# Accelerated AC Contingency Calculation on Commodity Multi-core SIMD CPUs

Tao Cui, *Student Member, IEEE,* Rui Yang, *Student Member, IEEE,* Gabriela Hug, *Member, IEEE,* Franz Franchetti, *Member, IEEE*

***Abstract*—Multi-core CPUs with multiple levels of parallelism (i.e. data level, instruction level and task/core level) have become the mainstream CPUs for commodity computing systems. Based on the multi-core CPUs, in this paper we developed a high performance computing framework for AC contingency calculation (ACCC) to fully utilize the computing power of commodity systems for online and real time applications. Using Woodbury matrix identity based compensation method, we transform and pack multiple contingency cases of different outages into a fine grained vectorized data parallel programming model. We implement the data parallel programming model using SIMD instruction extension on x86 CPUs, therefore, fully taking advantages of the CPU core with SIMD floating point capability. We also implement a thread pool scheduler for ACCC on multi-core CPUs which automatically balances the computing loads across CPU cores to fully utilize the multi-core capability. We test the ACCC solver on the IEEE test systems and on the Polish 3000-bus system using a quad-core Intel Sandy Bridge CPU. The optimized ACCC solver achieves close to linear speedup (SIMD width multiply core numbers) comparing to scalar implementation and is able to solve a complete N-1 line outage AC contingency calculation of the Polish grid within one second on a commodity CPU. It enables the complete ACCC as a real-time application on commodity computing systems.**

## I. Introduction

AC contingency calculation (ACCC) is the fundamental tool for power system steady state security assessment. It evaluates the consequences of power grid component failures, assesses the system security given the failures and further helps to provide corrective or preventive actions for decision making. ACCC is a basic module for most offline power system planning tools [1]. It is also a critical functionality of most SCADA/EMS systems. ACCC has also been widely used in market applications, such as the simultaneous feasibility test for market dispatch security [2]. As a complete ACCC computation on a practical power grid often requires a large amount of load flow computations resulting in a large computational burden, the complete ACCC often remains as offline / non real-time applications on commodity computing systems.

Recent large scale integrations of variable renewable generation and large variance in load (e.g. electric vehicle charging) introduce significant uncertainties and largely-varying grid conditions. Moreover, the increasing loads and generations drive today's power grid closer to its limits, resulting in higher possibility of contingencies as well as more serious consequences. The largely-varying grid conditions on the already stressed power grid require merging of most conventional offline analyses into online even real-time operation to satisfy the unprecedented security requirements. Therefore, an optimized ACCC solver for online and real-time operation would be an important tool given the new security assessment challenges.

This work was supported by NSF through awards 0931978 and 1116802.

In the computing industry, the performance capability of the computing platform has been growing rapidly in the last several decades at a roughly exponential rate [3]. The recent mainstream commodity CPUs enable us to build inexpensive computing systems with similar computational power as the supercomputers just ten years ago. However, these advances in hardware performance result from the increasing complexity of the computer architecture and they actually increase the difficulty of fully utilizing the available computational power for a specific application [4]. This paper focuses on *fully* utilizing the computing power of modern CPUs by code optimization and parallelization for specific hardware, enabling the real-time complete ACCC application for practical power grids on commodity computing systems.

**Related work**. Contingency analysis has long since been a joint research field of both power system and high performance computing domains [5] [6]. In [7], a workload balancing method is developed for massive contingency calculation on Cray supercomputer. In [8], a hybrid approach is proposed using Cray XMT's graphical processing capability. A graphical processing unit based DC contingency analysis has been implemented in [9]. Recently, some commercial packages such as PSS/E also work actively to include parallel processing on multi-core CPU to boost the performance of ACCC [1]. However, most previous approaches are focusing on task level parallelism, or using specified parallel math solvers. In order to fully utilize the computing power of a commodity CPU with multi-level parallel hardware and other performance enhancement features, specific algorithmic transforms and code optimizations for ACCC are presented in this paper.

