A Distributed Power Grid Analysis Framework from Sequential Stream Graph

Chun-Xun Lin
Dept. of ECE, UIUC
IL, USA
clin99@illinois.edu

Tsung-Wei Huang
Dept. of ECE, UIUC
IL, USA
twh760812@gmail.com

Ting Yu
Austin, TX, USA
yuting20031918@gmail.com

Martin D. F. Wong
Dept. of ECE, UIUC
IL, USA
mdfwong@illinois.edu

ABSTRACT

The ever-increasing design complexities have overwhelmed what is offered by existing EDA tools. As a result, the recent EDA industry is driving the need for distributed computing to leverage large-scale compute-intensive problems, in particular, *power grid analysis*. In this paper, we introduce a distributed power grid analysis framework based on the stream graph model. We show that the stream graph model has better programmability over the MPI and enables flexible domain decomposition without limited by hardware resource. In addition, we design an efficient scheduling policy for this particular workload to maximize the cluster utilization to improve the performance. The experimental results demonstrated the promising performance of our framework that scales from single multi-core machines to a distributed computer cluster.

ACM Reference Format:
Chun-Xun Lin, Tsung-Wei Huang, Ting Yu, and Martin D. F. Wong. 2018. A Distributed Power Grid Analysis Framework from Sequential Stream Graph. In *GLSVLSI '18: 2018 Great Lakes Symposium on VLSI, May 23–25, 2018, Chicago, IL, USA*. ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3194554.3194560

1 INTRODUCTION

As the technology continues to advance, analyzing a power distributed network that corporates billions of transistors becomes a critical challenge. Traditionally, power analysis engineers partitioned the problem into smaller and manageable pieces, and ran each on a single multi-threading machine. However, according to [1], analyzing a power grid with 136 million nodes on a single multi-core machine can take hundreds of GBs of memory and several hours to finish. Building such a high-end computer is expensive and unscalable to the ever-increasing design complexities. As a result, EDA vendors are driving the need of distributed power grid analysis.

Researchers have proposed parallel computing methods for power grid analysis [2] [3] [4] [5] [6]. Existing works are based on either multi-threading in a shared memory storage or distributed computations across different nodes. The work by [2] [3] developed parallel power grid simulators by taking the advantage of multi-cores with shared memory to speed up the computing. Although the shared memory model is advantageous in data communication, it relies on expensive hardware resources to gain more scalability. The work by [4] [5] designed parallel computing schemes by partitioning data and distributing the computations across multiple machines using the low-level massage passing interface (MPI) library [7]. While MPI provides a layer of abstraction over the network communication, it suffers from many distinct notations to express the parallelism. The bottom-up design principle of MPI is analogous to assembly languages in terms of writing parallel code. For example, users have to manually name the machines for process mapping and hard-code message passing for serialization and deserialization. It also requires a significant amount of coding efforts when the software changes to the next generation. Putting these issues together discourages developers from being productive and innovative. Nevertheless, building a distributed power grid analysis beyond MPI remains an open problem.

While existing big-data tools offer many promises in distributed computing [8], EDA researchers remain skeptical about the applicability for many reasons [9]. First, power grid analysis is compute-intensive whereas the big data computing focuses on I/O processing. Second, MapReduce paradigm assumes data can be split into independent chunks while the power grid data are not easily separable. Third, the mainstream programming languages of the big data are JVM languages that do not appeal to the language need of power grid (C/C++). As a consequence, we need a specialized distributed framework for power grid analysis.

In this paper, we introduce a distributed power grid analysis framework based on the stream graph model. The goal of this paper is, instead of solving the power grid analysis with domain-specific techniques, to investigate into the programmability, extensibility, and scalability of distributed power grid analysis at framework level. We summarize our contributions as follows:

- We show that with the use of the *stream graph* programming paradigm, programming distributed power grid analysis can be greatly simplified. Unlike MPI which is based on low-level message passing API, the stream graph is a higher-level abstraction to express parallelism. Users can focus on developing the framework based on the algorithmic specification, without wrestling with system-specific implementation details.
- We show that with a customized scheduler, we are able to maximize the resource utilization in a cluster. Our scheduler is tailored for the compute-intensive power grid analysis. We demonstrate our scheduler can effectively leverage the CPU usage for this particular workload.

