Optimized Parallel Distribution Load Flow Solver on Commodity Multi-core CPU

Tao Cui, Franz Franchetti
Department of ECE., Carnegie Mellon Univeristy
Email: {tcui,franzf}@ece.cmu.edu

***Abstract*—Solving a large number of load flow problems quickly is required for Monte Carlo analysis and various power system problems, including long term steady state simulation, system benchmarking, among others. Due to the computational burden, such applications are considered to be time-consuming, and infeasible for online or realtime application. In this work we developed a high performance framework for high throughput distribution load flow computation, taking advantage of performance-enhancing features of multi-core CPUs and various code optimization techniques. We optimized data structures to better fit the memory hierarchy. We use the SPIRAL code generator to exploit inherent patterns of the load flow model through code specizlization. We use SIMD instructions and multithreading to parallelize our solver. Finally, we designed a Monte Carlo thread scheduling infrastructure to enable real time operation. The optimized solver is able to achieve more than 50% of peak performance on a Intel Core i7 CPU, which translates to solving millions of load flow problems within a second for IEEE 37 test feeder.**

I. Introduction

Load flow computation is an essential routine for power system analysis. The main purpose of load flow is to compute the system's complete state based on load and generation status. Load flow computation often lies in the critical path of most power system analysis programs. Solving a large number of load flow problems is required for various applications including Monte Carlo style analysis, long term steady state simulation, system benchmarking, among others. These applications are generally viewed to be computational intensive, time consuming and infeasible for online applications.

With the recent development of smart grid technologies like integration of renewable energy resources or plug-in electrical vehicles connected to the distribution system, there is a higher demand for computational performance for distribution system analysis. Fast load flow analysis for distribution system smart grid applications ensuring the system's efficiency, reliability, economics, and sustainability under various conditions are being actively developed [1], [2], [3]. The computing performance requirements (high throughput or realtime) make embedded high performance computing technology one of the most important enabling technologies for various smart grid applications in distribution networks.

In contrast to the transmission system load flow and general circuit simulation, the branch current based *Forward / Backward Sweep* (FBS) method is the most efficient method for radial distribution system analysis [1], [4]. In FBS method, the system is modelled as a tree, and the core computation of the FBS method is to iteratively traverse the tree in forward and backward order to update the unknown system state on each component (link or node) according to the circuit laws. Inside each iteration, on each link, small 3×3 complex matrix-vector multiplication is the main computation. The following properties of the distribution load flow computation allow us to specifically optimize the FBS for commodity multicore platforms: 1) most normal load flow cases converge at same iteration step, 2) for a fixed system, the system and data structures also have a fixed pattern, 3) there is a tradeoff between modeling convenience and computation efficiency, and 4) real time Monte Carlo benefits from a relaxed scheduling and threading model. In this work we are exploiting above mentioned features and developed a high performance computing framework to solve high throughput distribution load flow problems. Our target hardware is a mainstream commodity multi-core CPU with SIMD instructions. Such a platform provides performance in the hundred Gflop/s to Tflop/s range (flop/s is floating-point operation per second), and is affordable enough to be potentially deployed down to to distribution substation level.

Contribution. In [5] we presented an initial version of a high performance solver for distribution system load flow on multi-core CPUs. In this paper, we provide a highly optimized extension of the solver. In particular, we applied aggressive algorithm level optimization as well as code synthesis and specialization using the SPIRAL code generator. The contribution of this paper is two-fold: First, the optimized solver reaches more than 50% of a CPU's machine peak performance using the 8-way single-precision AVX SIMD instructions and the 4 cores of a SandyBridge CPU. This translates into about 50x speedup compared to the best compiler-optimized baseline code on the target platform. The performance is also scalable with the hardware's parallel capabilities (multiple cores and SIMD vector width). Second, we detail the necessary optimization techniques to provide such speed-up levels for power grid solvers. We apply algorithmic and data structure optimizations, code synthesis, explicit SIMD code generation, and low-overhead multi-threading.

Synopsis. The paper is organized as follows: In Section II we review the distribution network load flow algorithm and the software framework for our solver. In Section III we describe our performance optimization methods. In Section IV we report and analyze performance results of our optimized solver for IEEE test feeders. In Section V we discuss an example application. Section VI concludes the paper.

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

II. Algorithm and Implementation Considerations

Distribution system load flow algorithm. The distribution system has some distinctive properties: 1) high resistance to reactance ratio and the actual value of impedance can vary, 2) radial structure, 3) unbalanced and multi-phase (usually three phases). Therefore most of the distribution load flow cases cannot be solved well with transmission load flow and general circuit simulation algorithm. The branch current-based forward/backward sweep (FBS) method is one of the most efficient methods for distribution system load flow computation. We use the FBS method and the generalized multi-phase unbalanced feeder models outlined in [4] and [1].

