High Throughput Computing for Massive Scenario Analysis and Optimization to Minimize Cascading Blackout Risk

Eric Anderson, Jeff Linderoth*

October 9, 2015

Abstract

We describe a simulation-based optimization method that allocates additional capacity to transmission lines in order to minimize the expected value of the load shed due to a cascading blackout. Estimation of the load-shed distribution is accomplished via the ORNL-PSerc-Alaska (OPA) simulation model, which solves a sequence of linear programs. Key to achieving an effective algorithm is the use of a High-Throughput Computing environment that allocates computational resources on a platform of more than 14,000 cores simultaneously among several users. We discuss also important implementation details necessary to achieve effective implementation in this massive-scale computing environment. In the end, we demonstrate a prototype computation that reduces the expected load shed by 76% allocating only 1.1% of the installed capacity. The massive-scale computation is made possible using the computational platform provided through HTCondor, effectively obtaining over five months of CPU time in just over one day.

1 Introduction

The ORNL-PSerc-Alaska (OPA) model is a blackout model that relies on a sequence of linear programs to simulate the load shed caused by a cascading failure of a power system [1,4,5]. Repeated batches of simulations can produce an estimate of the distribution of load shed for a given, exogenous triggering event. The independent nature of the replicated simulations makes OPA a natural candidate for implementation in high-performance and high-throughput computing environments. In this paper, we demonstrate how to leverage the CPU cycles provide by the HTCondor distributed computing software [16] to efficiently

*Department of Industrial and Systems Engineering, University of Wisconsin-Madison. email: {eanderson4,jlinderoth}@wisc.edu

estimate the impact of a cascading blackout. The vast CPU power available for simulation allows us to also consider the optimization of the network topology and operating parameters in order to minimize the impact of a triggering cascade event. A focus of the paper is on the implementation details necessary to achieve high utilization in this massive-scale computing environment. The paper demonstrates the utility of using federations of (possibly non-dedicated), loosely-coupled, shared computational resources for scenario-based analysis and optimization for power systems problems.

In Section 2, we give background on the mathematical and simulation models used to estimate the load-shed distribution of a cascading power failure. We also describe an optimization problem that allocates additional line capacity in order to minimize the impact of a cascading power failure and the search procedure used by our algorithm. Section 3 describes the computational infrastructure used by our algorithm. In Section 4, we briefly describe computational enhancements that improve the efficiency of our algorithm. A case study demonstrating the utility of our algorithm for reducing expected load shed and the significant computational power available in our high-throughput computing environment is given in Section 5.

2 Background

We view the load shed $F(\cdot)$ caused by a cascading blackout in a power network as a function of four arguments

$$F(x, \mathbf{d}, \boldsymbol{\xi}, \boldsymbol{\omega}),$$

where

- x Operating parameters of the network, e.g. line capacities;
- $\mathbf{d}$ Real power demands, (possibly random);
- $\boldsymbol{\xi}$ A random (exogenous) cascade-triggering event;
- $\boldsymbol{\omega}$ A random variable encapsulating the evolution of the cascading event.

Throughout the paper, we adhere to the convention that random variables are typeset in bold, and realizations of a random variable are given in normal typeface. In this section, we describe a simulation model that estimates the distribution of the random variable $F(x, \hat{d}, \hat{\xi}, \boldsymbol{\omega})$, an optimization model for the problem of minimizing the expected load shed, and the optimization method we use for approximately solving the model.

2.1 The OPA Simulation Model

The "inner loop" of the OPA model is designed to estimate the distribution of the random variable $F(\hat{x}, \hat{d}, \hat{\xi}, \boldsymbol{\omega})$—the load shed given operating parameters $\hat{x}$, fixed loads $\hat{d}$, and a

triggering event $\hat{\xi}$. A simplified version of a power network may be represented as a graph $(\mathcal{B}, \mathcal{L})$ with a set of buses $\mathcal{B}$, a set of lines $\mathcal{L}$, a set of generation nodes $\mathcal{G} \subseteq \mathcal{B}$, and a set of demand nodes $\mathcal{D} \subseteq \mathcal{B}$. The initial triggering event $\hat{\xi}$ results in an initial set of lines $\mathcal{O}(\hat{\xi}) \subseteq \mathcal{L}$ that may no longer carry flow. In OPA, the distribution of the random variable $F(\hat{x}, \hat{d}, \hat{\xi}, \boldsymbol{\omega})$ is estimated by solving a sequence of linear programs. As we will ultimately be interested in how changing line capacity affects load shed, in our version of the linear program, the additional capacity on lines is parameterized with the vector $\hat{x}$. With these definitions, the linear program used by OPA has the following form:

$$\min_{f,p,\theta,t} \quad \sum_{i \in \mathcal{G}} c_i p_i + \Lambda \sum_{i \in \mathcal{D}} t_i \qquad (LP(\hat{x}, \hat{d}, \mathcal{O}))$$

$$\text{s.t.} \quad \sum_{j \in \delta^+(i)} f_{ij} - \sum_{j \in \delta^-(i)} f_{ij} = \begin{cases} p_i & \forall i \in \mathcal{G} \\ \hat{d}_i - t_i & \forall i \in \mathcal{D} \\ 0 & \text{otherwise} \end{cases} \tag{1}$$

$$f_{ij} - B_{ij}(\theta_i - \theta_j) = 0 \quad \forall (i,j) \in \mathcal{L} \setminus \mathcal{O} \tag{2}$$

$$-\overline{f}_{ij} - \hat{x}_{ij} \leq f_{ij} \leq \overline{f}_{ij} + \hat{x}_{ij} \quad \forall (i,j) \in \mathcal{L} \tag{3}$$

$$\underline{P}_i \leq p_i \leq \overline{P}_i \; \forall i \in \mathcal{G} \tag{4}$$