GLSVLSI '18, May 23–25, 2018, Chicago, IL, USA

ACM ISBN 978-1-4503-5724-1/18/05…$15.00
https://doi.org/10.1145/3194554.3194560

- We show that our framework is a more flexible and scalable alternative to MPI-based solutions. We can flexibly partition the power grid to different subdomains regardless of the number of cores, which is instead impossible in MPI due to its architectural limitation.

We implement our framework on DtCraft[1] [10], a distributed execution engine for high-performance applications, for our experiment. The experimental results show that our distributed power grid framework achieves comparable performance to MPI-based solutions. We also demonstrated the effectiveness of our scheduler over the default scheduler of DtCraft in an emulated production environment.

2 DISTRIBUTED POWER GRID ANALYSIS

The goal of power grid analysis is to solve following system of equations extracted from the associated circuit:

$$GV = I,$$

G : A matrix formed by the conductance of components

V : A vector consists the voltage of nodes (unknown)

I : A vector consists the independent current sources

By solving above linear system, the voltage drop at each node can be derived by comparing the node voltage V with the supply voltage. One feasible way to solve the system is the domain decomposition [3] [11] [12] which partitions the problem into subsets and solves them in parallel. The Additive Schwarz Method (ASM), one type of the domain decomposition methods, is especially suitable for large sparse system [4]. In this paper, we adopt the geometric ASM method with 2D partitioning proposed by [4] for distributed direct current (DC) analysis, which is proved to have minimum data communication. The geometric ASM method for DC analysis can be summarized as four steps:

(1) Partition the circuit into subdomains.
(2) Solve each subdomain independently.
(3) Synchronize and exchange the boundary values of subdomains.
(4) Go to (2) if any of the subdomains does not converge.

The geometric ASM method is a natural fit for distributed computing as it can be directly parallelized by assigning the subdomains to different processors.

2.1 Existing Works and Limitations

Based on the geometric ASM method, researchers developed a number of distributed power grid analysis systems using MPI [4] [11] [12]. The MPI programming model is processor-centric. A MPI program consists of several processes with each process attached to a processor, and a typical MPI program can have number of processes less than or equal to the number of available processors. Even though oversubscription is possible in MPI, it's discouraged by the official due to performance degradation. The processes form a communication group and each process has a unique number called rank for identification. Processes can send or receive data through using the rank in a set of APIs. Based on the message passing model, a distributed DC analysis program with MPI is shown in Algorithm 1

Algorithm 1: MPI-based Distributed DC analysis

```
   Input: C: circuit, W: width, H: height
1  MPI_Init();
2  rank ← MPI_Rank();
3  subdomains ← ∅;
4  if rank == 0 then
5  |  PartitionGrid(C, W, H);
6  end
7  MPI_Sync();
8  subdomains[rank] ←ReadGrid(rank);
9  bd_value_num ← CountBD(subdomains[rank]);
10 MPI_Gather( 0, bd_nums, bd_value_num);
11 if rank == 0 then
12 |  bd_array ← CreateBoundaryArray(bd_nums)
13 end
14 converge ← False;
15 while !converge do
16 |  solution ← Solve(subdomains[rank]);
17 |  bd_value ← ExtractBoundary(solution);
18 |  converge ← Check(solution);
19 |  MPI_Gather( 0, bd_array, bd_value);
20 |  MPI_Gather( 0, result, converge);
21 |  if rank == 0 then
22 |  |  converge ← IsConverge(result);
23 |  |  Reorder(bd_array);
24 |  end
25 |  MPI_Scatter(0, bd_array, bd_value);
26 |  UpdateBD(bd_value, solution);
27 |  MPI_broadcast(0, converge);
28 end
```