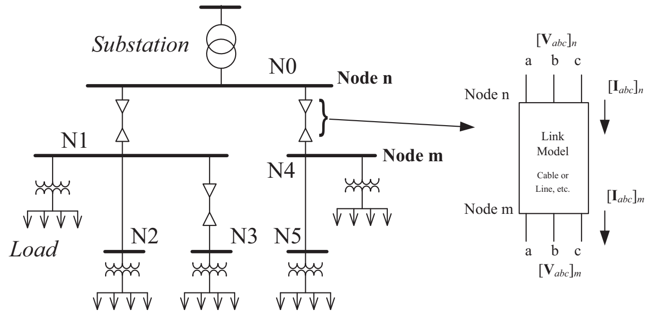


Fig. 1. Link model (left) and Single line diagram (right).

$$[\mathbf{I}_{abc}]_n = [\mathbf{c}]\,[\mathbf{V}_{abc}]_m + [\mathbf{d}]\,[\mathbf{I}_{abc}]_m \tag{1}$$

$$[\mathbf{V}_{abc}]_m = [\mathbf{A}]\,[\mathbf{V}_{abc}]_n - [\mathbf{B}]\,[\mathbf{I}_{abc}]_m \tag{2}$$

Fig. 1 shows the radial system and the link model for a branch connecting two nodes, $\mathbf{n}$ is the node closer to the substation, and the reference forward direction is from $\mathbf{n}$ to $\mathbf{m}$. The relations of the complex three phase voltages $[\mathbf{V}_{abc}]_n$ $[\mathbf{V}_{abc}]_m$ and complex three phase currents $[\mathbf{I}_{abc}]_n$ $[\mathbf{I}_{abc}]_m$ are given in (1) and (2), The 3×3 complex matrices $\mathbf{A}$, $\mathbf{B}$, $\mathbf{c}$, $\mathbf{d}$ are constant matrices derived from each equipment model [1].

In distribution network load flow model, the substation is the constant voltage source at the root of the tree. The electric load on each node (e.g. Node N1 to N5 in Fig. 1) usually have pre-specified power consumption. The load flow algorithm solves the three phase voltages on every node which are the states of the system. All other system quantities can be derived from the knowledge of the voltages.

The FBS method works as following: given an initial guess for the voltages, in each iteration first a *Backward Sweep* traverses from all the leaves to the root, using the *Kirchhoff Current Law* on each node and (1) on each branch to update currents. Next, a *Forward Sweep* traverses from root to all the leaves, using (2) to update voltages. The iterations consisting of backward and forward sweeps are repeated until each node's power mismatch is smaller than an error limit. Subsequent iterations use the results of the previous iteration as starting point for the backward sweep.

As we can see, the FBS method is highly related to the physical models. It is possible to trade off modelling accuracy and computational efficiency to achieve the necessary performance withing the required error bounds.

Implementation considerations. In this paper we focus on a high throughput scenario requiring a large number of load flow problems solved in real time. These problems have the following properties: 1) The system topology and component model are mostly unchanged for all computation cases. 2) The load on every node and other continuous value may be different across problems. 3) All the solved problems are mostly independent, however, they may have some shared data or logic that need to be scheduled and synchronized. Important power system applications share these properties, including as Monte Carlo simulation to solve probabilistic load flow, monthly, yearly steady state simulation and some system benchmarking considering varying load and generation.

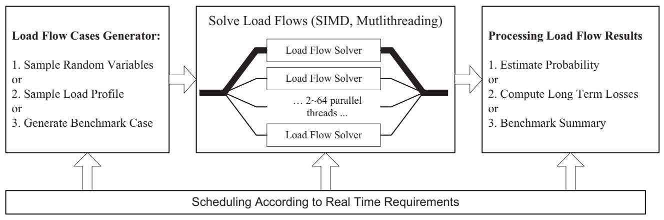

Load Flow Cases Generator:
1. Sample Random Variables
or
2. Sample Load Profile
or
3. Generate Benchmark Case
Solve Load Flows (SIMD, Mutlithreading)
Load Flow Solver
Load Flow Solver
… 2~64 parallel threads ...
Load Flow Solver
Processing Load Flow Results
1. Estimate Probability
or
2. Compute Long Term Losses
or
3. Benchmark Summary
Scheduling According to Real Time Requirements

Fig. 2. General structure of our realtime high throughput load flow solver.

Our general software structure to take advantage of these properties is shown in Fig. 2. Our approach is intrinsically *embarrassingly parallel*. However, in order to achieve peak performance and enable real time operation, aggressive code optimization and providing a proper threading infrastructure is still a big challenge.

