MASTER: A JAVA Based Work-Stealing Technique For Parallel Contingency Analysis

Technical Report · January 2016

DOI: 10.13140/RG.2.1.2589.9121

1 author:

S.K. Khaitan
Iowa State University
46 PUBLICATIONS **399** CITATIONS

SEE PROFILE

MASTER: A JAVA Based Work-Stealing Technique For Parallel Contingency Analysis

Siddhartha Kumar Khaitan
ECE Deptarment, Iowa State University, Iowa, USA.
skhaitan@iastate.edu

***Abstract*—In this paper, we present MASTER, a Java based multithreaded work-stealing technique for parallel contingency analysis in power systems. MASTER analyzes contingencies using time domain simulation and scales contingency analysis task to multiple cores using multithreading in Java. To achieve load balancing, MASTER uses efficient implementation of work-stealing algorithm. We discuss several implementation issues and design time choices which are crucial for achieving efficient implementation. Experiments performed with contingencies of a 13029 bus power system shows that MASTER provides high computational gains and provides much better load-balancing than the conventional scheduling techniques.**

***Index Terms*—Concurrency, multithreading, work-stealing, power systems, parallel contingency analysis**

I. Introduction

As power systems are growing in their size and complexity, the probability of critical failures is also increasing. To take preventive and corrective actions towards such disturbances, power system operators rely on obtaining information from analysis of a large number of contingencies. Conventionally, due to the small size of power systems, the number of contingencies to be analyzed has also remained small and hence, sequential computing platforms have fulfilled the computational needs of the contingency analysis task. However, with increasing system size and requirements of analyzing higher order $(N - x)$ contingencies, sequential computing platforms are proving to be insufficient in catering to the computational demands of the contingency analysis. Recently, researchers have proposed use of high-performance computing (HPC) resources , however, parallelization also brings the issue of achieving load balancing[1] and maximizing resource usage efficiency [1]. The conventional scheduling approaches (e.g. [2–4]) either provide poor load-balancing or do not scale well to large number of processors. Thus, the state-of-the-art in power systems calls for efficient parallelization of contingency analysis task, in a manner that provides load balancing and resource usage efficiency maximization.

In this paper, we present MASTER, a Java based multithreaded work-stealing technique for parallel contingency analysis in power systems. MASTER conducts time domain simulation of power system for analyzing contingencies. For parallelization of contingency analysis task, MASTER employs multithreading using Java (Section III). To achieve load balancing on the available cores, MASTER uses the work-stealing algorithm (Section IV). While work-stealing algorithm has been used in other domains (e.g. [5, 6]), to the best of our knowledge, our work is among the first to implement a multithreaded JAVA implementation of work-stealing to achieve load-balancing in power systems.

While work-stealing is known to be efficient in theory, achieving the expected efficiency in practice requires careful tuning of the parameters depending on the problem domain and the hardware platform used. An inefficient implementation is likely to significantly slow-down the algorithm [7, 8]. For this reason, we explore several algorithmic factors which impact the efficiency of work-stealing implementation and also discuss the actual choices made in MASTER for efficient implementation. We also present the details of MASTER scheduler along with its implementation in Java.

MASTER is envisioned as a security assessment tool for efficiently analyzing a large number of contingencies and assisting the system personnel in power system control centers. By virtue of using JAVA implementation, MASTER is portable and architecture independent. We have evaluated MASTER on a large power system with 13029 buses. Dynamic contingency analysis simulations have been performed on a multicore processor using 2, 4, 8 and 16 threads. The results show that MASTER offers high computational gains and provides much better load-balanced allocation than a conventional scheduling technique, namely centralized task-queue scheduling.

The rest of the paper is organized as follows. Section II discusses the related work in power system contingency analysis and scheduling techniques. Section III presents the design and architecture of MASTER. Section IV discusses the implementation of work-stealing algorithm. Section V presents the experimental results on evaluation of MASTER and comparison with a conventional scheduling approach. The conclusion and the future work are discussed in Section VI.

