Large-Scale Transient Stability Simulation of Electrical Power Systems on Parallel GPUs

Vahid Jalili-Marandi, *Student Member*, *IEEE*, Zhiyin Zhou, *Student Member*, *IEEE*, and Venkata Dinavahi, *Senior Member*, *IEEE*

Abstract—This paper proposes large-scale transient stability simulation based on the massively parallel architecture of multiple graphics processing units (GPUs). A robust and efficient instantaneous relaxation (IR)-based parallel processing technique which features implicit integration, full Newton iteration, and sparse LU-based linear solver is used to run the multiple GPUs simultaneously. This implementation highlights the combination of coarse-grained algorithm-level parallelism with fine-grained data-parallelism of the GPUs to accelerate large-scale transient stability simulation. Multithreaded parallel programming makes the entire implementation highly transparent, scalable, and efficient. Several large test systems are used for the simulation with a maximum size of 9,984 buses and 2,560 synchronous generators all modeled in detail resulting in matrices that are larger than $20{,}000 \times 20{,}000$.

Index Terms—Graphics processors, instantaneous relaxation, large-scale systems, multiple GPUs, Newton-Raphson method, parallel multithreaded programming, power system simulation, power system transient stability, sparse direct solvers.

1 Introduction

Electrical power systems are the largest man-made distributed nonlinear structures that are composed of a variety of equipment such as generators, transformers, transmission lines, and customer loads, strewn across a vast geographic area spanning thousands of kilometers. Continuous growth in electricity demand accompanied by system expansion and complex interconnections within the grid are creating major operational and control problems which require significant computational resources for real-time or near-real-time intervention by system operators in energy control centers. Maintaining power system stability is paramount for a secure and uninterrupted supply to the customers. Transient stability simulations analyze the impact of potential grid disturbances (contingencies) in a transient time frame which is normally up to about 10 seconds after a disturbance.

Online dynamic security assessment (DSA) has become the linchpin for improved reliability and security of modern stressed power systems. According to [1] DSA is the process of determining the degree of risk in a power system's ability to survive imminent contingencies without interruption to customer service. Many utilities world wide are implementing online DSA in their control centers to prevent crippling economic fallout from massive blackouts such as the one that occurred in Northeastern US and Canada on August 14, 2003. The five main stages of online DSA are [2]: measurements, modeling, computation, visualization, and control. A real-time snapshot of the power system under study is captured through measurements by supervisory control and data acquisition (SCADA), and a base model is established using the state estimator and system equivalencing. Detailed time-domain simulations are then carried out for a list of credible contingencies, and security indices (for transient and voltage stability) are determined for reporting to system operators. The entire DSA cycle is required to be completed within a 10-30 minute time frame to allow operators sufficient time to initiate preventive control actions before the next snapshot is taken. The complete hardware and software architecture for implementing online DSA can be found in [3]. Essentially the main hardware components consist of the energy management system (EMS), the data server, the DSA client stations, the computational servers, and several user workstations, all of which communicate on a secure LAN. The conventional methods to improve the cycle time for online DSA includes three strategies: 1) utilizing distributed computation on multiple CPU-based computational servers, 2) contingency screening to identify critical contingencies for which run the time-consuming simulations, 3) early termination of simulations based on improved security indices. Nevertheless, computational speed still remains a bottleneck owing to the sequential operation of the CPUs that are required to solve several thousand nonlinear differential algebraic equations (DAEs) in each time-step due to the detailed mathematical modeling of grid components. Using a software tool developed by the Electric Power Research Institute (EPRI) [4], the computer time to perform hundreds of contingency simulations was

- *V. Jalili-Marandi is with the OPAL-RT Technologies Inc., 1751 Richardson, Suite 2525, Montreal, Quebec, Canada H3K 1G6. E-mail: vahidj@ualberta.ca.*
- *Z. Zhou and V. Dinavahi are with the Department of Electrical and Computer Engineering, University of Alberta, ECERF 9107 116 Street, Edmonton, Alberta, Canadas T6G 2V4. E-mail: {zhiyin, dinavahi}@ualberta.ca.*

Manuscript received 17 June 2011; revised 2 Sept. 2011; accepted 8 Nov. 2011; published online 29 Nov. 2011.
Recommended for acceptance by S. Ranka.
For information on obtaining reprints of this article, please send e-mail to: tpds@computer.org, and reference IEEECS Log Number TPDS-2011-06-0397.
Digital Object Identifier no. 10.1109/TPDS.2011.291.

reduced to about 20 to 30 minutes; however, improved numerical algorithms and computer systems are still needed today, to reduce this time to less than 5 to 10 minutes for real-time operation. The objective of this paper is to explore the use of multiple graphics processing units (GPUs) as the core computational engine for the transient stability simulation of large-scale power systems.

The field of high performance computing (HPC) has been dominated by GPUs recently. Many intractable problems that have traditionally required a supercomputer are now being solved on GPU-based desktop computers. GPU clusters have tremendously accelerated many HPC applications such as molecular dynamics, bioinformatics, computational fluid dynamics, finance, weather, and atmosphere modeling [5], [6], [7], [8], [9], to name just a few. The GPU is a massively parallel processor composed of hundreds of computational cores known as stream processors (SPs) which execute identical instructions on threads that are mapped to individual data elements. Unlike a multicore CPU which has fewer computational cores and consequently can execute only a few parallel threads at a time, the GPU is classified as a many-core processor which can execute hundreds of threads simultaneously based on a single instruction multiple data (SIMD) architecture. The GPU can be programmed by the compute unified device architecture (CUDA) language [10], [11] which has a C-like syntax and shares many of its constructs. The GPU cannot work as a stand-alone processor; it needs the CPU to initiate and keep track of kernel (functional program) execution. In fact, many of the currently available off-the-shelf HPC servers are hybrid GPU-CPU servers, in that they integrate one or more multicore CPUs with multiple GPUs on the same chassis. It is conceivable that clusters made up of such hybrid servers can be used for online DSA computations in energy control centers. In [12], the transient stability simulation of large-scale systems on a single GPU was presented. The classical transient stability simulation approach was used and compared with CPU-based implementation to illustrate advantages of general purpose GPU (GPGPU) computations for large-scale systems. Although, no sparse methods are exploited for system solution in [12], the application of GPU for power system dynamic simulation was proven to be promising.

This paper proposes a multi-GPU implementation of large-scale transient stability simulation based on an accurate and robust relaxation method. The instantanous relaxation (IR) method, first suggested and implemented on a CPU-based distributed real-time simulator [13], uses implicit Trapezoidal integration, full Newton-Raphson iterations, and sparse LU factorization methods to solve dynamic equations of the subsystems of a decomposed network. The IR method has proven to be a coarse-grained program-level parallel technique; thus, its implementation on a multi-GPU computational server where each of the individual GPUs has a fine-grained parallel architecture enables significant acceleration of the transient stability simulation.

The paper is organized as follows. Section 2 provides a brief overview of the transient stability simulation and the application of parallel processing in this area. Section 3 explains IR method and its application for transient stability simulation. In Section 4, the multi-GPU system hardware and the proposed multithreaded programming aspects are presented. A sparse linear solver suitable for the transient stability simulation and the SIMD-based architecture of GPUs is proposed in Section 5. In Section 6, the experimental results of multi-GPU-based transient stability simulation for various large-scale test systems and a comprehensive discussion are presented. Finally, Section 7 gives the conclusion of the work.

2 Transient Stability Simulation

2.1 Overview

Power system stability is the ability of an electric power system, for a given initial operating condition, to regain a state of operating equilibrium after being subjected to a physical disturbance, with most system variables bounded so that practically the entire system remains intact [1]. If an interconnected network is subjected to a perturbation, the analysis of the ability of the power system to keep its rotating machines in synchronism, during and after the perturbation, is the transient stability analysis. From the system theory viewpoint, power system transient stability is a strongly nonlinear problem. It can be mathematically described by a set of nonlinear differential-algebraic equations (DAEs) as follows:

$$\dot{\mathbf{x}} = \mathbf{f}(\mathbf{x}, \mathbf{v}, t), \tag{1}$$

$$\mathbf{0} = \mathbf{g}(\mathbf{x}, \mathbf{v}, t), \tag{2}$$

$$\mathbf{x}(t_0) = \mathbf{x_0}, \tag{3}$$

where $\mathbf{x}$ is the vector of state variables, $\mathbf{x_0}$ is the initial values of state variables, and $\mathbf{v}$ is the vector of network bus voltages. $\mathbf{f}(.)$ and $\mathbf{g}(.)$ are nonlinear functions, and (1) describes the dynamic behavior of the system resulting from the synchronous machines, while (2) describes the algebraic network constraints on (1), such as load demands and transmission line power flows. Further detail on the system representation is given in the Appendix.