III. Optimization and Parallelization

Baseline implementation. We implemented a baseline code in object-oriented C++, similar to the implementation of GridLabD [3]. In our baseline implementation, the distribution system is modeled as a tree and implemented using the Standard Template Library (STL), which provides template data structures for trees and traversals, among others. The forward and backward sweeps iterate over the tree. While an STL-based object oriented implementation provides convenience and productivity from an software engineering perspective, these benefits come with a (potentially) huge performance penalty due to various overheads and uncontrolled memory allocation/access patterns.

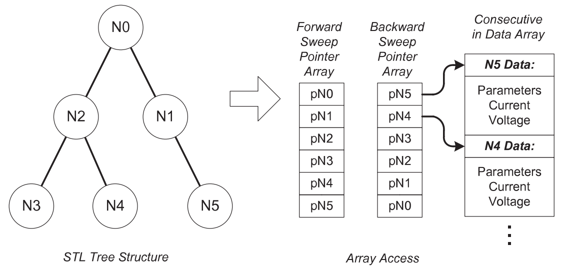


Fig. 3. Data structure optimization (same network as in Fig. 1).

Data structure optimization. The major data structure optimization is to flatten the tree data into an 1D array (Fig. 3).

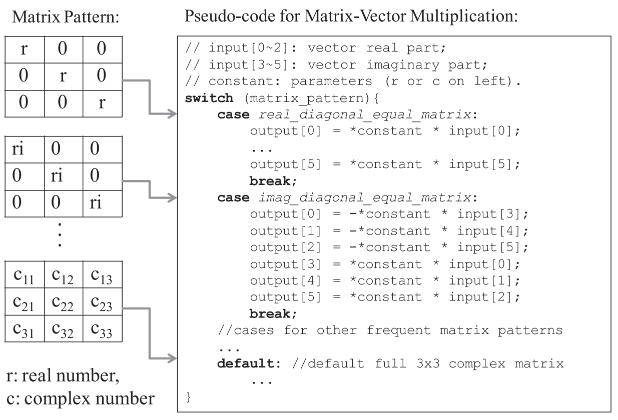
This is a one-time effort as the tree stays unchanged throughout the Monte Carlo simulation or other high-throughput applications, and thus negligible. The data needed for FBS on each node or branch is placed together in a block (N5 data in Fig. 3). Blocks are then placed in the 1D array according to the traverse sequence of FBS iterations. In this way, the tree traversals with object data access through member functions are converted into streaming memory access to a raw data array. We further introduce two pointer arrays (one for the forward sweep and one for the backward sweep) to enable pointer-based implementation of node and branch computations. The sweeps just linearly traverse the pointer arrays which in turn linearly (upwards) or almost linearly (downwards) traverses the data array. Thus, the FBS computation preserves temporal and spatial locality of the data streams introduced by the data structure optimization. The data structure optimized code takes advantages of the memory hierarchy and yield much better performance than the baseline C++ code.

Specialization through code synthesis. The main per-node or per-branch operations in the FBS iteration are small matrix-vector multiplications described in (1) and (2). The matrices are complex-valued and of size 3×3 due to the three phases of the distribution system. All 3×3 complex matrices $\mathbf{A}, \mathbf{B}, \mathbf{c}, \mathbf{d}$ are constant and not all of them are full matrices as the elements in these matrices are related to the self and mutual relations among the phases in the three-phase system. 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 due to the modeling of the nodes. These pattern are fixed once the system's physical elements are known.

To avoid unnecessary additions and multiplications with zero as well as the storage of known zero values, we synthesize special matrix-vector multiplication kernels that inline the matrix structure into the kernel. We need one specialized kernel per matrix pattern and a dispatch mechanism that invokes the correct kernel. The savings can be considerable as small 3×3 matrix-vector product kernels can be fully unrolled and a switch statement-based jump table can be used. Our approach is similar to [6], which introduces a pattern-based sparse matrix multiplication kernel, and to [7] where a similar code synthesis technique is used.

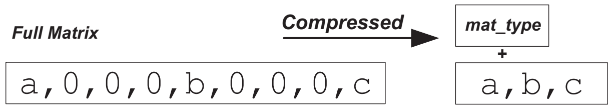
Fig. 4 shows example sparse matrices and a corresponding code fragment. Fig. 4(a) shows that for each matrix pattern only the necessary operations are present. The unnecessary floating point operations are removed. Fig. 4(b) shows the compact storage scheme for the sparse matrix. Since the patterns are known, only an identifier per pattern is needed in addition to the non-zero values. The kernel code for a given pattern has the addressing for the constants inlined. This reduces memory access operations and address calculations. This further reduces the runtime of the FBS iteration.

