Achieving Load-Balancing in Power System Parallel Contingency Analysis Using X10 Programming Language

Siddhartha Kumar Khaitan and James D. McCalley

Department of Electrical and Computer Engineering
Iowa State University, Ames, Iowa, USA.
{skhaitan, jdm}@iastate.edu

Abstract

Due to recent trends of expansion and deregulation in power systems, the stress level of power systems has increased which has highlighted the importance of conducting stability analysis. Further, due to increasing emphasis on analyzing $N-k$ contingency, the number of contingencies which are required to be analyzed has greatly increased. To address this challenge, researchers have used parallel computing resources, however, in absence of efficient load-balanced scheduling, parallelization leads to wastage of computation resources. In this paper, we present an approach to parallelize power system contingency analysis using X10 language. We discuss the features of X10 which enable us to achieve high performance gains. Our approach is evaluated using a large 13029-bus power systems. We parallelize contingency analysis over 2, 4, 8 and 16 threads and use efficient work-stealing algorithm to achieve load-balancing.

The results have shown that our approach scales effectively with the number of cores and provides large computational gains. Also, it outperforms a conventional scheduling technique, namely master-slave scheduling.

General Terms Algorithm, design

Keywords X10 programming language, concurrency, parallelization, HPC (high-performance computing), work-stealing, scheduling, load-balancing, power system simulation, control center, time domain simulation.

1. Introduction

In recent years, rising electricity demands and deregulation in power systems have led to significant increase in their sizes. This along with the emphasis on analyzing $N-k$ contingency[1] has led to huge increase in the number of contingencies to be analyzed for the purpose of ensuring stability of the power system. Due to this, serial execution platforms are proving to be insufficient in fulfilling the demands of contingency analysis and hence, in the absence of high-performance computing resources, the power system operators are unable to analyze a large number of contingencies in a short amount of time. This prevents the operators from taking suitable preventive and corrective actions towards failures which may lead to catastrophic events such as blackouts (e.g. the one that happened in August 2003 [5]). To address this, parallelization of contingency analysis is extremely important. However, parallelization also necessitates achieving load-balancing since a load-unbalanced scheduling is likely to lead to wastage of processor resources and in worst case, may even nullify the advantage obtained from parallelization.

In this paper, we present an approach for parallelization of power system contingency analysis while also achieving load-balancing. To achieve parallelization, we use X10 [10, 13] a state-of-the-art open-source parallel programming language which is being developed by IBM. We evaluate our approach using a large 13029-bus power systems. We simulate 500, 1000, 1500 and 2000 contingencies. Also, we simulate contingencies using serial execution and parallel execution using 2, 4, 8 and 16 threads with work-stealing based scheduling[2] for load-balancing. Further, we compare our approach with master-slave scheduling algorithm, which is a well-known conventional scheduling algorithm. The results show that our approach scales well with the number of threads and outperforms master-slave scheduling algorithm.

In this paper, we make three important contributions. First, we highlight the need of using HPC in power system contingency analysis and demonstrate parallelization of

X10'13 June 20, 2013, Seattle, WA, USA.

[1] In power systems, a contingency refers to the possible event of failure of a single or more components. Further, in a power system with a total of N components, a failure in at most k components is termed as $N-k$ contingency.

[2] Note that the X10 runtime also implements a work-stealing scheduler, however our implementation is totally different from that scheduler, as we show in Section 5.

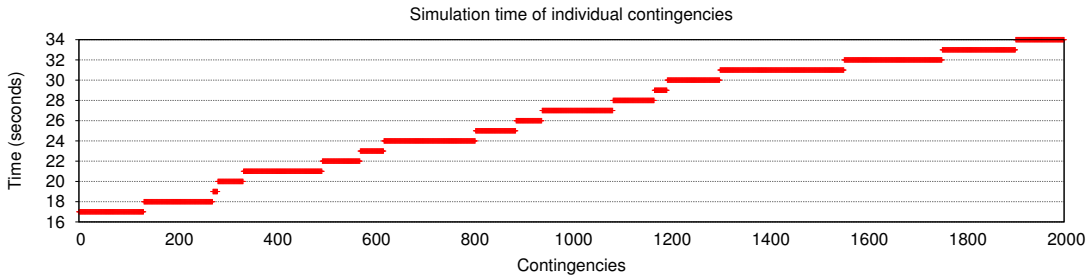

Figure 1: Simulation time variation across contingencies (sorted in ascending order of time)