**Contribution**. This paper presents an accelerated ACCC that builds upon several algorithmic and computer architectural optimizations. At the data level, we use Woodbury matrix identity with fast decoupled power flow algorithm to formulate a fine grain vectorized implementation of ACCC. It solves multiple cases simultaneously using SIMD (Single Instruction Multiple Data) floating point units of a single CPU core. At the task/core level, we implement a thread pool scheduler on multi-core CPUs that automatically balances the computing loads across multiple CPU cores. Together with some other aggressive code optimizations for data structure, sparse kernels and instruction level parallelism, our solver is able to complete a full line outages ACCC of the Polish 3120-bus system within a second. It enables real-time complete ACCC for a practical grid on inexpensive computing systems.

**Synopsis.** The paper is organized as follows: the feature of computing platforms are reviewed in Section II. The SIMD transformation of ACCC is described in Section III. The multi-core and other optimizations are described in Section IV. We report the performance results of the optimized solver in Section V. Section VI concludes the paper.

978-1-4799-6415-4/14/$31.00 ©2014 IEEE

## II. Modern Computing Platform

Fig. 1 shows the structure of the quadcore Intel Core i7 2670 QM Sandy Bridge CPU for mid-range laptops. It has 4 physical cores (Core P#0-4), three levels (L1-L3) of CPU cache memories. It also supports Intel's new AVX (Advanced Vector eXtension) instruction set for single instruction multiple data (SIMD) operations. It has a clock rate of 2.2 GHz. The theoretical single precision peak performance is 140 Gflop/s (or Gflops, $10^9$ floating point operations per second) [10]. In terms of this value, this performance oriented CPU in 2012 has the similar performance as the top supercomputers in the world in just year 2001 (Cray T3E1200, 138 Gflop/s on Top500 List) [3]. However, the peak hardware performance assumes that one can *fully* utilize all the performance enhancement features of the CPU, which becomes very difficult on modern CPUs given the more and more complicated hardware. We mainly look into the following hardware aspects explicitly available to software development:

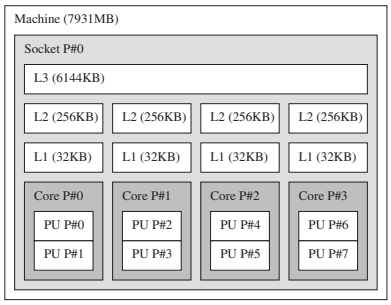


Fig. 1. Core i7 2670QM CPU system structure: 4-core, 3-level cache

**Multiple levels of parallelism** have become the major driving force for the hardware performance. The followings are two types of explicit parallelism on modern CPU:

1) Single Instruction Multiple Data (SIMD) uses vector instructions and registers to perform the same operation on multiple data at the same time: The Streaming SIMD Extensions (SSE) or the new Advanced Vector eXtensions (AVX) instruction sets on Intel or AMD CPUs perform floating point operations on 4 floating point or 8 floating point data packed in vector registers at the same time (single precision). As illustrated in Fig. 2 for example, the scalar `fadd` performs add operation on one data slot, the SSE version `addps` or AVX version `vaddps` instruction performs the add operation on four or eight data slots simultaneously. Many new or under-

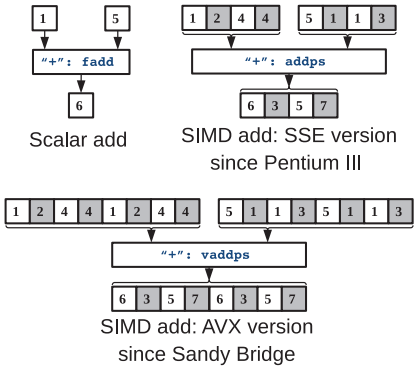


Fig. 2. Illustration of scalar add versus SIMD add

developing micro-architectures such as Intel's new Xeon Phi architecture further expands the SIMD processing width to 16 data. With the help of SIMD, the performance can be significantly increased for particularly formed problems.

2) Multithreading on multi-core CPUs enables multiple threads to be executed simultaneously and independently on different CPU cores while communicating via shared memories. A proper scheduling and load balancing strategy is necessary to fully utilize multiple CPU cores.