We use the SPIRAL [8] program generation and autotuning system as code synthesizer and source-to-source compiler to generate the jump table and the small specialized matrix-vector multiplication kernels. In particular, we are utilizing the SPL compiler, SPIRAL's domain-specific compiler which includes a lower-level compiler infrastructure to translate high-



```
// input[0~2]: vector real part;
// input[3~5]: vector imaginary part;
// constant: parameters (r or c on left).
switch (matrix_pattern){
    case real_diagonal_equal_matrix:
        output[0] = *constant * input[0];
        ...
        output[5] = *constant * input[5];
        break;
    case imag_diagonal_equal_matrix:
        output[0] = -*constant * input[3];
        output[1] = -*constant * input[4];
        output[2] = -*constant * input[5];
        output[3] = *constant * input[0];
        output[4] = *constant * input[1];
        output[5] = *constant * input[2];
        break;
    //cases for other frequent matrix patterns
    ...
    default: //default full 3x3 complex matrix
        ...
}
```

(a) Jump table dispatching pattern-specific matrix-vector kernels.



(b) Compressed storage for a matrix pattern.

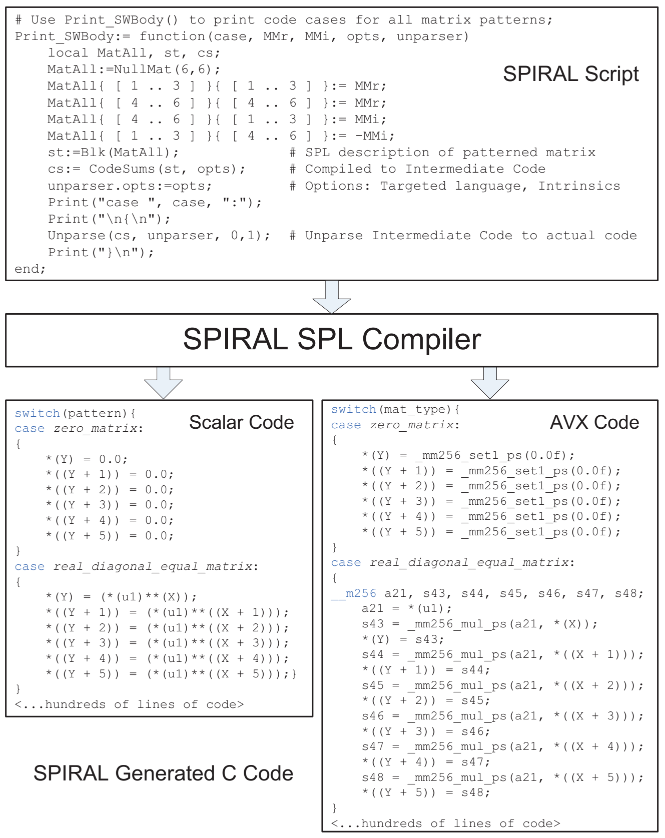
Fig. 4. Pattern-based optimized sparse matrix-vector multiplication.

level matrix pattern descriptions into unrolled, highly-efficient specialized matrix-vector multiplication code.

We show our approach in Fig. 5. We show the input script to the SPIRAL code synthesis engine and the code it synthesizes. The input script constructs a matrix-level representation of a given sparse 3×3 complex matrix with known pattern but unknown entry values. It sets the corresponding matrix entries according to the input parameters to real variables, complex variables or zero, and reuses variables to indicate symmetry. Then, the SPL compiler is invoked to compile this symbolic matrix into a straight-line kernel in internal code representation implementing the corresponding matrix-vector product, applying all the necessary basic block optimizations (strength reduction, common subexpression elimination, copy propagation, array scalarization) [9] that are enabled by the particular pattern of symmetry and reuse indicated through the variables. In a final step the generated code in internal representation is unparsed (pretty-printed) as C switch statement. SPIRAL supports synthesis of scalar as well as SIMD vector code.

SIMD vectorization. SIMD vector instructions like Intel's SSE and AVX instruction set extensions, as well as AltiVec/VMX and ARM's NEON ISA allow the parallel operation on multiple floating-point data elements. The data is held in vector registers, and special vector instructions operate on the usually 2–8 slots. These instruction sets usually feature mutually incompatible restrictions in their functionality and are hard to use but offer substantial speed-up.