thousands of contingencies to achieve computational gains. With 16 threads we achieve more than 14× speedup over serial execution and with such speedup, a workload requiring one day of execution time can be executed in less than 2 hours. Second, we propose an approach for accelerating *legacy* software code using widely available general-purpose processors (CPUs). Thus, our approach does not require special purpose hardware (e.g. GPUs or FPGAs) and also does not incur the cost of rewriting and porting legacy code to new platforms. Third, we discuss the details that went into the use of HPC, specifically X10 parallel programming language and work-stealing scheduling algorithm, in the context of contingency analysis in power systems. HPC techniques have been used in several application-domains [3, 14, 15, 23, 26] and we believe that the insights presented in this paper will be extremely helpful for power system operators to bring the benefits of HPC in modern control centers also. Our work provides the power system operators with a decision support tool for making crucial decisions affecting the operation of power systems. The rest of the paper is organized as follows. Section 2 reviews the use of HPC in power systems and also presents a brief background on the languages used. Section 3 discusses the time domain simulation and overall parallelization approach. Section 4 explains the work-stealing scheduling algorithm. Section 5 discusses the features of our implementation. Section 6 presents the experimental platform and results. Finally, we discuss the conclusion and future work in Section 7.

2. Related Work and Background

2.1 Power System Contingency Analysis

Modern control centers routinely analyze a large number of contingencies to assess the ability of power system to sustain any possible component failures which may lead to disastrous consequences such as blackouts. This also enables them to devise suitable preventive and corrective actions. While some of the previous works perform steady state contingency analysis (e.g. [16]), we perform dynamic contingency analysis, which requires more computational resources and execution time.

2.2 HPC in Power Systems

Due to the limits imposed by power dissipation, the clock frequency of computing systems can no longer be increased at the rate that had been sustained during the last several years. Instead, state-of-the-art computing systems are using multiprocessor systems to leverage parallelism instead of frequency scaling for achieving high computational capabilities. Driven by this paradigm shift, in recent years, high performance computing (HPC) has been used in several domains to accelerate computation intensive tasks [22, 24]. In the field of power systems also, HPC techniques have been used to achieve large computational gains in power system stability analysis [17, 18, 21]. Researchers have used several high-performance computing languages and libraries, such as OpenMP [9], Pthreads [20], Cilk [6], D [4], Go [2], Scala [25] and Chapel [8]. In this paper, we use X10 and discuss its unique features below.

2.3 A Brief Background of X10

X10 is a strongly typed, concurrent objectoriented programming language and is designed for providing a higher-level of abstraction than several existing languages (e.g. MPI). The design of X10 is guided by the motivation of providing safety from errors, scalability and flexibility. X10 provides automatic garbage collection and array bounds checking, which significantly simplifies memory management and also avoids a large number of errors, as typically observed in C/C++ programs. X10 aims to improve programmer productivity by making distributed programming easier. Other features of X10 include ability to integrate with C++/Java code. Thus, X10 provides significant advantages compared to conventional languages such as Java or C/C++ [10].

X10 is based on the APGAS (Asynchronous Partitioned Global Address Space) model, where a computation is divided among a set of places or threads, which host single or multiple activities (e.g. computation, I/O etc.). A X10 “place” refers to a single operating system process.

X10 provides programming constructs to allow the programmer to explicitly identify potentially concurrent computations and to easily synchronize them as per the requirements. Thus, it allows easily expressing parallelism and effi-