Discrete solution of these equations in time-domain requires the use of numerical integration techniques which are either explicit or implicit [14]. Modified Euler [15] and Trapezoidal [16] methods are the most commonly used explicit and implicit techniques, respectively, for transient stability simulations. Explicit methods are easy to implement and fast, but are prone to numerical instability with large step-sizes; however, implicit methods are numerically stable and can tolerate larger step-sizes. In explicit techniques (1) and (2) are solved separately, whereas an implicit method allows simultaneous solution of (1) and (2) wherein it is possible to discretize (1), and then lump the resulting equations with (2) to make a larger set of nonlinear algebraic equations. However, an implicit method needs an extra iterative procedure such as the Newton-Raphson [17] or the Predictor-Corrector [18] technique to linearize the implicit equations. Ultimately, a set of linear algebraic equations must be solved, in each iteration and time-step, to find the value of the state and algebraic variables. LU factorization along with and forward and back substitution is a common approach for this purpose.

2.2 Application of Parallel Processing

The complexity of power systems simulation has increased with the system size and modeling detail with a concomitant increase in computational time. Due to the time critical nature of such computation, the need for parallel processing for the transient stability simulation of realistic systems was recognized decades ago.

In the 1950s, G. Kron developed a solution method for large networks known as "diakoptics" [19]. The basic idea of diakoptics is to solve a large system by tearing it apart into smaller subsystems. These subnetworks are then analyzed independently as if they were completely decoupled, and then the solutions of the torn parts are combined and modified to yield the solution of the original problem. The solution of the entire network can be obtained by injecting back the link currents into the corresponding nodes. The result of the procedure is identical to one that would have been obtained if the system had been solved as one. The *parallel-in-space* algorithms are step-by-step methods based on partitioning the original system into subsystems and distributing their computation among the parallel processors [20]. These algorithms address the task-level parallelism wherein serial algorithms are converted into various smaller and independent tasks that may be solved in parallel. In the transient stability calculation of a large-scale power system the obvious part where parallelism can be exploited is the solution of linear algebraic equations. Despite the sequential nature of the initial value problem which derives from the discretization of differential equations, *parallel-in-time* approaches have been proposed for parallel processor implementation. The idea of exploiting the parallelism-in-time in power system applications was first proposed in [21] and [22] to concurrently find the solution for multiple time-steps. In this method, the simulation time is divided into a series of blocks that contain a number of steps which lead to the solution of the system. The *waveform relaxation* (WR) method [23], was an attempt to exploit both space and time parallelism for the transient stability problem. The WR method is an iterative approach for solving the system of DAE over a finite time span where the original DAE is partitioned into smaller groups that can be solved independently. Each subsystem uses the previous iterate waveforms of other subsystems as guesses for its new iteration. Nevertheless as shown in [13] the WR method has drawbacks which preclude its implementation for real-time transient stability simulation.

From the hardware point-of-view various types of parallel processing architectures such as multiple-instruction multiple-data (MIMD), single-instruction multiple-data, distributed memory, and shared memory [24], [25] machines have been employed for the transient stability problem. Supercomputers [26], multiprocessor networks [27], [28], array-processors [29], [30], and real-time simulators based on clustered general purpose processors [13], [31] all have been examined and reported for this application. Although these hardware-based approaches helped to speed up the simulation, they were limited by many factors such as cost, communication issues, programmability, and the system size.

Recently, graphics processing units which were originally developed for rendering detailed real-time visual effects in the video gaming industry, have become programmable to the point where they are a viable general purpose programming platform [10], [11]. General purpose programming on the GPU (also called GPGPU) is garnering a lot of attention in various scientific communities due to the low cost and significant computational power of recent GPUs. The use of parallel GPUs as an alternative to the parallel CPU-based cluster of computers in simulations that need highly intensive computations has become a real possibility that has not yet been investigated in power system dynamic stability simulation.

This paper proposes to combine several aspects of parallelism to utilize the full capability of GPUs as efficiently as possible. Three types of parallelism are used in the transient stability implementation.

- Algorithm-level. This is a *top-level* or *coarse-grain* parallelism which happens before any numerical method starts solving the system equations. It is also known as *program-level* parallelism [23]. Here, the objective is not to address task definition and scheduling, but the parallelism inherent in the overall algorithm.
- Task-level. In this type of parallelism, the traditional serial algorithm is converted into various smaller and independent tasks which may be solved in parallel. For example, to solve a linear set of equations in the form of $Ax = b$, the task-level parallelism entails decomposing the A into various submatrices that can be solved in parallel. Using sparsity-based solution methods, or converting A into a block-bordered diagonal matrix are examples of the task-level parallelism approach.
- Data-parallelism. This is the *fine-grained* type of parallelism that can be used on the SIMD-based architectures such as vector processors and GPUs. A given problem must have the capability to be expressed in the data-parallel format in order to take advantages of the SIMD hardware.

Both the algorithm-level and task-level approaches can take advantage of data-parallel techniques. In the next sections, it will be shown how the proposed algorithm-level parallel method, i.e., instantaneous relaxation (IR), and a task-level parallel sparse matrix solver, can be implemented on data-parallel architecture of multiple GPUs the transient stability simulation.

3 Instantaneous Relaxation for Large-Scale Transient Stability Simulation

The Instantaneous Relaxation technique utilizes a coarse-grained parallel method that decomposes a system, mathematically described by a set of nonlinear DAEs, into smaller subsystems and solves each subsystem independently in one time-step. Each subsystem is solved for its internal variables while the required external variables are kept constant during the time-step. At the end of each time-step the external variables of all subsystems are exchanged and updated for the next time-step. Thus, IR method offers

a program-level parallelization that enables equal distribution of computation load on parallel computers [13].

3.1 Formulation

In the IR technique, the analysis to determine independent blocks of work (subsystems) that may subsequently be processed in parallel is performed in an initial step. To apply the IR method for the solution of a set of nonlinear DAEs, as described in (1) and (2), the preliminary step is to cluster variables into groups which can be solved independently. After partitioning the system into n subsystems, the following set of DAEs are prepared to describe the dynamics of each subsystem

$$\dot{\mathbf{x}}^{int} = \mathbf{f}(\mathbf{x}^{int}, \mathbf{x}^{ext}, \mathbf{v}^{int}, \mathbf{v}^{ext}, t), \tag{4}$$

$$\mathbf{0} = \mathbf{g}(\mathbf{x}^{int}, \mathbf{x}^{ext}, \mathbf{v}^{int}, \mathbf{v}^{ext}, t), \tag{5}$$

where $\mathbf{x}^{int}$ and $\mathbf{v}^{int}$ denote state and algebraic variables that define the dynamic behavior of Subsystem i, and $\mathbf{x}^{ext}$ and $\mathbf{v}^{ext}$ denote the state and algebraic variables that define all subsystems external to subsystem i. Therefore, $\mathbf{x} = \mathbf{x}^{int} \bigcup \mathbf{x}^{ext}$ and $\mathbf{v} = \mathbf{v}^{int} \bigcup \mathbf{v}^{ext}$ are the set of all state and algebraic variables that describe the original-size system. Using the Gauss-Jacobi iteration scheme the pseudocode for the GJ-IR method is as follows.

```
initialize x and v;
t=0;
repeat{
  t=t+1;
  for all subsystems (i=1,2,...,n) solve
  ẋ^int(t) = f(x^int(t), x^ext(t−1), v^int(t), v^ext(t−1));
  0 = g(x^int(t), x^ext(t−1), v^int(t), v^ext(t−1));
  update all x^ext and v^ext;
}until (t>T)
```

Fig. 1 illustrates the above mentioned steps to apply the IR method for a power system. It should be noted that the system decomposition must be done in such a way that components inside each subsystem ($\mathbf{x}_i^{int}$) are tightly interdependent while the dependency between components in two different subsystems ($\mathbf{x}_i^{int}$ and $\mathbf{x}_j^{int}$) is weak to ignore the interconnection. This loose dependency allows subsystems to be *relaxed* from their external connection during each iteration, and thus the Gauss-Jacobi iteration scheme can be efficiently utilized at the top-level.