**Memory hierarchy**, e.g. multiple levels of caches, should also be considered for performance tuning. 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 (L1, L2, L3 in Fig. 1), the levels closer to CPU cores are faster in speed but smaller in size. An optimized data storage and access pattern is important to utilize the caches to increase the performance.

Given the hardware features, proper parallelization model and code optimization techniques are of crucial importance to *fully* utilize the computing power for ACCC performance. In the following sections, we show the proposed approaches and the benefits when applying performance tuning and parallel programming models on modern CPU hardware for ACCC.

## III. Mapping Contingencies to Data Parallelism

This section describes the proposed data parallel model that can process multiple contingency cases simultaneously on a single CPU core. The main idea is to transform most parts of the ACCC computation into the *same* and *fixed* instruction sequence on *different* data for different contingencies to utilize the CPU's SIMD capability (e.g. the SIMD add in Fig. 2).

### A. Base algorithm: fast decoupled power flow

The widely used fast decoupled power flow (FDPF) algorithm is the base algorithm for our ACCC. FDPF is originated from full Newton-Raphson (NR) method, by considering some unique properties of the transmission network. FDPF method often has fewer total floating point operation counts to the NR method [12]. In each iteration of FDPF, the following equations are solved consecutively to update the state $(\theta, \mathbf{V})$. Suppose the state of the current iteration is $(\theta^k, \mathbf{V}^{(k)})$, the state of the next iteration $(\theta^{(k+1)}, \mathbf{V}^{(k+1)})$ is computed as:

$$\begin{aligned} \mathbf{\Delta V}^{(k)} &= -\mathbf{B''}^{-1}\mathbf{\Delta Q}(\theta^{(k)}, \mathbf{V}^{(k)})/\mathbf{V}^{(k)} \\ \mathbf{V}^{(k+1)} &= \mathbf{V}^{(k)} + \mathbf{\Delta V}^{(k)} \end{aligned} \tag{1}$$

$$\begin{aligned} \mathbf{\Delta}\theta^{(k)} &= -\mathbf{B'}^{-1}\mathbf{\Delta P}(\theta^{(k)}, \mathbf{V}^{(k+1)})/\mathbf{V}^{(k+1)} \\ \theta^{(k+1)} &= \theta^{(k)} + \mathbf{\Delta}\theta^{(k)} \end{aligned} \tag{2}$$

In equation (1) and (2), $\mathbf{\Delta Q}(\cdot)$ and $\mathbf{\Delta P}(\cdot)$ compute the power mismatch by evaluating power flow equations via matrix-vector product styled operations.

Two linear systems of $\mathbf{B'}$ and $\mathbf{B''}$ (the inverse in (1)(2) ) are solved to obtain the adjustments $\Delta\theta$ and $\Delta\mathbf{V}$. Using a direct linear solver, the $\mathbf{B'}$ and $\mathbf{B''}$ are pre-factorized into the product of lower triangle ($L$) and upper triangle ($U$) matrices before

the iteration: $\mathbf{B}' = L'U'$,$\mathbf{B}'' = L''U''$. During the iteration, only forward and backward substitutions using pre-computed LU factors are needed to solve the linear systems.

### B. ACCC by modifying base case

The core computation of ACCC is to solve multiple cases of AC power flow given different component failures. These contingency cases can be considered as power flow base case (without failure) plus different modifications on the equation and/or parameters to consider contingencies. Following shows the typical modifications:

**Line outage cases**. In these cases, the system parameters stay unchanged, suppose the failed line section from bus $i$ to bus $j$ is taken out of the system. As a result, a $2\times2$ matrix $\Delta y$ is added to the corresponding slots of the original admittance matrix $Y$ to form the new admittance matrix $\tilde{Y}$.

$$M = \begin{bmatrix} 0, ..., \underset{i}{1}, 0...0, 0, ..., 0 \\ 0, ..., 0, 0...0, \underset{j}{1}, ..., 0 \end{bmatrix}^T \tag{3}$$

$$\tilde{Y} = Y + M\Delta y M^T \tag{4}$$

$$\Delta y = \begin{bmatrix} y_{ij} + b_{ij} & -y_{ij} \\ -y_{ij} & y_{ij} + b_{ij} \end{bmatrix} \tag{5}$$

