	128	6	ELL*	ELL*	CSR*	-
	8192	256	12	ELL*	CSR*	CSR*	-
	32768	256	12	ELL*	CSR*	CSR*	-
	131072	256	12	ELL*	CSR*	CSR*	-
	524288	128	3	CSR*	ELL*	CSR*	-

*: pattern storage
-cat: concatenated storage
-BD: block diagonal storage

Band / Volume 679
Entwicklung von nickelbasierten katalysatorbeschichteten Diaphragmen für die alkalische Wasserelektrolyse
C. B. Karacan (2025), 146 pp
ISBN: 978-3-95806-860-5

Band / Volume 680
Bewertung lokaler Eigenspannungsverteilungen bei der lokalen Bauteilreparatur durch Kaltgasspritzen
J.-C. Schmitt (2025), 154, xxvii pp
ISBN: 978-3-95806-861-2

Band / Volume 681
First principles study of the effect of substitution\doping on the performance of layered oxide cathode materials for secondary batteries
N. Yaqoob (2025), iii, 126 pp
ISBN: 978-3-95806-864-3

Band / Volume 682
Field assisted sintering technology/spark plasma sintering in the direct recycling of hot-deformed Nd-Fe-B scrap and PM T15 steel swarf
M. T. M. Keszler (2025), viii, 173 pp
ISBN: 978-3-95806-866-7

Band / Volume 683
Assessment of erosion in recessed areas of fusion devices using multi-scale computer simulations
S. W. Rode (2025), viii, 196 pp
ISBN: 978-3-95806-867-4

Band / Volume 684
Europäische Energiewende – Deutschland im Herzen Europas
T. Klütz, P. Dunkel, T. Busch, J. Linssen, D. Stolten (2025), IV, 56 pp
ISBN: 978-3-95806-870-4

Band / Volume 685
Performance and stability of solar cells and modules: From laboratory characterization to field data analysis
T. S. Vaas (2025), xvii, 146 pp
ISBN: 978-3-95806-871-1

Schriften des Forschungszentrums Jülich
Reihe Energie & Umwelt / Energy & Environment

Band / Volume 686
From Soil Legacy to Wheat Yield Decline: Studying the Plant-Soil Feedback Mechanisms in Wheat Rotations
N. Kaloterakis (2025), XXIX, 188 pp
ISBN: 978-3-95806-874-2

Band / Volume 687
Entwicklung von Beschichtungsverfahren für die Herstellung von Wärmedämmschichten auf additiv gefertigten Komponenten
M. Rüßmann (2026), ix, 188 pp
ISBN: 978-3-95806-877-3

Band / Volume 688
Model Perovskite Oxide Electrocatalysts for the Oxygen Evolution Reaction and their Material Sustainability Evaluation
L. Heymann (2026), vi, 174 pp
ISBN: 978-3-95806-878-0

Band / Volume 689
Development of an oxygen ion conducting solid oxide electrolysis cell based on gadolinium-doped cerium oxide as fuel electrode and electrolyte material
D. Ramler (2026), ix, 162 pp
ISBN: 978-3-95806-879-7

Band / Volume 690
Design of Local Multi-Energy Systems: Impact of Coupled Energy Vector Integration and Grid Service Participation
P. S. Glücker (2026), xxviii, 145 pp
ISBN: 978-3-95806-880-3

Band / Volume 691
A Parallel-in-Space Simulator for Accelerating Power System Simulation on Graphics Processing Units
J. Zhang (2026), 112 pp
ISBN: 978-3-95806-882-7

Weitere ***Schriften des Verlags im Forschungszentrum Jülich*** unter
http://wwwzb1.fz-juelich.de/verlagextern1/index.asp

Energie & Umwelt / Energy & Environment
Band / Volume 691
ISBN 978-3-95806-882-7

Mitglied der Helmholtz-Gemeinschaft

JÜLICH
Forschungszentrum