In Algorithm 1, the power grid is partitioned into $W \times H$ subdomains and the MPI program launches $W \times H$ processes with each process assigned a subdomain. Notice that in line 8, a process handles the subdomain based on the rank automatically assigned by MPI and the rank is limited by the number of available CPU cores. Although users can implement a distributed computing program by directly including the MPI library and utilizing the low-level APIs, there are several disadvantages of the MPI-based implementation:

- The number of subdomains is limited by the available processors. Also, this is a constraint to launching the program (`mpirun -n [number of cores]`). This fundamentally restricts our problem-solving logic to deliver an effective and scalable solution.
- To manage all processes running concurrently in the MPI model, an MPI program needs to explicitly use conditional instructions or branch predicate to separate the execution flows of different processes. This complicates the program structure and also makes the MPI program difficult to be extended to incremental analysis [13], where some processes might change the values in subdomains.

[1]We use DtCraft version 0.0.1 for the implementation.

As a result, it's desirable to have a novel distributed computing framework that does not suffer from the above issues.

3 DISTRIBUTED POWER GRID ANALYSIS BASED ON STREAM GRAPH

3.1 Stream Graph Model

Stream graph [10] is a new programming model that aims for distributed computing, especially for high performance (compute-intensive) applications. A stream graph is a high-level abstraction that describes the program as a directed graph, where vertices and edges encapsulate the data flow and a sequence of computations. The computations are asynchronous, i.e. computations are only executed when the associated data arrive. This makes the stream graph a competitive solution for performance-driven applications.

3.2 DC Analysis in Stream Graph

Based on the stream graph programming paradigm, we formulate the DC analysis as a stream graph with two types of vertices: synchronization vertex and worker vertex. The stream graph of DC analysis consists of one synchronization vertex and N worker vertices where N is the number of subdomains, and there are two directed edges connecting the synchronization vertex and each worker vertex. In general, the synchronization vertex serves as a hub that exchanges data between worker vertices and determines whether the solution converges or not, while the worker vertex is responsible for solving a subdomain and reporting the result to the synchronization vertex. Algorithm 2 presents the stream graph for DC analysis. A synchronization vertex is first inserted into the graph in line 5. Then we insert a worker vertex and two directed edges to the graph (line 7 - 11) and execute the graph in line 12.

Algorithm 2: DC analysis using stream graph

```
   Input: C: circuit, W: width, H: height
1  Graph G;
2  workers ← {} ;
3  to_worker ← {} ;
4  to_sync ← {} ;
5  sync ← InsertV(G, sync_cb(C,W,H,to_worker));
6  N ← W ∗ H ;
7  for i = 1 to N do
8  |  workers[i] ← InsertV(G, worker_cb());
9  |  to_worker[i] ← InsertE(G, sync, workers[i], worker_edge_cb(
   |   )) ;
10 |  to_sync[i] ← InsertE(G, workers[i], sync, sync_edge_cb()) ;
11 end
12 dispatch(G);
```

The program initializes required data from invoking the synchronization vertex's callback once to prepare subdomains. Then the synchronization vertex notifies the worker vertices of the corresponding subdomains by sending a signal through edges. Algorithm 3 presents the callback of the synchronization vertex. In line 1, the power grid is first partitioned into W×H subdomains and then each subdomain index is passed to a worker vertex along the directed edge (line 2 - 7).

Algorithm 3: Callback of a synchronization vertex

```
   Input: C: circuit, W: width, H: height, edges: edges to worker
          vertices
1  PartitionGrid(C, W, H);
2  for i = 1 to W do
3  |  for j = 1 to H do
4  |  |  id ← SubdomainId(i, j);
5  |  |  send( edges[i][j], id );
6  |  end
7  end
```

For the input edge callbacks of both types of vertices, we use finite state machines to establish a communication protocol to react to different types of input data. Algorithm 4 shows the details in the callback of a synchronization vertex at input side. The callback has two states: *CHECK* and *RECV*. In line 3, the CHECK state gathers the results from worker vertices and informs all worker vertices the global status once all results are received (line 4 - 11). The callback is removed when reaching convergence (line 10). From line 13 to 23, in the RECV state the synchronization vertex collects and sends the new boundary values to the worker vertices.