II. Background

High performance computing (HPC) is a promising approach for accelerating computation intensive and performance-critical tasks using parallelization technique [9–11]. However, parallelization also brings the requirement of efficient scheduling for achieving load-balancing [12, 13]. For accelerating contingency analysis using parallelization,

[1]In this paper, we use 'load-balancing' as it is used in parallel processing field (distribution of tasks to threads in a manner that the completion time of different workers is as close as possible) and not as it is used in power-systems.

researchers have used different platforms such as MPI (message passing interface) [4] and GPU (graphics processing unit). For parallelization, we use multithreading and discuss the reasoning behind the choice in Section III.

Scheduling techniques for achieving load-balancing in HPC, can be broadly divided into two categories, namely "static" and "dynamic" scheduling. Static scheduling works by using a fixed assignment of tasks on the available processors. The limitation of static scheduling is that when tasks have heterogeneous execution times, static scheduling fails to achieve load balancing. Work-stealing [14] is a dynamic scheduling method, which works by allocating tasks to worker threads to finish and letting a worker with no pending task (called thief) steal a task from another worker (called victim).

While Mittal et al. [4] perform load-balancing for steady state contingency analysis, we conduct dynamic contingency analysis which requires higher computation time and resources and hence, also imposes larger penalty of load imbalance.

In this paper, we evaluate single-order contingencies, although our framework permits analysis of cascading contingencies also. Previous work [15, 16] has proposed methods for identifying higher order contingencies and capturing their time delayed cascading effects on the network. Different events of contingencies can be seen as taking place as function of time along the dynamic event tree (DET), which accounts for both slow and fast dynamics. Each of these points along the DET can serve as the initiating event for the contingency analysis in the MASTER framework. These would include low frequency, high impact higher order contingencies. We also allow dynamic updating of system conditions which could make a high order contingency more probable due to changes in system configuration or maintenance. In this way, dependent events can be properly captured in the MASTER framework.

III. MASTER: Design and Architecture

MASTER uses a multithreaded scheduler for implementing work-stealing algorithm. In comparison to multiprocessing, our choice of multithreading is guided by the fact that threads are lightweight compared to processes and hence, thread-switching generally incurs less overhead than switching between the processes. Further, threads share the same address space and hence intercommunication between them is also less expensive than that between processes.

MASTER scheduler is designed using Java. The multithreading built into the Java language and runtime platform provides support for running multiple concurrent threads. Compared to languages such as C and C++, the benefit of using Java is that it includes multithreading primitives as part of the language itself and as part of its libraries. Thus, a portable implementation of multithreading can be easily achieved. In contrast, languages such as C do not have built-in multithreading capabilities and hence, to implement multithreading, they must make calls to operating system multithreading primitives, which harms efficiency and portability.

The work-flow of MASTER is explained in Figure 1. The MASTER code is compiled into platform-independent byte-code, which runs on the Java virtual machine (JVM). The JVM runs as a single process which internally spawns many threads. When the MASTER code running inside the JVM asks for another thread, JVM starts another thread which can run on a different CPU core. In this manner, multithreading is achieved on a multicore processor. In hardware, the management of threads is done by the operating system (OS), and the JVM uses the facility provided by the OS. More implementation details are provided in Section IV.

GPUs and MPI are other widely-used parallelization platforms. However, the limitation of GPUs is that they do not share memory with CPUs and are generally slower than CPUs in executing serial tasks. Hence, a significant fraction of execution time is spent in transferring data from GPU to CPU and vice-versa. Similarly, use of MPI requires reserving the nodes which leads to delays and resource wastage. Also inter-process communication used in MPI is expected to be slower than the inter-thread communication used in multithreading. Further, a common limitation of these platforms is that modern ISOs have simulators with thousands of lines of legacy code and porting them to these platforms will incur significant economic and time overhead. In contrast, MASTER scheduler runs separate to the main application program and provides ability to parallelize. Moreover, MASTER uses multithreading with Java and hence, it can be easily implemented on the commonly available multicore processors.

For contingency analysis, MASTER performs time domain simulation using a high-speed cross-platform power system simulator [17, 18]. The simulator uses efficient numerical algorithms (linear, non-linear and integration solvers) to solve the DAEs (differential and algebraic equations) appearing in power-system modeling [19]. The simulator provides different models of power system components such as governors, exciters and generators. The overall contingency analysis steps are explained in Figure 2.