In FDPF, this affects the mismatch computation with the new $\tilde{Y}$ as well as the linear solver with the new $\tilde{\mathbf{B}}'$ and $\tilde{\mathbf{B}}''$:

$$\tilde{\mathbf{B}}' = \mathbf{B}' + M'\Delta b' M'^T \tag{6}$$

$$\tilde{\mathbf{B}}'' = \mathbf{B}'' + M''\Delta b'' M''^T \tag{7}$$

**PV bus outage cases**. Such a case happens when the generator fails to maintain the voltage of a PV bus. The PV bus changes to a PQ bus. The $Y$ matrix does not change. Another equation for the new PQ bus' Q and $V$ is added to the power flow equation set, resulting in another column and row added to the bottom and right of $B''$:

$$\tilde{\mathbf{B}}'' = \begin{bmatrix} \mathbf{B}'' & N \\ N^T & a \end{bmatrix} \tag{8}$$

**Generator outage without PV bus outage cases**. These cases only affect the specified active power injection value on PV bus, without modifying the power flow equation system.

The main idea is to transform most parts of the ACCC computation into the *same* instruction sequence on *different* data for different contingencies in order to use the CPU's SIMD hardware units. The mismatch computation part can be simply transformed by plugging-in different data in $Y$ matrix when necessary for different contingencies without changing the instruction sequence. The linear solver part is transformed by compensation method as follows:

### C. Data parallelism by compensation

Transforming linear solver parts of ACCC into data level parallelism is based on the *Woodbury matrix identity*. Suppose:

$$\tilde{A} = A + MaN^T \tag{9}$$

The inverse of $\tilde{A}$ is

$$\tilde{A}^{-1} = A^{-1} - A^{-1}M(a^{-1} + N^T A^{-1} M)^{-1} N^T A^{-1} \tag{10}$$

Numerical solution of $A^{-1}$ in (10) is the forward/backward substitutions using the LU factors of $A$. Hence (10) provides a way to solve the modified liner system $\tilde{A}$ using the LU factors of the base case $A$ plus some compensations. The $A$ in (10) can be $\mathbf{B}'$ or $\mathbf{B}''$ in FDPF, therefore, the linear system of ACCC can be solved using the same base case factorization with compensations [11].

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

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

Based on (10), the solution for (11) can be found as follows:
Step 1: forward substitution:

$$F = L'^{-1}b \tag{12}$$

Step 2: compensate:

$$W = L'^{-1}M' \tag{13}$$

$$\tilde{W}^T = M'^T U'^{-1} \tag{14}$$

$$c = (\Delta b^{-1} + \tilde{W}^T W)^{-1} \tag{15}$$

$$\Delta F = -Wc\tilde{W}^T F \tag{16}$$

$$F = F + \Delta F \tag{17}$$

Step 3: backward substitution:

$$\tilde{x} = U'^{-1}\tilde{F} \tag{18}$$

Note in the above compensation steps (13) to (16), the computation is only determined by the new system topology. Therefore, these compensation matrices can be pre-computed before the ACCC. Also $W$ and $\tilde{W}^T$ have the same dimension as $M'$ and $M'^T$, and with a proper ordering scheme, these two matrices can be sparse with small floating-point operation counts and memory footprints. Therefore, during the ACCC, (12) and (18) are fixed procedures for all cases. Different contingency cases can be computed using the compensation steps based on pre-computed $W$, $\tilde{W}^T$, $c$ and (16) and (17).

$\mathbf{B}''$ can be treated in a similar way for line outage cases.

**Handling PV bus outage**. We need to solve $\tilde{x}$ in the following linear system of $\tilde{\mathbf{B}}''$:

$$\tilde{\mathbf{B}}''\tilde{x} = \begin{bmatrix} \mathbf{B}'' & N \\ N^T & a' \end{bmatrix} \tilde{x} = \begin{bmatrix} b \\ b' \end{bmatrix} \tag{19}$$

Solving (19) can be decomposed as follows:
Step 1: solving $x_0$ in:

$$(\mathbf{B}'' - N(\frac{1}{a'})N^T)x_0 = b - (b'/a')N \tag{20}$$

Step 2: solve $\tilde{x}$:

$$x_1 = \frac{1}{a'}(b' - N^T x_0) \tag{21}$$

$$\tilde{x} = \begin{bmatrix} x_0 \\ x_1 \end{bmatrix} \tag{22}$$

Note (20) can be solved in the same way using compensation as we solve (11) based on the same $L''$ and $U''$ factors of the base case $\mathbf{B}''$ for different contingencies.

