Electric Power Systems Research 114 (2014) 118–125

Contents lists available at ScienceDirect

Electric Power Systems Research

journal homepage: www.elsevier.com/locate/epsr

SCALE: A hybrid MPI and multithreading based work stealing approach for massive contingency analysis in power systems

Siddhartha Kumar Khaitan*, James D. McCalley

Department of Electrical and Computer Engineering, Iowa State University, Ames, IA, USA

ARTICLE INFO

Article history:
Received 8 March 2014
Received in revised form 19 April 2014
Accepted 21 April 2014
Available online 13 May 2014

Keywords:
Hybrid MPI and multithreading
Dynamic contingency analysis
Massive parallelization
Time domain simulation

ABSTRACT

In this paper, we present SCALE, a hybrid message passing interface (MPI) and multithreading based work Stealing approach for massive Contingency AnaLysis in powEr systems. SCALE performs time domain simulation of power systems using efficient numerical algorithms and scales the approach for analyzing a large number of contingencies using MPI. For achieving dynamic load balancing, SCALE uses efficient implementation of the work stealing algorithm. SCALE uses a hybrid MPI and multithreading based implementation, where MPI is used for communication between different nodes (processors) and multithreading is used within each processor to facilitate implementation of non-blocking version of work-stealing algorithm. We have evaluated SCALE on a large, 13,029 bus system using thousands of contingencies. Also, we have compared the dynamic load balancing based scheduling approach of SCALE with a state-of-art scheduling approach, namely master–slave scheduling approach. The results show that SCALE is effective in analyzing a large number of contingencies and scales well to a large number of processors.

1. Introduction

With increasing emphasis on analyzing $N-k$ contingencies for ensuring power system security, modern energy management systems routinely perform analysis of a large number of contingencies. Since modeling of power system equations leads to highly stiff differential and algebraic equations (DAEs), simulation of power systems is extremely computation intensive and hence, analysis of a large number of contingencies requires excessive simulation time.

To address this issue, researchers have proposed techniques to parallelize contingency analysis using high performance computing (HPC) resources. However, a limitation of most existing methods of contingency analysis is that they use static scheduling techniques. The static scheduling works by using a fixed allocation of tasks to processors and thus, incurs no dynamic scheduling overhead. However, in the worst case, the difference in the completion time of the fastest and the slowest processor can become extremely large. This happens when the contingencies require different simulation times. In such cases, the static scheduling algorithm leads to poor load balancing. This, in turn, leads to wasting computation resources and increase in the actual completion time of the overall simulation. A few other techniques use master–slave dynamic scheduling methods, however, due to the possibility of contention for access to the master processor, these techniques do not offer sufficient computational efficiency. Thus, the state-of-the-art in power system contingency analysis calls for accelerating contingency analysis and also achieving load balancing to maximize the resource usage efficiency.

In this paper, we present SCALE, a hybrid MPI and multithreading based work Stealing approach for massive Contingency AnaLysis in powEr systems. SCALE performs time domain simulation (TDS) of power systems using efficient numerical algorithms (Section 3.1). To solve the DAEs arising in power system modeling, SCALE uses linear solvers, nonlinear solvers and integration solvers. Further, the analysis of contingencies is parallelized using MPI in C++(Section 3.2). Finally, using efficient work stealing based scheduling, load balancing is achieved on the running processors (Section 3.3).

For implementing work stealing, SCALE leverages a hybrid MPI and multithreading based implementation. SCALE uses MPI for communication between different nodes (processors) and multithreading for communication within each processor to facilitate implementation of non-blocking version of work stealing algorithm (Section 4). To offer insights into the utility of scheduling techniques in power systems, we discuss several issues related to implementation of work stealing (Section 4). We also analyze the advantages, limitations and implementation overhead of SCALE and compare it to that of master–slave scheduling techniques (Section 4.3).

* Corresponding author. Tel.: +1 515 294 5499; fax: +1 515 294 4263.
E-mail addresses: skhaitan@iastate.edu (S.K. Khaitan), jdm@iastate.edu (J.D. McCalley).

http://dx.doi.org/10.1016/j.epsr.2014.04.017
0378-7796/

Simulation of thousands of contingencies have been performed for a large power system with 13,029 buses, 431 generators, 12,488 branches and 5950 loads (Section 5). The results show that SCALE is effective in reducing the simulation time and achieving load balancing. Since load balancing naturally translates into avoidance of idle time of the processors, SCALE also enables significant economic gains, energy savings and enables analysis of larger number of contingencies within the same time-budget.

2. Related work

Recent years have witnessed a phenomenal growth in the application of High performance computing (HPC) to accelerate computation intensive problems in several areas, such as power systems [1–3]. The widely used HPC implementation techniques include distributed memory model (MPI) [4] and shared memory model using threads (OpenMP or POSIX threads) [5].

One of the challenges in the use of parallelization techniques is to achieve load balancing to maximize the resource usage efficiency and reduce the completion time of the task. This requires efficient scheduling techniques. In the literature several techniques have been proposed for scheduling which also accomplish load balancing. Some researchers have used master–slave scheduling technique [6–8] for contingency analysis. We discuss master–slave scheduling technique in more detail in Section 3.3.

Work stealing is an efficient load balancing technique which, when using P processors, achieves P-fold speedup in the parallel part and uses at most P times more memory than what it uses for a single processor [9]. In the literature, several variations of work stealing have been proposed (e.g. [10–14]). Zhou et al. [15] use work stealing to achieve load-balancing in GPUs. Blumofe et al. [16] propose a programming language, named Cilk which uses work stealing for enabling multithreaded parallel computing. Van et al. [17] use work stealing for wide-area networks [17].

