Power System Probabilistic and Security Analysis on Commodity High Performance Computing Systems

Tao Cui
Carnegie Mellon University
5000 Forbes Ave.
Pittsburgh, PA 15213
tao.cui@ieee.org

Franz Franchetti
Carnegie Mellon University
5000 Forbes Ave.
Pittsburgh, PA 15213
franzf@ece.cmu.edu

ABSTRACT

Large scale integration of stochastic energy resources in power systems requires probabilistic analysis approaches for comprehensive system analysis. The large-varying grid condition on the aging and stressed power system infrastructures also requires merging of offline security analyses into online operation. Meanwhile in computing, the recent rapid hardware performance growth comes from the more and more complicated architecture. Fully utilizing the computation power for specific applications becomes very difficult. Given the challenges and opportunities in both the power system and computing fields, this paper present the unique high performance commodity computing system solution to the following fundamental tools for power system probabilistic and security analysis: 1) a high performance Monte Carlo simulation (MCS) based distribution probabilistic load flow solver for real time distribution feeder probabilistic solution. 2) A high performance MCS based transmission probabilistic load flow solver for transmission grid analysis. 3) A SIMD accelerated AC contingency calculation solver based on Woodbury matrix identity on multi-core CPUs. By aggressive algorithm level and computer architecture level performance optimizations including optimized data structures, optimization for superscalar out-of-order execution, SIMDization, and multi-core scheduling, our software fully utilizes the modern commodity computing systems, makes the critical and computational intensive power system probabilistic and security analysis problems solvable in real time on commodity computing systems.

Categories and Subject Descriptors

G.4 [**Mathematical Software**]: Parallel and vector implementations; I.6.8 [**Simulation and Modeling**]: Types of Simulation—*Monte Carlo, Parallel*

Keywords

Code optimization, performance tuning, power system computation, SIMD, multi-core

Prepared for HiPCNA-PG'13 Denver, CO, USA

1. INTRODUCTION

The electric power system has been experiencing significant transition in the past decades. The large scale integration of stochastic and variable renewable energy resources such as wind and solar energy as well as plug-in of new loads with large variance such as electrical vehicles introduce significant uncertainties in the power grids. Most of the new power injections are stochastic and non-dispatchable in nature, large penetration level results in significant impacts on almost every aspects of the power grids. Moreover, the more active and increasing loads and generations drive today's aged and stretched power grid closer and closer to its limits, thus result in higher possibility of component failures as well as more serious consequences of any contingencies.

In response to the new challenges in uncertainties and security analysis requirements, North-American Electric Reliability Corporation (NERC) has suggested using new *probabilistic* analysis approach for power grid analysis from distribution feeders to bulk power grids [18]. NERC also suggests applying N-1 contingency to N-k contingency analysis for even *real time* operation [17]. Given the large uncertainties and more strict security assessment requirements, an efficient and generally applicable computational analysis framework that can analyze and monitors the power grid using probabilistic approaches and assess system security comprehensively in real time would be an critical important tool for the efficient and reliable operation of the power grids.

On the computing side, the last decades have seen an enormous growth in the performance capabilities of computing platforms. A current Intel server processor has a double precision peak performance of more than 200 Gflop/s (10^9 additions/subtractions/multiplications per second) thanks to an 8-core CPUs with AVX vector instructions [13]. In term of this value, the single chip desktop CPU has similar peak performance comparing to the No.1 fastest supercomputer (Fujitsu NWT, 280 Gflop/s) in 1995 and to the No.500 fastest supercomputer (Cray T3E1200, 138 Gflop/s) in just 2001 [16]. However, the recent advances in computing performance are mainly thanks to the more and more complicated hardware architecture such as deep memory hierarchy, multiple levels of parallelism (data level, instruction level, task level, etc). Without the awareness of hardware architecture, most software can only utilize a very small fraction of the CPU's computing power and cannot benefit from the current and future growth of computer hardware capability. To fully exact the computing power out of the modern

hardware architecture becomes very difficult. It requires the knowledge and efforts from both the application domain and the computer architecture domain, including but not limited to algorithm level optimization, data structure optimization, special hardware instructions and parallel programming, etc. In most cases, the specific numerical application may need to be carefully redesigned and tuned to fully utilize the modern hardware capability.

This paper targets on the fundamental computational kernels for above power system challenges, specifically consisting of the followings: 1) a high performance Monte Carlo simulation (MCS) based three phase distribution probabilistic load flow (DPLF) solver for real time feeder probabilistic analysis and monitoring. MCS methods for probabilistic load flow (PLF) are considered to be robust, generally-applicable and can be accurate in theory, therefore are often used as accuracy reference for other method. However, MCS methods are also believed to be computational intensive and impractical for real time application. With aggressive code optimization, multi-level parallelization and task-decomposition for real time application, we are pushing the computing speed to the machine peak on commodity multi-core CPUs, building the highly optimized solver with order-of-magnitude speedup comparing to the baseline software. 2) a high performance Monte Carlo simulation based transmission probabilistic load flow (TPLF) solver. Based on fast decoupled load flow algorithm, we investigated and developed efficient linear solver and related elementary functions for parallel massive amount load flow computations specifically for real time MCS solution of TPLF. 3) An accelerated AC contingency calculation (ACCC) is also proposed and developed for fast and comprehensive steady state security assessment. Based on Woodbury matrix identity, different contingency cases can be transformed into the SIMD (single instruction multiple data) computation model, together with a thread pool scheduler, our implementation fully utilizes the computing power of commodity system, making comprehensive ACCC efficient and feasible for real time application on commodity computing systems.

This paper is organized as following: the background of commodity hardware is reviewed in Section 2, the DPLF solver is presented in Section 3, the TPLF solver is presented in Section 4, the ACCC solver is presented in Section 5, Section 6 concludes the paper.

2. COMMODITY COMPUTING SYSTEMS

The main architecture we are targeting is the new Intel CPU with Sandy Bridge (or later generation) micro-architecture with deep memory hierarchy, superscalar out-of-order scheduler, AVX (Advanced Vector eXtension) instruction set extensions and multiple CPU cores.

Fig. 1 shows the block diagram of an example Sandy Bridge CPU (Core i7 2670QM) topology. It has 4 CPU cores (Core P#0 to P#3), each core has two logical processing units (PU) due to Hyper-Threading (Intel's term for simultaneous multithreading or SMT). The CPU system has three levels of cache memories (L1, L2 and L3). Multi-core and deep memory hierarchy (three level of cache memories) are the most relevant features in this figure at the CPU core level.