To solve each subsystem, (4) is first discretized resulting in a new set of nonlinear algebraic equations. Here, the implicit Trapezoidal Rule is used to discretize the differential equations as follows:

$$0 = \mathbf{x}^i - \frac{h}{2}[\mathbf{f}^i(\mathbf{x}^i, \mathbf{v}^i, t) + f^i(\mathbf{x}^i, \mathbf{v}^i, t-h)], \tag{6}$$

where $i = 1, 2, \ldots, n$ indicates the subsystem, and h is the integration step-size. Then, (5) and (6) can be linearized by the Newton-Raphson method (for the jth iteration) as follows:

$$J(\mathbf{z}_{j-1}^i) \cdot \Delta \mathbf{z}^i = -\mathbf{F}^i(\mathbf{z}_{j-1}^i), \tag{7}$$

where J is the Jacobian matrix, $\mathbf{z}^i = [\mathbf{x}^i, \mathbf{v}^i]$, $\Delta \mathbf{z}^i = \mathbf{z}_j^i - \mathbf{z}_{j-1}^i$, and $\mathbf{F}^i$ is the vector of nonlinear function evaluations. Equation (7) is a set of linear algebraic equations which can be solved with LU factorization and forward and back substitution.

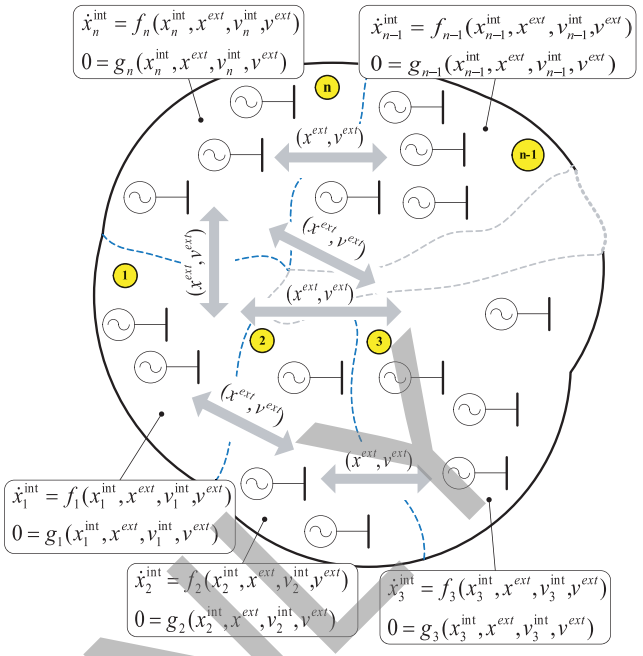


Fig. 1. Decomposing a power system into n subsystems to apply the IR algorithm.

The transient stability simulation includes two main solvers: differential-solver and the network-solver. What is explained up to this point was the differential solver that solves and updates the state variables of the system. The network-solver, however, solves the nodal equations of the network to provide the link between electrical machines and the network that includes transmission lines, transformers, and loads. The nodal equation of a power system is of the form $\mathbf{i} = \mathbf{Y}\mathbf{v}$, where $\mathbf{Y}$ is the admittance matrix of the system, $\mathbf{i}$ is the vector of injected currents at each node, and $\mathbf{v}$ is the vector of node voltages. This equation is solved for $\mathbf{v}$. The network-solver used in this paper is based on Gauss-Seidel iterations. The two solvers work alternately, i.e., at each time-step the network-solver updates the differential-solver with the bus voltages, and the differential-solver updates the network solver with the machine injected currents at the buses the machines are connected to. Then, the two solvers continue with their new inputs to the next time-step.

Benchmarking revealed that a majority of execution time in the transient stability simulation is spent for the nonlinear solution. By using the IR method, however, and by distributing the subsystems over several parallel GPUs, a large-scale system is decomposed into individual subsystems whose Jacobian matrices as well as its admittance matrix are smaller, resulting in faster computations. Following the convergence of iterative solutions in all subsystems, the external state and algebraic variables are updated.

3.2 System Partitioning

A prerequisite for implementing the IR method on parallel processors is the decomposition of the original system into subsystems in which tightly coupled variable are grouped together. The *slow coherency* method [33] is used in this paper

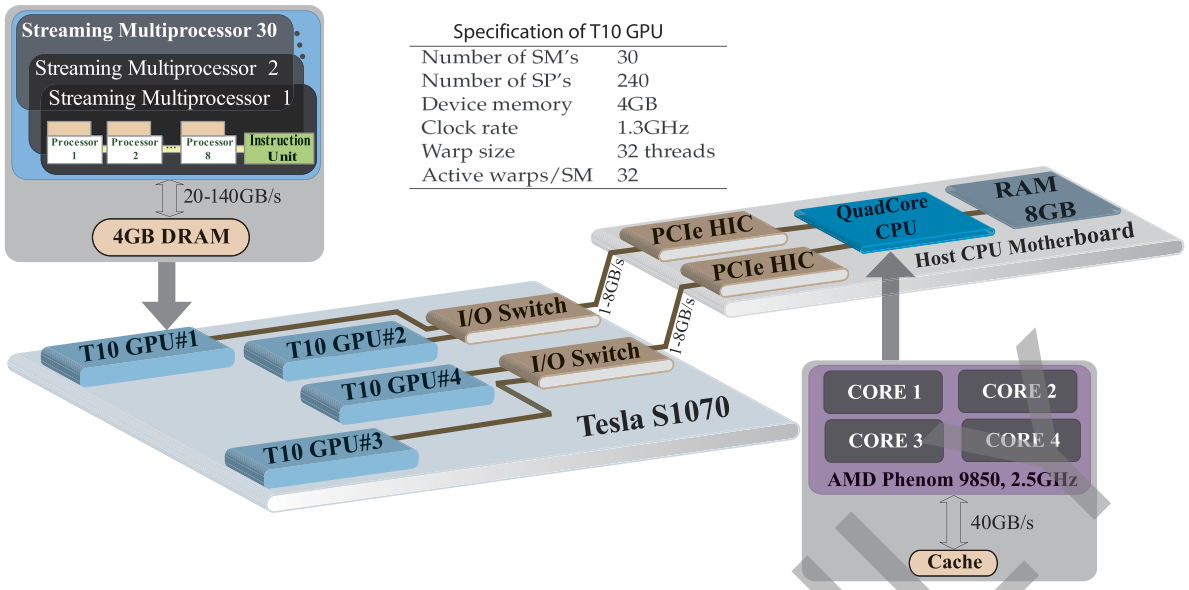


Fig. 2. Hardware configuration of Tesla S1070 GPU server connected to a host CPU motherboard.

for the system partitioning. While there are other methods [32] for system partitioning, they are stymied by the computational load balancing problem or implementation issues. Finding tightly coupled variables in a partitioned system has a physical meaning from the power system point of view. Coherent generators are those whose rotors oscillate in lock-step within any given area. Following a large disturbance in the system, some generators lose their synchronism with the rest of the network. Thus, the system is naturally partitioned into several areas in which generators are in-step while there are oscillations among the different areas. Since the coherency characteristic reflects the level of dependency between generators in an area, coherent generators can be grouped into the same subsystem which can then be solved independently from other subsystems using the IR method. The partitioning achieved using the coherency property has the advantages [33] that the coherent group of generators are independent of the location or the severity of disturbance (i.e., topology of the system), and the level of detail used in the generator modeling.

4 Graphics Processors

4.1 Hardware Architecture

The hardware used in this work is one unit of Tesla S1070 GPU server from NVIDIA [34]. The Tesla S1070 server is equipped with four independent T10 GPUs each with 4 GB of memory so that the total memory of the S1070 unit is 16 GB. The internal configuration of Tesla S1070 and its connection to the host CPU are shown in Fig. 2.

The GPU and CPU communicate via the PCIe 2.0×16 bus that supports up to 8 GB/s transfer rate. When a GPU instruction is invoked, blocks of threads (with the maximum size of 512 threads per block) are defined to assign one thread to each data element.

Each streaming multiprocessor (SM) shown in Fig. 2, includes eight stream processor cores, an instruction unit, and on-chip memory that comes in three types: registers, shared memory, and cache. Threads in each block have access to the shared memory in the SM as well as to a global memory in the GPU. When a SM is assigned to execute one or more thread blocks, the instruction unit splits and creates groups of parallel threads called *warps*. The threads in one warp are managed and processed concurrently on the eight stream processors.

Each pair of the T10 GPUs is connected to one I/O Switch. On the host motherboard a host interface card (HIC) must be plugged into the PCIe bus to establish connection between each pair of T10 GPUs and the host CPU. This implies that the Tesla S1070 cannot behave as a unified processor. To circulate data between the 4 GPUs of the Tesla S1070, data from one GPU is first transferred to the host's main memory, and it is then uploaded to the other GPUs. This deficiency is alleviated by using multiple CPU threads as explained in the Section 4.2.