Flood et al. [18] propose a multiprocessor garbage collection framework which utilizes work stealing to balance the work of tracing the object graph. For MPI platforms, Pezzi et al. propose a hierarchical work stealing (HWS) algorithm [19]. HWS employs a hierarchical structure of managers (i.e. master) and workers (i.e. slaves), which are arranged in a binary tree structure. The inner-nodes of the tree operate as managers and the leaf nodes operate as the worker nodes. Since in MPI platforms, processes can only communicate if they share an inter communicator, enabling one-to-one communication between large number of workers requires high implementation cost. HWS aims to reduce this cost by using managers which facilitate the task of communication between workers by mediating. However, their technique has the limitation of requiring several extra master (manager) nodes. Further, since each work stealing request must go through the manager nodes, compared to one-to-one communication, each work stealing request is slowed-down. Another variant of work stealing is cluster aware hierarchical work stealing [20,21]. This technique provisions that the processors be arranged in a specific topology (e.g., tree topology). This minimizes the communication-cost between processors.

Dinan et al. [22] implement work stealing for 8192 processors with a distributed memory systems. In their method, each process maintains a local task-queue, which is split into public and private portions to enable non-blocking stealing. For implementation, the authors use PGAS (Partitioned Global Address Space) programming model. The PGAS model provides a global view of physically distributed data and efficient one-sided access. Their method stores the distributed task queues in the global address space. Using this flexibility, the steal-operations take place without interrupting the victim process. Saraswat et al. [23] discuss an approach for efficiently extending work stealing to distributed memory systems. In their method, when a node is unable to find work after a certain number of unsuccessful steals, it is tagged as dormant and does not make attempts to further steal the requests for a certain time. Through this, their method reduces the total number of stealing requests.

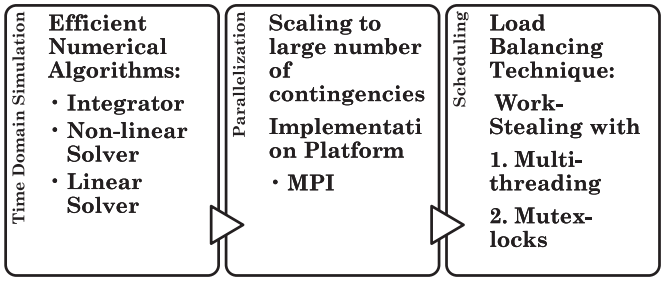

Fig. 1. Overview of SCALE approach.

Chen et al. [24] evaluate the impact of work stealing based thread scheduling algorithm on on-chip cache sharing in the context of multithreaded programs. They also compare the work stealing based scheduler with another greedy scheduler, namely parallel depth first scheduler. In a CMP (chip multiprocessors) environment, many multithreaded programs provide opportunities for sharing the cache constructively, such that threads do not interfere with each other, rather they mutually share the cache. By using work stealing based scheduling, the scheduling of tasks of a parallel merge-sort algorithm to processors can be altered in such a way that the cooperative threads that share an address space use the same cache. Thus, effective use of work stealing leads to reduction in the number of off-chip misses and large saving in the processor energy consumption.

Chiang et al. [25,26] propose TEPCO-BCU method for enabling fast analysis of contingencies in power systems. Their method works by computing the controlling UEP (unstable equilibrium point) of a reduced state model (instead of the original model), and relates the computed controlling UEP to the controlling UEP of the original model. In contrast, SCALE approach uses the computational-capability of multiple nodes and multiple cores on a computing system to accelerate the task of contingency analysis. Thus, our work is orthogonal to their work and can be synergistically integrated with it.

Thus, work stealing has been widely studied and used in several fields. In this paper, we propose a MPI based multithreaded work stealing implementation, which addresses several shortcomings of work stealing implementations in the previous works. The more details of our approach are provided in the next two sections.

3. SCALE: system design

Fig. 1 shows the overview of the SCALE approach. In this section, we describe each of the components of SCALE.

3.1. Time domain simulation

For time domain simulation of power system, SCALE uses a high-speed power system simulator [27,28]. The simulator models the power system by a large number of DAEs. To solve these DAEs, three classes of algorithms are used, namely integrator, nonlinear solver and linear solver.

Newton methods are the ubiquitous choice for the solution of nonlinear equations. We have developed several Newton-based nonlinear solvers, including a conventional Newton solver and the relaxed Newton method to avoid repeated Jacobian calculations, which is commonly called very dishonest Newton method (VDHN).

The simulator is fully modular to interface any linear solver. A number of linear solvers have been interfaced and tested. Of all the linear equation solvers interfaced (KLU, UMFPACK, SuperLU, PARDISO, etc.), [29–31] KLU is found to be most computationally efficient and the fastest [32,33]. KLU is a serial platform based linear equation solver.

The simulator interfaces with a number of implicit and explicit integrators including trapezoidal method, Euler method and the BDF integrator IDAS [34]. IDAS is the most efficient of all integrators. All results presented in this paper are with KLU solver used within the IDAS integrator.

3.2. Parallelization: scaling-up contingency analysis

To simulate a large number of contingencies, SCALE parallelizes the contingency analysis using MPI. MPI is a well-known standard message passing library for the distributed memory systems. The choice of MPI is guided by the fact that it is a vendor-independent library. Also, MPI is a industry standard which has been adopted by many commercial vendors and allows easy integration with C++code. Finally, MPI allows point-to-point communication and collective operations, thus giving flexibility and efficiency to the designer. Thus, use of MPI enables SCALE to be portable.