Algorithm 5 presents the callback of a worker vertex at input side. The callback has three states: *INIT*, *COMPUTE* and *WAIT_RESULT*. In the INIT state, each worker vertex first receives a subdomain index from the synchronization vertex (line 2 - 6). Then a worker vertex solves its own subdomain and replies the result and transits to the WAIT_RESULT state (line 22 - 27). In the WAIT_RESULT state, the worker vertex waits for the global result. The callback is removed if the whole solution converges (line 9 - 11); otherwise the worker vertex sends the boundary values to the synchronization vertex and transits to the COMPUTE state (line 12 - 14). In the COMPUTE state (line 17 - 19), when a worker vertex receives the updated boundary values, it proceeds to solve the subdomain with the new values and send the result back.

Our proposed distributed framework has several benefits over the MPI model:

- In contrast to the static (manual) mapping of processes to processors in MPI, the callbacks in the stream graph can be executed on any core in an asynchronous manner, allowing users to create more partitions than the available processors.
- By packaging the callbacks (sequential block) into a parallel program, the stream graph formulation has better code readability and makes debugging easier, whereas the MPI program is more complex as processes with different execution trajectories are put in the same block.
- The stream graph formulation lets users to assign the resource requirements for individual subgraph, which allows the scheduler to make a more effective cluster resource utilization.

Combining above benefits, our framework has better programmability and scalability than the MPI. We believe our framework stands out as a unique solution to distributed power grid analysis, considering the software design and the architectural decision we made.

Algorithm 4: Input edge callback of a synchronization vertex

```
Input: id: edge id, N: number of worker vertices, edges: edges to
       worker vertices
switch state do
    case CHECK do
        recv( results[id] );
        if all workers are recv then
            done ← AllConverge(results) ? True : False;
            for i = 1 to N do
                send( edges[i], done );
            end
            state = RECV;
            return done ? REMOVE_THIS_CB:DEFAULT;
        end
    end
    case RECV do
        recv( bd_vectors[id] );
        if all workers are recv then
            Reorder(bd_vectors);
            for i = 1 to N do
                send( edges[i], bd_vectors[i] );
            end
            state = CHECK;
            return DEFAULT;
        end
    end
end
```

Algorithm 5: Input edge callback of worker vertex

```
Input: id: edge id, edge: edge to synchronization vertex
switch state do
    case INIT do
        recv( subdomain_id );
        my_subdomain ← ReadGrid( subdomain_id );
        go to 22;
    end
    case WAIT_RESULT do
        recv( result );
        if result then
            return REMOVE_THIS_CB;
        end
        state ← COMPUTE;
        send(edge, bd_value);
        return DEFAULT;
    end
    case COMPUTE do
        recv( bd_value );
        UpdateBD(bd_value, solution);
        go to 22;
    end
end
solution ← Solve(my_subdomain) ;
bd_value ← ExtractBoundary(solution) ;
converge ← Check(solution);
state ← WAIT_RESULT;
send( edge, converge );
return DEFAULT;
```

4 APPLICATION-SPECIFIC RESOURCE CONTROL PLUG-IN

Job scheduling is an important issue in distributed computing as the scheduling has a huge impact on overall system performance. In this section, we first outline the default scheduler in DtCraft, then we introduce a scheduler that is tailored for CPU bound applications such as the power grid analysis to enhance the system performance.

4.1 Default Scheduler

The default scheduler in DtCraft adopts a best-fit method to match a job's tasks to machines based on their resource (CPU + memory) requirements. Unlike CPUs that are shared among processes, memory claimed by a process will not be available to others during execution. As a result, memory is regarded as a hard constraint and any process violates the memory constraint will be terminated. The policy of the default scheduler is first-come-first-serve and non-preemptive. Whenever the scheduler receives a job from users, it seeks to find a feasible scheduling for the job if no jobs are waiting ahead. The scheduler first takes a snapshot of the current status of machines, then for each task in the job, the scheduler collects the machines that have enough memory to accommodate the task, and among those candidates the best-fit machine, the one with the least amount of memory, is matched to the task. A job cannot be scheduled if any of its task fails to be matched to a machine. A failed job will be stored in a queue for future processing. Whenever a job finishes execution and releases the memory, the scheduler will examine the queue to process the waiting jobs. The idea of this method is to reduce memory fragmentation which could spare more room to have more jobs scheduled.