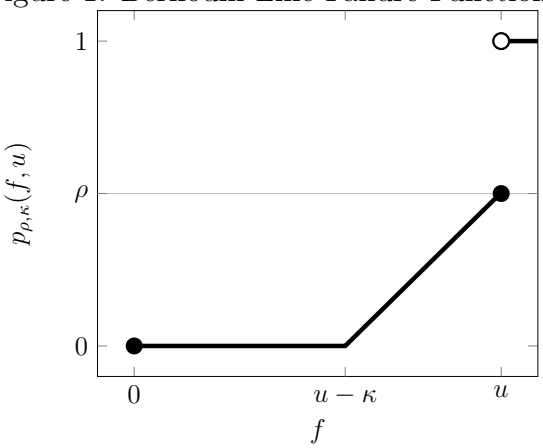
$$f_{ij} = 0 \quad \forall (i,j) \in \mathcal{O} \tag{5}$$

$$t_i \geq 0 \qquad \forall i \in \mathcal{D}.$$

In $(LP(\hat{x}, \hat{d}, \mathcal{O}))$, the variables f_{ij} denote the (real) power flow on line $(i,j) \in \mathcal{L}$, the variable p_i is the real power generation at generator $i \in \mathcal{G}$, the variables θ_i denote the phase angles at each bus $i \in \mathcal{B}$, and the variables t_i represent the load shed at nodes $i \in \mathcal{D}$. The parameter Λ is a large constant that prioritizes minimizing the load shed over the generation cost in the objective. The equations (1) are the power-flow balance equations, and equations (2) enforce the DC-power flow requirement that the flow on line $(i,j) \in \mathcal{L}$ is proportional to the voltage angle difference between the endpoints of the line. Inequalities (3) limit the absolute value of the real power a line to be no more than the installed capacity, where $\overline{f}_{ij}$ is the original installed capacity and $\hat{x}_{ij}$ is the additional capacity (that is fixed in this subproblem). Each generator $i \in \mathcal{G}$ has a lower ($\underline{P}_i$) and upper ($\overline{P}_i$) bound on the level of power it may supply. Lines in the set $\mathcal{O}$ that have failed are not allowed to carry flow by equations (5), and these lines also no longer must satisfy the DC-power flow equations (2).

The OPA algorithm solves $(LP(\hat{x}, \hat{d}, \mathcal{O}))$ to obtain real power flows f^* and load shed t^*. Next, lines are added to the failed set $\mathcal{O}$ based on the outcome of a random event and the loading of the line. This random event, encapsulated in the random variable $\boldsymbol{\omega}$, mimics the *evolution* of the cascade. In OPA, additional line failures $\boldsymbol{\omega}$ are modeled as Bernoulli random variables, whose parameter $p = p(|f^*_{ij}|, \overline{f}_{ij} + \hat{x}_{ij})$ is a function of the the optimal line flow f^*_{ij} and installed capacity $\overline{f}_{ij} + \hat{x}_{ij}$. Figure 1 shows a typical Bernoulli line failure function $p_{\rho,\kappa}(f,u)$. The function $p_{\rho,\kappa}(f,u)$ captures the behavior that lines whose flow f is

Figure 1: Bernoulli Line Failure Function



near or at the capacity u are more likely to fail. The function $p_{\rho,\kappa}()$ has two parameters, the value ρ, which represents the probability that the line will fail if $f = u$; and κ, which represents the amount under u that f must be in order to have a positive probability of failing. In [2], the authors suggest using values $\rho = 0.5, \kappa = 0$. We primarily used this parametrization in our experiments.

For each line $(i, j) \in \mathcal{L}$, Bernoulli random variables ω_{ij} with parameter $p_{\rho,\kappa}(|f^*_{ij}|, \overline{f}_{ij}+\hat{x}_{ij})$ are drawn and if $\omega_{ij} = 1$, then line (i, j) is added to $\mathcal{O}$. If lines are added to $\mathcal{O}$, then $(LP(\hat{x}, \hat{d}, \mathcal{O}))$ is solved again. If no lines are added to $\mathcal{O}$, the final value of $\sum_{i\in\mathcal{D}} t^*_i$ gives an estimate of the load shed $F(\hat{x}, \hat{d}, \hat{\xi}, \boldsymbol{\omega})$. A complete description of the inner loop of the OPA algorithm is given in Figure 2.

In order to estimate the *distribution* of the load shed, The algorithm in Figure 2 can be repeated for a number of trials T. Figure 3 shows a typical load-shed distribution obtained from running $T = 512$ trials of the OPA algorithm (2) on the well-known IEEE118 benchmark instance. In this simulation, we used an initial exogenous triggering event ξ_1 of the failure of lines indexed [12, 14, 34, and 111], and we set $\hat{x} = 0$, which means that we did not add additional capacity to the lines. The IEEE118 benchmark instance has a nominal demand of $\sum_{i\in\mathcal{D}} \hat{d}_i = 3668$ MW. The simulation experiment was run using a Bernoulli line failure function $p_{\rho,\kappa}(\cdot)$ with parameters $\rho = 0.5$ and $\kappa = 0$. To create Figure 3, Algorithm 2 was called $T = 512$ times, with each trial producing an estimate z_i of the load shed. In total, more than 2000 linear programs were solved to produce this empirical distribution. The sample average of the 512 trials was $\sum_{i=1}^{T} z_i/512 = 130.845$, so we would estimate $\mathbb{E}_{\mathbb{P}_{\boldsymbol{\omega}}}[F(0, \hat{d}, \xi_1, \boldsymbol{\omega})] = 130.845$. The sample standard deviation of the 512 trials was 85.124, implying a standard error in our estimate of the mean load shed of $85.124/\sqrt{512} = 3.762$. Thus, using this experiment, we could construct an approximate 95% confidence interval of the mean load shed under the initial contingency ξ_1 to be $[123.472, 138.219]$.