3.3. Scheduling: achieving load balancing

Scheduling techniques can be broadly divided into static and dynamic scheduling techniques. In this paper, we focus on dynamic scheduling techniques. We first briefly discuss master–slave scheduling and then describe work stealing based scheduling in detail.

Master–slave dynamic scheduling: Master slave scheduling technique works by using a processor as the master and rest of the processors as the workers (slaves). The master thread assigns one (or more) task to each of the workers in the beginning of the execution. Afterwards, the workers start executing their tasks. At the completion of their tasks, the slaves request new tasks from the master. Algorithm 1 shows the pseudo code master–slave based dynamic load balancing algorithm.

Algorithm 1 *(The Master–slave Scheduling Algorithm).*

```
Input: A task-list T and a slave processor-list S and a Master-node m
Output: A load-balanced (best-effort) allocation of tasks to processors
//Initialization;
foreach Slave-Processor s in S do
    if T is empty then
        break;
    end
    Remove a task t from T;
    Assign t to s;
end
Algorithm for Master-node m ;
while T is not empty do
    Wait for the task-request from a slave;
    if a task-request arrives from slave s then
        Remove a task t from T ;
        Allocate t to s;
    end
end
Algorithm for any slave-node s ;
while there is a task to be run do
    Finish the task;
    Request a task from m;
    if no task is available then
        break;
    end
end
```

In general, master–slave scheduling provides much better load balancing than the static scheduling method. Compared to work stealing (discussed next), master–slave scheduling method has the advantage that it does not require a special hardware where every node (processor) is able to communicate with every other node. Rather, only the master node communicates with the other nodes and the worker nodes only communicate with the master node. Despite these advantages, the disadvantage of the master–slave scheduling method is that it leads to wastage of the master processor, since the master node does no useful work. Also, if multiple worker processors finish their tasks and then send task requests to the master node simultaneously, contention arises which needs to be addressed using proper synchronization. As the number of the processors increase, this problem is likely to become worse. Some implementations [8] utilize the concept of multiple masters with multi-counters to reduce contention at a single master. However, the limitation of this method is that if one master exhausts its work queue, it requests pending works from other masters which leads to high latency. Additionally, since multiple masters do not perform useful work, the overhead of implementation of the method increases.

Work Stealing Dynamic Scheduling: Work stealing (also called task-stealing or random-stealing) [35,9,17] is a well known dynamic scheduling method, which is based on a simple observation that scheduling with load-balancing can be achieved, if a processor with no pending task is allowed to steal a task from the processor with excess tasks. The difference between master–slave scheduling and work stealing is that, in master–slave a single master allocates the tasks to all the slaves, while in the work stealing, all processors can communicate with the other processors. Thus, work stealing is a distributed algorithm and hence, it scales much better than the master–slave scheduling method. Several researchers have presented theoretical analysis to characterize the performance of work stealing. Blumofe and Leiserson [9] have shown that the expected time of executing a fully-strict (Fully-strict computation is one where the data dependencies of a worker go to its parent only.) computation with P processors, using their work stealing based scheduler, is given by:

$$t = \frac{t_1}{P + O(t_\infty)} \tag{1}$$

Here t_1 denotes the minimum execution time of the computation with a single processor (i.e. serially). Also, t_∞ denotes the minimum execution time with infinite number of processors. The expected total cost of communication of the algorithm is $P \times t_\infty(1 + N_d) \times S_{max}$, where N_d is the maximum number of times a thread synchronizes with its parent, and S_{max} is the size of largest activation record of any thread [9]. Further, the storage space required by the algorithm is given by $s_1 P$, where s_1 is the minimum space required with a single processor. Thus, the work stealing algorithm is efficient in terms of time, space and communication [9]. While these theoretical bounds hold for fully-strict computations, work stealing has also been shown to be efficient for those programs which are not fully-strict [36].

SCALE implements a non-blocking version of the work stealing algorithm [35]. Compared to other versions of work stealing, the non-blocking version allows work stealing by other processors even when the current processor is busy executing its own task. Thus, the non-blocking version leads to reduced overhead of synchronization and idle-wait time. We also note that different contingencies can be analyzed independent of each other, since their analysis does not depend on the result of analysis of other contingencies. Thus, the theoretical bounds on the efficiency of work stealing (as shown in [9]) apply in the context of contingency analysis.

The SCALE approach also provides several advantages over previous task-scheduling approaches, such as [37]. The proactive task scheduling approach in [37] improves upon master–slave scheduling and requires the use of a master to add the tasks in the queue

of slave processors and transfer those tasks to the queue of other processors toward the end to achieve load-balancing. However, as the number of slave processors increase, this will incur increasing overhead and hence, this approach does scale well. In general, any master–slave based dynamic load-balancing will have higher communication requirements and thus higher chances of contention especially as the number of processors increase. On the other hand, when the number of processors are less, significant fraction is wasted as in the case of 2 processors, where 50% is wasted. In comparison, in SCALE approach, different workers complete their task in independent manner, without the need of any synchronization and hence, SCALE approach scales well. SCALE approach does not require any processor to act as a master (scheduler) and hence, it does not lead to wastage of computational power.

Algorithm 2 shows the pseudo code of our work stealing based dynamic load balancing algorithm.