4.2 Proposed Scheduler

One deficiency of the default scheduling policy is the underutilization of CPUs since the default scheduler tends to assign jobs to the machines that are partially loaded while there still exists idle machines. To have better utilization of the cluster resource, we propose a scheduler for balancing the workload of cluster machines. In order to evenly distribute the workload we integrate the CPU usage and average CPU load in the past one minute into scheduling to decide the deployment. We record the CPU demands of tasks allocated on each machine and define the ratio of total CPU demands to the number of CPUs on the machine as load index. During the job scheduling, we first collect the machines that satisfy the memory requirement. Then, rather than selecting the machine with the least available memory, a task is matched to the machine with the smallest load index and in case of a tie, the machine with smaller average CPU load in the past one minute is preferred. The goal of using load index to determine the task placement is to proportionally distribute the workload. Algorithm 6 presents the algorithm of the proposed scheduler. In Line 8 - 15, the scheduler finds the machines with enough memory and then deploys the task on the least utilized machine by comparing their load indices and average load over past one minute for tie-breaking.

Algorithm 6: Load-aware scheduling algorithm

Input: M: machines, J:a job
Output: P:packings

```
snapshot ← {};
foreach m ∈ M do
    snapshot ← snapshot ⋃ m;
end
foreach t ∈ J do
    best ← null;
    foreach s ∈ snapshot do
        if s.memory >= t.memory then
            if s.load < best.load or best == null then
                best ← s;
            end
            else if s.load == best.load and
              s.loadavg < best.loadavg then
                best ← s;
            end
        end
    end
    if best == null then
        P ← ∅;
        break;
    end
    else
        P ← P ⋃ (t, best);
        best.memory− = t.memory;
        best.load+ = t.cpu/best.cpu;
    end
end
```

5 EXPERIMENTAL RESULTS

We first compare two implementations of distributed DC analysis: the stream graph and the MPI model on both the single machine and the distributed environment. Next we compare the proposed scheduler with the default scheduler in an emulated production environment.

5.1 Stream Graph versus MPI

We conduct experiments on a set of power grid benchmarks released by IBM [14]. We use the network file system (NFS) to allow file sharing across the machines. In the single machine experiment, the machine is equipped with a 2.4 GHz quad-core CPU and 35 GB memory. Due to the available number of cores, we partition the circuit into four (2×2) subdomains to evaluate the MPI program. Since the stream graph does not have the processor binding issue, we further test the stream graph model with the 3×3 and 4×4 partitions to investigate possible performance improvement.

Table 1 lists the results of the single machine experiment. We record the total execution time (including the generation of partitioned files) and the matrix solving time. For the 2×2 partitions, the runtime of stream graph is only moderately higher than the MPI's and both exhibit a similar performance scale. Considering the 3×3 and 4×4 partitions, the performance is further improved by partitioning the circuit into smaller subdomains to reduce the matrix solving time.

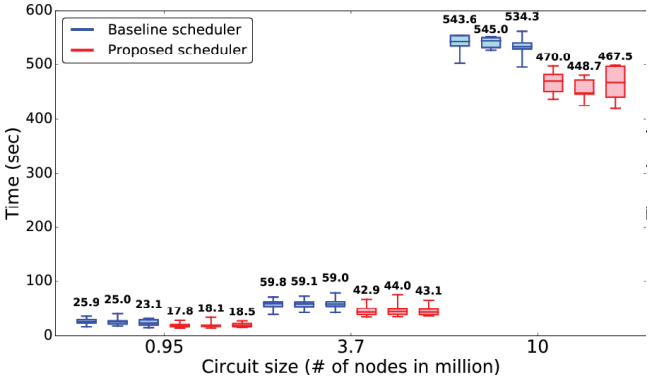

Figure 1: The runtime (sec) distribution for the three sizes of benchmarks in all runs. The number on the top of each box is the median value, and the top and bottom whiskers represent the maximum and minimum values.