4.2 Multithreaded Parallel Programming

The Tesla S1070 is a CUDA-enabled device. A CUDA program consists of multiple phases that are executed on either the CPU (*host*) or the GPU (*device*). The GPU runs its own user specified CUDA *kernel* independently but is controlled by the CPU [35]. A single T10 GPU can execute only one kernel at any given time; whereas a multi-GPU server such as the Tesla S1070 can run multiple kernels simultaneously. To have 4 GPUs working in parallel, in a Tesla S1070 4 CPU cores are required to control the GPUs simultaneously. This minimizes the overhead that occurs in data copying and kernel invocation. Therefore, multithreaded CPU programming is required to manage the parallel CPU cores. In this work, the C-runtime library is used for this purpose. Synchronizing the resource access between CPU threads is a common problem when writing CPU multithreaded applications [36]. Having two or more CPU threads simultaneously access the same data can lead to undesirable and unpredictable results. For example, one CPU thread could be updating the contents of a structure while another CPU thread is reading the contents of the same structure. To control synchronization among

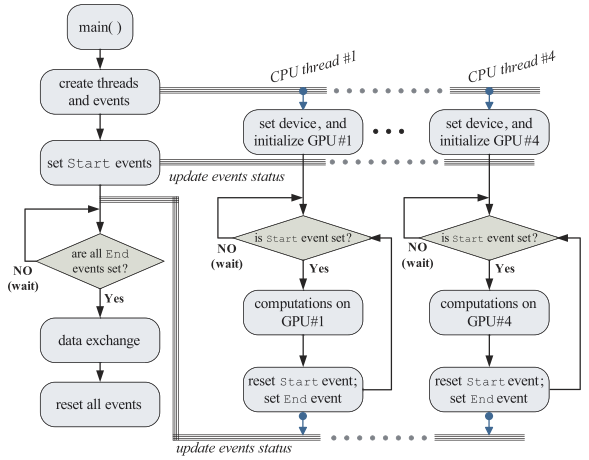


Fig. 3. Multithreaded program flow to control 4 GPUs using a quadcore CPU.

the CPU threads, *events* are used. Events allow CPU threads to be synchronized by forcing them to pause until a specific event is set or reset.

Fig. 3 illustrates the flowchart of multithread programming to manage operations of up to 4 GPUs connected to a quad-core CPU. The main thread on the CPU first creates four child threads which are responsible for handling each GPU. Child threads begin execution by setting one individual device (GPU) and initializing it, and then wait until the *Start* event is set. As soon as the main thread sets the *Start* event the computations in the individual GPU belonging to each child thread are activated. The main thread goes into a waiting state until all the threads are finished by the GPU computations before setting the *End* event. Thereafter, the four child threads exchange and update the required data with the aid of the CPU, and the simulation continues.

5 Parallel Direct Sparse Linear Solver for Transient Stability Simulation on GPU

A direct, noniterative, method for matrix solution is developed for this work. Although iterative methods such as the conjugate gradient algorithm [37] have been implemented on the GPU, the parallelism is only available within a single iteration. Since the next iteration depends on the previous one, iterations cannot be processed in parallel although the same algorithm is used within the iterations.

Moreover, since the number of iterations determined by the convergence of each matrix is dissimilar even for matrices of the same dimension, extra considerations are necessary to synchronize with other parallel tasks.

Compared with iterative methods, direct methods can give conformable solutions with multielement parallelism and stable solution time regardless of the elements in the matrix, although at the cost of higher algorithmic complexity. In the literature GPU accelerated direct solvers have been proposed for symmetric sparse matrices using Cholesky decomposition [40], and multifrontal computations [41], showing a significant speedup. However, since the Jacobian matrix in power system stability computations is unsymmetrical it requires a specific method for an efficient solution.

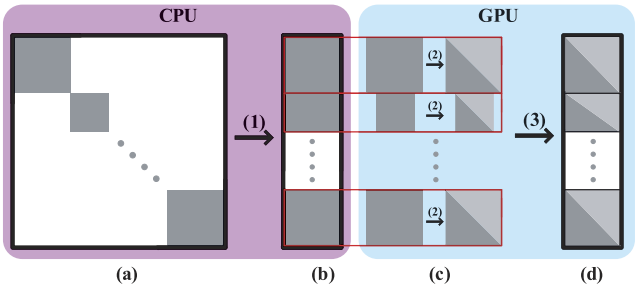


Fig. 4. The block node sparse LU algorithm for a typical Jacobian matrix. (a) Original sparse matrix in dense data structure on CPU. (b) Sparse data structure on CPU; Step 1 is dense to sparse conversion. (c) Sparse data structure on GPU; Step 2 is parallel LU factorization. (d) LU factors on GPU in the sparse data structure (Step 3).

As a direct method for solving a linear equation set $A\mathbf{x} = \mathbf{b}$, LU factorization is applied to decompose the matrix into the product of lower and upper triangular matrices

$$PA = LU, \tag{8}$$

where P is a permutation matrix for solution stability formed by exchanging the rows of A.

Forward substitution is used to solve for $\mathbf{y}$ as

$$L\mathbf{y} = P\mathbf{b}, \tag{9}$$

and then backward substitution is used to solve for $\mathbf{x}$

$$U\mathbf{x} = \mathbf{y}. \tag{10}$$

The parallel dense LU algorithm on GPU has already been efficiently implemented, and commercial packages exist such as CULA (GPU-accelerated LAPACK) [42]. The Jacobian matrix for the transient stability formulation, however, is nonsymmetric and sparse, as shown in Fig. 4a. In contrast to a dense algorithm, a sparse algorithm can significantly reduce the computational burden by only dealing with the nonzero elements. The computation of sparse matrices in Matlab is rightly based on this idea. By improving the traditional Fortran BLAS routine and development with C [43], the `lu()` function in Matlab is a more efficient sparse algorithm compared with other commercial and noncommercial sparse software on CPU.

For a nonsymmetric matrix, the method is depended on the pattern of asymmetry of the matrix. Because of the nonregular structure of the general sparse matrix, it is difficult to parallelize the LU algorithm due to the weak support for dynamic and branching operations on the GPU. Fortunately, the Jacobian matrix has an important feature that alleviates the problem: it has a block diagonal structure which enables its implementation on the GPU. Since all blocks in the matrix are decoupled, they can be decomposed in parallel. Each block can be treated as a dense node, as shown in Fig. 4b and 4c. Thus, the data structure of this sparse matrix is in the format of block node sparse. The typical size of the matrix blocks is 9×9; although a few of them have various sizes, all the blocks are within 16×16. The number of threads per CUDA block can adapt to a variety of block node dimensions so as to maximize efficiency.

Effective dense LU methods and linear solvers on GPU exist [38], [39] albeit they are used for solving a single large dense system. However, the power system dynamic

```
for all block nodes of size n{
    for i = 0 to n-1{
        find the max of column i;
        get the row id r_max;
        if A(i,rmax) ≠ A(i,i)
            swap row i and row r_max;
        update L;
        update U and remainder;
    }
}
```

(a)

```
b = blockIdx.x;  n = blockDim.x;
r = threadIdx.x; c = threadIdx.y;
A = &MatrixData[b];
for(i=0;i<n-1;i++){
    if(A[i][r] == max(A[i])){
        r_max = r;}
    if(A[i][r_max] != A[i][i]){
        swap(A[c][i],A[c][r_max]);}
    if(r>i){
        A[i][r] = A[i][r]/A[i][i];}
    if(r>i&&c>i){
        A[c][r] -= A[c][i]*A[i][r];}
}
```

(b)

Fig. 5. (a) Pseudocode for the sparse LU algorithm, and (b) its CUDA map.

Jacobian has a block node sparse structure with individual dense blocks. Therefore, a high performance dense matrix LU algorithm is designed to process the LU factorization for each dense block of the sparse Jacobian matrix. Because the size of each block is small enough (typically 9×9), all computations of LU decomposition can be implemented inside the shared memory of the CUDA block, which significantly improves the data bandwidth by avoiding global memory access, one of the bottlenecks of GPU implementation.

Since there is no overlap between the L and U matrices, a matrix A

$$A = L + U - I, \tag{11}$$

is used to store them both, where I is the identity matrix. Before partial pivoting, the maximum element of the column i of A is found as $A(i, r_{max}) = max(A(i, i+1 : n-1))$, where r_{max} denotes the row index of the maximum element of the column i of A.