From an architectural perspective, the simulator operates in two modes, namely anticipatory mode and emergency mode. The simulator analyzes most of the critical disturbances in anticipatory mode and storing the results for future use. This helps in providing the power system operator the ability to take timely corrective response at the crucial time after the fault. It provides a cross-platform simulation framework, and has been developed using object-oriented programming (OOP) approach, which facilitates easy development and extension of simulator features.

IV. Work-stealing Based Scheduling

For achieving load-balancing on available threads, MASTER uses work-stealing algorithm. It has been shown in [14], that using work-stealing, the time taken in executing a fully-strict computation[2] with P workers is given by:

[2]Fully-strict computation is one where the data dependencies of a thread go to its parent only and not to any other thread. Since contingencies are independent and thus, there are no data dependencies. Hence, the theoretical guarantee provided in [14] holds for our experiments also.

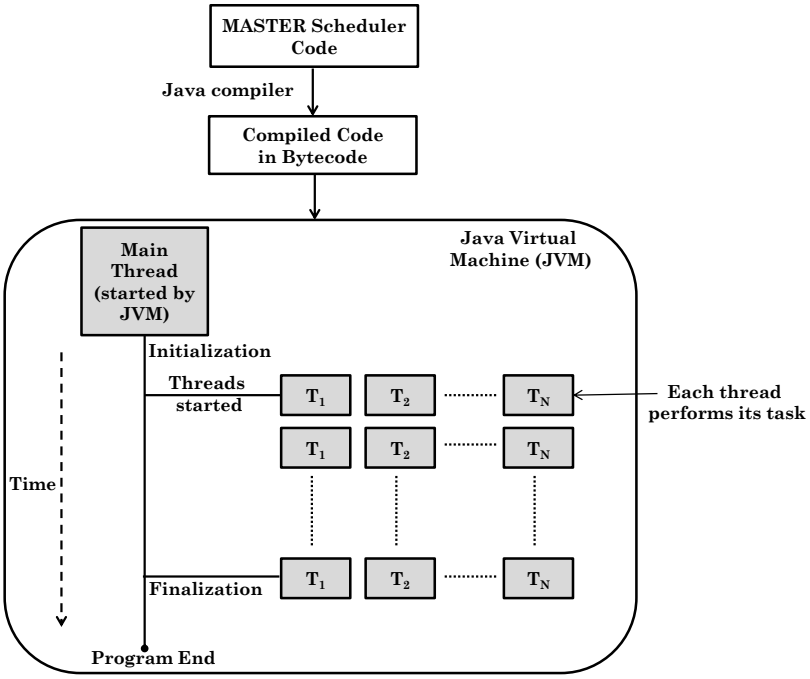

Fig. 1: Implementation of multithreading in MASTER using Java

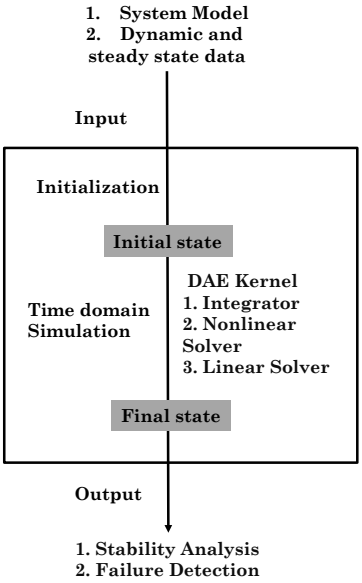

Fig. 2: Single Contingency Analysis

$$T = T_1/P + O(T_\infty) \tag{1}$$

Here T_1 denotes the serial execution time and T_∞ denotes the estimated execution time with infinite number of workers. Also, the storage space required scales only linearly with the number of processors [14]. Thus, work-stealing algorithm is efficient in terms of time and space requirement. Work-stealing has been shown to impose low overheads and has good scalability [20–23] and hence, it has been used in several parallel programming models.

However, due to practical considerations, several algorithmic factors influence the efficiency of work-stealing implementation [24, 25]. In the following, we discuss them in more detail and also mention the actual choices made in MASTER.