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

0) Pre-computation of the base case LU factors and the compensation matrices for different contingencies.
1) Fixed mismatch calculation using admittance matrices with modified values, e.g. set outage line's y to 0.
2) Fixed forward/backward substitutions for all cases.
3) Different compensation steps for different contingencies.

The *fixed* computation steps in the above list are mostly the same instruction sequence on different data. Only the compensation steps are different for different contingency cases to accurately consider the contingencies for the solution. With the above decomposition of the computing procedure, the *fixed* computation steps of the ACCC can be well mapped on to fine grain data level parallelism and can use SIMD instructions to perform the computation on multiple cases simultaneously.

### D. Programming model: SIMD parallelism on single core

The proposed SIMD model for ACCC is shown in Fig. 3 lower part. The upper part of the figure is the original scalar version code on CPU's floating-point unit: the contingency cases are evaluated sequentially. The lower part shows the proposed SIMD version code using CPU's SIMD units: the fixed forward/backward substitution of the linear solver and the mismatch computation are vectorized. 4 cases (on SSE) or 8 cases (on AVX) are processed simultaneously on SIMD units. The compensation steps for different cases are evaluated using pre-computed compensation matrices and then are plugged into the corresponding slots in SIMD units.

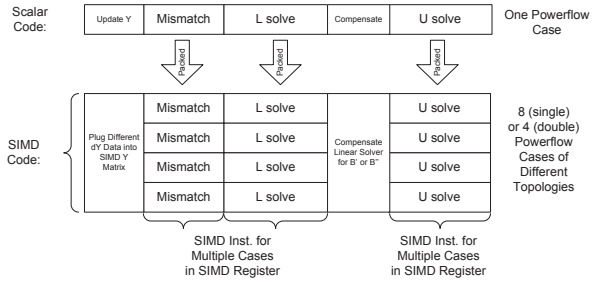


Fig. 3. Scalar and SIMD model of ACCC: iteration segment for (1) or (2)

## IV. Multi-core Task Scheduling and Others

### A. Task scheduling over multiple cores

Load balancing is one of the most important considerations for designing a parallel program. It is also one of the important targets of most ACCC research projects and commercial products [1] [7]. 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.

We implemented a thread pool based scheduler for the ACCC. As shown in Fig. 4, a pool of worker threads (Worker Thd 1 to N) are created and pinned to Core 1 to Core N to process the SIMD vectorized ACCC 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 process the ACCC results. The three elements of the thread pool scheduler design are the task queue data structure and the two types of threads:

1. Task queue data structure includes: a queue buffering the task pointers to the SIMD ACCC cases to be processed.
2. Worker thread: wait if the queue is empty, otherwise pop the task from the task queue and process the task using the SIMD ACCC solver.
3. Dispatch thread: keep dispatching 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 a new task from the queue. In our ACCC application, there are usually a large amount of small tasks and the loads can be dynamically balanced among worker threads on different physical cores.

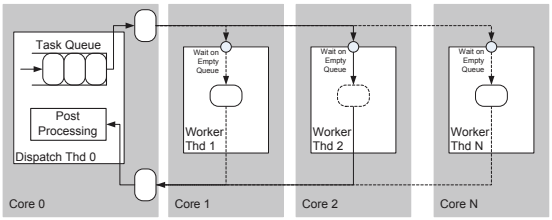


Fig. 4. Thread pool scheduler on multi-core CPU

### B. Other code optimization

Besides the above explicit parallelization, the following algorithm and code optimization techniques are applied for the FDPF computing kernel on x86 CPU.