Our realtime high-throughput power flow solver can benefit from these SIMD vector instructions, as shown in Fig. 6. For power system problems targeted by our approach and for a given distribution system, the instruction sequence in each load flow iteration is fixed across multiple instances, only the input



```
# Use Print_SWBody() to print code cases for all matrix patterns;
Print_SWBody:= function(case, MMr, MMi, opts, unparser)
    local MatAll, st, cs;
    MatAll:=NullMat(6,6);
    MatAll{ [ 1 .. 3 ] }{ [ 1 .. 3 ] }:= MMr;
    MatAll{ [ 4 .. 6 ] }{ [ 4 .. 6 ] }:= MMr;
    MatAll{ [ 4 .. 6 ] }{ [ 1 .. 3 ] }:= MMi;
    MatAll{ [ 1 .. 3 ] }{ [ 4 .. 6 ] }:= -MMi;
    st:=Blk(MatAll);               # SPL description of patterned matrix
    cs:= CodeSums(st, opts);       # Compiled to Intermediate Code
    unparser.opts:=opts;           # Options: Targeted language, Intrinsics
    Print("case ", case, ":");
    Print("\n{\n");
    Unparse(cs, unparser, 0,1);  # Unparse Intermediate Code to actual code
    Print("}\n");
end;
```

```
switch(pattern){
case zero_matrix:
{
    *(Y) = 0.0;
    *((Y + 1)) = 0.0;
    *((Y + 2)) = 0.0;
    *((Y + 3)) = 0.0;
    *((Y + 4)) = 0.0;
    *((Y + 5)) = 0.0;
}
case real_diagonal_equal_matrix:
{
    *(Y) = (*(u1)**(X));
    *((Y + 1)) = (*(u1)**((X + 1)));
    *((Y + 2)) = (*(u1)**((X + 2)));
    *((Y + 3)) = (*(u1)**((X + 3)));
    *((Y + 4)) = (*(u1)**((X + 4)));
    *((Y + 5)) = (*(u1)**((X + 5)));}
}
<...hundreds of lines of code>
```

```
switch(mat_type){
case zero_matrix:
{
    *(Y) = _mm256_set1_ps(0.0f);
    *((Y + 1)) = _mm256_set1_ps(0.0f);
    *((Y + 2)) = _mm256_set1_ps(0.0f);
    *((Y + 3)) = _mm256_set1_ps(0.0f);
    *((Y + 4)) = _mm256_set1_ps(0.0f);
    *((Y + 5)) = _mm256_set1_ps(0.0f);
}
case real_diagonal_equal_matrix:
{
__m256 a21, s43, s44, s45, s46, s47, s48;
    a21 = *(u1);
    s43 = _mm256_mul_ps(a21, *(X));
    *(Y) = s43;
    s44 = _mm256_mul_ps(a21, *((X + 1)));
    *((Y + 1)) = s44;
    s45 = _mm256_mul_ps(a21, *((X + 2)));
    *((Y + 2)) = s45;
    s46 = _mm256_mul_ps(a21, *((X + 3)));
    *((Y + 3)) = s46;
    s47 = _mm256_mul_ps(a21, *((X + 4)));
    *((Y + 4)) = s47;
    s48 = _mm256_mul_ps(a21, *((X + 5)));
    *((Y + 5)) = s48;
}
<...hundreds of lines of code>
```

Fig. 5. Pattern matrix vector product: specification and synthesized code.

sample data are different. This enable us to simply run multiple FBS iterations in parallel–one per vector slot. We simply pack one input sample data element per FBS instance into the vector variables, and convert the original instructions into SIMD instructions. When the input samples of different load flow instances are close to each other, the load flows instances usually converge at the same iteration step. Therefore multiple load flow cases can be solved simultaneously, resulting in an almost linear speedup with respect to the processing width of SIMD instructions. Further iteration on already converged instances (to converge the remaining slots) is rare and does not introduce any complications.

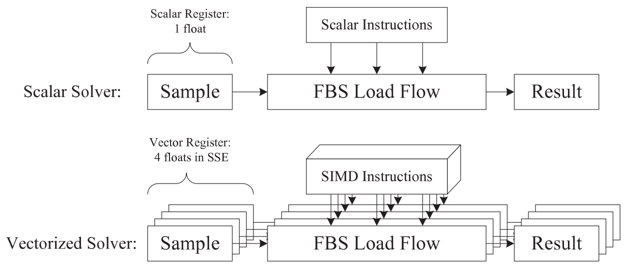


Fig. 6. Vectorization of load flow solver

Realtime threading infrastructure. Extracting full performance on modern multicore CPUs requires special attention to multithreading. Our target applications have some important properties that allow highly efficient implementations. Real time power system applications like Monte Carlo simulation require the solution of a large number of independent power flow problems. 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 same number of simulations. Further, much of the data is constant and shared; only the Monte Carlo result and the seeding varies across threads. Using multiple results buffers and on-the-fly thread-local seeding we can keep the Monte Carlo simulation running without gap, extracting maximum useful work from the processor.

We implemented a light-weight worker thread infrastructure that allows for fast buffer switching and seeding, shown in Fig. 7. A master thread orchestrates the computation 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 FBS solver in parallel on independent problems.