Algorithm 2 *(The Work Stealing Based Scheduling Algorithm).*

```
Input: A processor-list P and a task-list T
Output: A best-effort load-balanced allocation of available tasks on processors
//Initialization Step
while True do
    foreach Processor p in P do
        if T is empty then
            break;
        end
        Remove a task rom T, let the task be t ;
        Assign the task t to p;
    end
end
//Each processor has two threads: 1. executor thread and 2. polling thread
For executor thread on processor p ;
while p has unfinished tasks do
    foreach unfinished task t at p do
        Complete the task t ;
    end
    foreach p' in P-{p} do
        Try stealing a task from processor p' ;
        if The stealing was successful then
            Assign stolen task to processor p ;
            break;
        end
    end
end
For polling thread on processor p ;
while True do
    if p' sends a stealing request then
        if p has an unstarted task then
            Let t be such task ;
            Remove t from task list of p ;
            Send t to p' ;
        else
            Return NULL to processor p' ;
        end
    end
end
Wait on a barrier for all the processors;
Terminate all threads and all processors;
```

Here, each processor executes two threads. These threads are called the executor/worker thread and the poller thread. The worker thread executes the tasks and the poller thread polls for steal requests coming from thief processors at regular intervals. This enables SCALE to facilitate answering the task-stealing request sent by the thief, while the victim continues to execute its own task.

4. SCALE implementation

4.1. Hybrid MPI and multithreading framework

In this section, we focus on the optimizations incorporated in parallelizing the contingency analysis. Parallelization involves intelligent management of division of work to the computing elements (either threads or processors) and aggregation of the result. Since different parallelization methods have different advantages and limitations, a careful choice of the parallelization technique is required for efficiently implementing the work stealing technique. Based on this observation, SCALE uses a hybrid MPI and multithreading based implementation (see Fig. 2). MPI enables communication between different nodes of a cluster and thus exploits the distributed memory architecture of the cluster, while multithreading enables sharing the resources of shared memory processors within a single node. The main difference between multithreading and MPI lies in their communication mechanism. Threads share the physical memory and hence, they can easily share intermediate variables and the final result without requiring any explicit communication. Thus, the time of communication is avoided although handling of multiple threads incurs a certain overhead. On the other hand, in the case of MPI, the nodes do not share the physical memory. Hence, they communicate with each other using the MPI communication functions. Thus, MPI incurs higher communication cost than the multithreading and is useful when the communication cost is negligible in comparison to the processing time of each task. To benefit from the strengths of both, SCALE uses MPI based communication between different nodes and multithreading between each node.

4.2. Optimizations incorporated

A naïve implementation of work stealing is likely to be inefficient due to constraints presented by real hardware and hence, it is unlikely to achieve the performance guarantees promised by theoretical analysis. For this reason, we have made significant efforts to optimize the implementation of work stealing in SCALE, which are discussed below.

First, to reduce the contention caused by communication, we implement a specific polling order. When the task queue of the processor is empty (called free-processor), its worker thread sends a stealing requests to other processes to find a pending work. When there are a large number of available processors, the number of options available for polling become very large and hence the order of polling becomes important. A naïve approach of polling might be to always poll in a fixed order, for example, polling processors 0, 1, 2, … in order. However, in this scenario, all free-processors will poll processors 0, 1, etc.; this is likely to create contention on the processes 0, 1, etc., which is likely to increase the overhead of implementation. To alleviate this, SCALE provisions random-polling. In other words, each free processor chooses a victim processor in a random order. Thus, by using statistical properties, SCALE provisions equal distribution of the stealing requests to all the processors.

Second, based on the responses received from a victim processor to the stealing request, the thief marks the victim as idle/active. Using this information, further requests are not sent to those processors which are known to be idle. This helps in reducing the request traffic.

Third, to ensure correctness and proper progress of the algorithm, it is imperative that no deadlocks occur and a task is executed exactly once. However, stealing of the task by a thief from the victim processor has the potential of creating a deadlock if, simultaneously, another thief or the victim processor also tries to schedule the process for running. To avoid this situation, while achieving

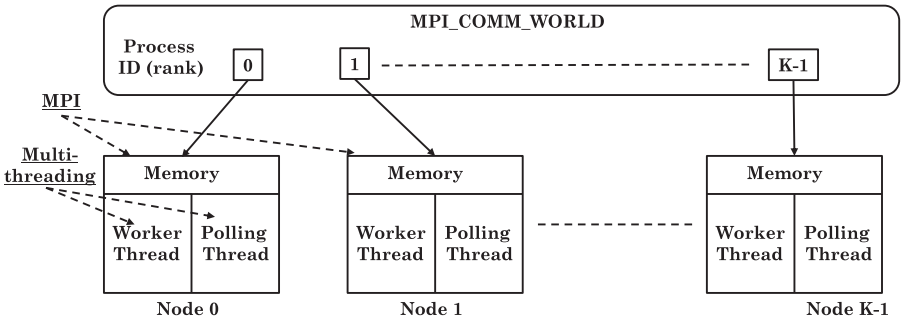

Fig. 2. Use of hybrid MPI and multithreading scheme (assuming one CPU per node).

synchronization, we use mutex locks [38] as shown in Fig. 3. This ensures that when the worker thread is stealing (removing) a task from the task list, the poller thread is not able to allocate this task to another thief. In this manner, synchronization ensures that critical sections of the task are executed atomically in the threads.