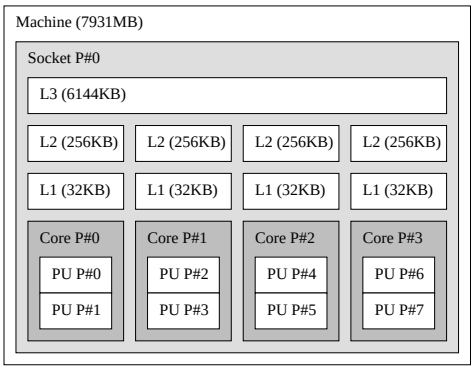

Figure 1: Core i7 2670QM: 4 physical cores, 3 levels cache (L1–L3), 8 GB memory

Fig. 2 shows the architecture inside each core (Core P#0 to P#3 in Fig. 1). This figure is taken from "Intel®64 and IA-32 Architectures Optimization Reference Manual" [12]. It shows an out-of-order superscalar scheduler in the middle, such scheduler is able to issue multiple independent instructions to independent arithmetic, floating point, or load/store functional units to exploit the instruction level parallelism.

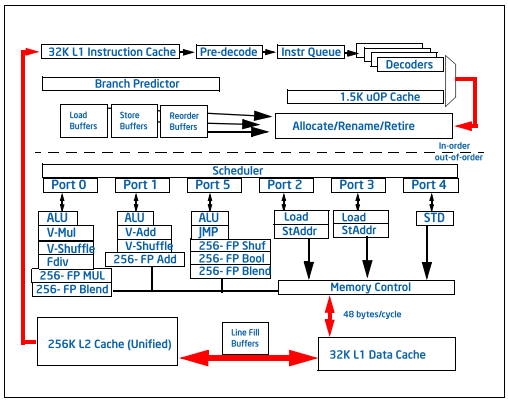

Figure 2: Intel Sandy Bridge micro-architecture inside each core

If we further look into the floating point functional units in each core on a Sandy Bridge CPU (e.g. `256-FP Add` in Fig. 2). The floating point unit is capable of processing multiple data by a single instruction. The so called Single Instruction Multiple Data units (or SIMD unit) exploit data level parallelism. As showed in Fig. 3, for example, we consider working on 32-bit single precision floating point data. On the `256-FP Add` unit, it can execute the scalar version ADD instruction `FADD` which adds one floating point to another at a time. It is also capably of execute SSE version ADD instruction `ADDPS` which can add an array of 4 floating point data to another array of 4 at a time. It also supports the new AVX version ADD instruction `VADDPS`, which can add an array of up to 8 floating point data (256-bit in total) to another array of 8 floating point data at the same time.

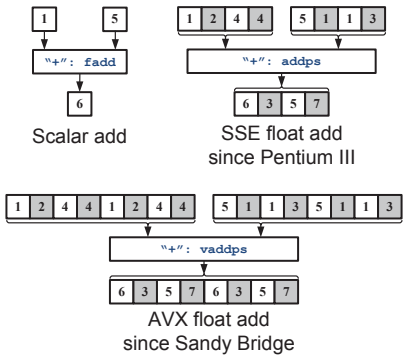

Figure 3: Illustration of SIMD operation

The clock frequency of Core i7 2670QM in is 2.2 GHz. By taking the above computing capabilities of the CPU in to consideration, the single floating point theoretical peak performance of this chip is 2.2 G $\times$ (1 floating `ADD` + 1 floating `MUL`) $\times$ 8 single floats in AVX $\times$ 4 cores = 140.8 Gflop/s. This is the theoretical single precision peak performance of this chip [13]. In terms of this value, this CPU in 2012 has the similar performance as the fastest supercomputer in the world in just year 2001 (Cray T3E1200, 138 Gflop/s on Top500 List) [16]. The rapid growth of computing capability implies Moore's law still applies on the modern hardware architectures [15].

However, from the description of the architecture and the calculation of hardware theoretical peak performance, to benefit from the high peak hardware performance, the application software has to *fully* utilize all the performance enhance features of the CPU. Given the more and more complicated hardware, fully utilizing the computation power for specific applications becomes very difficult on modern CPU. In this paper, from the hardware perspective, we are particularly looking into the following aspects to improve the computational performance of our proposed power system probabilistic and security analysis applications.

Memory hierarchy. Memory hierarchy includes main memory and multiple levels of caches. The cache is a small but fast memory that automatically keeps and manages copies of the most recently used and the most adjacent data from the main memory locations in order to bridge the speed gap between fast processor and slow main memories. There could be multiple levels of caches (such as L1,L2,L3 in Fig. 1), the levels closer to CPU cores are faster in speed but smaller in size. An optimized data access pattern is important to utilize the cache functions to increase the performance.

Multi-level parallelism. Utilization of multilevel parallelism inside each CPU core and among CPU cores can have significant impact on computational performance. We are looking into the following aspects that are relevant to our applications.

1. Instruction level: Superscalar and out-of-order architecture exploits the instruction level parallelism. Within a single CPU core, superscalar processor executes more than one instruction during a clock cycle by simultaneously dispatching multiple instructions to multiple functional units on the processor (as showed in the middle of Fig. 2). Out-of-order execution re-order the instructions according to their dependency, and independent instructions within a instruction dispatch window can be executed simultaneously on multiple functional units. Code optimization techniques such as loop unrolling, mixture of independent instructions, using bigger un-branched code blocks, etc. can be used to exploit the instruction level parallelism on superscalar hardware architectures [4] [6].

2. Data level: Single Instruction Multiple Data (SIMD): The Streaming SIMD Extensions (SSE) or the Advanced Vector eXtensions (AVX) instruction sets on Intel or AMD's x86 CPUs can perform floating point arithmetic operations on 4 single precision floating point (SSE) or 8 single precision floating point (AVX) data packed in vector register at the same time. Besides SSE and AVX which are already available on commodity systems, many new or under-developing microarchitectures such as Intel's new Larrabee architecture are further expanding the processing width of SIMD units (to 16 single precision floating point data).

3. Task-level: Multicore CPUs enables multiple threads to be executed simultaneously and independently on different CPU cores while communicate via shared memories. A proper scheduling and synchronization strategy is necessary for real time applications. And balancing the work load among the cores is one the most important consideration for parallel programming.