Each thread is exclusively pinned to a physical core. All threads use the same shift-register random number generation algorithm seeded such that they are far apart in the cycle and thus practically independent. Double-buffering and signaling allows non-blocking synchronization that extracts maximum useful computation from the multicore CPU. The memory is organized as in Fig. 8. Access to the data block is guided by pointer array. The pointer may point to shared memory for constant data, but may also point to thread local stored data for thread private read write access. In this way, multiple threads run simultaneously with limited contention on hardware resources and fully utilize the hardware.

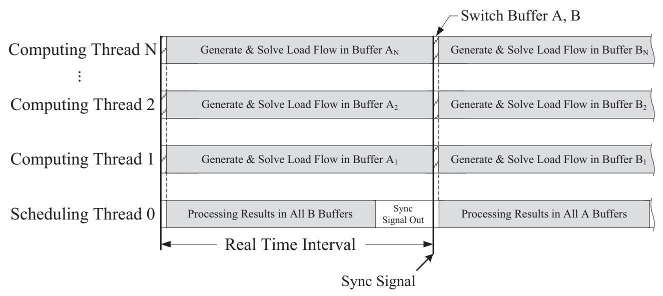

Switch Buffer A, B
Computing Thread N
Generate & Solve Load Flow in Buffer A_N
Generate & Solve Load Flow in Buffer B_N
⋮
Computing Thread 2
Generate & Solve Load Flow in Buffer A_2
Generate & Solve Load Flow in Buffer B_2
Computing Thread 1
Generate & Solve Load Flow in Buffer A_1
Generate & Solve Load Flow in Buffer B_1
Scheduling Thread 0
Processing Results in All B Buffers
Sync Signal Out
Processing Results in All A Buffers
Real Time Interval
Sync Signal

Fig. 7. Multi-threaded real-time high-throughput load flow solver.

IV. Experimental Result

Performance evaluation. We evaluated the performance of our solver across network sizes and multicore platforms. We use the Intel C++ compiler 11.0 and -O3 optimization level. To assess the performance across problem sizes, we connected

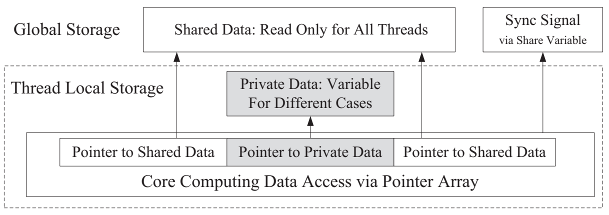

Global Storage
Shared Data: Read Only for All Threads
Sync Signal via Share Variable
Thread Local Storage
Private Data: Variable For Different Cases
Pointer to Shared Data
Pointer to Private Data
Pointer to Shared Data
Core Computing Data Access via Pointer Array

Fig. 8. Storage of shared and private data.

multiple IEEE 4-bus test feeders [10] into larger trees, and varied the resulting network size between 4 busses and 2,048 busses. Results are shown in Fig. 9 for a 2.2 GHz quad-core Intel Core i7 CPU with the machine peak of 140 Gflop/s. We show scalar vs. SIMD vector code (AVX), sequential vs. multi-threaded code, and the impact of pattern-based code synthesis. In the best case the fully optimized multi-core solver is able to achieve around 60% of the CPU's machine peak. Vectorization, parallelization and code synthesis provide speed-up as expected.

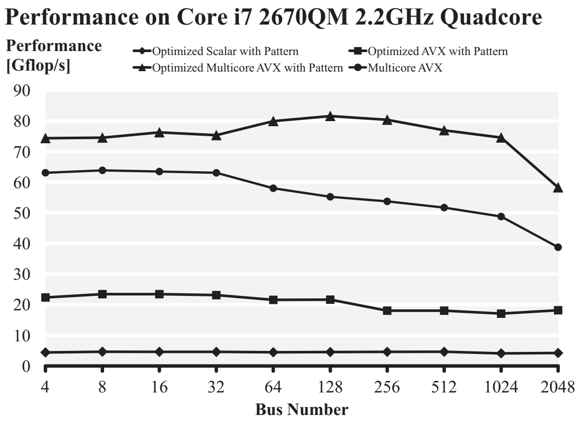


Fig. 9. Performance result across different network sizes (on Core i7).

To assess performance across various multicore platforms we evaluated our optimized solver with the IEEE 37-bus test feeder [10]. We performed experiments on four Intel systems consisting of one or two multicore CPUs (2.66 GHz Core2 Extreme, 3.33 GHz Xeon X5680, 2.2 GHz Core i7 and dual 2.27 GHz Xeon 7560) and support for SSE or AVX. As show in Fig. 10, the performance increases with the increase of SIMD width (SSE to AVX), and with the increase of number of CPU cores. A single multicore CPU can provide up to 85 Gflop/s performance for our solver. Our solver scales almost linearly with increasing hardware parallelism (cores and vector width).