Intuitively, in the beginning of the execution of the algorithm, most processors remain busy with the tasks allocated to them. Work stealing actually begins when the fastest processor (or the one with smallest amount of workload) finishes its allocated tasks. Using this observation, the task accomplished by the poller thread can be reduced. For this purpose, we provision that in the beginning of the execution, the poller thread polls after long intervals (5 s in our implementation) and later, after receiving the first steal request, the period of polling is reduced (2 s in our implementation). Using this strategy enables us to allocate more resources to the executor thread for efficiently executing tasks in the beginning of the execution, while still being responsive to the steal requests in the later phases of the execution.

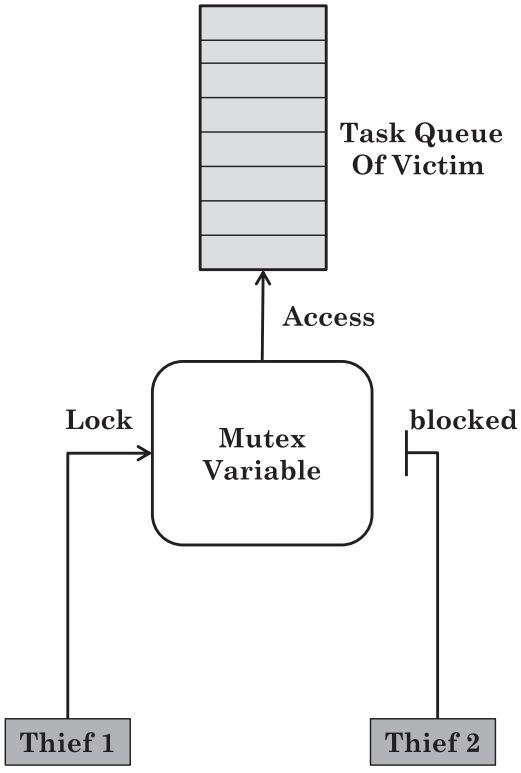

Fig. 3. Example of use of mutex lock for avoiding deadlock. (When Thief 1 is accessing the task-queue of a victim for stealing, Thief 2 is blocked and cannot access the task-queue.)

4.3. Implementation overhead of SCALE

We now focus on the implementation overhead of SCALE. For enabling work stealing, each processor uses two threads. With the above mentioned optimizations, the worker thread remains active for most of the time, while the poller thread remains active for a much smaller fraction of time. Hence, most of the processor resources are used by the worker thread. When the task lengths are sufficiently large, the overhead of communication and polling become a much smaller fraction of the computation time, and hence the overhead of implementation reduces. Since polling requests are distributed randomly and processors send stealing requests *only* when their task lists are empty, the probability of contention is much smaller. Comparatively, the master slave method uses only a single thread in both the master and the slave. However, since all free processors send task requests to the master, the probability of contention is much higher. Higher contention also translates into increased idle time of the processors, which in turn increases the completion time and resource wastage. Thus, compared to master slave method, SCALE incurs only small additional overhead. However, as shown in Section 5, SCALE offers significant computational gains, and hence its use is justified.

4.4. Limitations of work stealing method

Some researchers have discussed the limits of work stealing algorithm. Saha et al. [39] investigate the scalability of work stealing on a CPU having up to 32 cores, where each core executes 4 hardware threads in a round-robin manner. They have observed that work stealing causes contention for accessing task-queues, which can limit or even degrade the application performance as the number of cores increases. In such scenarios, static scheduling proves to be advantageous since it does not incur the overhead of contention. Similarly, Vrba et al. [40] have shown that a careful static assignment of the tasks to the available processors can match the performance of the work stealing algorithm on finely-granular parallel applications. However, in operating the power systems, the exact simulation time of contingencies depends on the nature of the disturbance and characteristics of the power system. Hence, a pre-computation of optimal assignment of contingency analysis tasks on available processors is infeasible. Also, on our experimentation platform, the communication cost between processors is negligibly small and hence the overhead of work stealing scheme is extremely small (Section 5).

5. Results and discussion

To evaluate SCALE approach, we simulate a large power system with 13,029 buses, 431 generators, 5950 loads and 12,488 branches. The different contingencies simulate different disturbance events such as branch faults, bus faults, branch-tripping,

Table 1
Simulation times with master slave (MS) method and SCALE (in s).

# Contingencies	$N=8$		$N=64$		N=128		$N=256$	
	MS	SCALE	MS	SCALE	MS	SCALE	MS	SCALE
1000	4256	3774	503	470	256	236	186	148
2000	8492	7416	1007	933	556	490	370	283
3000	13,849	11,141	1465	1368	776	718	504	434
4000	16,988	14,767	1945	1821	1059	956	628	570
5000	21,136	18,555	2392	2251	1270	1177	856	688
10,000	42,439	36,812	4820	4535	2499	2345	1577	1420

generator tripping and generator fault and combinations of them. The simulation time of different contingencies varies between 10 and 25 s. The power flow program with the software solves the system for initialization of the time domain simulation. At the point of each contingency (or event) time instant, the contingency scenario is set by modifying the Y matrix and reinitializing the dynamics within the IDAS integrator.

The experiments were performed on a large cluster. The performance metric is taken as the wall-clock time, since this represents the total time needed for task completion. We simulate between 1000 to 10,000 contingencies, using 8, 64, 128 and 256 processors. Since we simulate a large number of contingencies, the amount of time required for simulating them using only a few processors will be very large. For example, by extrapolating the simulation time with 8 processors to estimate the time required for a single processor, we find that the time is nearly 4 days which represents an infeasible amount of time. This also explains the importance of using the SCALE approach for analyzing massive contingencies.