Obtaining a more precise estimate of the expected load shed is possible by increasing the number of trials T. Since we are estimating the mean value using a standard Monte-

Figure 2: One Trial of the OPA Simulation

procedure OPA($\hat{x}, \hat{d}, \xi$)
  Solve ($LP(\hat{x}, \hat{d}, \emptyset)$) for base load shed $z_0 = \sum_{i \in \mathcal{D}} t_i^*$.
  Let $\mathcal{O} = \{(i,j) \in \mathcal{L} \mid \hat{\xi}_{ij} = 1\}$
  Stage $s \leftarrow 1$; DONE $\leftarrow$ FALSE
  while Not DONE **do**
    Solve ($LP(\hat{x}, \hat{d}, \mathcal{O})$) for flows f^* and load shed t^*.
    $\forall (i,j) \in \mathcal{L}, \omega_{ij} \sim$ BERNOULLI($p_\kappa(|f_{ij}^*|, \overline{f}_{ij} + \hat{x}_{ij})$)
    $\mathcal{N} := \{(i,j) \in \mathcal{L} \mid \omega_{ij} = 1\}$
    if $\mathcal{N} = \emptyset$ **then**
      $z \leftarrow \sum_{i \in \mathcal{D}} t_i^*$
      DONE $\leftarrow$ TRUE
    else
      $\mathcal{O} \leftarrow \mathcal{O} \cup \mathcal{N}$
      $s \leftarrow s + 1$
  Return $z - z_0$ ▷ the Load Shed $F(\hat{x}, \hat{d}, \hat{\xi}, \omega)$ is $z - z_0$

Carlo method, the error in our estimate reduces at a rate of $\sigma/\sqrt{T}$ where σ^2 is the variance of $F(x, \hat{d}, \hat{\xi}, \boldsymbol{\omega})$. There is some historical evidence that the load-shed from blackouts may be best described by a power-law distribution [8], implying that the true variance σ^2 is quite large. Therefore, obtaining accurate estimates of load shed may require significant computational effort, warranting the use of high performance or high-throughput computing platforms.

While the goal in our work is focusing on minimizing the expected load shed, it is worth nothing that the required computation to accurately estimate different statistics of the load shed may require even more computational effort. Take, for example, the problem of estimating a quantile $\boldsymbol{q}_\eta = H^{-1}(\eta)$, where H^{-1} is the inverse of the distribution function of the load shed random variable $F(x, \mathbf{d}, \boldsymbol{\xi}, \boldsymbol{\omega})$. The most natural way to estimate $\boldsymbol{q}_\eta$ involves taking the η-quantile of the empirical distribution produced by repeated application of the OPA algorithm in Figure 2. Then, to get obtain a confidence interval around this estimate, *repeated* distributions must be produced, which would require significantly more computational effort than for the simple estimation of $\mathbb{E}_{\mathbb{P}_{\boldsymbol{\omega}}}[F(x, \hat{d}, \hat{\xi}, \boldsymbol{\omega})]$. Thus, to calculate and optimize more complicated statistics of the load-shed, we should definitely consider the use of large-scale computational resources provided in high-performance and high-throughput computing environments.

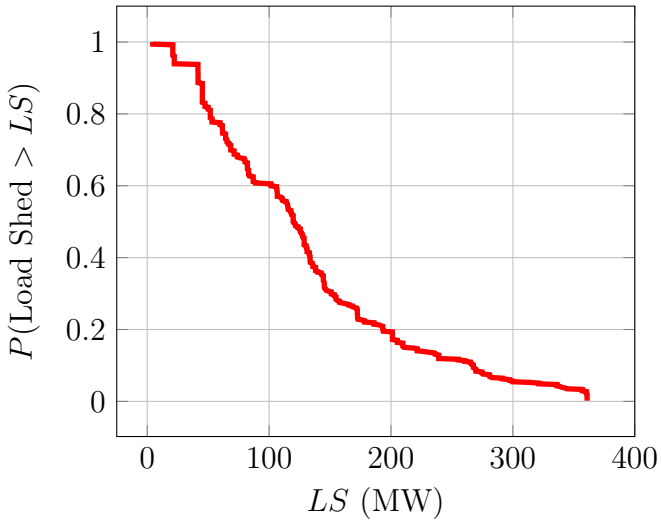


Figure 3: Sample Load Shed Distribution

2.2 Transmission Expansion

In our definition of the linear program that drives the OPA simulation, $(LP(\hat{x}, \hat{d}, \mathcal{O}))$, we have the ability to adjust the line capacity with input variables x. Given a budget constraint

$$X = \left\{ x \in \mathbb{R}^{|\mathcal{L}|_+} \mid \sum_{(i,j)\in\mathcal{L}} b_i x_i \leq K \right\},$$