The binary scan operation [44] is applied to parallelize partial pivoting, which reduces the step complexity from $O(N)$ to $O(log_2 N)$. The update for column of L is $A(k,i) = A(k,i)/A(i,i)$, while the update for row of U and the rest of the matrix is $A(k,k) = A(k,k) - A(k,i) * A(i,k)$, where $k = i+1 : n-1$. The entire algorithm and its CUDA map is shown in Fig. 5.

This algorithm can handle various block dimensions by introducing a index vector for the block compress sparse data structure. The LU factorization of a singular square matrix whose rank is less than its order can also be supported by this algorithm. Table 1 shows a comparison of the sparse LU algorithm implementation on the CPU (using Matlab) by leveraging the power of the 4 cores of the AMD Phenom 9850 CPU, and the GPU for various Jacobian matrices gleaned from test systems described in the Section 6. As can be seen for increasing size of the matrix the GPU implementation can be highly efficient.

TABLE 1
Comparison of Sparse LU Implementation on CPU and GPU for Power System Jacobian

Jacobian matrix	Sparsity (%)	LU on CPU (Matlab)	LU on GPU	Speed-up
87 × 87	89.9	0.070s	0.118s	0.59
174 × 174	94.95	0.131s	0.121s	1.08
348 × 348	97.47	0.253s	0.126s	2.01
696 × 696	98.74	0.501s	0.136s	3.68
1392 × 1392	99.37	1.013s	0.201s	5.04
2784 × 2784	99.68	2.103s	0.291s	7.23
5568 × 5568	99.86	4.203s	0.507s	8.29
11136 × 11136	99.92	8.156s	0.800s	10.19
22272 × 22272	99.96	16.447s	1.575s	10.44

6 Experimental Results and Discussion

In this section, we demonstrate results of the IR-based transient stability implementation on multiple GPUs. The entire simulation code is written in C++ integrated with CUDA and CUBLAS library [45] and all calculations employ 64-bit floating point number representation. The accuracy of the simulation has been verified using the PSS/E software from Siemens Energy Inc.

6.1 Test Systems

As shown in Fig. 6, several test systems of increasing size were constructed to explore the efficiency of the multi-GPU-based simulations. The specifications of each test system in terms of the number of generators, buses, the total number of DAEs, as well as dimensions of the Jacobian matrix required for the system modeling are listed in Table 2. The $Scale\ 1$ system is the IEEE's New England test system with 10 generators and 39 buses. This system was duplicated several times and interconnected via appropriate number of transmission lines to create larger systems that are highly meshed. Thus, test systems with $Scale$ of 2, 4, 8, 16, 32, 64, 128, and 256 were obtained. The $Scale\ 256$ system was constructed based on the maximum compute capacity of the current host computer hardware. The steady-state and dynamic stability of these systems have been examined and verified by Siemens' PSS/E software. A flat start was used for all the state variables in the system, i.e., voltage and angle of all buses set to $1.0\angle 0^{\circ}$p.u., and PSS/E was used to find the initial load flow results. The developed GPU software is interfaced with PSS/E's *.raw file format, so that the load flow data can be fed directly into the prepared simulation codes.

6.2 Implementation of the IR Algorithm on the Tesla S1070

The IR method was implemented on the multi-GPU Tesla S1070 server. This implementation is a combination of algorithm-level-parallelism (coarse-grain) and data-parallelism (fine-grain). Fig. 7 illustrates the flowchart of the IR implementation. In Table 2, the computation time to simulate a duration of 1,580 ms using 1, 2, and 4 GPUs is

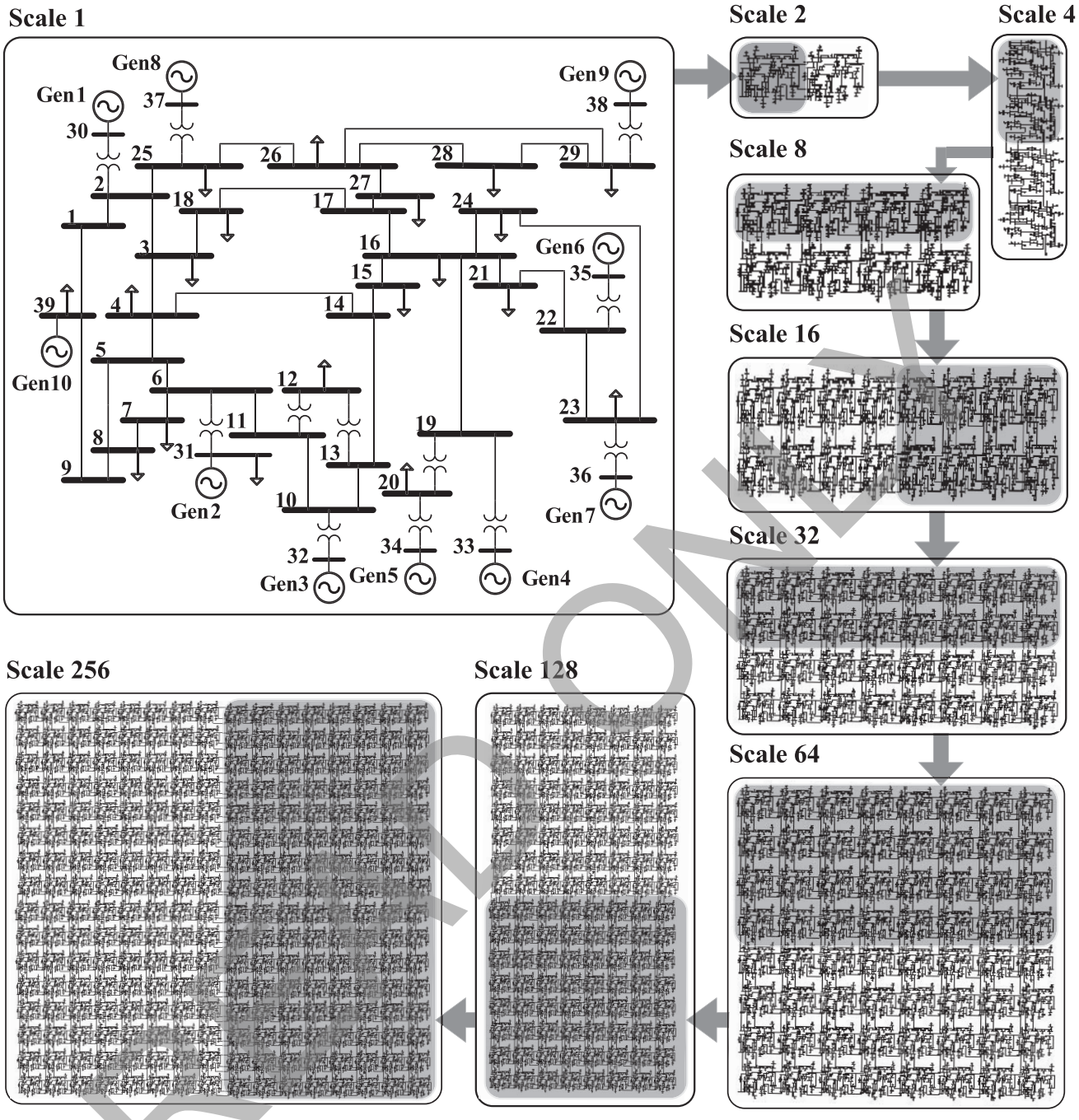


Fig. 6. Construction of test systems for transient stability simulations. The *Scale 1* system shows the one-line diagram of the IEEE's New England Test System.

listed. In the case of 1 GPU, the serial IR method was implemented, where the subsystems are solved sequentially instead of simultaneously. For 2 and 4 GPUs, however, the parallel IR method has been used as described in Section 2. With 4 GPUs the *Scale* 1 system was ignored because the communication and computation times are too similar to reveal any computational advantage of using 4 GPUs.

Figs. 8 and 9 illustrate time-domain results for the terminal voltages and rotor angles of the 30 generators closest to the disturbance location in the *Scale* 256 system. This system entirely occupies Tesla S1070 to solve 27,392 DAEs. The network is perturbed by a three-phase fault at time 0.5 s that is cleared after 80 ms. Obviously, buses close to the fault area are affected more than buses geographically farther from the disturbance. Similar results were observed for other system scales as well.