1. Sparse LU factorization using approximated minimal degree scheme (AMD) for solving $\mathbf{B}'$ and $\mathbf{B}''$ [13].
2. Optimized usage of trigonometric functions: using optimized trigonometric functions to achieve same precision with smaller number of CPU cycles [14].
3. Unrolling sparse kernel, pre-generate source code for consecutive columns of the sparse L and U factors to build bigger code block and using jump table to avoid branching in the inner loop. Similar approaches on unrolling techniques are discussed in [15] [16] [17].

## V. Performance Results

All code tested was written in C and compiled by Intel C++ Compiler V12 with -O3 flag (full compiler optimization) on 64-bit Linux. The performance results are as follows:

Fig. 5 shows the performance breakdown of the data parallel implementation of our ACCC solver on a single CPU core for different test systems (including IEEE standard test systems from 14-bus to 300-bus and the Polish grid of 2383 buses and 3120 buses). The performance results are given in terms of Gflop/s. The base algorithm is the FDPF load flow algorithm with AMD based sparse LU factors. The first bar is the baseline implementations directly using sparse kernel from the CXSparse package in SuiteSparse [18]. The second bar is the optimized scalar implementation with the optimization techniques on sparse kernel, math functions and unrolling discussed in Section IV-B. Based on the optimized scalar implementation, the third bar shows the speed results of the 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 the same time and a close to linear speedup can be observed. The last bar is the SIMD implementation using

AVX instruction extensions available on Intel Sandy Bridge CPU since 2011. Using AVX, we pack 8 single precision floating point data and process the packed data using AVX instruction. Another speedup can be observed.

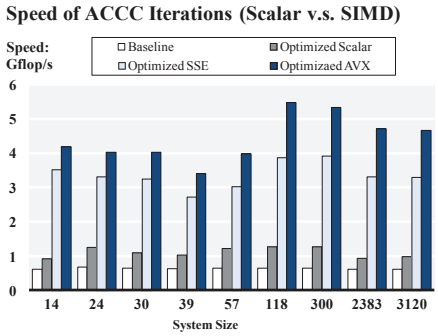


Fig. 5. Speedup result by SIMD transformation

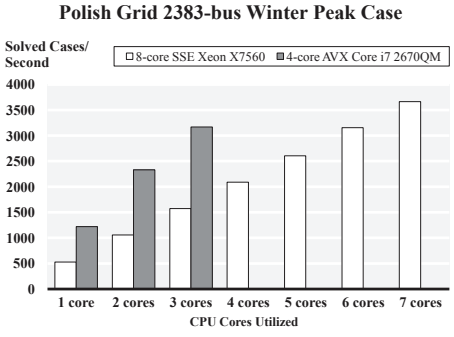


Fig. 6. Thread pool performance of Polish 2383-bus system

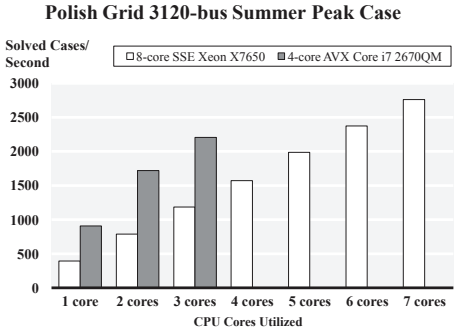


Fig. 7. Thread pool performance of Polish 3120-bus system

Fig. 6 and Fig. 7 shows the results in terms of how many contingency cases can be solved every second for the Polish grid. The maximal iteration is 20 and the maximal mismatch is 1kW. Most N-1 cases converge using FDPF. We show the test results on two machines, the darker bars are the results on a quad-core 2.2GHz Intel Core i7 2670QM Sandy Bridge CPU supporting AVX instructions. The lighter bars are the results of an 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, the rest cores are for load flow computations. In these tests on both machines, we observed a linear speedup for ACCC with the increased core numbers, thanks to the dynamic balance of the thread pool design. Also, the 4-core machine is able to achieve higher performance thanks to the AVX capability with wider SIMD processing width. In Fig. 6 and Fig. 7, our ACCC is able to finish a complete N-1 line outage screening for the Polish grid using each of these two CPUs in around a second. Therefore, our implementation enables ACCC as a real-time application for practical sized power grids to meet the new challenges in the future electric power grid.