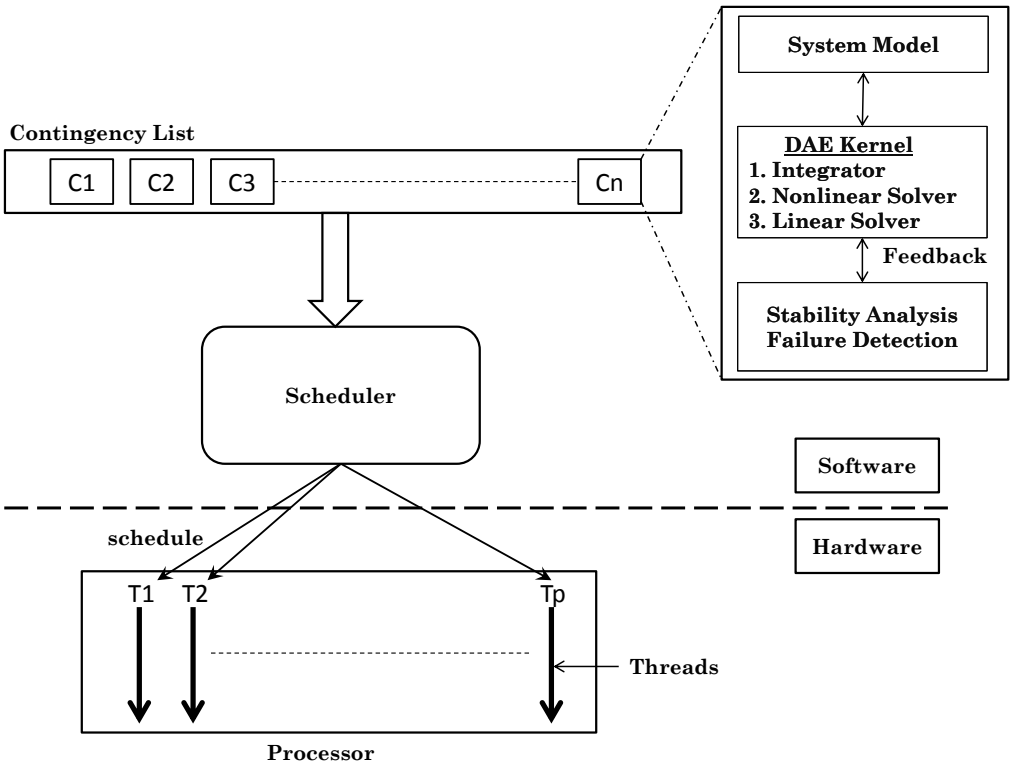

Figure 2: Overall Flow Diagram of Our Approach

ciently exploiting the hardware resources on modern multi-core and clustered architectures.

X10 has been shown to scale to more than 32,000 cores [1], and thus, as a highly scalable language, it offers a promising approach to address the computational demands of contingency analysis.

2.4 Scheduling Techniques For Achieving Load-balancing

As shown in figure 1, the simulation time of contingencies varies a lot. For this reason, efficient scheduling techniques are required to avoid resource wastage. In literature, both static and dynamic scheduling techniques have been used. For the scenarios where task lengths differ widely, the static scheduling techniques lead to poor resource-usage efficiency. In this paper, we do not study static scheduling, since in real-world scenario, the worst-case performance of static scheduling can be very poor. Moreover, obtaining load-balancing using static scheduling generally requires a priori knowledge of the completion time of different tasks, which may be unavailable in some domains as in dynamic contingency analysis.

Among dynamic scheduling techniques, master-slave based dynamic scheduling algorithm has been widely used [16]. Work stealing [7] is a dynamic scheduling method which works on the principle that scheduling with load balancing can be achieved if a worker with no remaining job is allowed to steal a job from other worker. We discuss these scheduling algorithms in more detail in Section 4.

Blumofe et al. have shown [7] that using work-stealing scheduling method with P workers, the time taken in executing a fully-strict computation is given by:

$$T = T_1/P + O(T_\infty) \quad (1)$$

Here T_1 denotes the execution time with 1 worker (i.e. serial execution) and T_∞ denotes the expected execution time with infinite number of workers. Further, the space requirement of the work-stealing method scales only linearly with the number of workers [7]. From this it is concluded that work-stealing algorithm is efficient in terms of space and time requirement. Note that a fully-strict computation is one where the data dependencies of a thread do not go to any thread other than its parent. In the context of this paper, the contingencies can be independently analyzed and hence, the theoretical guarantee provided by them hold for our work also.

3. System Architecture and Design

Figure 2 shows the overall diagram of our approach. In the following sections, we discuss each component of our approach in detail.