A. *Choice of Algorithmic Options*

1. Task dependency and ordering: In the application domains where tasks have interdependence or ordering-constraints (e.g. [26]), scheduling algorithm needs to fulfill these constraints for correctness. In our work, each task represents analysis of a single contingency and hence, tasks are mutually-independent, and maintaining ordering is not required. Further, the policy used for enqueuing tasks in the task-queues also becomes unimportant and hence we safely ignore this factor in the design of MASTER.

2. Organization and management of task-queues: The choice of data-structure used for storing task-queues affects the efficiency of task enqueuing and dequeuing. A naïve approach is to keep a single-ended queue where both owner and thief dequeue from the same end. However, this is likely to lead to contention. Another choice is to keep double-ended queue, where the owner dequeues from one end and the thief dequeues from another [27]. A variant of this is partitioning the per-thread task queue into a private and a public region [26]. The thief only accesses the public region of the queue, while the private region is accessed by the owner thread only. This alleviates the need of providing locks in the private region of the queue. However, this also requires separate management of the private and public region and suitably transferring the task from one region to another, when an imbalance is created. In general, double-ended queues (known as *deque*) are expected

to be simpler than the queue with private-public region. For this reason, we have used double-ended queues in MASTER.

3. Victim selection: The choice of a victim-selection strategy (also called polling strategy) impacts the communication overhead incurred in each stealing action. Further, when tasks have ordering-constraints, completing the tasks in a desired order may greatly reduce the overall completion time. Some commonly used victim selection policies are:

1) Fixed: The victim is selected in fixed order (e.g. $1, 2, 3, \ldots$ etc.).
2) Random: Any of the available workers is selected randomly as the victim [28].
3) Local: The thief tries to steal from the nearest-neighbor or one of the close neighbors. Guo et al. [29] provision stealing within a pre-defined locality domain. Min et al. [26] propose a "hardware topology-aware hierarchical victim selection" strategy where a victim is selected in hierarchical manner, i.e., victim processor is first searched in the same socket [30] and only if no victim is found, the algorithm searches for victims in other sockets or nodes [30, 31].
4) Circular: The victim is selected in circular manner, i.e., a thief i searches in the order $i{+}1, i{+}2, \ldots, N, 1, \ldots, i{-}1$.

Of these, MASTER uses circular polling since unlike fixed polling, it does not cause contention on initial threads, viz. 1, 2,3 etc. Further, unlike random polling, it does not require generating random numbers and is likely to spread the stealing requests more uniformly to the available threads, thus reducing contention. Also, since the tasks do not have ordering constraints and threads share memory, choosing victim in topology-aware manner is not required.

4. Task-stealing granularity: Task-stealing granularity refers to the number of tasks which are allowed to be stolen in each stealing request. While some implementations provision stealing a fixed number of tasks [29, 32, 33], other implementations allow stealing a variable number of tasks in each stealing action [26, 27, 34], which can be as large as half the number of tasks available on the victim process (termed as "steal-half"). For sake of simplicity, in this paper, we provision stealing only one task at a time. This enables fine-grain load balancing and also keeps the quantum of work stolen small. For higher number of threads, steal-half policy may provide better load-balancing [27]. Exploration of this policy is planned as part of future work with larger number of threads.

5. Proactive task-pushing: A variant of work-stealing algorithm provisions actively giving tasks to the starving threads, when the amount of local tasks exceeds a given threshold so that the idle-wait time of the starving threads is reduced [35, 36]. The disadvantage of task-pushing, however, is that it increases communication overhead and causes congestion in the worst case. For this reason, MASTER does not employ proactive task-pushing.

B. Algorithm Pseudo-code

Algorithm 1 depicts the pseudo-code for MASTER scheduling algorithm. For sake of simplicity, in this paper, we assume that all the tasks have already arrived, although MASTER works equally well for the case when tasks arrive dynamically.