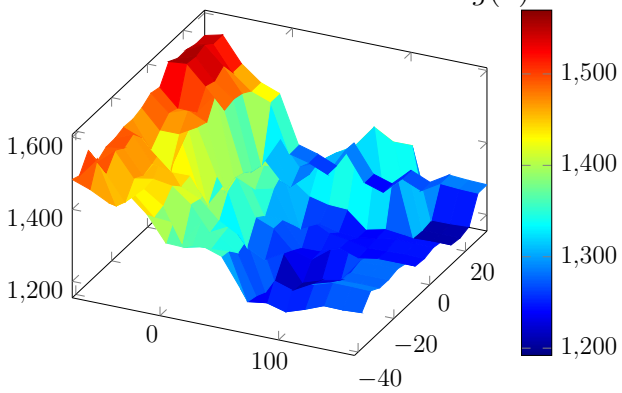
we use massive computational effort to optimize the function

$$\min_{x\in X} g(x) := \mathbb{E}_{\mathbb{P}_{\boldsymbol{\xi}}, \mathbb{P}_{\boldsymbol{\omega}}} F(x, \hat{d}, \boldsymbol{\xi}, \boldsymbol{\omega}). \tag{6}$$

That is, given a fixed demand $\hat{d}$, a distribution $\mathbb{P}_{\boldsymbol{\xi}}$ of exogenous initial triggering events $\boldsymbol{\xi}$, and using the OPA simulation as our surrogate for the distribution $\mathbb{P}_{\boldsymbol{\omega}}$, how should one best deploy additional capacity to minimize the expected value of load shed? This is an extremely difficult optimization problem, in part due to the complicated nature of the objective function $g(x)$. Our simulations have demonstrated that $g(x)$ is neither convex nor monotonic in x. Specifically, adding capacity to lines may result in *larger* expected load shed. Figure 4 demonstrates the effect of varying the capacity on two fixed lines in the IEEE118 network on the expected total load shed $g(x)$.

To produce Figure 4, the distribution $\mathbb{P}_{\boldsymbol{\xi}}$ was taken to be a very simple discrete distribution consisting of four equally-likely contingencies (given specifically in Section 5). The number of trials T used to estimate $g(x)$ was very large, so that the standard error of the estimate of the mean load-shed was less than 1 MW. This simple simulation experiment clearly demonstrates that the expected load shed function $g(x)$ is neither smooth nor monotonic in x, so we should employ an appropriate optimization method.

Figure 4: Estimate of two dimensions of $g(x)$ for IEEE118



2.3 Derivative Free Optimization

In order to approximately optimize our estimate of the function $g(x)$, we rely on a family of derivative-free optimization methods known as *pattern search* [3,9]. We employ a derivative-free method primarily because the derivative of the function $g(\cdot)$ is not available analytically or through automatic differentiation, and estimating the gradient via finite differences was deemed to be too costly from a computational perspective. Pattern search methods work by searching for improved function values along a collection of possible directions that sufficiently span the search space. Simultaneous searching in multiple directions has the ability to provide robustness against noise that may mislead gradient-based methods that use a single search direction. We combine our pattern-search method with a line search, so that numerous candidate points in each of the directions are evaluated. By using this simple line search, we can find step sizes that are more likely to escape local minima or "high frequency effects" of the objective function $g(x)$ that are displayed in Figure 4. To illustrate this point, we implemented the simplest pattern search method, *compass search*, where the search directions are simply the coordinate axes, on a reduced 2-dimensional subspace. We ran 4 different compass searches with initial trial steps of 1, 25, 50, and 75. The first portion of Figure 5 shows the value of the objective function for iterations of the compass search with these different step sizes. The second portion of Figure 5 plots the final point to which the algorithm converged for the four different initial step-sizes. The lowest objective value found in this subspace was obtained using an initial step size of 50. These figures demonstrate the significant impact that step size can have on solution value. Therefore, to avoid getting stuck in local minima, we employ a line search strategy wherein we evaluate $P = 30$ points along each search direction.

Initial computational experience with the OPA simulation revealed that there was a relatively small subset of lines $\mathcal{C} \subseteq \mathcal{L}$ that were most likely to fail from a cascading blackout. Restricting our search to the space induced by the lines $\mathcal{C}$ allows us to achieve a better solution for the same computational budget. At each major iteration k of our pattern search, we begin by finding a subset $\mathcal{C}_k$ of the lines over which to search for improvements

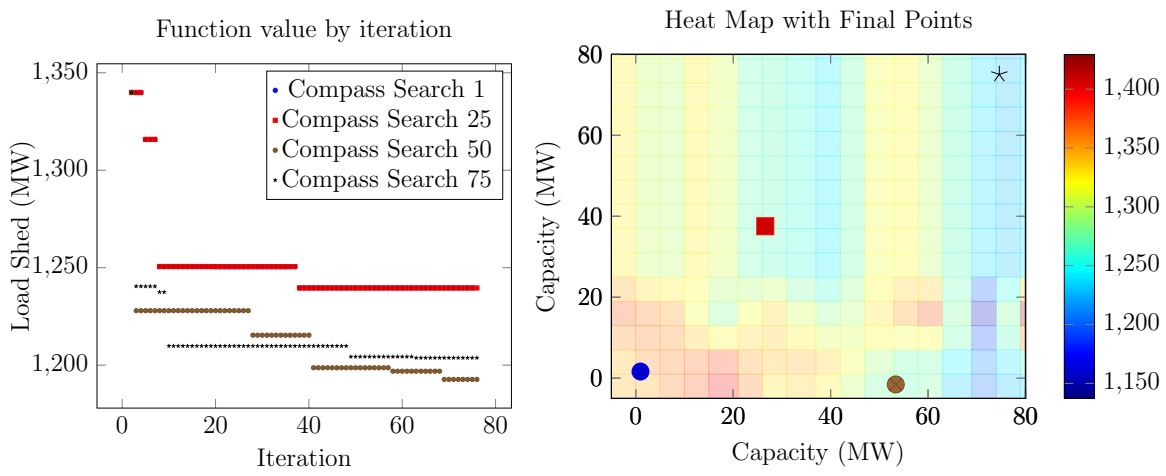


Figure 5: Resulting points from compass search with different initial step size values

in the objective function $g(x)$. The candidate set $\mathcal{C}_k$ is obtained by examining the OPA simulation results from the function evaluations involved in iteration $k-1$. Lines for which the frequency of failure was sufficiently high (e.g. $\geq 10\%$) over all the OPA cascade simulations were included in the candidate set for the next iteration.