3.1 Power System Simulation

To analyze power system contingencies, we use time domain simulation (TDS) approach, where we simulate the response and characteristics of the power system for each time-step. For this purpose, we use a high speed power system simulator [19] which has been verified against commercial software packages. This simulator models the power system differential and algebraic equations (DAEs). These DAEs are solved using three different classes of numerical solvers, namely an integration solver, a linear solver and a nonlinear solver. For each of these classes, the simulator provides several numerical solvers. For integration solver, we use IDAS integrator [29] since it provides high-performance and handles the stiff power system dynamics in efficient and accurate manner. For linear solver, we use KLU [12], which has been found to provide better performance than other solvers such as UMFPACK and SuperLU etc. For non-linear solver we have used VDHN (very dishonest Newton method) solver.

Contingency analysis is regularly performed in all power systems for ensuring safe and reliable operation. For contingency analysis, firstly, a simulator is used for modeling the power system being studied. Then, a set of critical contingencies are identified for which the analysis needs to be performed. Each contingency represents a combination of different events taking place within the simulation time-frame. This is simulated by modeling the effect of these events on system variables such as voltage, rotor angle etc. This is achieved through event-driven modeling approach in the power system simulator. At the end of simulation, the response obtained is analyzed to decide whether the power system is stable for the contingency being analyzed. If the power system is not stable for a particular contingency, further investigation is being made to identify suitable preventive and corrective measures.

3.2 Parallelization Approach

Data parallelism and task parallelism are two widely-used approaches for parallelization. As the name implies, in data parallelism, parallelization is achieved across parts of the data (e.g. pixels in a large image) and in task parallelism, parallelization is achieved across different tasks (e.g. rendering different objects in a scene). Given that computing the response of the system for each time-step requires using multiple solvers, it is evident that TDS does not offer large data-parallelism for simulation of individual contingencies. For this reason, in this paper, we use task-parallelism where analysis of a single contingency is referred to as a single task.

As for programming model, in parallel programming, both shared memory and distributed memory computing have been used. Compared to distributed memory computing, the advantage of shared memory programming is that it uses threads or lightweight processes which typically have small overhead. Moreover, shared memory programming reduces the overhead of data communication between processes. The limitation of shared memory programming is that it does not scale well to large number of cores. In this paper, we use shared memory model. Use of distributed memory computing is planned as a future work.

3.3 Using X10 For Parallelization

We now discuss the basic programming constructs used for parallelization and scheduling and then discuss the actual details of the algorithm in the next section.

In X10, computation is seen in terms of one or more asynchronous *activities* which happens at one or more *places*. To start a new asynchronous activity, we use `async` statement, which represents the fundamental concurrency construct in X10. The newly created activity has access to the same heap of data objects as the current activity, but has its own control stack. Thus, an activity represents a very lightweight thread of execution. An activity can also access the local variables which are declared outside of the `async` block. Unlike the threads in Java, the activities in X10 are not named. Also, unlike Java threads, the activities cannot be interrupted and on its completion, an activity 'dies' (i.e. releases its resources) on its own without requiring the programmer to specify a command. Thus, use of asynchronous activity provides a foundation for lightweight "threads". For sake of convenience, in the remainder of the paper, we use the word 'thread' to denote an X10 activity.

In work-stealing algorithm access to worker queues may be done by multiple workers and hence, it represents a shared data. To protect these shared data from race conditions, we have used `atomic` command. When an atomic block is executed, all the other concurrent activities in the same place are suspended and hence the block is executed in a single step. Note that although an atomic statement might itself involve execution of multiple statements; with respect to all other activities of the system, it is performed in a single step. Since atomic execution leads to serialization of execution, we have taken due care to keep its use minimum.

To provide a barrier synchronization and wait till the completion of all threads, we use `finish` statement. This ensures that till all the activities spawned within the body of `finish` have terminated, the subsequent statement cannot be executed. This is referred to as global termination. In the context of work-stealing algorithm, `finish` command is used for detecting algorithm termination. Use of `atomic` command ensures integrity of shared data in the multithreaded context. This provides the flexibility while leaving the responsibility of lock management and other mechanisms for actually enforcing atomicity to the underlying implementation itself. It has been shown that any program written using the parallel constructs (viz. async, finish, atomic etc.) will never deadlock [28].

To execute each task (i.e. the contingency), we have used `Runtime.execForWrite()` function which runs the specified command in a separate process. We also use the

`close()` function, which ensures that the control of the main program waits till the simulation of individual contingency in the separate process is completed.