3. MCS SOLVER DISTRIBUTION PLF

In this section, we present a high performance parallel Monte Carlo simulation (MCS) framework for distribution PLF on multicore CPUs [7] [8] [9]. We use forward backward sweep based load flow algorithm for distribution network load flow solutions [14]. We applied aggressive code optimization including data structure optimization that transforms forward backward sweep into array access for better memory hierarchy utilization, algorithm level optimization and code generation considering the sparse property of equipment models, multi-level parallelism including Single Instruction Multiple Data (SIMD) model and task-decomposition based scheduling on multicore CPUs for real time Monte Carlo simulation applications. For the proposed MCS type applications, our optimized load flow solver is able to achieve more than 50% of a CPU's theoretical peak performance. That is about 50x speedup comparing to the best compiler-optimized baseline code on a quad-core CPU. The optimized MCS solver is able to solve millions of load flow of IEEE 37 Test Feeders [11] within a second on a quadcore Sandy Bridge CPU, therefore enabling MCS as real-time, high-accuracy and generally applicable solution for the real time PLF analysis on distribution feeders. Most of the work in this part has been discussed in [8] [9]. In this section we briefly highlight the key approaches and results.

3.1 Code Optimization

The forward backward sweep (FBS) load flow method solves the radial distribution load flow by traversing over the radial distribution network (tree) from substation (root) to

each the loads (leaves) using (2) to update voltages, and traverses back from leaves to root using (1) to update branch currents until voltages converges. The basic computation is the complex 3×3 matrix and 3×1 vector multiplication in following equation (1) (2) [14].

$$[\mathbf{I}^n_{abc}]_{3\times1} = [\mathbf{c}]_{3\times3}\,[\mathbf{V}^m_{abc}]_{3\times1} + [\mathbf{d}]_{3\times3}\,[\mathbf{I}^m_{abc}]_{3\times1} \quad (1)$$

$$[\mathbf{V}^m_{abc}]_{3\times1} = [\mathbf{A}]_{3\times3}\,[\mathbf{V}^n_{abc}]_{3\times1} - [\mathbf{B}]_{3\times3}\,[\mathbf{I}^m_{abc}]_{3\times1} \quad (2)$$

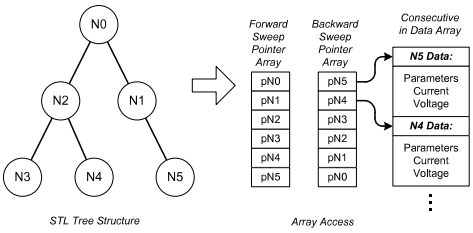

Figure 4: Data structure optimization

Data structure optimization. The baseline implementation model the tree using C++ object oriented programming. Starting from the baseline, the major data structure optimization is to flatten the tree object into an 1D array (Fig. 4). In this way, the tree traversals with object data access through member functions are converted into streaming memory accesses to a raw data array. The sweeps are turned into linearly (upwards) or almost linearly (downwards) traversals on the data array. Thus, the optimized FBS computation preserves temporal and spatial locality of the data streams. The data structure optimized code takes advantages of the memory hierarchy and yield much better performance than the baseline C++ code.

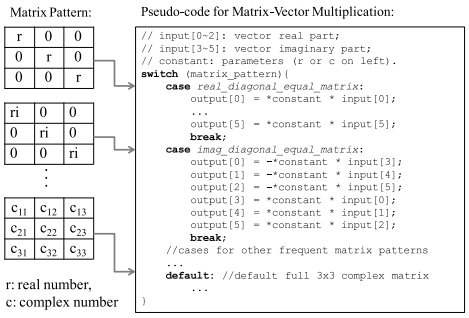

Figure 5: Pattern-based optimized sparse matrix-vector multiplication.

Specialization through code synthesis. The main per-node or per-branch operations in the FBS are small matrix-vector multiplications in (1) (2). The $\mathbf{A}, \mathbf{B}, \mathbf{c}, \mathbf{d}$ matrices are constant matrices. Due to the link model's physical properties, most of these matrices are symmetric, diagonal or even identity matrices, and there is a limited number of sparsity pattern. These patterns are fixed once the system's physical elements are given. We synthesize special matrix-vector multiplication kernels that inline the matrix structure into the kernel as showed in Fig 5: we generate one specialized kernel per matrix pattern and use a jump-table dispatch mechanism (switch-case) that invokes the correct kernel for each pattern. The savings can be considerable as small 3×3 matrix-vector product kernels can be fully unrolled. Bigger code blocks and unrolled loop exploit *instruction level parallelism* of modern CPUs. Our approach is similar to [3], which introduces a pattern-based sparse matrix multiplication kernel. We also compress the 3×3 matrix by its pattern and its non-zero elements.

3.2 Explicit Parallelization

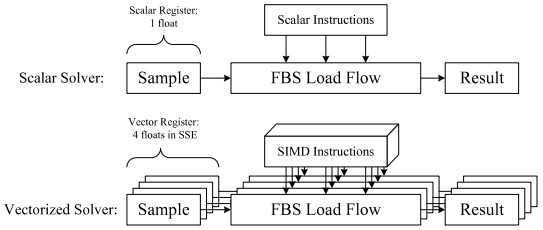

Figure 6: Vectorization of load flow solver for MCS

SIMD vectorization. SIMD exploits the fine-grained *data level parallelism*. The repeat load flow computation in MCS is same load flow algorithm (same instruction sequence) operates on different sample values (multiple data). We vectorize the solver for x86 Streaming SIMD Extentions (SSE) or Advanced Vector eXtention (AVX). These instruction set extensions use vector registers to hold 4 or 8 single precision floating point data and use SIMD instructions to process same arithmetic operations on these multiple data at the same time. As show in Fig. 6 we pack multiple samples into SIMD vector registers, and convert scalar instructions to SIMD instructions. In this way, multiple load flows can be solved at the same time.