Table 1 shows the simulation times taken by the master–slave scheduling method and the SCALE work-stealing scheduling method. Clearly, SCALE outperforms master–slave scheduling for all the cases tested. For 10,000 contingencies and 8 processors, SCALE provides nearly 1.5 h of saving compared to master slave scheduling method. The computation time saved by SCALE can allow the online analysis of a much larger number of contingencies within the same time budget.

To see how the simulation times vary with the number of processors, we take the simulation time with master–slave scheduling method for 8-processors as the baseline and define a speedup metric $\Upsilon(N, C)$ as follows.

$$\Upsilon(N, C) = \frac{T_{\text{masterslave}}(8, C)}{T(N, C)} \tag{2}$$

Here N represents the number of processors and C represents the number of contingencies analyzed. $T_{\text{masterslave}}(8, C)$ denotes the simulation time for C contingencies with master–slave scheduling using 8 processors. $T(N, C)$ denotes the simulation time with either master–slave method or SCALE, for N processors and C contingencies. The speedup values are shown in Figs. 4 and 5.

With 8, 64, 128 and 256 processors, SCALE provides up to 1.24×, 10.1×, 19.3× and 31.91× speedup, respectively. Thus, compared to master–slave scheduling method, SCALE offers excellent speedup. With the computational advantage offered by SCALE, the time consumed in contingency analysis can be reduced by an order of magnitude. This proves the effectiveness and utility of SCALE in modern energy management systems.

To gain further insights and see the scaling behavior, we take the simulation times with the SCALE method itself with 8-processors as the baseline and compute speedup values. These values are shown in Fig. 6. Clearly, the speedup offered by SCALE increases linearly with the number of processors with slight degradation for 256 processors. There are several factors which contribute to high computational gains in SCALE, such as (1) the use of work-stealing

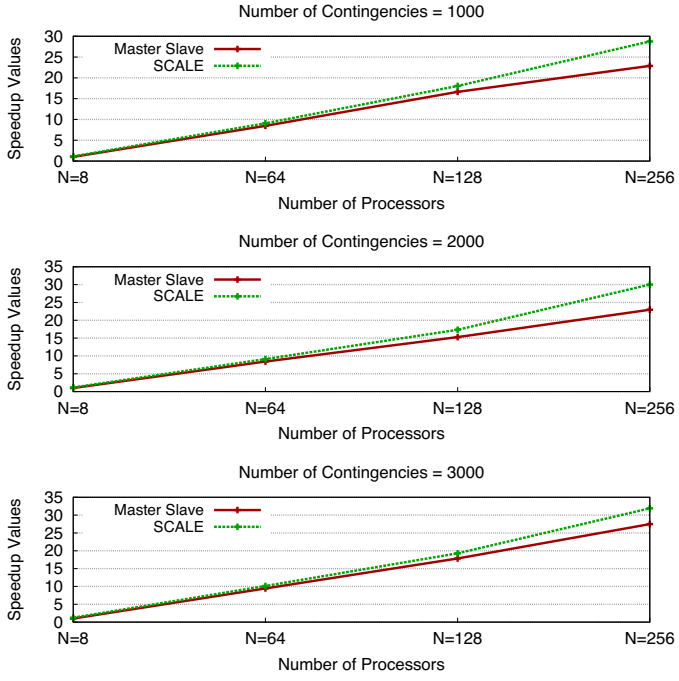

Fig. 4. Speedup results for master slave method and SCALE for different contingencies (for $C=1000$, 2000 and 3000).

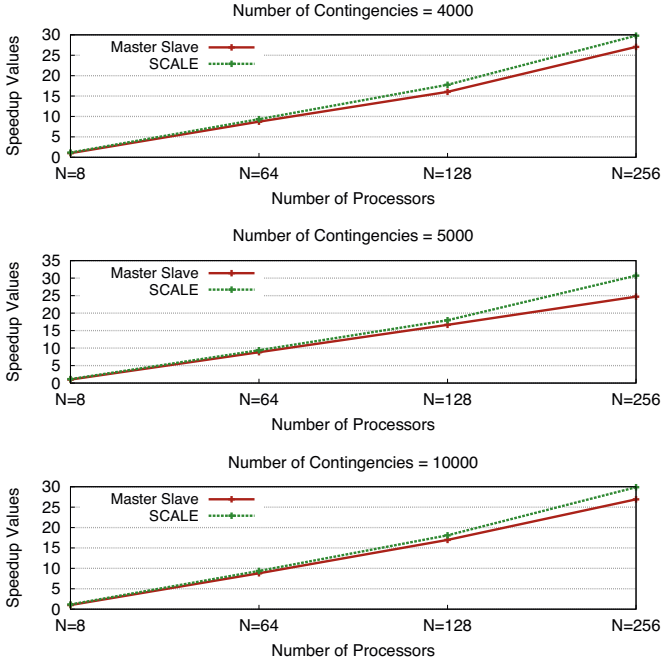

Fig. 5. Speedup results for master slave method and SCALE for different contingencies (for $C=4000$, 5000 and 10,000).

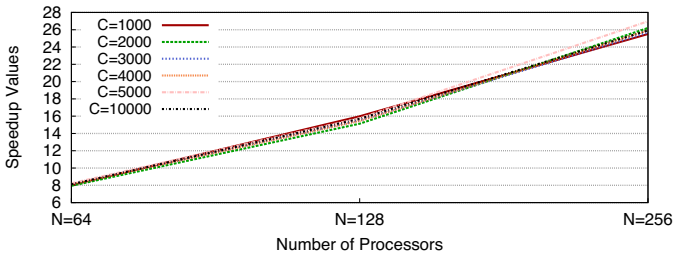

Fig. 6. Speedup results for SCALE for different contingencies (baseline is SCALE with 8 processors).

algorithm which has theoretically proven efficiency, (2) there is no node wastage as in master–slave, (3) multithreading to implement the worker and the polling thread, (4) efficient polling algorithm and (5) dynamic load-balancing.