Next we evaluate their performance in a cluster consisting of 9 machines with each has a 3.2 GHz quad-core CPU and 24 GB memory. We experiment four partition sizes: 3×3, 4×4, 5×5 and 6×6. For the sake of fairness, in the stream graph model a subdomain is assigned one CPU core. Table 2 shows the matrix solving and total runtime (including the latency of transferring partitioned files on NFS). In all types of partitions, the matrix solving time of stream graph is close to the MPI model's and the difference does not scale with the circuit size, indicating the performance of stream graph is comparable to the MPI model.

5.2 Production-Mode Evaluation

The scheduler experiments are undertaken on Amazon's Elastic Compute Cloud and we use 10 EC2 instances where each instance has 4 CPUs and 16 GB memory. The first experiment is to evaluate the schedulers on handling workload composed of jobs in different scales. We select three types of circuits whose power grids have 0.95, 3.7 and 10 million nodes respectively to represent jobs with small, medium and large scale. The stream graph each has 4 (small), 8 (medium) and 16 (large) worker vertices respectively. There are one hundred jobs in total and the number of jobs for each types are 27, 68 and 5, which is distributed normally to simulate the job composition in realistic situations. The jobs are randomly permuted and we submit a job every 10 seconds.

We run three times for both the default scheduler and the proposed scheduler and record all results. Table 3 shows the total time from submitting the first job to the finish of the last job. Compared with the baseline scheduler, the proposed scheduler effectively reduces the total time by an average of 10%.

To understand the impact of schedulers on the runtime of each job, Figure 1 records the distribution of completion time on jobs with different sizes. With the proposed scheduler, the average runtime of the small, medium and large-sized jobs is reduced by 24%, 22% and 14% respectively.

Lastly, we evaluate the schedulers with jobs arriving in Poisson distribution manner. We set the average arrival rate to 0.1 (i.e. the

Table 1: Runtime (sec) of MPI versus Stream graph on single machine

Testcase	Size	Solve Time (2x2)		Total Time (2x2)		Solve time (stream graph)		Total Time (stream graph)	
		MPI	Stream graph	MPI	Stream graph	3x3	4x4	3x3	4x4
y200	10513442	1,061.45	1,246.63	1,133.46	1,302.94	544.44	749.20	592.30	795.65
y250	6727562	628.15	717.60	676.21	754.30	262.03	266.03	294.22	296.09
y300	4688899	251.82	294.99	284.67	320.52	156.08	154.70	178.36	176.37
y400	2627442	48.72	68.38	66.82	82.49	45.96	49.61	58.14	61.25
y500	1680602	25.77	36.31	37.94	45.43	25.76	25.29	33.69	32.95
y600	1171822	12.50	18.41	20.77	24.74	18.25	15.50	23.96	20.99
y800	655896	6.44	10.71	11.07	14.51	7.82	7.49	10.97	10.68
y1000	419522	2.85	5.27	5.79	7.61	4.21	4.17	6.27	6.18

Table 2: Runtime (sec) of MPI versus Stream graph (ours) on a cluster with 9 machines

Testcase	Decomposition	Solve Time		Total Time	
		MPI	Ours	MPI	Ours
y200	6 x 6	90.60	109.12	149.42	163.40
y250	6 x 6	34.40	45.36	70.583	82.29
y300	5 x 5	22.52	28.06	43.95	52.59
y400	5 x 5	7.52	10.04	19.35	22.75
y500	4 x 4	5.21	6.77	14.79	16.87
y600	4 x 4	3.26	5.03	9.96	12.54
y800	3 x 3	2.25	3.65	7.54	7.97
y1000	3 x 3	1.10	2.58	4.41	5.63

Table 3: The execution time (minutes) for the three runs.