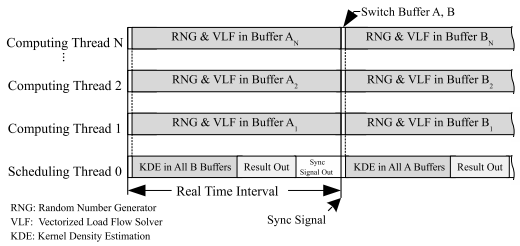

Figure 7: Real time multi-thread MCS on multicore

Multi-threading for real time MCS application. In particular for Monte Carlo simulation the exact number of problems to be solved is not of utmost importance as long as the accuracy is maintained. Thus we can run multiple Monte Carlo simulations independently on the different cores and collect Monte Carlo results after a pre-specified (long enough) time without having to ensure that all thread perform the exactly given number of simulations. We implemented a light-weight worker thread infrastructure that allows for fast buffer switching, shown in Fig. 7. A master scheduling thread orchestrates the computation on multiple computing threads and collects and post-processes the results. At the end of every real-time interval, the master thread sends a sync signal to all worker threads, so that all worker threads switch to new buffers. Once they signal back

that they switched the master thread collects the results from the old buffers of all computing threads to post process. The remaining cores are saturated with worker threads running the SIMD vector load flow solver in parallel on independent problems. In this way, the real time scheduler exploit the *task level parallelism*, the speed for solving large amount of load flows is only limited by the number of cores present in the CPU. In another words, the MCS implementation fully utilizes the computing power of the hardware.

3.3 Performance Results

We show the performance results measured in Gflop/s (floating point operations divided by runtime). The detail of performance results on the solver on quadcore Core-i7 CPU with AVX are shown in Fig. 8. We duplicate multiple IEEE 4-bus test feeder and interconnect them at the root to build a bigger case for benchmarking the computing speed. The highest curve is the speed of fully optimized solver using optimized data structure, AVX, multithread and pattern based matrix vector product. The peak speed of the solver reaches 80 Gflop/s, which is around 65% of the machine's theoretical peak [13]. When the system becomes bigger, the performance drops mainly because the data cannot be completely fitted into the CPU caches.

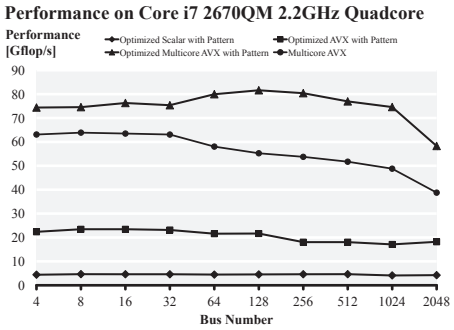

Figure 8: Performance on different network sizes

Table 1: Approximate Runtime of MCS DPLF

1M Load Flow		Optimized Code		Baseline
System	Flops	Core 2	Core i7	C++(O3)
IEEE 37	≈ 60G	< 2s	< 1s	> 60s
IEEE 123	≈ 200G	< 10s	< 3.5s	> 200s

To solve 1 million load flow cases of IEEE 37-bus test system and IEEE-123 bus test system, the approximate runtime of MCS is showed in Table 1. We can see that on new Intel Sandy Bridge CPU (Core-i7) with quad-core and AVX, 1 million load flows can be solved within 4 seconds, which is less than the update time interval of most SCADA system. For most PLF cases, 1 millions load flow samples can achieve accurate enough MCS results. The baseline runtime results including fully-compiler-optimized C++ code (-o3) are also showed for reference. Clearly, the baseline programs without hardware-aware optimization fail to produce the similar performance results under real time constraint.

We also tested the MCS solver on IEEE 37-bus test feeder on different machines [11]. As show in Fig. 9, the performance increases with the increase of SIMD width (SSE to AVX), and with the increase of number of CPU cores. This figure implies that optimized MCS solver is a well fitted application for modern computer architecture. It sees an almost *linear speedup* with the increase of hardware parallel capacity. Therefore, with the trend of increasing parallelism on modern CPUs, further performance increase can be expected by using new CPU hardware (e.g. Intel's Larrabee and MIC architecture).

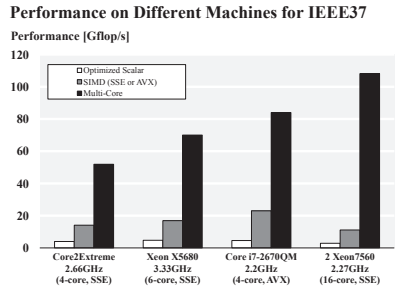

Figure 9: Performance on different platforms.

4. MCS SOLVER TRANSMISSION PLF

Based on the similar MCS framework, in this part, we focus on MCS based probabilistic load flow for transmission network using fast decoupled power flow (FDPF) algorithms. We present an algorithmically and architecturally optimized MCS based transmission PLF solver on commodity system. At algorithm level, we optimized the fast decoupled power flow implementation with highly optimized math functions (e.g. sparse LU factorization, efficient math functions implementation and utilization). At the architecture level, we apply aggressive code optimization techniques including optimized sparse data structure for better cache performance, loop unrolling for sparse kernel to exploit the supersaclar out-of-order architecture, Single Instruction Multiple Data (SIMD), multithreading on multiple CPU cores and task-decomposition scheduling for real time MCS application. As a result of our optimization, we show our solver is able to solve up to 1 million load flow sample cases of the IEEE 118-bus system and ∼50K sample cases of the Polish 2383-bus system within 5 seconds on an Intel Sandy Bridge quadcore CPU. Our work shows a fast, accurate, reliable and generally applicable MCS solver as the transmission PLF solution on inexpensive commodity computing system.

4.1 Code and Algorithm Optimization

In this section we discuss the code optimization of MCS solver for transmission PLF on multicore CPUs.

LU Factorization In fast decoupled power flow algorithm, we need to solve two linear systems (3) (4) in each iteration:

$$-\mathbf{B}'\boldsymbol{\Delta}\theta = \boldsymbol{\Delta}\mathbf{P}/\mathbf{V} \quad (3)$$

$$-\mathbf{B}''\boldsymbol{\Delta}\mathbf{V} = \boldsymbol{\Delta}\mathbf{Q}/\mathbf{V} \quad (4)$$

These matrices are factorized as the product of lower $\mathbf{L}', \mathbf{L}''$

and upper $\mathbf{U}', \mathbf{U}''$ triangle matrices as following:

$$\mathbf{P}'\mathbf{B}'\mathbf{Q}' = \mathbf{L}'\mathbf{U}' \quad (5)$$

$$\mathbf{P}''\mathbf{B}''\mathbf{Q}'' = \mathbf{L}''\mathbf{U}'' \quad (6)$$

$\mathbf{P}', \mathbf{P}''$ are partial pivoting permutation matrices. $\mathbf{Q}', \mathbf{Q}''$ are column reordering permutation matrices. During the iteration, solving the linear equation becomes two forward and backward substitutions using $\mathbf{B}'$ and $\mathbf{B}''$'s LU factors.