We have also confirmed that the output (in terms of values of voltage, etc., and determination of stability/instability) obtained using serial execution are identical with those obtained using SCALE approach. This is also expected, since SCALE approach does not use any approximation or change of numerical algorithm or the procedure of time domain simulation. Similarly, the robustness is not affected since the changes in the code are minimal. In fact, in terms of real-world application, SCALE approach enables obtaining the results of contingency analysis much more quickly which facilitates robust operation of control centers, since the operators can become equipped with situational awareness.

6. Conclusion and future work

In this paper, we presented SCALE, a technique for massively parallel contingency analysis using dynamic load balancing. To optimize the load distribution across multiple processors for the application of parallel power system contingency analysis, SCALE leverages efficient implementation of work-stealing on a large number of processors. The results of experiments performed on a large power system with thousands of contingencies show that SCALE approach is effective in simulating a large number of contingencies and also offers significant speedup over master–slave based scheduling method. Our future work will focus on evaluating SCALE with much larger number of contingencies and further exploring the opportunities of accelerating SCALE. Also, we are currently investigating the effect of task-length on the speedup achieved by the SCALE algorithm. This will help us in further making SCALE a valuable approach for providing situational awareness to the operational personnel.

Acknowledgements

The authors gratefully acknowledge the use of Iowa State University's supercomputers, namely Cystorm and Cyence for developing the code and running the experiments shown in the paper.

References

[1] R. Green, L. Wang, M. Alam, C. Singh, Intelligent and parallel state space pruning for power system reliability analysis using MPI on a multicore platform, in: Innovative Smart Grid Technologies (ISGT), 2011 IEEE PES, IEEE, 2011, pp. 1–8.
[2] V. Jalili-Marandi, Z. Zhou, V. Dinavahi, Large-scale transient stability simulation of electrical power systems on parallel GPUs, IEEE Trans. Parall. Distrib. Syst. (2011) 1.
[3] S.K. Khaitan, J.D. McCalley, EmPower: an efficient load balancing approach for massive dynamic contingency analysis in power systems, in: High Performance Computing, Networking, Storage and Analysis (SCC), SC Companion, 2012, pp. 289–298.
[4] A. Agrawal, S. Misra, D. Honbo, A. Choudhary, Mpipairwisestatsig: parallel pairwise statistical significance estimation of local sequence alignment, in: Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing, ACM, 2010, pp. 470–476.
[5] L. Dagum, R. Menon, Openmp: an industry standard api for shared-memory programming, IEEE Comput. Sci. Eng. 5 (1998) 46–55.
[6] A. Mittal, J. Hazra, N. Jain, V. Goyal, D. Seetharam, Y. Sabharwal, Real time contingency analysis for power grids, in: Euro-Par 2011 Parallel Processing, 2011, pp. 303–315.
[7] I. Gorton, Z. Huang, Y. Chen, B. Kalahar, S. Jin, D. Chavarria-Miranda, D. Baxter, 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 (e-Science'09), 2009, pp. 277–283.
[8] Y. Chen, Z. Huang, D. Chavarr’í-Miranda, Performance evaluation of counter-based dynamic load balancing schemes for massive contingency analysis with different computing environments, in: Power and Energy Society General Meeting, 2010 IEEE, IEEE, 2010, pp. 1–6.
[9] R. Blumofe, C. Leiserson, Scheduling multithreaded computations by work stealing, in: 35th Annual Symposium on Foundations of Computer Science, 1994 Proceedings, IEEE, 1994, pp. 356–368.
[10] Y. Guo, R. Barik, R. Raman, V. Sarkar, Work-first and help-first scheduling policies for async-finish task parallelism, in: IEEE International Symposium on Parallel & Distributed Processing, 2009 (IPDPS 2009), IEEE, 2009, pp. 1–12.
[11] T. Hiraishi, M. Yasugi, S. Umatani, T. Yuasa, Backtracking-based load balancing, in: ACM Sigplan Notices, vol. 44, ACM, 2009, pp. 55–64.
[12] A. Tzannes, G. Caragea, R. Barua, U. Vishkin, Lazy binary-splitting: a run-time adaptive work-stealing scheduler, in: ACM SIGPLAN Notices, vol. 45, ACM, 2010, pp. 179–190.
[13] M. Michael, M. Vechev, V. Saraswat, Idempotent work stealing, in: ACM Sigplan Notices, vol. 44, ACM, 2009, pp. 45–54.
[14] D. Cederman, P. Tsigas, Dynamic Load Balancing Using Work-Stealing, 2011.
[15] K. Zhou, Q. Hou, Z. Ren, M. Gong, X. Sun, B. Guo, Renderants: interactive Reyes rendering on GPUs, in: ACM Transactions on Graphics (TOG), vol. 28, ACM, 2009, p. 155.
[16] R. Blumofe, C. Joerg, B. Kuszmaul, C. Leiserson, K. Randall, Y. Zhou, Cilk: an efficient multithreaded runtime system, in: Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'95), vol. 30, ACM, 1995.
[17] R. Van Nieuwpoort, T. Kielmann, H. Bal, Efficient load balancing for wide-area divide-and-conquer applications, in: ACM SIGPLAN Notices, vol. 36, ACM, 2001, pp. 34–43.
[18] C. Flood, D. Detlefs, N. Shavit, X. Zhang, Parallel garbage collection for shared memory multiprocessors, in: Proceedings of the 2001 Symposium on Java TM Virtual Machine Research and Technology Symposium, vol. 1, USENIX Association, 2001, p. 21.
[19] G. Pezzi, M. Cera, E. Mathias, N. Maillard, On-line scheduling of MPI-2 programs with hierarchical work stealing, in: 19th International Symposium on Computer Architecture and High Performance Computing, 2007 (SBAC-PAD 2007), IEEE, 2007, pp. 247–254.
[20] J. Baldeschwieler, R. Blumofe, E. Brewer, Atlas: an infrastructure for global computing, in: Proceedings of the 7th Workshop on ACM SIGOPS European Workshop: Systems Support for Worldwide Applications, ACM, 1996, pp. 165–172.
[21] M. Backschat, A. Pfaffinger, C. Zenger, Economic-based dynamic load distribution in large workstation networks, in: Euro-Par'96 Parallel Processing, Springer, 1996, pp. 631–634.
[22] J. Dinan, D. Larkins, P. Sadayappan, S. Krishnamoorthy, J. Nieplocha, Scalable work stealing, in: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis, ACM, 2009, p. 53.
[23] V. Saraswat, P. Kambadur, S. Kodali, D. Grove, S. Krishnamoorthy, Lifeline-based global load balancing, in: Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming, ACM, 2011, pp. 201–212.
[24] S. Chen, P. Gibbons, M. Kozuch, V. Liaskovitis, A. Ailamaki, G. Blelloch, B. Falsafi, L. Fix, N. Hardavellas, T. Mowry, et al., Scheduling threads for constructive cache sharing on cmps, in: Proceedings of the Nineteenth Annual ACM Symposium on Parallel Algorithms and Architectures, ACM, 2007, pp. 105–115.
[25] H.-D. Chiang, Direct Methods for Stability Analysis of Electric Power Systems: Theoretical Foundation, BCU Methodologies, and Applications, John Wiley & Sons, Hoboken, New Jersey, 2011.
[26] H.-D. Chiang, Y. Tada, H. Li, Power System On-line Transient Stability Assessment, Wiley Encyclopedia of Electrical and Electronics Engineering, Hoboken, New Jersey, 2007.
[27] S.K. Khaitan, J.D. McCalley, High Performance Computing in Power and Energy Systems POWSYS, Springer, New York, 2012, pp. 43–69.
[28] S. Khaitan, C. Fu, J. McCalley, Fast parallelized algorithms for on-line extended-term dynamic cascading analysis, in: Power Systems Conference and Exposition, 2009 (PSCE'09), IEEE/PES, IEEE, 2009, pp. 1–7.
[29] T. Davis, Algorithm 832: Umfpack v4.3—an unsymmetric-pattern multifrontal method, ACM Trans. Math. Softw. 30 (2004) 196–199.
[30] T. Davis, K. Stanley, Klu: a" clark kent" sparse lu factorization algorithm for circuit matrices, in: 2004 SIAM Conference on Parallel Processing for Scientific Computing (PP04), 2004.
[31] O. Schenk, K. Gärtner, Solving unsymmetric sparse systems of linear equations with pardiso, Future Gener. Comput. Syst. 20 (2004) 475–487.