6.3 Observations

6.3.1 Transparency

In the developed C++ code the IR method was implemented to work with 1 to 4 GPUs, depending on the required compute capacity. Transparency in parallel processing software development refers to the ease with which software written for a set number of processors can be reformulated for another group of processors [46]. While the SIMD hardware architecture of the GPU and the way kernels are

TABLE 2
System Scale versus GPU Computation Time for Various Configurations for a Simulation Duration of 1,580 ms

System Specifications					Serial IR Method		Parallel IR Method	
System scale	Generators	Buses	Nonlinear DAEs	Jacobian matrix	1 GPU 2 Areas	1 GPU 4 Areas	2 GPUs 2 Areas	4 GPUs 4 Areas
1	10	39	107	87 × 87	0.439s	N/A	0.223s	N/A
2	20	78	184	174 × 174	0.488s	1.122s	0.247s	0.295s
4	40	156	428	348 × 348	0.557s	1.125s	0.280s	0.298s
8	80	312	856	696 × 696	0.764s	1.404s	0.386s	0.376s
16	160	624	1712	1392 × 1392	1.600s	3.143s	0.870s	0.866s
32	320	1248	3424	2784 × 2784	2.379s	4.617s	1.296s	1.254s
64	640	2496	6848	5568 × 5568	10.207s	20.252s	5.843s	5.606s
128	1280	4992	13696	11136 × 11136	67.005s	132.938s	34.242s	34.119s
256	2560	9984	27392	22272 × 22272	616.534s	1237.990s	310.769s	313.777s

invoked and run by multiple threads offers a high degree of transparency, the developed software itself is effectively controllable by changing one variable at the compile time to work with increasing number of parallel GPUs.

6.3.2 Data Structure

Table 2 shows that the Jacobian matrix size increases quadratically with the system scale. The memory (DRAM) required in GB to store a dense Jacobian matrix of size m with 4 bytes per floating point number is $(m * m * 4)/(1{,}024^3)$; for system scales 128 and 256 this number is quite large—of the order of 0.5-2 GB during the simulation run. Moreover, this matrix is the interface between the sparse linear solver explained in Section 5 and the transient stability model. While the theoretical data transfer bandwidth between GPU and CPU is 1-8 GB/s, in the current hardware configuration it is only 1.35 GB/s. Transferring data in the order of a few GB with a slow transfer rate takes significant time. Therefore, to eliminate the storage space requirement and to overcome the transfer bandwidth limitation a sparse data structure was used to save only the nonzero elements of large-scale matrices.

6.3.3 Scalability

Another important issue in a parallel processing-based simulation technique is the scalability characteristic [46]. The scaling factor (SF) is defined as follows:

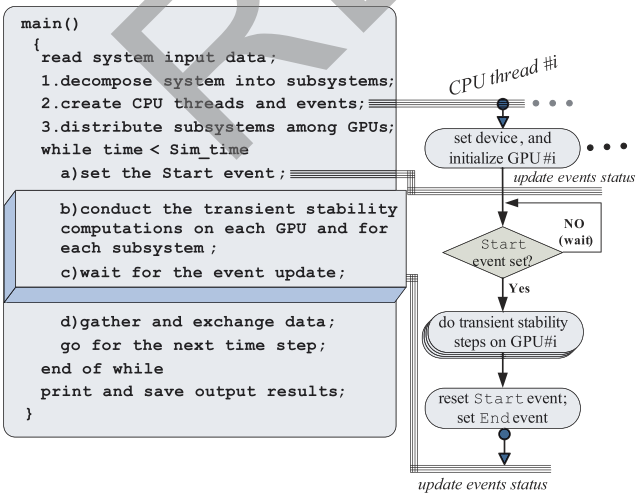


Fig. 7. Pseudocode of IR algorithm implementation on multiple GPUs; CPU thread $\#\mathrm{i} = 1, 2, 3$, or 4.

$$SF = \frac{computation\ time\ of\ Single\ GPU}{computation\ time\ of\ Multiple\ GPUs}. \quad (12)$$

The SF reveals how efficient the parallel multi-GPU simulation is in comparison with the single-GPU simulation. Ideally, using n parallel processors running simultaneously to solve a problem which takes Ts on a single processor, the simulation time would break down to $\frac{T}{n}$ s. However, this is not true in practice due to several software development issues such as task scheduling, processors' communication, and the parallel processing algorithm. Thus, SF is always less than n, and the closer it is to n, the higher the efficiency of the multiprocessing-based technique. This factor was computed for the IR method for 2 and 4 GPUs, and for all system scales. The results are depicted in Fig. 10. For the IR method in both the 2 and 4 GPUs, as the test system size expands, the SF grows closer to the number of parallel GPUs in use. This means that the IR method is scalable, and if the GPU-CPU communication overhead could be reduced, for example,

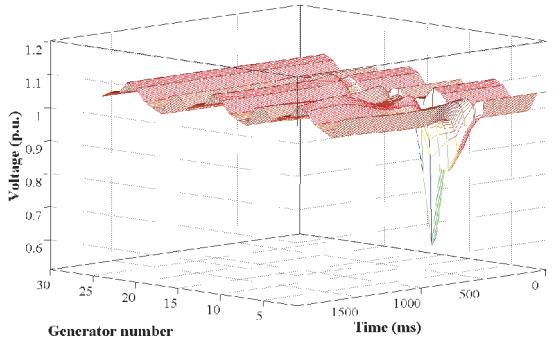


Fig. 8. Terminal voltages of 30 generators in the Scale 256 system.

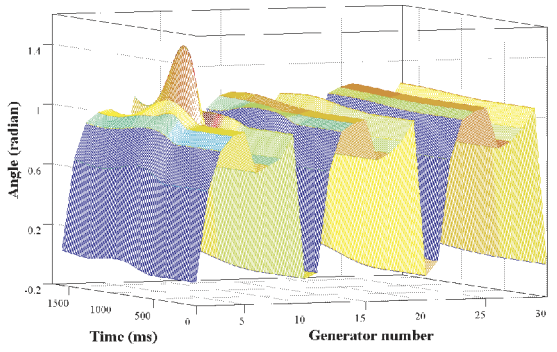


Fig. 9. Rotor angles of 30 generators in the Scale 256 system.

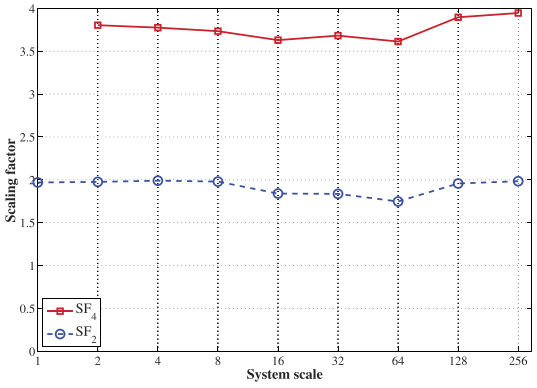


Fig. 10. Scaling factor for the IR method using 2 and 4 GPUs, SF_2 and SF_4, respectively.

by hardware upgrade or advancement in GPU cluster technology, the SF will increase further.

6.3.4 Communication Overhead and Efficiency

As explained earlier, the GPUs inside the Tesla S1070 have no internal communication, i.e., their only recourse is to exchange data using the CPU as the conduit. This unavoidable communication adds extra latency which is negligible for small scale systems. For large-scale systems with large data sets, however, this GPU-CPU communication results in significant communication overhead. As shown in Fig. 2, the data transfer bandwidth between the GPU and CPU is limited to 8 GB/s, which is quite slow compared to that of each GPU and its own memory (20-140 GB/s). Therefore, from the programming viewpoint, higher efficiencies can be realized by minimizing the amount of data transfer to the CPU. In the developed CUDA kernel global memory access is avoided as much as possible in order to maximize efficiency. The impact of this issue on the simulation time is observable in Table 2 for 2 and 4 GPUs. In 4 GPU cases, there are more GPU-CPU communications which shadow the benefits of dividing the system into four smaller subsystems in comparison with the 2 GPU cases.

7 Conclusion

This paper investigated the potential of using multiple GPUs for transient stability simulation which is a main component of any dynamic security assessment (DSA) tool. For implementation on a multi-GPU computational server a robust relaxation-based method was used as a coarse-grained program-level parallel processing-based technique. The instantaneous relaxation algorithm exploits implicit Trapezoidal integration rule, Newton-Raphson iteration, and sparse LU decomposition methods. Traditional methods for transient stability simulation do not have a structure suitable for exploiting the parallel architecture of GPUs. The GJ-IR algorithm, however, reveals a two-level parallel structure: individual blocks of the Jacobian matrix are independent, and each of these blocks may be inverted by a data-parallel LU decomposition algorithm. Therefore, the GJ-IR is an ideal algorithm for mapping to modern GPUs, and is able to achieve both good scaling and high overall performance.