4. Work-Stealing Based Load Balancing Algorithm

Algorithm 1 Work-Stealing Scheduling Algorithm For Load-Balancing on P Threads

Require: task-list T
Ensure: A best-effort load-balanced scheduling of tasks.
Allocate P task-queues and assign tasks in T to these task-queues
Spawn P threads
Wait for completion of all the threads
Algorithm Terminated. Return

Algorithm For Each Worker Thread i
(Assumption: If no task is returned, a NULL is returned.)
while true **do**
  Dequeue a task t_i from task-queue of i
  if t_i is not NULL **then**
    Finish the task t_i
  else
    Choose a victim k with remaining tasks
    Steal a task t_k from from k
    if t_k is NULL **then**
      Break from while loop
    else
      Finish the task t_k
    end if
  end if
  if stealing request arrives from q **then**
    if an unstarted task t_r is remaining **then**
      return t_r to q
    else
      return NULL to q
    end if
  end if
end while
Mark thread i as completed. Return.

Algorithm 1 shows the pseudo-code of the work-stealing algorithm, which works as follows. We first allocate P task queues and distribute the available works to these queues. These queues are used by individual workers. We then start P activities (threads) which act as worker and start executing their tasks (contingency analysis). Each worker deques a task from its queue for execution. On finishing its own tasks, it tries to steal a task from another worker (called victim). However, if no other worker has an extra task, it indicates that either all other workers are finished or they are executing their last tasks. In such a case, the worker thread is terminated. When all the threads are terminated, the algorithm is also completed.

To enable efficient implementation, we have made the following choices in the implementation of the algorithm.

1. Since contingencies can be independently analyzed, the design of work-stealing algorithm is simplified, compared to the scenarios where tasks need to be analyzed in a specific order (e.g. [27]).
2. In work-stealing algorithm, polling strategy (also referred to as victim selection strategy) has significant impact on communication overhead of each stealing event. In literature, several polling strategies have been proposed such as random selection (i.e. a victim is randomly chosen), fixed-order selection (i.e. a victim is chosen in a fixed order, e.g. 1, 2, 3...) and circular selection (i.e. a thief q searches in the order $q+1, q+2, \ldots, N, 1, \ldots, q-1$). Of these, we choose circular selection strategy, since unlike fixed-order selection, it does not create contention on the initial threads and it also avoids the need of generating random numbers, while distributing the stealing requests to the available threads.
3. The granularity of task-stealing is chosen as one task, since it leads to fine-grain load-balancing and also avoids the possibility where the victim itself may be left with few tasks. Using a large value of task-stealing granularity leads to aggressive stealing which is useful when the communication cost between workers is high. In our experiments, a multicore processor where the communication cost between different workers is small and hence, using a task-stealing granularity of one is sufficient.
4. To reduce the idle-wait of the starving threads, a variant of work-stealing [30] provisions actively giving tasks to the starving threads when the amount of local tasks exceed a given threshold. Since, in worst-case, this may lead to congestion, we do not use this option.

For comparison purposes, we have also implemented master-slave scheduling (MSS) method. The MSS algorithm works as follows. It uses a single worker as the master and the remaining workers are designated as slaves. In MSS, initially a single task is allocated to each slave. Afterwards, each slaves works to finish its task and then requests a new task from the master. When all tasks available at master are finished, the algorithm is terminated. MSS achieves load-balancing by always allocating the new task to a free slave. The psuedo-code for MSS is ommitted for brevity.

5. Salient Features of Our Approach

In this section, we discuss the optimizations incorporated and the salient features of our approach.

5.1 Enabling Legacy Code Reuse

For achieving high computation capabilities, HPC platforms such as GPU (graphics processing unit) have also been used.