At iteration k, there are $|\mathcal{C}_k|$ directions, one for each line in the candidate set. The direction d^ℓ we employ for line $\ell \in \mathcal{L}$ is to increase the capacity on line ℓ, while simultaneously (and equally) reducing the capacity of all other lines in $\mathcal{C}_k$. Specifically, for each $\ell \in \mathcal{C}_k$, the direction vector d^ℓ has components

$$
\begin{aligned}
d^\ell_a &= 1 \quad \text{for } a = \ell, \\
d^\ell_a &= -1/|\mathcal{C}_k| \quad \text{for } a \neq \ell, a \in \mathcal{C}_k, \\
d^\ell_a &= 0 \quad \text{for } a \notin \mathcal{C}_k.
\end{aligned}
$$

This choice of search direction ensures that point to be evaluated remains feasible. (In case the transmission capacity becomes negative, it is set to 0). For each search direction d^ℓ, we range the capacity addition for transmission element ℓ up to Δ, a step length tolerance that is modified by the pattern search algorithm. Pseudocode for our implementation of the pattern search algorithm is given in Figure 6.

There are natural places in Algorithm 6 to parallelize the computation. Specifically, there are many simultaneous trial points $p_{k\ell t}$ for which we are required to compute an estimate of $g(p_{k\ell t})$. Each of these estimated function evaluations may be completely independently. This type of naturally parallel computation, requiring very little inter-process communication is perfect for the loosely-coupled collection of computational resources provided by the HTCondor system.

Figure 6: Pattern Search

procedure PS($g : \mathbb{R}^n \to \mathbb{R}, k_{\max}, \Delta_{\text{tol}}$)
  $k \leftarrow 0$, $x_0 \in X$ Initial guess
  Evaluate $g_0 \leftarrow g(x_0)$
  while $k \leq k_{\max}$ and $\Delta \geq \Delta_{\text{tol}}$ **do**
    $k \leftarrow k + 1$
    Create $\mathcal{C}_k$: List of candidate lines
    for all $\ell \in \mathcal{C}_k$ **do**
      for $t \leftarrow 1, P$ **do**
        $\alpha_t \leftarrow \frac{t\Delta}{P}$
        $p_{klt} = x_{k-1} + td^{\ell}$
        Evaluate $g(p_{klt})$
    $g_k = \min_{\ell,t} g(p_{k\ell t})$, with $\ell^*, t^* \in \arg\min_{\ell,t} g(p_{k\ell t})$
    if $g_k < g_{k-1}$ **then**
      $x_k = p_{k\ell^* t^*}$
    else
      $g_k \leftarrow g_{k-1}$, $x_k \leftarrow x_{k-1}$
      $\Delta \leftarrow \frac{1}{2}\Delta$

3 HTCondor for Scenario Analysis and Optimization

In this section, we discuss the high-throughput computing software tools used to create and harness large federations of computing resources to solve our capacity expansion problem.

3.1 HTCondor

Our software framework relies on HTCondor. HTCondor is a job management system for compute-intensive jobs [13, 16]. HTCondor provides a job queueing mechanism, a scheduling policy for controlling jobs, and resource monitoring and management. HTCondor is especially designed to effectively handle computational tasks in a *high-throughput* manner. High throughput computing refers to the use of many computing resources over long periods of time to accomplish a computational task. This is in contrast to high-performance computing, wherein jobs require large computing power (and perhaps significant data sharing and inter-process communication) over a relatively shorter interval. The computational resources provided by HTCondor are shared in a manner wherein "owners" of the resources have priority. Additionally, users who have used a smaller percentage of the total resource in the recent past have higher priority on those resources. Jobs from users with a higher priority may preempt running jobs, and HTCondor ensures that jobs that are removed from a resource are started again on another free resource and ultimately completed. Sharing

resources in this manner creates an environment in which users have access to a significant pool of computational cycles, including cycles that may have otherwise gone unused. The disadvantage of running computational tasks in an HTCondor environment is that the application must be agile and flexible to the random nature of the available resources for a specific task.

The HTCondor software comes packaged with additional software tools that were useful to manage our computation. Specifically, we made use of the HTCondor DAGMan. DAGMan is a meta-scheduler that manages dependencies between jobs. These dependencies are represented as a directed acyclic graph (DAG). The nodes of the graph are the jobs (programs), and the edges identify the dependencies (or precedence relationship) on the jobs. The HTCondor DAGMan submits jobs to HTCondor in an order represented by a DAG and processes the results. We used the DAGMan to help control iterations of our algorithm so as to not overwhelm the shared computational resource.

3.2 Software Infrastructure

The software infrastructure for the parallel search running in the HTCondor environment consists of

- A master python program coordinating the computation on the submission node; and
- A shell script, compiled C-executable, and python script to be run on the executing nodes.

The master process is responsible for implementing the logic of the Pattern Search algorithm in Figure 6, while the shell script, compiled C-executable, and python scripts run on the executing nodes only perform function evaluations—the **Evaluate** steps of the algorithm in Figure 6. The evaluation of new trial points is only initiated after the entire current iteration is complete. A more sophisticated version of the optimization algorithm would work in an asynchronous fashion, starting new iterations before the previous iterations have completed, as suggested in [11, 12, 14]. We leave such algorithmic enhancements to future research. In fact, one purpose of this work is to demonstrate how simple it is to get a parallel algorithm running in the HTCondor environment.