A GPU-based efficient parallel sparse linear solver is proposed that exploits the block diagonal structure of the Jacobian matrix. This solver relies on the binary scan-based partial pivoting, block node sparse data structure, and SIMD dense model. It is demonstrated that this linear solver is at least 2-10 times faster than an efficient sparse CPU-based solver.

The simulation codes for the GPU implementation are quite flexible and extensible; they are written entirely in C++ integrated with GPU-specific functions. The accuracy of the simulation was validated by the PSS/E software. The efficiency was evaluated for several large test cases by utilizing 1, 2, and 4 GPUs working in parallel. The largest scale system includes 2560 generators all modeled in detail within 27,392 DAEs. The developed software that takes advantage of CPU-based multithread programming style was shown to be transparent, scalable, and efficient from the data storage and GPU-CPU communication viewpoint. Performance of the multi-GPU algorithm can be further increased by direct GPU-to-GPU communication such as the new GPUDirect technology which allows peer-to-peer between GPUs on the same PCIe bus. Moreover, newer generation of GPUs, for example the Fermi architecture from NVIDIA, have the capability to run multiple parallel kernels which would make it easier to implement a parallel processing-based transient stability technique, and alleviate the GPU-CPU data bandwidth bottleneck.

Appendix

Power System Representation for the Transient Stability Analysis

The mathematical modeling of the transient stability phenomena involves a set of nonlinear differential equations modeling the dynamics of the synchronous machines and a set of nonlinear algebraic equations that model nonrotating components. In this paper, each synchronous generator is represented by a detailed 9th order model with the excitation system including the automatic voltage regulator (AVR) and the power system stabilizer (PSS). The state equations of the machine are given as follows:

$$\dot{\delta}(t) = \omega_R.\Delta\omega(t), \tag{13}$$

$$\Delta\dot{\omega}(t) = \frac{1}{2H}[T_e(t) + T_m - D.\Delta\omega(t)], \tag{14}$$

$$\dot{\psi}_{fd}(t) = \omega_R.[e_{fd}(t) - R_{fd}i_{fd}(t)], \tag{15}$$

$$\dot{\psi}_{1d}(t) = -\omega_R.R_{1d}i_{1d}(t), \tag{16}$$

$$\dot{\psi}_{1q}(t) = -\omega_R.R_{1q}i_{1q}(t), \tag{17}$$

$$\dot{\psi}_{2q}(t) = -\omega_R.R_{2q}i_{2q}(t), \tag{18}$$

$$\dot{v}_1(t) = \frac{1}{T_R}[v_t(t) - v_1(t)], \tag{19}$$

$$\dot{v}_2(t) = K_{stab}.\dot{\Delta\omega}(t) - \frac{1}{T_w}v_2(t), \tag{20}$$

$$\dot{v}_3(t) = \frac{1}{T_2}[T_1\dot{v}_2(t) + v_2(t) - v_3(t)]. \tag{21}$$

The stator voltage equations are

$$e_d(t) = -R_a i_d(t) + L''_q i_q(t) - E''_d(t), \tag{22}$$

$$e_q(t) = -R_a i_d(t) - L''_d i_d(t) - E''_q(t),$$

where

$$E''_d \equiv L_{aq}\left[\frac{\psi_{q1}}{L_{q1}} + \frac{\psi_{q2}}{L_{q2}}\right], \tag{23}$$

and

$$E''_q \equiv L_{ad}\left[\frac{\psi_{fd}}{L_{fd}} + \frac{\psi_{d1}}{L_{d1}}\right].$$

The electrical torque of the machine is given as

$$T_e = -(\psi_{ad} i_q - \psi_{aq} i_d), \tag{24}$$

where

$$\psi_{ad} = L''_{ad}\left[-i_d + \frac{\psi_{fd}}{L_{fd}} + \frac{\psi_{d1}}{L_{d1}}\right], \tag{25}$$

and

$$\psi_{aq} = L''_{aq}\left[-i_q + \frac{\psi_{q1}}{L_{q1}} + \frac{\psi_{q2}}{L_{q2}}\right].$$

ω_R, H, D, R_{fd}, R_{1d}, R_{1q}, R_{2q}, R_a, L_{fd}, L_{d1}, L_{q1}, L_{q2}, L''_d, L''_q, L_{ad}, L_{aq}, L''_{ad}, L''_{aq}, T_R, T_w, T_1, T_2, and K_{stab} are constant system parameters whose definition can be found in [43]. According to this formulation the vector of state variables in (1) and (2) for the synchronous generator is given as

$$\mathbf{x} = [\delta \;\; \Delta\omega \;\; \psi_{fd} \;\; \psi_{d1} \;\; \psi_{q1} \;\; \psi_{q2} \;\; v_1 \;\; v_2 \;\; v_3]^t. \tag{26}$$

The algebraic equations are based on the nodal equations for the network

$$\mathbf{Yv} = \mathbf{i}, \tag{27}$$

where $\mathbf{i}_{(n+r)\times 1} = [\mathbf{i}_{n\times 1} \;\vdots\; 0_{r\times 1}]^t$. n denotes the number of generator nodes and r denotes the number of remaining nodes. $\mathbf{Y}$ and $\mathbf{v}$ are the admittance matrix and the voltage vector of the network. This equation is solved for $\mathbf{v}$.

ACKNOWLEDGMENTS

Financial support from the Natural Science and Engineering Research Council of Canada (NSERC) is gratefully acknowledged.

REFERENCES

[1] IEEE/CIGRE Joint Task Force on Stability Terms and Definitions, "Definition and Classification of Power System Stability," *IEEE Trans. Power Systems,* vol. 19, no. 2, pp. 1387-1401, May 2004.

[2] K. Morison, L. Wang, and P. Kundur, "Power System Security Assessment," *IEEE Power and Energy Magazine,* vol. 2, no. 5, pp. 30-39, Sept./Oct. 2004.

[3] L. Wang and K. Morison, "Implementation of Online Security Assessment," *IEEE Power and Energy Magazine,* vol. 4, no. 5, pp. 46-59, Sept./Oct. 2006.

[4] EPRI TR-104352, "Analytical Methods for Contingency Selection and Ranking for Dynamic Security Analysis," *Power and Energy Soc. General Meeting, Conversion and Delivery of Electrical Energy in the 21st Century,* Project 3103-03 Final Report, Sept. 1994.

[5] D. Blythe, "Rise of the Graphics Processor," *Proc. IEEE,* vol. 96, no. 5, pp. 761-778, May 2008.

[6] J.D. Owens, M. Houston, D. Luebke, S. Green, J.E. Stone, and J.C. Phillips, "GPU Computing," *Proc. IEEE,* vol. 96, no. 5, pp. 879-899, May 2008.

[7] W. Liu, B. Schmidt, G. Voss, and W. Muller-Wittig, "Streaming Algorithms for Biological Sequence Alignment on GPUs," *IEEE Trans. Parallel and Distributed Systems,* vol. 18, no. 9, pp. 1270-1281, Sept. 2007.

[8] R. Weber, A. Gothandaraman, R.J. Hinde, and G.D. Peterson, "Comparing Hardware Accelerators in Scientific Applications: A Case Study," *IEEE Trans. Parallel and Distributed Systems,* vol. 22, no. 1, pp. 1045-9219, Jan. 2011.

[9] W.J. van der Laan, A.C. Jalba, and J.B.T.M. Roerdink, "Accelerating Wavelet Lifting on Graphics Hardware Using CUDA," *IEEE Trans. Parallel and Distributed Systems,* vol. 22, no. 1, pp. 132-146, Jan. 2011.

[10] NVIDIA CUDA: Programming Guide, June 2008.

[11] T.D. Han and T.S. Abdelrahman, "hiCUDA: High-Level GPGPU Programming," *IEEE Trans. Parallel and Distributed Systems,* vol. 22, no. 1, pp. 78-90, Jan. 2011.

[12] V. Jalili-Marandi and V. Dinavahi, "SIMD-Based Large-Scale Transient Stability Simulation on the Graphics Processing Unit," *IEEE Trans. Power Systems,* vol. 25, no. 3, pp. 1589-1599, Aug. 2010.

[13] V. Jalili-Marandi and V. Dinavahi, "Instantaneous Relaxation Based Real-Time Transient Stability Simulation," *IEEE Trans. Power Systems,* vol. 24, no. 3, pp. 1327-1336, Aug. 2009.