Table 1: Simulation time in seconds. P=1 refers to serial execution

	C=500		C=1000		C=1500		C=2000	
P	Master-slave	Work-stealing	Master-slave	Work-stealing	Master-slave	Work-stealing	Master-slave	Work-stealing
1	11140	11140	22551	22551	33006	33006	45115	45115
2	11252	5569	21541	10704	33336	16699	44729	22664
4	3841	2854	7242	5483	11303	8519	15081	11468
8	1683	1476	3361	2881	4977	4401	6631	5966
16	858	846	1772	1644	2629	2504	3346	3193

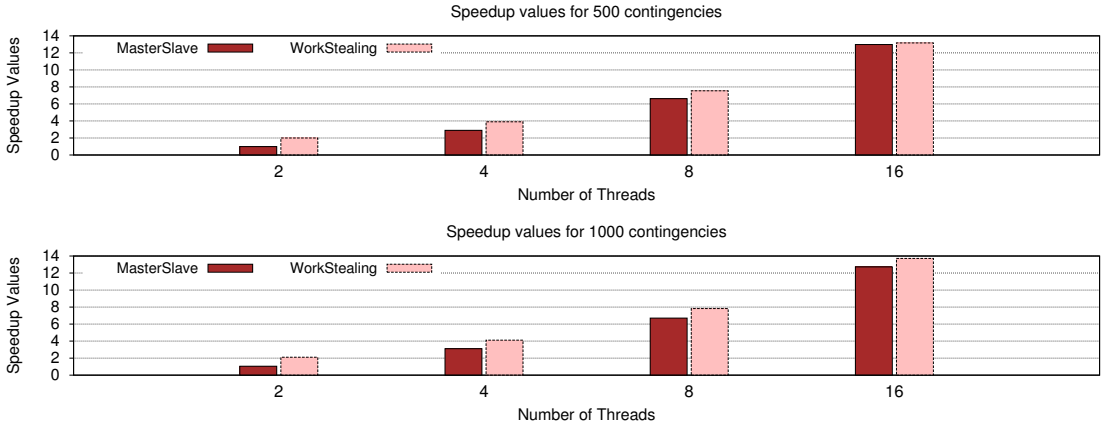

Figure 3: Performance Scaling Results with master-slave scheduling and our approach

The limitation of GPU is that they do not share memory with CPU and hence, use of GPU may lead to large overhead of data transfer between CPU and GPU. Further, using GPU may require significant remodeling and rewriting of the code. Modern power-system control-centers use legacy code with tens of thousands of lines of code and porting them to GPU would incur high overhead. Also, given the highly mathematical nature of code, porting may also introduce bugs and would require significant testing. In contrast, our approach runs on conventional desktops and multicore processors and hence, our approach allows legacy code reuse.

5.2 Optimizations For Achieving High Performance

We have X10, which can be compiled on a large number of platforms, such as Linux, Windows, X86 machines, MacOS, supercomputer, cluster etc. Moreover, compared to the languages such C/C++, X10 provides concurrent programming facilities as the part of language itself, which leads to portable implementation. X10 also provides several features which are not present in Java. Compared to the thread-creation in Java, X10 provides a much simpler way of creating asynchronous activities. Using `async`, a programmer can create multiple activities, which perform arbitrary computation, thus leading to a flexible design.

Currently, X10 compiler works by converting the X10 code into Java or C++ code[3]. Afterwards, using Java (resp. C++) compiler, bytecode (resp. executable-binary) is created. Since C++ code uses native execution, it is expected to have higher performance and hence, we have compiled X10 with `x10c++` to use C++ backend. Note that the compilation of X10 into C++/Java source code ensures portability and thus, our scheduler for contingency analysis can be used on a wide variety of platforms (hardware/operating systems). Moreover, this features enables utilizing platform-specific optimization capabilities of classical C++/Java compilers.

5.3 Difference from the Runtime Scheduler of X10

The runtime scheduler of X10 uses work-stealing scheduling [13]. However, our work-stealing scheduler is completely different from the X10 scheduler. Firstly, we did not utilize the X10 work-stealing scheduler. Second, as mentioned by Grove et al. [13], the X10 work-stealing scheduler uses double-ended queue, however we use single-ended queue. Further, the X10 work-stealing scheduler selects a victim in random fashion, however, we use circular victim-selection strategy. Our scheduler runs in user-space. Also, we compare our work with master-slave scheduler and use the scheduler

[3] Other possibilities such as converting to CUDA code is outside the scope of this work

Figure 4: Performance Scaling Results with master-slave scheduling and our approach

in power systems to show the superiority and effectiveness of our approach over existing methods.