On the submitting node, a master python process is run that queries the HTCondor system to determine if any HTCondor jobs associated with the Pattern Search are running. If no jobs are running, then the function evaluations from an iteration of the algorithm are complete. Then the python code performs the logic in Algorithm 6 to

1. Determine the new incumbent point x_k (if any);
2. Create the next set of candidate lines $\mathcal{C}_{k+1}$; and
3. Calculate the trial points $p_{k+1,\ell t}$ to be evaluated.

Candidate lines for the upcoming iteration are obtained by parsing the output files from the recently completed iteration and calculating the observed failure percentage of each line. Preparing the next trial points is accomplished by creating appropriate input files and the necessary directory and file structure required by the HTCondor DAGMan. As earlier described, HTCondor DAGman was used as a mechanism for throttling the job submissions. At a given iteration, we may have over 2000 trial points to be evaluated. DAGMan will process these jobs ensuring that the local HTCondor scheduler is not overwhelmed and that not too many jobs idle appear in the HTCondor queue. These two features ensure efficient execution of all jobs while keeping our HTCondor user-priority low. A lower user priority results in a larger share of resources at each iteration. For our case study computation in Section 5, we instructed DAGMan to run at most 500 jobs simultaneously. The most important feature of the computing infrastructure is that HTCondor (and DAGMan) will automatically ensure that *all* submit jobs are completed, even if the resources are claimed by higher-priority users.

On the worker execution side, a shell script is run that calls a compiled C-executable that runs a specified number of trials T of the OPA simulation. The input to the program consists of files specifying the instance: topology, line susceptances B_{ij}, line limits for the trial point $\overline{f}_{ij} + \hat{x}_{ij}$, the specification of the distribution function $p_{\rho,\kappa}()$ for failure information, and a random number seed. The use of a common random number stream can reduce the variance of estimate of the (difference) between function evaluations, as explain in Section 4. The C-executable was linked with the CPLEX (v12.6) software for solving the linear programs ($LP(\hat{x}, \hat{d}, \mathcal{O})$) in the OPA simulation.

The HTCondor overhead for starting a program is considerable. Thus, it is desirable that the grain size of the computational task is sufficiently large (on the order of a few minutes) in order to amortize the startup costs. For our larger-scale computations detailed in Section 5, the number of replications of the OPA algorithm in Figure 2 for a given trial point was calibrated so that the standard error of our estimate of the mean load shed was $< 1\%$. We found that $T = 15,000$ replications was sufficient. For example, for one trial point, the estimated mean load shed was 110.9, with a sample standard deviation of 108.7. For $T = 15,000$, the error in our estimate of the mean is $108.7/\sqrt{15000} = 0.8875$. The $T = 15,000$ trials typically required around 15 minutes of CPU time, so the estimation of the mean load shed for one trial point $p_{k\ell t}$ was a reasonable grain size for the computation.

On the evaluation machine, considerable output is produced by the executable, including a detailed description of the lines that failed at each stage of the simulation for each trial. These additional statistics could be used to optimize a different measure of load shed or by a more sophisticated optimization algorithm, but our simple pattern search method in Figure 6 requires only aggregated statistical information. Thus, after completion of the executable, the shell script runs a python script to summarize the necessary statistics about the computation and writes this to a small file, which HTCondor passes back to the submission node, signaling completion of the function evaluation $g(p_{k\ell t})$.

The folders created as part of the DAGMan submission process contain log files for each job and the summary output files computed by the post-execution python script. The output folders can be used to trace what has happened up to the current point in the algorithm in order to allow the python master process to continue off where it last was if it was restarted—giving our method a very natural checkpointing mechanism.

3.3 Computational Environment

The primary HTCondor cluster used for our computations consists of 341 cores, located in the Wisconsin Institute of Discovery at the University of Wisconsin-Madison. This HTCondor pool was connected via HTCondor flocking [6] to the primary HTCondor pool in the Center for High Throughput Computing (CHTC) at the University of Wisconsin-Madison. The CHTC provides computational resources for UW and affiliated researchers and, as of September, 2015, hosts an HTCondor Pool with 13,900 cores. Condor flocking is a mechanism by which jobs submit to run in one HTCondor pool are scheduled to run in a different pool. Thus, the total number of cores that were available for our computation was 14,241.

The majority of the machines in the Condor clusters at UW-Madison use the Scientific Linux (versions 5 and 6) distributions. Executables that perform a specified number of trials of the OPA simulation (Algorithm 2) were created for each platform. HTCondor DAGMan was configured to automatically transfer the appropriate executable (and additional necessary run-time libraries) depending on the Linux distribution of machine designated to run the job.

Our jobs typically have a very low memory footprint, so using the Condor scheduling/matchmaking mechanism [15], we could make requests for computing resources that did not necessarily have a large amount of RAM. This helped us obtain more resources for our computations.

4 Computational Enhancements

In this section, we briefly describe two simple computational techniques that increased the efficiency of our simulation and optimization procedure.

Hot-Starts Between iterations of Algorithm 2, the set of failed lines $\mathcal{O}$ changes. This has the effect of removing constraints (2) and fixing variables (5) on the linear program $(LP(\hat{x}, \hat{d}, \mathcal{O}))$. However, the solution to $(LP(\hat{x}, \hat{d}, \mathcal{O}))$ at one iteration remains *dual* feasible after changes to the set of failed lines $\mathcal{O}$. Therefore, it can be computationally advantageous to employ the *dual simplex method* to solve the sequence of linear programs in the OPA simulation, as subsequent linear programs can be warm started using the dual feasible basis from the previous solve. We ran a small experiment to measure the computational speedup

Table 1: Average # Iterations and CPU Time/LP Solve

	# It./LP	**Time/LP** (s)
No Warm Start	2970	0.666
Warm Start	192	0.139

obtained by hot-starting the dual simplex method. This experiment was run on a power system consisting of $|\mathcal{B}| \approx 2200$ buses and $|\mathcal{L}| \approx 2800$ lines.