	1st		2nd		3rd	
	Base	Ours	Base	Ours	Base	Ours
Runtime	25.32	23.05	26.0	22.58	25.42	22.77

average arrival time of a job is 10 seconds) and submit 100 medium-sized jobs. Table 4 shows that the proposed scheduler's average completion time of a job is around 20% smaller than the default scheduler's. We observe that the number of vertices deployed on each machine can vary greatly in the default scheduler, resulting in a low resource utilization and slower performance.

6 CONCLUSION

This paper introduces a distributed power grid analysis framework based on the stream graph programming model. The framework enables flexible power grid decomposition regardless of the available CPU cores. Moreover, a load aware scheduler is proposed to balance the machine workloads and effectively promote the overall system resource utilization. The experimental results show that the framework has comparable performance as the MPI-based framework and the effectiveness of the load aware scheduler. We believe we open a new direction for the distributed power grid analysis. Our idea can inspire EDA engineers to rethink the way to parallelize EDA algorithms.

Table 4: The average runtime (sec) of a benchmark for the three runs.

	1st		2nd		3rd	
	Base	Ours	Base	Ours	Base	Ours
Runtime	62.53	48.44	63.51	42.95	62.47	42.89

7 ACKNOWLEDGMENT

This work is partially supported by the National Science Foundation under Grant CCF-1421563 and CCF-171883.

REFERENCES

[1] C. J. Wei, H. Chen, and S. J. Chen. Design and Implementation of Block-Based Partitioning for Parallel Flip-Chip Power-Grid Analysis. *TCAD*, 31(3):370–379, March 2012.
[2] James W. Demmel, John R. Gilbert, and Xiaoye S. Li. An Asynchronous Parallel Supernodal Algorithm for Sparse Gaussian Elimination. *SIAM J. Matrix Anal. Appl.*, 20(4):915–952, July 1999.
[3] V. Y. Voronov and N. N. Popova. Parallel power grid simulation on platforms with multi core processors. In *2009 International Conference on Computing, Engineering and Information*, pages 144–148, April 2009.
[4] T. Yu, Z. Xiao, and M. D. F. Wong. Efficient parallel power grid analysis via Additive Schwarz Method. In *IEEE/ACM ICCAD*, pages 399–406, Nov 2012.
[5] Laura Grigori, James W. Demmel, and Xiaoye S. Li. Parallel Symbolic Factorization for Sparse LU with Static Pivoting. *SIAM Journal on Scientific Computing*, 29(3):1289–1314, 2007.
[6] Q. He, W. Au, A. Korobkov, and S. Venkateswaran. Parallel power grid analysis using distributed direct linear solver. In *2014 IEEE International Symposium on Electromagnetic Compatibility (EMC)*, pages 866–871, Aug 2014.
[7] MPICH. https://www.mpich.org/.
[8] M. Zaharia, M. Chowdhury, A. Dave T. Das, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *NSDI*, pages 2–2. USENIX Association, 2012.
[9] T. W. Huang, M. D. F. Wong, D. Sinha, K. Kalafala, and N. Venkateswaran. A distributed timing analysis framework for large designs. In *ACM/IEEE DAC*, pages 1–6, June 2016.
[10] T.-W. Huang, C.-X. Lin, and Martin D. F. Wong. DtCraft: A Distributed Execution Engine for Compute-intensive Applications. In *IEEE/ACM ICCAD*, pages 757–765, Nov 2017.
[11] Kai Sun, Quming Zhou, Kartik Mohanram, and D. C. Sorensen. Parallel domain decomposition for simulation of large-scale power grids. In *IEEE/ACM ICCAD*, pages 54–59, Nov 2007.
[12] PETSC. http://www.mcs.anl.gov/petsc/.
[13] P. Sun, X. Li, and M. Y. Ting. Efficient incremental analysis of on-chip power grid via sparse approximation. In *ACM/IEEE DAC*, pages 676–681, June 2011.
[14] Sani R. Nassif. Power grid analysis benchmarks. In *IEEE/ACM ASP-DAC*, pages 376–381, 2008.