Algorithm 1 MASTER work-stealing algorithm

Input: task-list T
Output: Best effort allocation of tasks to threads
isFinished[1:N]=false, allQueuesEmpty =false
CODE FOR MAIN THREAD
Spawn N threads, let H be the list of threads.
Allocate task-queues for each thread h_i in H.
Assign tasks in T to task-queues of different threads. Start all threads
while true **do**
  Set value of allQueuesEmpty based on status of task-queues of all threads
  if allQueuesEmpty is true **then**
    break;
  else
    sleep 1 second
  end if
end while
Wait till isFinished[i] is true for all $1 \leq i \leq N$
Algorithm Terminated. Return
CODE FOR EACH WORKER THREAD i
while allQueuesEmpty is false **do**
  dequeue a task t_j from task-queue of i
  if t_j is NONE **then**
    Choose a victim k
    Steal task(s) from from k and enqueue in the task-queue of i
  else
    Finish the task t_j
  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
isFinished[i]=true

At the time of initialization, MASTER creates a number of threads using the `Thread` class. All the threads carry the same priority. Each object of the `Thread` class, encapsulates the data and functions (methods) associated with separate threads of execution. Each of these threads has task queues to which it enqueues the newly created tasks and from which it dequeues the tasks for execution. Since the contingencies can be analyzed independently, the tasks can be executed in any order. Each task is run using `exec()` command, which

executes a given command in a separate process. Through this command, the MASTER scheduler interfaces with the power system simulator for consistency analysis. This command returns a `Process` object, which is used by the scheduler for managing the process created, which includes waiting for the process to finish and handling errors (if any) occurred in the process execution.

The task queues are implemented as deques (i.e. double ended queues, which support element insertion and removal at both ends). The worker executes the task from the front side and stealing requests are responded to from the rear side. For this, Java's `BlockingDeque` class is used.

A main thread handles initial task distribution and periodically checks whether all the task-queues are empty. For implementation of periodic wait, the `sleep()` method is used which causes a thread to enter the *not runnable* state until the specified time has expired. Use of `sleep()` provides an efficient means of making processor resources available to the other threads. Either for polling or stealing, a worker thread needs exclusive and consistent access to the task-queues of the victim thread, while the victim thread itself needs to dequeue the task for execution. To ensure consistency, MASTER uses the built-in mechanism of Java to provide synchronization. The thread which first accesses the deque gets the lock on it and meanwhile, no other thread can acquire the lock.

To avoid the overhead of creating threads for every task and destroying them at the completion of the task, MASTER uses thread pools. The tasks are passed to the thread pool and when a thread becomes idle, the task is assigned to that thread. If no new task arrives, the idle threads wait for a new task. By reusing threads for multiple tasks, the overhead of thread-creation is amortized over many tasks. Further, use of thread-pool enables limiting the number of running threads which helps in avoiding resource thrashing caused by excessive number of threads.

In a multithreaded scheduling algorithm, termination detection requires reading the status of task-queues of all threads. In work-stealing, detecting termination is even more challenging, since threads can steal tasks from other threads, and hence, finishing of the task-queues of a thread does not necessarily imply algorithm termination. To detect termination, MASTER works as follows. The main thread periodically reads the status of all task-queues and when all of queues are empty, it sets *allQueuesEmpty* to true. When this is true, all threads stop trying to steal more tasks. Each thread i, which has finished its tasks, sets *isFinished[i]* to true. Meanwhile, the main thread periodically checks whether all the threads have finished and when that is true, the algorithm is terminated.

C. Features and Efficiency

From Amdahl's law [37], the speedup obtained from parallelizing a program is limited by the fraction of time spent in the sequential part. For this reason, use of a single (global) locked queue might present a scalability bottleneck since the accesses to the global queue are required to be serialized. Since the MASTER scheduler uses work-stealing with distributed task-queues, it automatically avoids the need of a global task-queue. Further, since the tasks are allocated from the local task-queue, the latency of access is reduced.

In power systems, contingencies can be analyzed independent of each other and maintaining an ordering between them is not required. MASTER leverages this fact to further speedup its work-stealing implementation, since regardless of whether other tasks have been completed, MASTER can schedule a task to the first free thread. Also, maintaining a table of completed tasks and reading from it to ensure task-ordering is also avoided, which leads to more optimized implementation.