[14] B. Stott, "Power System Dynamic Response Calculations," *Proc. IEEE,* vol. 67, no. 2, pp. 219-241, Feb. 1979.

[15] F.F. de Mello, J.W. Felts, T.F. Laskowski, and L.J. Opple, "Simulating Fast and Slow Dynamic Effects in Power Systems," *IEEE Computer Applications in Power,* vol. 5, no. 3, pp. 33-38, July 1992.

[16] H.W. Dommel and N. Sato, "Fast Transient Stability Solutions," *IEEE Trans. Power Apparatus and Systems,* vol. PAS-91, no. 4, pp. 1643-1650, July 1972.

[17] W. Gear, "Simultaneous Numerical Solutions of Differential-Algebraic Equations," *IEEE Trans. Circuit Theory,* vol. CT-18, no. 1, pp. 89-95, Jan. 1971.

[18] L. Elden and L. Wittmeyer-Koch, *Numerical Analysis—An Introduction.* Academic Press Inc., 1990.

[19] G. Kron, *Diakoptics-Piecewise Solutions of Large Systems,* vols. 158/162, General Electric and Also Published by McDonald, 1963.

[20] M. La Scala, G. Sblendorio, A. Bose, and J.Q. Wu, "Comparison of Algorithms for Transient Stability Simulations on Shared and Distributed Memory Multiprocessors," *IEEE Trans. Power Systems,* vol. 11, no. 4, pp. 2045-2050, Nov. 1996.

[21] F.L. Alvarado, "Parallel Solution of Transient Problems by Trapezoidal Integration," *IEEE Trans. Power Apparatus and Systems,* vol. PAS-98, no. 3, pp. 1080-1090, May/June 1979.

[22] M. La Scala, R. Sbrizzai, and F. Torelli, "A Pipelined-in-Time Parallel Algorithm for Transient Stability Analysis," *IEEE Trans. Power Systems,* vol. 6, no. 2, pp. 715-722, May 1991.

[23] M.L. Crow and M. Ilic, "The Parallel Implementation of the Waveform Relaxation Method for Transient Stability Simulations," *IEEE Trans. Power Systems,* vol. 5, no. 3, pp. 922-932, Aug. 1990.

[24] M.J. Flynn, "Very High Speed Computing Systems," *Proc. IEEE,* vol. 54, no. 12, pp. 1901-1909, Dec. 1966.

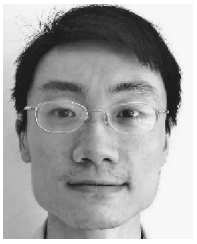
[25] J.R. Gurd, "A Taxonomy of Parallel Computer Architectures," *Proc. Int'l Specialist Seminar Design and Application of Parallel Digital Processors,* pp. 57-61, Apr. 1988.

[26] H.H. Happ, C. Pottle, and K.A. Wirgau, "An Assessment of Computer Technology for Large Scale Power System Simulation," *Proc. IEEE Conf. Power Industry Computer Applications,* pp. 316-324, May 1979.

[27] F.M. Brasch, J.E. Van Ness, and S.C. Kang, "Simulation of a Multiprocessor Network for Power System Problems," *IEEE Trans. Power Apparatus and Systems,* vol. PAS-101, no. 2, pp. 295-301, Feb. 1982.

[28] S.Y. Lee, H.D. Chiang, K.G. Lee, and B.Y. Ku, "Parallel Power System Transient Stability Analysis on Hypercube Multiprocessors," *IEEE Trans. Power Systems,* vol. 6, no. 3, pp. 1337-1343, Aug. 1991.
[29] H. Taoka, S. Abe, and S. Takeda, "Fast Transient Stability Solution Using an Array Processor," *IEEE Trans. Power Apparatus and Systems,* vol. PAS-102, no. 12, pp. 3835-3841, Dec. 1983.
[30] M. Takatoo, S. Abe, T. Bando, K. Hirasawa, M. Goto, T. Kato, and T. Kanke, "Floating Vector Processor for Power System Simulation," *IEEE Trans. Power Apparatus and Systems,* vol. PAS-104, no. 12, pp. 3360-3366, Dec. 1985.
[31] P. Forsyth, R. Kuffel, R. Wierckx, J. Choo, Y. Yoon, and T. Kim, "Comparison of Transient Stability Analysis and Large-Scale Real Time Digital Simulation," *Proc. IEEE Porto Power Tech,* vol. 4, pp. 1-7, Sept. 2001.
[32] J.S. Chai and A. Bose, "Bottlenecks in Parallel Algorithms for Power System Stability Analysis," *IEEE Trans. Power Systems,* vol. 8, no. 1, pp. 9-15, Feb. 1993.
[33] H. You, V. Vittal, and X. Wang, "Slow Coherency-Based Islanding," *IEEE Trans. Power Systems,* vol. 19, no. 1, pp. 483-491, Feb. 2004.
[34] NVIDIA, "Specification: Tesla S1070 GPU Computing System," Oct. 2008.
[35] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym, "NVIDIA Tesla: A Unified Graphics and Computing Architecture," *IEEE Micro,* vol. 28, no. 2, pp. 39-55, Mar./Apr. 2008.
[36] Visual Studio Developer Center, http://msdn.microsoft.com/en-us/library/, 2012.
[37] L. Buatois, G. Caumon, and B. Lévy, "Concurrent Number Cruncher: A GPU Implementation of a General Sparse Linear Solver," *Int'l J. Parallel, Emergent and Distributed Systems,* vol. 24, no. 3, pp. 205-223, June 2009.
[38] M. Fatica, "Accelerating Linpack with CUDA on Heterogenous Clusters," *Proc. Second Workshop General Purpose Processing on Graphics Processing Units,* pp. 46-51, Mar. 2009.
[39] S. Tomov, R. Nath, H. Ltaief, and J. Dongarra, "Dense Linear Algebra Solvers for Multicore with GPU Accelerators," *Proc. IEEE Symp. Parallel and Distributed Processing (IPDPS '10),* pp. 1-8, Jan. 2010.
[40] G.P. Krawezik and G. Poole, "Accelerating the ANSYS Direct Sparse Solver with GPUs," *Proc. Symp. Application Accelerators in High Performance Computing (SAAHPC '10),* 2010.
[41] R.F. Lucas, G. Wagenbreth, D.M. Davis, and R. Grimes, "Multifrontal Computations on GPUs and Their Multi-Core Hosts," *Proc. Ninth Int'l Conf. High Performance Computing for Computational Science (VECPAR '10),* vol. 6449, pp. 71-82, 2011.
[42] CULA, http://www.culatools.com/, 2012.
[43] J.R. Gilbert, C. Moler, and R. Schreiber, "Sparse Matrices in Matlab: Design and Implementation," *SIAM J. Matrix Analysis and Applications,* vol. 13, pp. 333-356, 1992.
[44] M. Garland, "Sparse Matrix Computations on Manycore GPU's," *Proc. 45th Ann. Design Automation Conf.,* pp. 2-6, June 2008.
[45] NVIDIA, "CUDA CUBLAS Library," Mar. 2008.
[46] IEEE Task Force on Computer and Analytical Methods, "Parallel Processing in Power Systems Computation," *IEEE Trans. Power Systems,* vol. 7, no. 2, pp. 629-638, May 1992.
[47] P. Kundur, *Power System Stability and Control.* McGraw-Hill, 1994.

Vahid Jalili-Marandi (S'06) received the BSc and MSc degrees in power systems engineering from the University of Tehran, Iran, in 2003 and 2005, respectively, and the Phd degree at the University of Alberta, Edmonton, Canada, in 2010. Currently, he is working at OPAL-RT Technologies Inc., Montreal, Canada. His research interests include transient stability studies, real-time simulations, parallel processing, and high-performance computing on GPUs. He is a student member of the IEEE.



Zhiyin Zhou (S'10) received the BSc degree in electronic engineering from Nanjing University, China, in 2000. Currently, he is working toward the MSc degree at the University of Alberta, Edmonton, Canada. His research interests include large-scale parallel and distributed computing, multithread processor system prallel programming, power system simulation, and electromagnetic transient studies. He is a student member of the IEEE.

Venkata Dinavahi (S'94-M'00-SM'08) received the PhD degree in electrical and computer engineering from the University of Toronto, in 2000. Currently, he is working as a professor at the University of Alberta. His research interests include real-time simulation of power systems and power electronic systems, large-scale system simulation, and parallel and distributed computing. He is a senior member of the IEEE.