6. Experimentation Platform and Results

We have tested our approach using a large power system with 13029 buses, 12488 branches, 5950 loads and 431 generators. We have used multiple contingencies, as shown below, which model different combinations of fault events such as bus faults, branch faults, and branch-tripping. To achieve high performance, we have compiled the code with `-optimize` flag. To utilize multiple cores of the processor, we set `X10_NTHREADS` environment variable to the desired number of threads. We present the wall-clock time of simulation, since this represents a metric which is of most importance for the power-system control center operators.

Table 1 shows the simulation time for different number of contingencies and cores. To study the scaling behavior, we define speedup quantity as follows.

$$S(C,P) = \frac{T(C,1)}{T(C,P)} \tag{2}$$

Here $S(C,P)$ refers to the speedup for C contingencies using P cores and $T(C,1)$ and $T(C,P)$ refer to simulation time using 1 (i.e. serial) core and P cores, respectively. The serial simulation experiments were performed without using the X10 scheduler, i.e. directly running the given number of contingencies and observing the simulation time.

Figure 3 and 4 show the value of speedup for different number of contingencies and different number of threads. We now analyze the results. Clearly, from Table 1, it is evident that for all contingencies, our work-stealing based scheduling approach provides higher performance than the master-slave scheduling method. For master-slave scheduling method, one thread (the master) becomes busy in scheduling and hence, performs no useful work. In contrast, for work-stealing scheduling method, all threads perform useful work.

For master-slave scheduling method, the simulation time with 2 threads is more than double with its value with 4 threads. This, however, does not represent super-linear speedup, since in going from 2 threads to 4 threads, the number of actual workers (i.e. slaves) is more than doubled. Similar argument also applies to the case with 8 and 16 threads.

When the number of threads is small, the speedup is nearly linear. As the number of threads become high (e.g. P = 16), the speedup value does not linearly increase with number of threads. This is because, on increasing the number of threads although the computation capability is linearly increased, other processor resources such as cache and memory bandwidth etc. do not increase linearly and this leads to sub-linear performance scaling, as observed by prior work also [11]. Further, for larger number of threads Scheduling overhead is small.

With 16 threads, we obtain more than 14$\times$ speedup over serial execution and with such speedup, a workload requiring one day of execution time can be executed in less than 2 hours. This shows the effectiveness of our approach.

The results presented in this section clearly demonstrate the performance benefits of our approach and its superiority over conventional scheduling method. Further, as the variation in simulation time of different contingencies increases, the load-balancing is increased which will further improve the utility of our technique.

7. Conclusion and Future Work

As single-core performance becomes power limited and multi-core processors with tens or even hundreds of cores

become widely available, use of parallel programming technique becomes a promising approach for achieving large computational gains. In this paper, we presented an approach for parallelization of power system contingency analysis using X10 parallel programming language. Using efficient work-stealing based scheduling method, we achieve dynamic load-balancing. The experimental results performed over a large test system show that our approach offers large computational gains and outperforms conventional scheduling technique. Our approach enables accelerating legacy code and avoids incurring the overhead of migrating and porting legacy code to new platforms. Our work has applications in analyzing power system stability which is extremely important for supporting critical infrastructures which depend on power systems.

Our future work will focus on analyzing even large number of contingencies and scaling our approach to larger number of cores using distributed computing approach. X10 is still being developed and it is expected that future developments will improve its productivity and performance even further.

References

[1] `http://x10-lang.org/home/news6.html`.

[2] The go programming language. `http://golang.org/`, 2012.

[3] A. Agrawal et al. Parallel pairwise statistical significance estimation of local sequence alignment using message passing interface library. *Concurrency and Computation: Practice and Experience*, 2011.

[4] A. Alexandrescu. *The D Programming Language*. Addison-Wesley Professional, 2010.

[5] G. Andersson et al. Causes of the 2003 major grid blackouts in north america and europe, and recommended means to improve system dynamic performance. *IEEE Transactions on Power Systems*, 20(4):1922–1928, 2005.

[6] R. Blumofe, C. Joerg, B. Kuszmaul, C. Leiserson, K. Randall, and Y. Zhou. *Cilk: An efficient multithreaded runtime system*, volume 30. ACM, 1995.

[7] R. Blumofe and C. Leiserson. Scheduling multithreaded computations by work stealing. In *Foundations of Computer Science, 1994 Proceedings., 35th Annual Symposium on*, pages 356–368. IEEE, 1994.