MASTER leverages several features of Java, such as easy memory management, platform-independence and portability. Thus, MASTER can be used on a wide variety of platforms. MASTER has been designed to be thread-safe and thus, the access to shared data (e.g. task-queues) does not cause conflict or race conditions. Also, due to the object-oriented design features of Java, MASTER scheduler can be easily extended.

D. Limitations and Possible Extensions

Multithreading introduces the overhead of thread-switching. When the processor switches from executing one thread to another, the context (e.g. local data and program pointer) of the current thread needs to be saved and the context of next thread needs to be loaded. With increasing number of worker threads, the contention for the shared resources also increases.

A possible improvement in load balancing can be done by provisioning stealing of the tasks from the thread with the largest number of remaining tasks. The overhead in this approach is the need of finding lengths of task-queues of all tasks and comparing them to find the largest sized queue. This extension is planned as a part of our future work.

In case different contingencies have different priorities (e.g., due to their severity or probability of occurrence), MASTER can be easily extended to set thread priorities and assigning tasks with high priorities to the workers (threads) with high priorities. Also, by allocating tasks such that the tasks with higher priorities are executed before those with lower priorities, task-priorities can be handled.

TABLE I: Simulation time (seconds). N=1 refers to serial execution, Centr. = centralized task-queue scheduling

	C=128		C=256		C=512	
N	Centr.	MASTER	Centr.	MASTER	Cent.	MASTER
1	2657	2657	5328	5328	10780	10780
2	2670	1334	5321	2698	10682	5353
4	894	672	1813	1367	3575	2703
8	413	358	817	717	1610	1404
16	228	199	436	393	893	790

V. Results

Experimental Platform: We simulate a large power system with 13029 buses, 431 generators, 12488 branches and 5950 loads. We show wall-clock time, since it is a metric of relevance to end-user. We simulate 128, 256 and 512 contingencies using 2, 4, 8 and 16 threads and compare the results with serial execution. Each contingency models fault occurrence

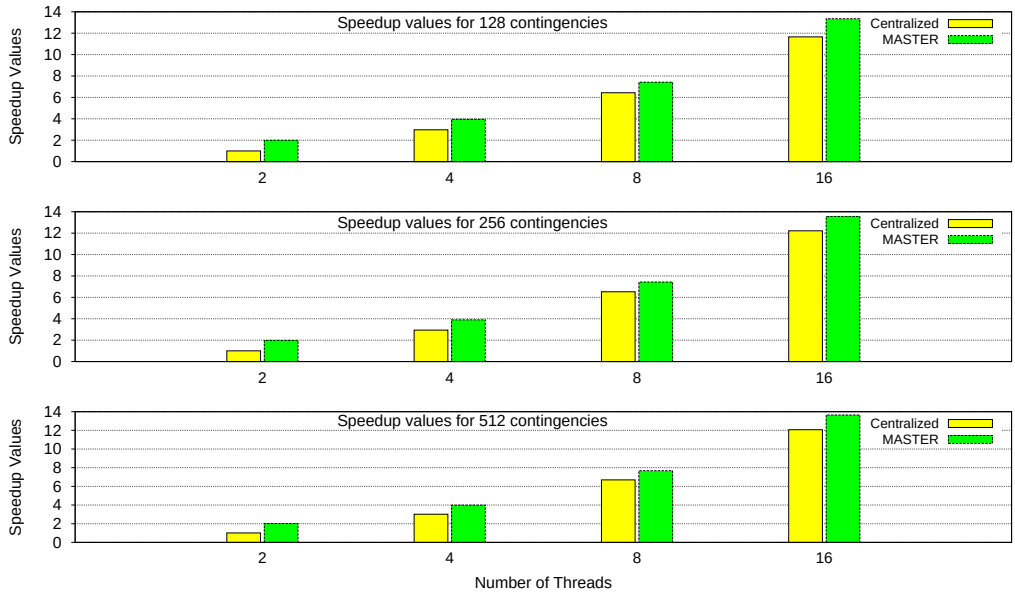

Fig. 3: Speedup Results with MASTER and conventional scheduling techniques for different number of worker threads and contingencies

and clearing in single or multiple system components such as bus and branch. We run the simulations on a multicore processor and different threads run on different cores.