Both $\mathbf{B}'$ and $\mathbf{B}''$ are sparse matrices originating from a circuit matrix, a proper ordering schemes can result in sparse LU factors and can significantly reduce the floating point operations in the forward and backward substitution steps. This is particularly important for our MCS application using FDPF algorithm.

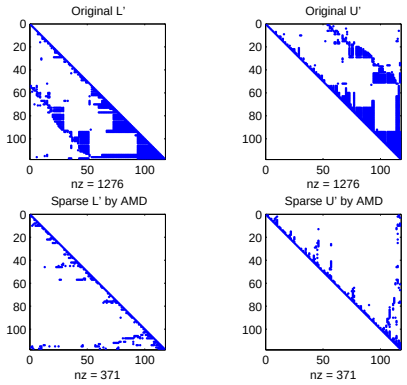

Figure 10: LU of IEEE118's B′: LU used in Matpower (top) and sparse LU using AMD (bottom)

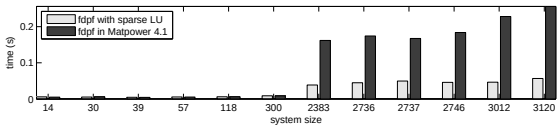

Figure 11: Improve performance by sparse LU

Fig. 10 shows the sparsity of factors L and U of the IEEE 118-bus system's $\mathbf{B}'$ matrix using different ordering schemes: e.g. approximated minimal degree (AMD) for circuit matrix [2], the factors L and U can be very sparse. Fig. 11 shows the overall runtime comparison of original FDPF implementation in Matpower 4.1 [20] and improved FDPF using sparse LU factorization (both coded in Matlab). The algorithm level optimization using sparse factors results in up to 4–5x overall speedup. Starting from here, our baseline code is built upon this AMD based sparse LU factorization.

Optimizing Data Storage During the mismatch computation, the trigonometric functions of angle differences (e.g. $\sin(\theta_i - \theta_j)$, $\cos(\theta_i - \theta_j)$) participate in the actual computation. On modern CPU, sin and cos operations can cost hundreds of CPU cycles, while `mul` and `add` usually cost less than 1 cycle. We use trigonometric identities to reduce the number of expensive sin and cos computations. For example, using $\sin(a-b) = \sin(a)\cos(b) - \cos(a)\sin(b)$ and $\cos(a-b) = \cos(a)\cos(b) + \sin(a)\sin(b)$ reduces the number of sin and cos operations from the number of branches to the number of buses. We store the $\sin(\theta)$,$\cos(\theta)$ values adjacent to the θ to exploit the data locality for better cache performance (as shown in upper part of Fig. 4). For the actual *sin* and *cos* computation, instead of using these functions in `libm`, we use an alternative high performance implementation in [19], which is especially suitable for extending to SIMD instructions for our MCS applications.

In the baseline code, the sparse L and U factors are stored in compressed column storage (CCS) format. During the substitution step of linear solver computation on CCS format sparse matrix, each data value and its row index are accessed at the same time. Storing the data value and row index consecutively as shown in the lower part of Fig. 4: the new mixed CCS format can exploit the data locality and improve the cache performance.

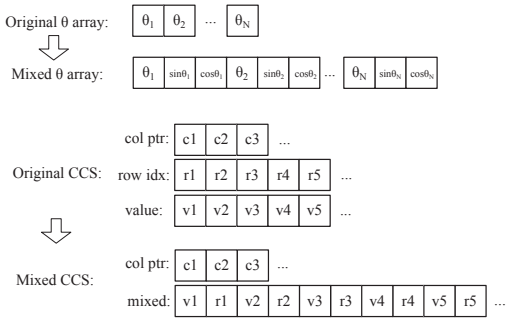

Figure 12: Optimizing data structure (upper: new θ array, lower: new CCS)

Unrolling Sparse Solver. In the most sparse solvers, the traversal over the sparse matrix is guided by nested loops, for example, the upper part of Fig. 13 shows the traversal on sparse matrix of compressed column storage format. The nested loops with only a few operations in the body result in unpredictable branches and limit the out-of-order execution and instruction reordering, hamper efficient register allocation and instruction scheduling [4]. In order to optimize the performance of sparse solver at the instruction level, we employ aggressive loop unrolling to combine consecutive columns into a bigger non-looping, non-branching code blocks. Since we use CCS format, the code for unrolled column is determined by the column size. We *pre-generate* multiple unrolled code blocks to cover various most common combinations of consecutive column sizes in sparse kernel computation, and use a `switch/case` statement to build a jump table dispatching instructions to different cases of these unrolled non-branching blocks. A similar technique for an accelerated SAT solver is in [6].

For example in Fig. 13, assume column i with 2 nonzeros is followed by column $i+1$ with 3 nonzeros. Instead of branching on every nonzero in the nested for-loops, we can pre-generate a bigger non-branching code block for the two columns of size 2 and 3 and place the code block in a `case` statement dispatched by the `case_pattern` 2 by 3. Based on this principle, we can generate different code blocks to unroll 1, 2 (this example in Fig. 13) or even more consecutive columns used in the sparse kernel computation.

```
for (col = 0; col < n; col++){
    for (row = col_ptr[col]; row < col_ptr[col+1]; row++){
        ...// access & compute on nonzero at (row, col)
    }
}
```

Unrolling

```
do{
    switch (case_pattern for 2 consecutive columns){
        case ...
        case pattern(2,3): {
            ...// access & compute on nonzero at (1, i)
            ...// access & compute on nonzero at (2, i)
            ...// access & compute on nonzero at (1, i+1)
            ...// access & compute on nonzero at (2, i+1)
            ...// access & compute on nonzero at (3, i+1)
            break;}
        case ...
    }
}while(!all columns visited)
```

Figure 13: Pseudo-code illustrating the loop unrolling in sparse matrix solver

Note all the `case` statements are *pre-generated* into source file. It increases the code size and the compiling time. In the runtime, only an extra sparse matrix analysis function which prepares the `case_pattern` for consecutive matrix column block is invoked once for each sparse matrix before all computation, the time is negligible comparing to MCS power flow computations. During the sparse kernel computation, instructions are dispatched to the bigger non-branching code blocks in the compiled `case` statements, which results in much better performance on modern superscalar out-of-order CPUs comparing to the code using nested loops.

4.2 Multilevel Explicit Parallelism

In this part, we directly use the SIMD approach and multicore scheduler in Section 3.2 to exploit the data level and task level parallelism for real time MCS application for transmission PLF. The parallelism stcuture and implementations are similar to Fig. 6 and Fig. 7.