Impact of optimization methods. Next we assess the impact of the various performance optimization techniques we applied. Fig. 11 shows performance for various optimization levels on the Core i7 CPU. The white bar shows the baseline C++ STL code compiled by Intel C Compiler with full optimization option (-O3). The *Scalar*, *AVX*, and *MultiThread AVX* code versions are using the flattened C array, and using scalar x87 code, are vectorized or both vectorized and multi-threaded. Bars called *Scalar Pattern*, *AVX Pattern*, and *MultiThread AVX Pattern* in addition also utilize matrix pattern based code synthesis. For the pattern based optimized code, we achieve a nearly linear speedup through vectorization and multithread. The fully optimized solver is about 50 times faster than the C++ STL baseline implementation subjected to full compiler optimization.

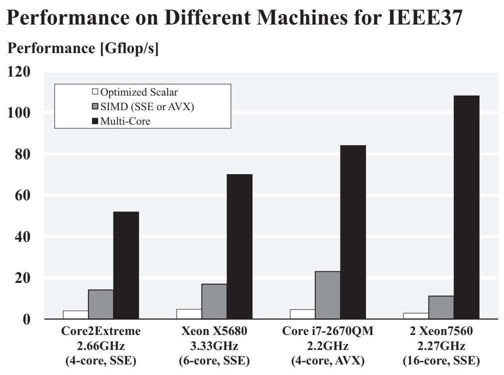


Fig. 10. Performance results across multicore platforms.

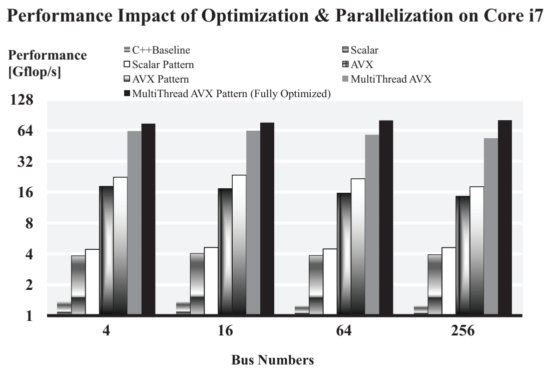


Fig. 11. Impact of performance optimization techniques (on Core i7).

Realtime Monte Carlo. For power systems the central control system (SCADA) has a cycle time of approximately 4 seconds. Thus, the number of simulations per SCADA interval as function of network size limits the achievable Monte Carlo accuracy. The approximate runtime of the solver for one million load flows solving a IEEE 37-bus test feeder and a IEEE 123-bus test feeder is showed in Table I. We observe that on the Intel Sandy Bridge CPU (Core i7) with quad-core and AVX, 1 million load flows can be solved within 4 seconds, which equals SCADA real time. The baseline runtime results including fully-compiler-optimized C++ code (-O3) and Matlab code are also showed for reference, falling far short of realtime.

V. Application Example: Probabilistic Load Flow

In this section, we show a possible application of the high performance solver to solve the probabilistic load flow (PLF) problem. The PLF problem is to model the uncertainties of

TABLE I
APPROXIMATE RUNTIME

1M Load Flow		Optimized Code on		Baseline on Core2	
System	Flops	Core2Extreme	Core i7	C++(-O3)	Matlab
IEEE 37	$\approx$ 60G	$<$ 2s	$<$ 1s	$>$ 60s	$>$ 5hr
IEEE 123	$\approx$ 200G	$<$ 10s	$<$ 3.5s	$>$ 200s	$>$ 10hr

some system parameters as the random variables (RV) in terms of probabilistic density functions (PDF), then based on the system's physical relations, PLF computes system states in terms of the RVs with PDFs as well [11]. Monte Carlo simulation (MCS) is one way to solve PLF which often serves as accuracy reference. MCS firstly samples the random variables. Then for each sample a load flow case is solved to obtain all states. Based on the the load flow results of all samples, the PDF of certain system states can be estimated.