Performance Results: We compare the work-stealing implementation to centralized task-queue scheduling where the main thread holds the queue of ready tasks and all worker threads request task from it on finishing their tasks.

Table I summarizes the time taken in simulation of different contingencies with different number of threads. Here C denotes the number of tasks (contingencies) and N denotes the number of threads. For C contingencies, let $T(C, N)$ denote the time taken with N threads and $T(C, 1)$ be the time taken using serial execution. To gain insights into the working of different scheduling methods, we define speedup values $S(C, N)$, for any scheduling method as follows.

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

Figure 3 shows these speedup values for different methods. Clearly, MASTER provides highest computational gains and outperforms the conventional techniques. A crucial limitation of centralized task-queue scheduling is that one thread (main thread) becomes busy with the task of allocating the works and performs no useful work.

With higher number of threads, contention increases and resultant synchronization serializes the execution. For this reason, the speedup does not scale linearly with the number of threads. Still, MASTER achieves high computation gains (of the order of 13.5× speedup with 16-threads) and much better load balancing than conventional scheduling techniques.

VI. Conclusion

In this paper, we presented MASTER, a multithreaded work-stealing technique for efficient parallelization and load-balancing of contingency analysis task in power systems. We discussed several design-time choices made for efficient implementation. The experiments show that MASTER offers large computation gains.

References

[1] S. K. Khaitan and J. D. McCalley, "Dynamic load balancing and scheduling for parallel power system dynamic contingency analysis," in *High Performance Computing in Power and Energy Systems*, 2013, pp. 189–209.

[2] J. Riquelme Santos, A. Gomez Exposito, and J. Martinez Ramos, "Distributed contingency analysis: practical issues," *IEEE TPWRS*, vol. 14, no. 4, pp. 1349–1354, 1999.

[3] Q. Morante, N. Ranaldo, A. Vaccaro, and E. Zimeo, "Pervasive grid for large-scale power systems contingency analysis," *IEEE Trans. Ind. Informat.*, pp. 165–175, 2006.

[4] 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*, pp. 303–315, 2011.

[5] K. Zhou, Q. Hou, Z. Ren, M. Gong, X. Sun, and B. Guo, "Renderants: interactive reyes rendering on GPUs," *ACM Transactions on Graphics (TOG)*, vol. 28, no. 5, p. 155, 2009.

[6] R. Van Nieuwpoort, T. Kielmann, and H. Bal, "Efficient load balancing for wide-area divide-and-conquer applications," in *ACM SIGPLAN Notices*, 2001, pp. 34–43.

[7] 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," *ACM SIGOPS OSR*, vol. 41, no. 3, pp. 73–86, 2007.

[8] Z. Vrba, P. Halvorsen, and C. Griwodz, "Evaluating the run-time performance of kahn process network implementation techniques on shared-memory multiprocessors," *CISIS*, pp. 639–644, 2009.

[9] M. P. Raju and S. K. Khaitan, "Domain decomposition based high performance parallel computing," *International Journal of Computer Science Issues*, vol. 5, 2009.

[10] M. Raju and S. Khaitan, "High performance computing using out-of-core sparse direct solvers," *International Journal of Mathematical, Physical and Engineering Sciences*, vol. 3, no. 2, pp. 377–383, 2009.

[11] S. K. Khaitan, "A Survey Of High-performance Computing Approaches in Power Systems," in *IEEE Power and Energy Society General Meeting (PES)*, 2016, pp. 1–5.

[12] S. K. Khaitan and J. D. McCalley, "Dynamic load balancing and scheduling for parallel power system dynamic contingency analysis," in *High Performance Computing in Power and Energy Systems*, ser. Power Systems. Springer Berlin Heidelberg, 2012, pp. 189–209.

[13] S. K. Khaitan and J. D. McCalley, "A High-Performance Parallelization and Load-Balancing Approach for Modern Power-Systems," *International Journal of Business Analytics (IJBAN)*, vol. 2, no. 2, pp. 62–74, 2015.

[14] R. Blumofe and C. Leiserson, "Scheduling multithreaded computations by work stealing," *Symp. on Foundations of Computer Science*, 1994.