## VI. Conclusion

In this paper we presented a multi-core high performance accelerated ACCC solver. By applying various performance optimizations and multi-level parallelization, especially the compensation based algorithm transformation, we transform the ACCC into a fine grained data parallel model. We also implemented a thread pool scheduler that can dynamically balance the work loads. The proposed ACCC fully utilizes the computing capability of the modern CPUs and the performance is scalable with the hardware parallel capacity. We tested the ACCC solver on the IEEE test systems as well as a real world national level Polish grid. Our ACCC solver is able to complete a full N-1 line outage screening of the Polish network within a second, enabling a complete ACCC solution for real-time operation on commodity computing systems.

## References

[1] Siemens-PTI, "PSS/E: Power system simulator for engineering."
[2] PJM, "Auction revenue rights," http://www.pjm.com/about-pjm/learning-center/markets-and-operations/arrs.aspx.
[3] H. Meuer, E. Strohmaier, J. Dongarra, and H. Simon, "Top 500 list," http://www.top500.org.
[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*, pp. 196–259, 2008.
[5] D. Falcão, "High performance computing in power system applications," *Vector and Parallel Processing – VECPAR'96*, pp. 1–23, 1997.
[6] Z. Huang and J. Nieplocha, "Transforming power grid operations via high performance computing," in *IEEE Power and Energy Society General Meeting, 2008*, Pittsburgh, PA, USA, July 2008.
[7] Z. Huang, Y. Chen, and J. Nieplocha, "Massive contingency analysis with high performance computing," in *IEEE Power and Energy Society General Meeting, 2009*, Calgary, AB, Canada, July 2009.
[8] I. Gorton, Z. Huang, Y. Chen, B. Kalahar, S. Jin, D. Chavarria-Miranda, D. Baxter, and J. Feo, "A high-performance hybrid computing approach to massive contingency analysis in the power grid," in *Fifth IEEE International Conference on e-Science, 2009.*, Dec. 2009, pp. 277–283.
[9] A. Gopal, D. Niebur, and S. Venkatasubramanian, "DC power flow based contingency analysis using graphics processing units," in *IEEE Power Tech, 2007*, Lausanne, Switzerland, July 2007, pp. 731 –736.
[10] Intel Corporation, "Intel®microprocessor export compliance metrics," http://www.intel.com/support/processors/sb/cs-017346.htm.
[11] O. Alsac, B. Stott, and W. Tinney, "Sparsity-oriented compensation methods for modified network solutions," *IEEE Transactions on Power Apparatus and Systems*, no. 5, pp. 1050–1060, 1983.
[12] B. Stott, "Review of load-flow calculation methods," *Proceedings of the IEEE*, vol. 62, no. 7, pp. 916 – 929, July 1974.
[13] P. Amestoy, T. Davis, and I. Duff, "Algorithm 837: AMD, an approximate minimum degree ordering algorithm," *ACM Transactions on Mathematical Software (TOMS)*, vol. 30, no. 3, pp. 381–388, 2004.
[14] N. Shibata, "Efficient evaluation methods of elementary functions suitable for simd computation," *Computer Science-Research and Development*, vol. 25, no. 1-2, pp. 25–32, 2010.
[15] T. Cui and F. Franchetti, "Autotuning a random walk boolean satisfiability solver," *Procedia Computer Science*, vol. 4, pp. 2176–2185, 2011.
[16] ——, "A multi-core high performance computing framework for probabilistic solutions of distribution systems," in *IEEE Power and Energy Society General Meeting, 2012*. San Diego, CA, USA, July 2012.
[17] ——, "Optimized parallel distribution load flow solver on commodity multi-core CPU," in *IEEE Conference on High Performance Extreme Computing, 2012*. Waltham, MA, USA, Sep 2012.
[18] T. Davis, I. Duff, P. Amestoy, J. Gilbert, S. Larimore, E. P. Natarajan, Y. Chen, W. Hager, and S. Rajamanickam, "SuiteSparse: a suite of sparse matrix packages."