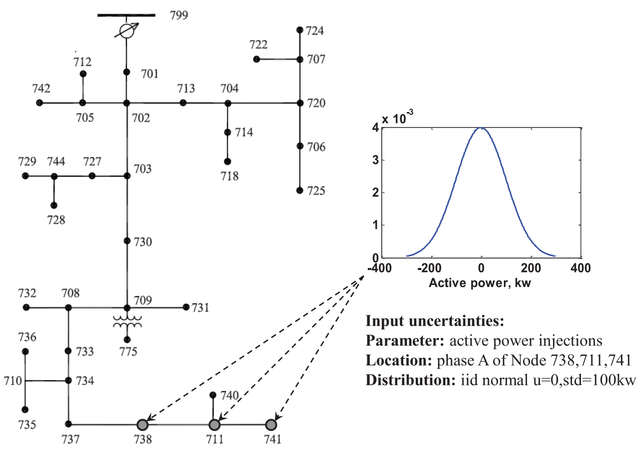


Fig. 12. Configuration of test case on IEEE 37-bus test feeder

Fig. 12 shows a typical problem setup: on the IEEE 37-bus test feeder, the active power load of phase A on node 738, 711 and 741 are assumed to be Gaussian RVs. The MCS method samples these RVs and solves the load flow for all samples. Fig. 13 shows the estimated PDFs of three phase voltages on node 738, from left to right, with the increasing number of Monte Carlo samples (100 to 10 million), the estimated PDFs on different random seeds converge to a single PDF. Assuming the random number generator can generate true random samples, by the Law of Large Numbers, the converged curve is the true PDF curve given the input RVs. In Fig. 13 we observe that even the very crude Monte Carlo (without any variance reduction) can obtain high accuracy around 1 million samples. The PDF curves at this accuracy level can be updated within one second using the optimized solver running on Core i7 CPU.

VI. Conclusion

In this paper we developed a multi-core high performance distribution load flow solver for high throughput distribution load flow problem solutions. We applied various aggressive performance optimizations including multi-level parallelization (SIMD vectorization and threading), data structure optimization, and pattern based code generation. The optimized solver is able to achieve more than 50% of a commodity multicore CPU's theoretical peak performance, and the performance is scalable with increased hardware parallelism. We evaluated the solver using IEEE test feeders on various Intel multicore CPUs (single CPU or dual CPU) featuring the SSE or AVX SIMD instruction set. Our optimized solver is able to solve millions of load flow problems within one second, enabling real-time application of various Monte Carlo style applications.

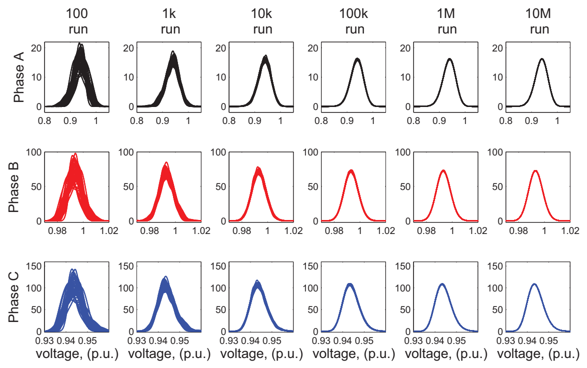


Fig. 13. Result of crude Monte Carlo simulation

References

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

[2] K. Schneider, D. Chassin, Y. Chen, and J. Fuller, "Distribution power flow for smart grid technologies," in *Power Systems Conference and Exposition, 2009. PSCE'09. IEEE/PES*. IEEE, 2009, pp. 1–7.

[3] D. P. Chassin, K. Schneider, and C. Gerkensmeyer, "GridLAB-D: An open-source power systems modeling and simulation environment," in *Transmission and Distribution Conference and Exposition, 2008. T&D. IEEE/PES*, 2008, pp. 1–5.

[4] D. Shirmohammadi, H. Hong, A. Semlyen, and G. Luo, "A compensation-based power flow method for weakly meshed distribution and transmission networks," *IEEE Transactions on Power Systems*, vol. 3, no. 2, pp. 753 –762, May 1988.

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

[6] 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*, ser. ICS '09. New York, NY, USA: ACM, 2009, pp. 100–109. [Online]. Available: http://doi.acm.org/10.1145/1542275.1542294

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

[8] M. Püschel, J. M. F. Moura, J. Johnson, D. Padua, M. Veloso, B. Singer, J. Xiong, F. Franchetti, A. Gacic, Y. Voronenko, K. Chen, R. W. Johnson, and N. Rizzolo, "SPIRAL: Code generation for DSP transforms," *Proceedings of the IEEE, special issue on "Program Generation, Optimization, and Adaptation"*, vol. 93, no. 2, pp. 232– 275, 2005.

[9] J. Xiong, J. Johnson, R. Johnson, and D. Padua, "SPL: A language and compiler for DSP algorithms," in *Proc. Programming Language Design and Implementation (PLDI)*, 2001, pp. 298–308.

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

[11] P. Chen, Z. Chen, and B. Bak-Jensen, "Probabilistic load flow: A review," in *Electric Utility Deregulation and Restructuring and Power Technologies, 2008. DRPT 2008. Third International Conference on*, april 2008, pp. 1586 –1591.