4.3 Runtime Performance Results

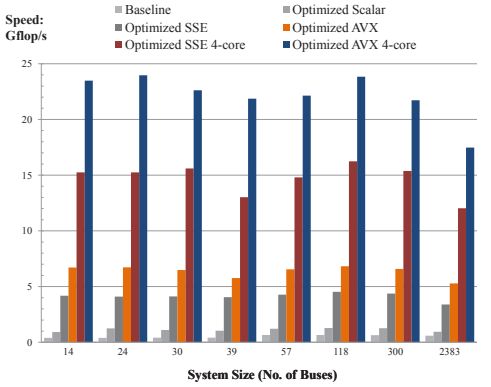

Figure 14: Optimization impact on computational speed (on Core i7 2670QM)

The performance results on a quadcore Core i7 2670QM is shown in Fig. 14, the *Baseline* code is the single thread scalar code using sparse LU factors and standard solver in SuiteSparse [10]. The *Optimized Scalar* code employs the data structure optimization and code unrolling. The *SSE* and *AVX* code are based on optimized scalar code and use SSE or AVX instructions. The *4-Core* versions run the SSE or AVX code on all CPU cores. All the codes are compiled by Intel C Compiler (`icc`) with optimization flag O3. The fully optimized code can achieve almost 50x speedup comparing to the compiler optimized baseline.

Table 2: Approximate Speed: Load Flow Cases Solved per Second on Core i7 2670QM

Test Cases		Approx. Speed [cases/s]	
Bus No.	Flops/Iteration	Baseline[1]	AVX 4-Core
14	1,034	39,000	2,270,000
24	1,788	23,000	1,340,000
30	2,242	19,000	1,010,000
39	2,715	23,000	805,000
57	4,467	15,000	495,000
118	9,130	7,000	261,000
300	23,370	3,000	92,900
2,383	175,365	340	9,960

1. **Baseline** is compiler optimized (`icc` & O3).

In terms of load flow cases solved per second, we fixed the iteration number to 10 to estimate a lower bound of the speed. Actual load flow cases in MCS would require less iterations since previous result can be used as the new initial guess. In Table 2, with the optimized MCS solver (AVX 4-Core), 200K load flow cases of the IEEE 118-bus system can be solved within a second on a Core i7 2670QM. 200K samples can achieve accurate converged PDF results for most PLF applications. Therefore, our optimized MCS solver enables real time, generally applicable, robust and accurate PLF solution for mid-size transmission grid.

5. SIMD ACCELERATED ACCC

In this part, we presented an accelerated AC contingency calculation (ACCC) solver. At algorithm level, we use Woodbury matrix identity within fast decoupled power flow algorithm to formulate a fine grain data parallel implementation of ACCC, which is especially suitable to be deployed onto modern CPU with SIMD instruction extension. At the architecture level, we applies aggressive code optimization for memory hierarchy, parallelization and thread pool based task scheduling. As results, we show our solver is able to solve the full contingency screening of a Polish 3120-bus system around 1 second on a quadcore Sandy Bridge CPU. It enables real time AC contingency analysis on commodity computing system.

5.1 Code Optimization

We use the fast decoupled power flow (FDPF) algorithm as the base load flow algorithm for ACCC screening. We applied the same optimization for the basic computing kernel showed in Section 4.1. Starting from the optimized FDPF computing kernel, in the following paragraph, we investigated and implemented the special transformation and optimization of ACCC to transform network outages into multilevel parallel program model to fully utilize the computing capability of modern CPUs.

5.2 Network Outages in FDPF Algorithm

The main difference between transmission PLF in Section 4 and the AC contingency calculation (ACCC) is that the ACCC need to consider the outages. In the power flow equations, the outage will change the structure of the network

and the structure of the power flow equations. However, we can still decompose the FDPF algorithm in to different steps, and use compensation based method to enable fine grain *data level parallelism* for most computations. The following paragraph shows an example of line outages:

Line outage cases. In the line outage cases, suppose the failed line section from bus i to bus j is taken out of the system. As a result, a 2×2 matrix Δy is added to corresponding slot of original admittance matrix $\mathbf{Y}$ to form the new admittance matrix $\tilde{\mathbf{Y}}$. We use a M matrix to indicate the location of the outage line in the $\mathbf{Y}$ matrix.

$$M = \begin{bmatrix} 0, ..., \underset{i}{1}, 0...0, 0, ..., 0 \\ 0, ..., 0, 0...0, \underset{j}{1}, ..., 0 \end{bmatrix} \tag{7}$$

$$\tilde{\mathbf{Y}} = \mathbf{Y} + M \Delta y M^T \tag{8}$$

$$\Delta y = \begin{bmatrix} y_{ij} + b_{ij} & -y_{ij} \\ -y_{ij} & y_{ij} + b_{ij} \end{bmatrix} \tag{9}$$

In FDPF computation, this will affect the mismatch computation which use the admittance matrix in the matrix vector product. One can simply compensate the $\mathbf{Y}$ to $\tilde{\mathbf{Y}}$ to compute the mismatch.

It will also affect the linear solver for (3) and (4) with similar modification on B' and B'' matrix:

$$\tilde{B'} = B' + M' \Delta b' M'^T \tag{10}$$

$$\tilde{B''} = B'' + M'' \Delta b'' M''^T \tag{11}$$

5.3 Data Parallelism of ACCC

Given the modifications on the FDPF algorithm for ACCC, transform FDPF based ACCC into data level parallelism is based on the *Woodbury matrix identity* (also called *Compensation method* in circuit simulation). Suppose:

$$\tilde{A} = A + MaN^T \tag{12}$$

The inverse of $\tilde{A}$ is

$$\tilde{A}^{-1} = A^{-1} - A^{-1}M(a^{-1} + N^T A^{-1} M)^{-1} N^T A^{-1} \tag{13}$$

Based on above formula, we can compute the contingency using the base case LU factors with minimal extra computation. Take B' in (3) as example, suppose B' is pre-factorized:

$$B' = L'U' \tag{14}$$

Handling line outage: In the base case, we need to solve x for $B'x = b$ in each iteration. While in the line outage cases, we need to solve x for modified B':

$$(B' + M' \Delta b' M'^T)x = b \tag{15}$$

Based on *Woodbury matrix identity*, the solution for (15) can be formulated as following steps:

Forward substitution:

$$F = L'^{-1} b \tag{16}$$

Compensate:

$$W = L'^{-1} M' \tag{17}$$

$$\tilde{W}^T = M'^T U'^{-1} \tag{18}$$

$$c = (\Delta b^{-1} + \tilde{W}^T W)^{-1} \tag{19}$$

$$\Delta F = -W c \tilde{W}^T F \tag{20}$$

$$F = F + \Delta F \tag{21}$$

Backward substitution:

$$\tilde{x} = U'^{-1} \tilde{F} \tag{22}$$

Note in compensation steps (17) to (20), the parameters and matrices can be determined by the new system topology, therefore, these compensation matrices can be pre-computed before the ACCC. Also W and W^T which have same dimension as M' and M'^T, and with a proper ordering scheme, these two matrices can be sparse with small floating-point operation numbers and memory footprint.