[32] S.K. Khaitan, J.D. McCalley, Tdpss: a scalable time domain power system simulator for dynamic security assessment, in: High Performance Computing, Networking, Storage and Analysis (SCC), 2012 SC Companion, IEEE, 2012, pp. 323–332.
[33] S.K. Khaitan, A. Gupta, High Performance Computing in Power and Energy Systems, Springer, New York, 2012.
[34] R. Serban, C. Petra, A.C. Hindmarsh, User Documentation for IDAS v1.0.0, 2009 https://computation.llnl.gov/casc/sundials/description/description.html
[35] N. Arora, R. Blumofe, C. Plaxton, Thread scheduling for multiprogrammed multiprocessors, in: Proceedings of the Tenth Annual ACM Symposium on Parallel Algorithms and Architectures, ACM, 1998, pp. 119–129.
[36] Ž. Vrba, H. Espeland, P. Halvorsen, C. Griwodz, Limits of work-stealing scheduling, in: Job Scheduling Strategies for Parallel Processing, Springer, 2009, pp. 280–299.
[37] S.K. Khaitan, J.D. McCalley, A. Somani, Proactive task scheduling and stealing in master–slave based load balancing for parallel contingency analysis, Electr. Power Syst. Res. 103 (2013) 9–15.
[38] A. Silberschatz, P. Galvin, G. Gagne, A. Silberschatz, Operating System Concepts, vol. 4, Addison-Wesley, Boston, MA, 1998.
[39] B. Saha, A. Adl-Tabatabai, A. Ghuloum, M. Rajagopalan, R. Hudson, L. Petersen, V. Menon, B. Murphy, T. Shpeisman, E. Sprangle, et al., Enabling scalability and performance in a large scale cmp environment, in: ACM SIGOPS Operating Systems Review, vol. 41, ACM, 2007, pp. 73–86.
[40] P. Zeljko Vrba, C. Halvorsen, Griwodz, Evaluating the run-time performance of kahn process network implementation techniques on shared-memory multiprocessors, in: International Workshop on Multi-Core Computing Systems, 2009.