[8] B. Chamberlain, D. Callahan, and H. Zima. Parallel programmability and the chapel language. *International Journal of High Performance Computing Applications*, 21(3):291–312, 2007.

[9] B. Chapman, G. Jost, and R. Van Der Pas. *Using OpenMP: portable shared memory parallel programming*, volume 10. MIT press, 2007.

[10] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. Von Praun, and V. Sarkar. X10: an object-oriented approach to non-uniform cluster computing. In *ACM SIGPLAN Notices*, volume 40, pages 519–538. ACM, 2005.

[11] K. Datta, M. Murphy, V. Volkov, S. Williams, J. Carter, L. Oliker, D. Patterson, J. Shalf, and K. Yelick. Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, page 4. IEEE Press, 2008.

[12] T. Davis and 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.

[13] D. Grove, O. Tardieu, D. Cunningham, B. Herta, I. Peshansky, and V. Saraswat. A performance model for x10 applications. *X10*, 11, 2011.

[14] S. Gupta et al. Guaranteed QoS with MIMO Systems for Scalable Low Motion Video Streaming Over Scarce Resource Wireless Channels. In *International Conference On Information Processing*. IK International Pvt Ltd, 2008.

[15] D. Honbo, A. Agrawal, and A. Choudhary. Efficient pairwise statistical significance estimation using fpgas. *Proceedings of BIOCOMP*, 2010:571–577, 2010.

[16] Z. Huang, Y. Chen, and J. Nieplocha. Massive contingency analysis with high performance computing. In *IEEE Power and Energy Society General Meeting 2009*. IEEE, July 2009.

[17] V. Jalili-Marandi, Z. Zhou, and V. Dinavahi. Large-scale transient stability simulation of electrical power systems on parallel GPUs. *Parallel and Distributed Systems, IEEE Transactions on*, (99):1–1, 2011.

[18] S. Khaitan and J. McCalley. Dynamic load balancing and scheduling for parallel power system dynamic contingency analysis. *High Performance Computing in Power and Energy Systems*, pages 189–209, 2012.

[19] S. K. Khaitan and J. D. McCalley. *High Performance Computing in Power and Energy Systems, POWSYS*, pages 43–69. Springer, 2012.

[20] B. Lewis, D. J. Berg, et al. *Multithreaded programming with Pthreads*, volume 2550. Sun Microsystems Press, 1998.

[21] A. Mittal, J. Hazra, N. Jain, V. Goyal, D. Seetharam, and Y. Sabharwal. Real time contingency analysis for power grids. *Euro-Par 2011 Parallel Processing*, pages 303–315, 2011.

[22] S. Mittal, S. Gupta, and S. Dasgupta. FPGA: An efficient and promising platform for real-time image processing applications. In *National Conference On Research and Development In Hardware Systems (CSI-RDHS)*, Kolkata, India, 2008.

[23] S. Mittal, A. Pande, L. Wang, and P. Kumar. Design exploration and implementation of simplex algorithm over reconfigurable computing platforms. In *IEEE International Conference on Digital Convergence*, pages 204–209, 2011.

[24] S. Mittal and Z. Zhang. Integrating sampling approach with full system simulation : Bringing together the best of both. In *IEEE International Conference On Electro/Information Technology (EIT)*, Indianapolis, USA, 2012. IEEE.

[25] M. Odersky, L. Spoon, and B. Venners. *Programming in Scala*. Artima Incorporated, 2008.

[26] A. Pande et al. Baywave: Bayesian wavelet-based image estimation. *International Journal of Signal and Imaging Systems Engineering*, 2(4):155–162, 2009.

[27] D. Sanchez, R. Yoo, and C. Kozyrakis. Flexible architectural support for fine-grain scheduling. *ACM Sigplan Notices*, 45(3):311–322, 2010.

[28] V. Saraswat and R. Jagadeesan. Concurrent clustered programming. *CONCUR 2005–Concurrency Theory*, pages 353–367, 2005.

[29] R. Serban, C. Petra, and A. C. Hindmarsh. User documentation for IDAS v1.0.0. `https://computation.llnl.gov/casc/sundials/description/description.html`, 2009.

[30] M. Wu and X. Li. Task-pushing: a scalable parallel gc marking algorithm without synchronization operations. In *IPDPS*, pages 1–10. IEEE, 2007.