Based on similar idea, other types of contingency such as PV bus outage can also be solved in the same way. The details are in [1] and [5].

From above compensation method, the contingency cases can be decomposed into following types of operation:

1) Pre-computation of LU factors for base case and compensation matrices for different contingencies
2) Fixed mismatch calculation using slightly changed admittance matrix
3) Fixed forward / backward substitution for all cases
4) Compensation steps for different contingencies

The fixed mismatch calculation in step 2 uses same instruction sequence but may use slightly different value in admittance matrix if there is line outage. The fixed forward / backward substitutions in step 3 use same L and U factors therefore are the same instruction sequence all contingency cases. Only the compensation steps are different for different contingency cases. With above decomposition of computing procedure, we can see most part of the different contingency calculation cases can be transformed into program model that uses the same instructions sequences, therefore, ACCC be well mapped on to finer grain parallelism: the original scalar operations in step 2) and step 3) can be transformed into SIMD instruction allowing forward/backward substitution and mismatch computation to be performance on multiple cases simultaneously, with the step 4) compensates the effects of different contingency cases. Step 1) can be precomputed before all online contingency analysis since it only relates to the topologies.

The SIMD model for different contingency calculation is showed in Fig. 15. The upper part of the figure shows original the scalar version code on CPU's floating-point unit: the contingency cases are evaluated sequentially. The lower part shows the SIMD version code using CPU's SIMD units:

the forward/backward substitution parts of linear solver, the mismatch computation are vectorized and 4 cases (on SSE) or 8 cases (on AVX) cases are processed simultaneously on SIMD units, while the compensation for different cases are evaluated using pre-computed compensation matrices and then are plugged into the corresponding slots in SIMD units.

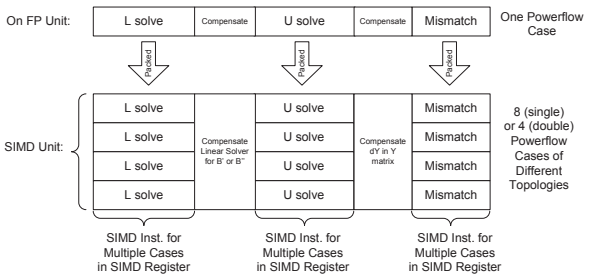

Figure 15: Scalar (upper) and SIMD (lower) model.

5.4 Load Balance via Thread Pool Scheduler

Load balancing is one of the most important considerations for parallel programming. In our ACCC application, we deal with the load balancing at the core level in a shared memory system: distributing and balancing the workload among multiple CPU cores to fully utilize the computing resources for ACCC computation.

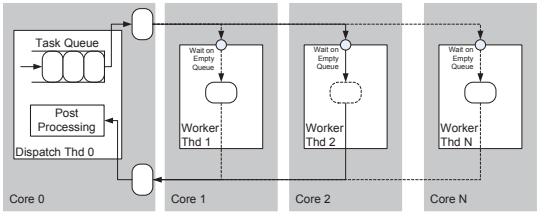

Figure 16: Thread pool scheduler on multi-core

We implemented a thread pool based scheduler for our ACCC application. As showed in Fig. 16, a pool of worker threads (Worker Thd 1 to N) are created and pinned to Core 1 to Core N to process the SIMD packed AC contingency computation tasks. One dispatch thread (Dispatch Thd 0) is created and pinned to Core 0 to manage the task queue and dispatch the work tasks into the thread pool, as well as post processing the ACCC results. Worker threads wait if the queue is empty, otherwise pop the task from the task queue to process using the SIMD data parallel solver in Section 5.3. Dispatch thread keeps dispatches the task into the task queue, once all tasks are dispatched, wait on the queue status. When the queue is empty, finish and clean up.

In this way, whenever any worker thread finishes the tasks and the queue is not empty, the worker will get new task from the queue. In our ACCC application, there are usually a large amount of small tasks, the loads can be dynamically balanced among worker threads on different physical cores by this thread pool design.

5.5 Results: Data Parallel on Single Core

Fig. 17 shows the performance breakdown of the data parallel implementation of ACCC solver on a single CPU core for different test systems (include IEEE standard test system from 14-bus to 300-bus and Polish grid of 2383 buses and 3120 buses). The performance results are showed in terms of floating point per seconds. The base algorithm is the FDPF load flow algorithm with AMD based sparse LU factors. The lowest bar is the baseline implementations directly using sparse kernel from CXSparse package in SuiteSparse [10]. The second lowest bar is the optimized scalar implementation with the optimization techniques on sparse kernel and math functions discussed in Section 4.1. Based on the optimized scalar implementation, the third bar shows the speed results of SIMD implementation using SSE instruction extensions which are available on most x86 CPUs. Using SSE, our accelerated implementation processes packed 4 single precision floating point data at a same time, a close to linear speedup can be observed. The highest bar is SIMD implementation using AVX instruction extensions available on Intel Sandy Bridge CPU since 2012. Using AVX, we pack 8 single precision floating point data and process the packed data using AVX instruction. Another speedup can be observed. Also we observed that the speedup increases with the increase of system size. Since the compensation steps in the middle are sparse vector / matrix with the size determined by the outage parts. For bigger system, the percentage of computation on compensation parts is smaller, more computation can be transformed onto SIMD model, a higher speedup number is expected.