[15] S. K. Khaitan, "On-line cascading event tracking and avoidance decision support tool," Ph.D. dissertation, Iowa State University, 2008.

[16] J. McCalley, K. Zhu, and Q. Chen, "Dynamic decision-event trees for rapid response to unfolding events in bulk transmission systems," *GM*, 2001.

[17] S. K. Khaitan and J. D. McCalley, "High performance computing for power system dynamic simulation," in *POWSYS*, 2012, pp. 43–69.

[18] S. K. Khaitan and 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.

[19] S. Khaitan, J. McCalley, and M. Raju, "Numerical methods for on-line power system load flow analysis," *Energy Systems*, vol. 1, no. 3, pp. 273–289, 2010.

[20] S. K. Khaitan and J. D. McCalley, "Achieving load-balancing in power system parallel contingency analysis using X10 programming language," in *Proceedings of the third ACM SIGPLAN X10 Workshop*, 2013, pp. 20–28.

[21] S. K. Khaitan and J. McCalley, "PARAGON: An Approach for Parallelization of Power System Contingency Analysis Using Go Programming Language," *International Transactions on Electrical Energy Systems*, 2014.

[22] R. Hoffmann, M. Korch, and T. Rauber, "Performance evaluation of task pools based on hardware synchronization," in *Proceedings of the 2004 ACM/IEEE conference on Supercomputing*, 2004, p. 44.

[23] S. K. Khaitan and J. D. McCalley, "Parallelizing power system contingency analysis using D programming language," in *IEEE Power and Energy Society General Meeting (PES)*, 2013, pp. 1–5.

[24] S. K. Khaitan and 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), 2012 SC Companion:*. IEEE, 2012, pp. 289–298.

[25] S. K. Khaitan and J. D. McCalley, "SCALE: A hybrid MPI and multithreading based work stealing approach for massive contingency analysis in power systems," *Electric Power Systems Research*, vol. 114, pp. 118–125, 2014.

[26] S. Min, C. Iancu, and K. Yelick, "Hierarchical work stealing on manycore clusters," *PGAS11*, 2011.

[27] J. Dinan, D. Larkins, P. Sadayappan, S. Krishnamoorthy, and J. Nieplocha, "Scalable work stealing," *SC*, p. 53, 2009.

[28] M. Fluet, M. Rainey, and J. Reppy, "A scheduling framework for general-purpose parallel languages," *ACM Sigplan Notices*, vol. 43, no. 9, pp. 241–252, 2008.

[29] Y. Guo, R. Barik, R. Raman, and V. Sarkar, "Work-first and help-first scheduling policies for async-finish task parallelism," in *IPDPS*, 2009, pp. 1–12.

[30] H. Matsuba and Y. Ishikawa, "Single IP address cluster for internet servers," in *IPDPS*, 2007, pp. 1–10.

[31] R. Rabenseifner, G. Hager, and G. Jost, "Hybrid MPI/OpenMP parallel programming on clusters of multi-core SMP nodes," in *PDP*, 2009, pp. 427–436.

[32] M. Frigo, C. Leiserson, and K. Randall, "The implementation of the Cilk-5 multithreaded language," *ACM Sigplan Notices*, vol. 33, no. 5, pp. 212–223, 1998.

[33] A. Narang, A. Srivastava, R. Jain, and R. Shyamasundar, "Dynamic distributed scheduling algorithm for state space search," *Euro-Par 2012 Parallel Processing*, pp. 141–154, 2012.

[34] G. Cong, S. Kodali, S. Krishnamoorthy, D. Lea, V. Saraswat, and T. Wen, "Solving large, irregular graph problems using adaptive work-stealing," in *ICPP*, 2008, pp. 536–545.

[35] M. Wu and X. Li, "Task-pushing: a scalable parallel GC marking algorithm without synchronization operations," *IPDPS*, pp. 1–10, 2007.

[36] S. K. Khaitan, J. D. McCalley, and A. Somani, "Proactive task scheduling and stealing in master-slave based load balancing for parallel contingency analysis," *Electric Power Systems Research*, vol. 103, pp. 9–15, 2013.

[37] G. Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," in *Spring joint computer conference*, 1967.