To get a fine-grained empirical distribution of the load shed random variable $F(\hat{x}, \hat{d}, \hat{\xi}, \boldsymbol{\omega})$, we perform $T = 15000$ trials, which required the solution of 67132 linear programs. Table 1 summarizes the results of this small experiment. We see that on average, using the warm-start information provided by the previous (dual) feasible solution can give a speedup per LP solve of around a factor of 5. This speedup factor (of at least 5 or 10) was typical during our experiments.

Common Random Numbers The stochastic nature of the cascading process, and its simulation via the OPA process described in Figure 2, often leads to a large variance for the load-shed random variable $F(x, \hat{d}, \hat{\xi}, \boldsymbol{\omega})$. Variance reduction techniques are commonly used in the simulation and stochastic programming communities, and we employ the well-known variance reduction technique of *common random numbers* (CRN) [7, 10].

The Bernoulli random variables to simulate line failures in OPA are generated using the inverse transform method. Specifically, given power flows f^*_{ij} and installed capacities $\overline{f}_{ij} + \hat{x}_{ij}$ for each line $(i, j) \in \mathcal{L}$, we draw a uniform random variable U_{ij} between 0 and 1. If $U_{ij} \leq p_{\rho,\kappa}(|f^*_{ij}|, \overline{f}_{ij} + \hat{x}_{ij})$ for the Bernoulli line failure function $p_{\rho,\kappa}()$ shown in Figure 1, then the line (i, j) fails at that iteration. The implementation was done so that at each (major) iteration k of the pattern search method described in Figure 6, the *same* stream of random numbers U was used for each arc $(i, j) \in \mathcal{L}$. This has the effect of using the same realizations of the random variables $\boldsymbol{\omega} \sim \omega_1, \omega_2, \ldots, \omega_T$ for each trial $t = 1, 2, \ldots, T$ across the different function evaluations. Specifically, when comparing points x_1 and x_2, the difference in estimated function values between the two points for trial t is

$$Z_t(x_1, x_2) = F(x_1, \hat{d}, \hat{\xi}, \omega_t) - F(x_2, \hat{d}, \hat{\xi}, \omega_t).$$

The estimate of the expected value of the difference between the two function values is then

$$T^{-1} \sum_{t=1}^{T} Z_t(x_1, x_2).$$

The variance of this estimate is

$$\mathsf{Var}\left[F(x_1, \hat{d}, \hat{\xi}, \omega_t)\right] + \mathsf{Var}\left[F(x_2, \hat{d}, \hat{\xi}, \omega_t)\right] - 2\mathsf{CoVar}\left[F(x_1, \hat{d}, \hat{\xi}, \omega_t), F(x_2, \hat{d}, \hat{\xi}, \omega_t)\right] \quad (7)$$

Table 2: Illustrative Estimates of Function Difference. $T = 750$ trials

	Estimate $F(x_1) - F(x_0)$	Standard Error	Estimate $F(x_2) - F(x_0)$	Standard Error
No CRN	-66.69	11.24	-57.12	11.99
CRN	-67.90	10.75	-55.48	1.15

According to Equation 7, in order to reduce the variance, we need

$$\mathsf{CoVar}\left[F(x_1, \hat{d}, \hat{\xi}, \omega_t), F(x_2, \hat{d}, \hat{\xi}, \omega_t)\right] > 0$$

By using a common random number stream for the failure probabilities of the lines, we increase the covariance between the load shed values for the two different configurations. This can have a very positive impact in reducing the variance of the estimate of the difference of function values between two points x_1, x_2, especially if x_1, x_2 are close together. Thus, for a fixed amount of computation, we are able to state with high certainty whether we are seeing an actual function decrease, or if the perceived decrease is merely an artifact of the noise from the simulation.

In our experiments, we observed cases where using common random numbers was quite beneficial, but this was not always the case. For example, consider Table 2. Here we show for two different trial points x_1 and x_2, the estimated function value difference between the trial point and a point x_0 as well as the standard error of the estimate. For the point x_1, we see no benefit to using CRN, while for x_2, we obtain a much higher-quality estimate by using CRN.

5 Case Study

In this section, we report on a case study demonstrating the utility of high throughput computing for performing a simulation-based optimization to minimize expected load-shed from a cascading power failure. We use the IEEE 118 Bus Test Case that represents a portion of the American Electric Power System (in the Midwestern US) as of December, 1962. This test system has 118 buses, 186 branches, and 32 generating nodes with positive real power injection. The largest load is requires 277 MW and the largest generator has a capacity of 607 MW. The total demand of the system is 3,668 MW and the total available generation is 4,951 MW, giving a reserve margin of 1,283 MW, or 35% of load.

The 118 bus test case did not have initial line limits $\overline{f}_{ij}$, so we created line limits that were calibrated to be $N-1$ robust to failures in the following manner. We started by solving the nominal (no failure) case with no line limits. We then created a problem wherein line limits were set to be 10% larger than the observed power flow, and we examined the power flow under all possible failures of a single component (line) of the system. For each line

Table 3: Triggering Contingencies $\boldsymbol{\xi}$ in Case Study

Scenario	**Lines That Failed**
1	12, 14, 34, 11
2	12, 2, 21
3	82, 5, 25, 34
4	13, 24

that was at its limit in the DC-power flow for one of the contingencies, we increased its capacity by 10%. The process of checking all possible single failures was iterated until no line limit constraints were active in any of the possible contingencies. After performing this process, our designed line limits had a total capacity of 34,859 MW, with the largest line limit being line 12 with 2,430 MW of transmission capacity. Many of the lines had limits under 50MW.