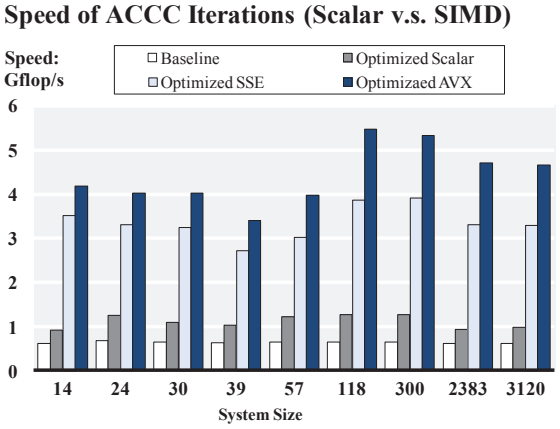

Figure 17: Speedup result by SIMD transformation

5.6 Results: Task Parallel on Multiple Cores

In this section, we show the results of thread pool scheduler on multiple cores for the accelerated ACCC application.

In order to show the benefit of thread pool for practical applications, we test a mid-sized power system case: the Polish grid 2383-bus system. The test cases are available in Matpower test cases [20]. We test the N-1 cases with rotating indexes, that is, the ACCC keeps running, whenever it solves the last contingency cases, it immediately begin to solve the first case again. In this way, the ACCC is assessing the security and taking the immediate varying grid condition into consideration.

Fig. 18 shows the results in terms of how many contingency cases can be solved every second for the Polish Grid. We show the test results on two machines, the darker bars are

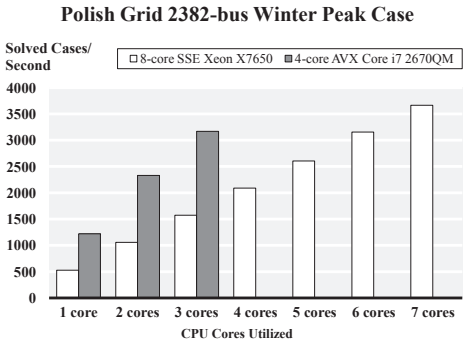

Figure 18: Thread pool performance of Polish 2383-bus system on different machines

the results on a quadcore 2.2GHz Intel Core i7 2670QM Sandy Bridge CPU supporting AVX instructions. The lighter bars are the results of a 8-core 2.26GHz Intel Xeon X7560 Nehalem CPU supporting SSE 4.1 instructions. On each CPU, one thread (one core) is reserved for scheduler and post-processing, therefore, the maximal available core number on these two machines are 3 and 7. In both tests on both machines, we observed a linear speedup for ACCC with the increased core numbers, thanks to the dynamic balance of thread pool design. Also, the 4-core machine is able to achieve higher performance thanks to the AVX capability with wider SIMD processing capability. From Fig. 18, our ACCC is able to complete a complete N-1 screen for the Polish grid on these two CPUs around a second. Therefore, it enables ACCC as an real time application for the real world mid-sized national level power grid to accommodate the fast varying grid conditions and help ensure the system security for the future smart power grid.

6. CONCLUSION

Given the new challenges and opportunities in both power system and computing performance engineering fields, this paper presented the contributions targeting the most fundamental and critical applications for power system probabilistic and security analysis including distribution probabilistic load flow, transmission probabilistic load flow and AC contingency calculation on commodity computing systems. By fully utilizing the computing power of commodity high performance computing system, we presented several unique solutions the new power grid challenges.

7. REFERENCES

[1] O. Alsac, B. Stott, and W. Tinney. Sparsity-oriented compensation methods for modified network solutions. *Power Apparatus and Systems, IEEE Transactions on*, (5):1050–1060, 1983.

[2] P. Amestoy, T. Davis, and I. Duff. Algorithm 837: AMD, an approximate minimum degree ordering algorithm. *ACM Transactions on Mathematical Software (TOMS)*, 30(3):381–388, 2004.

[3] M. Belgin, G. Back, and C. J. Ribbens. Pattern-based sparse matrix representation for memory-efficient smvm kernels. In *Proceedings of the 23rd international conference on Supercomputing*, ICS '09, pages 100–109, New York, NY, USA, 2009. ACM.

[4] S. Chellappa, F. Franchetti, and M. Püschel. How to write fast numerical code: A small introduction. *Generative and Transformational Techniques in Software Engineering II*, pages 196–259, 2008.

[5] T. Cui. *Power System Probabilistic and Security Analysis on Commodity High Performance Computing Systems*. PhD thesis, Carnegie Mellon University, 2013.

[6] T. Cui and F. Franchetti. Autotuning a random walk boolean satisfiability solver. *Procedia Computer Science*, 4:2176–2185, 2011.

[7] T. Cui and F. Franchetti. A multi-core high performance computing framework for distribution power flow. In *North American Power Symposium (NAPS), 2011*, pages 1–5. IEEE, 2011.

[8] T. Cui and F. Franchetti. A multi-core high performance computing framework for probabilistic solutions of distribution systems. In *Power and Energy Society General Meeting, 2012 IEEE*, pages 1–6, 2012.

[9] T. Cui and F. Franchetti. Optimized parallel distribution load flow solver on commodity multi-core cpu. In *High Performance Extreme Computing (HPEC), 2012 IEEE Conference on*, pages 1–6. IEEE, 2012.

[10] T. Davis, I. Duff, P. Amestoy, J. Gilbert, S. Larimore, E. P. Natarajan, Y. Chen, W. Hager, and S. Rajamanickam. Suite Sparse: a suite of sparse matrix packages.

[11] IEEE PES Distribution System Analysis Subcommittee. Distribution test feeders. `http://ewh.ieee.org/soc/pes/dsacom/testfeeders/index.html`.

[12] Intel Corporation. Intel®64 and ia-32 architectures optimization reference manual. `http://www.intel.com/content/www/us/en/architecture-and-technology/64-ia-32-architectures-optimization-manual.html`.

[13] Intel Corporation. Intel®microprocessor export compliance metrics. `http://www.intel.com/support/processors/sb/cs-017346.htm`.

[14] W. Kersting. *Distribution system modeling and analysis*. CRC, 2006.

[15] J. Larus. Spending Moore's dividend. *Commun. ACM*, 52:62–69, May 2009.

[16] H. Meuer, E. Strohmaier, J. Dongarra, and H. Simon. Top 500 list. `http://www.top500.org`.

[17] NERC. Transmission System Standards – Normal and Emergency Conditions.

[18] NERC. Special report: Accommodating high levels of variable generation, 2009.

[19] N. Shibata. Efficient evaluation methods of elementary functions suitable for simd computation. *Computer Science-Research and Development*, 25(1-2):25–32, 2010.

[20] R. D. Zimmerman, C. E. Murillo-Sánchez, and R. J. Thomas. Matpower: Steady-state operations, planning, and analysis tools for power systems research and education. *Power Systems, IEEE Transactions on*, 26(1):12–19, 2011.