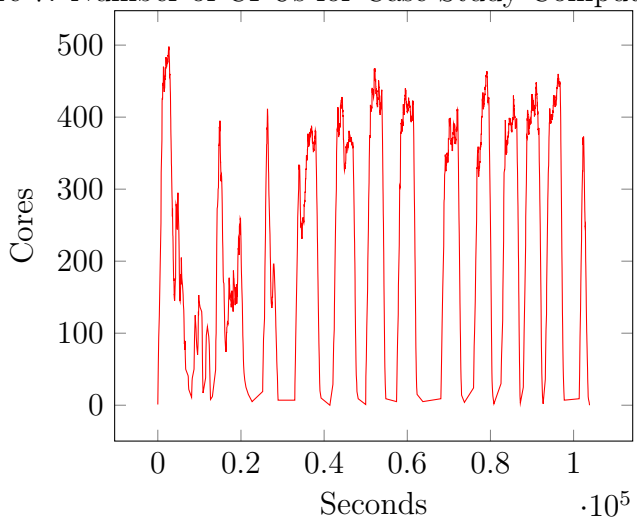
We next created a distribution of the random variable $\boldsymbol{\xi}$, the exogenous initial failure(s) that would trigger a cascading load-shed event in the system. Possible trigger events were evaluated by randomly removing between 2 and 4 lines from the network and running the OPA simulation described in Figure 2. Based on this analysis, we selected our random variable $\boldsymbol{\xi}$ to have four equally likely outcomes, listed in Table 3.

Our transmission expansion problem (6) is to allocate a budget of 400MW additional capacity to minimize the expected load shed. This 400 MW of transmission capacity is only 1.1% of the total installed capacity (34859MW) on the lines in the system.

In this experiment, we look at adding an incremental capacity to the existing transmission network versus creating new lines. In practice, there may be limitations on increasing capacity of transmission elements. However, this case study is merely a prototype designed to demonstrate the combination of simulation, optimization, and high-throughput computing for solving an important power system problem. Additionally, transmission capacity expansion may be possible by improving line clearance or rebuilding sections of the transmission lines to increase the nominal line rating. The extension of our simulation and optimization framework to cases considering the addition of new lines or cases where capacity may only be increased by certain fixed amounts can be easily accomplished through an appropriate modification of the Pattern Search method described in Figure 6.

One striking outcome of our experiment is the amount of computational power obtained by running in the HTCondor computing environment described in Section 3. The total CPU time used for our case study computation was 15,350,868 secs, or 177.67 CPU days. This was accomplished in 103,833 (wall clock) seconds, or 28.8 hours. Thus, there were on average 147.8 cores participating in the computation over the 28.8 hours. Figure 7 shows the number of machines participating in the evaluation of the objective function $g(x)$ over the course of the computation. The picture clearly shows the synchronization points of the algorithm. In all, $k = 13$ major iterations were performed.

Figure 7: Number of CPUs for Case Study Computation



A second striking outcome of our prototype case study is that we were able to make a large improvement in the expected load shed for a given modest budget of 400 MW of transmission capacity. Without adding any capacity, we estimate the expected load shed (in MW) to be

$$g(0) = 464.7 \pm 1.65.$$

After allocating the additional 400 MW of line capacity as suggested by the optimization method, the new system had an estimated expected load shed of

$$g(x^*) = 110.9 \pm 0.89.$$

The new system reduced the expected load shed by around 353 MW, or 76% of the original mean load shed value. The 400MW of additional transmission capacity was allocated to 29 lines, with the highest increase in capacity of 89 MW going to line #115. The majority of the capacity increases where small—less than 20 MW.

6 Conclusion

In this paper we demonstrated how to instrument a simulation-based optimization procedure to run on the HTCondor computing system. The simulation estimates the load-shed distribution of a cascading blackout caused by an exogenous random triggering event. The simulation was combined with a pattern-search optimization algorithm to effectively allocate a fixed amount of transmission capacity to minimize the expected load shed. Computational techniques such as warm-starting and common random numbers were important implementation details for an effective algorithm. Using the HTCondor DAGMan tool, supported with some simple scripts allowed us to implement the prototype algorithm very

easily. We hope our work demonstrates that not only is High Performance Computing an important tool for building more resilient and efficient power grids, but also High Throughput Computing can be an important ingredient in computational approaches to tackling difficult planning and operational problems for next-generation power grids.

Acknowledgment

This research was supported in part by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research, Applied Mathematics program under contract number DE-AC02-06CH11357. This research was performed using the compute resources and assistance of the UW-Madison Center For High Throughput Computing (CHTC) in the Department of Computer Sciences. The CHTC is supported by UW-Madison, the Advanced Computing Initiative, the Wisconsin Alumni Research Foundation, the Wisconsin Institutes for Discovery, and the National Science Foundation, and is an active member of the Open Science Grid, which is supported by the National Science Foundation and the U.S. Department of Energy's Office of Science.

References

[1] B. A. Carreras, V. E. Lynch, I. Dobson, and D. E. Newman. Critical points and transitions in an electric power transmission model for cascading failure blackouts. *Chaos: An Interdisciplinary Journal of Nonlinear Science*, 12(4):985–994, 2002.

[2] B. A. Carreras, V.E. Lynch, I. Dobson, and D. E. Newman. Complex dynamics of blackouts in power transmission systems. *Chaos*, 14(3):643,652, September 2004.

[3] Andrew R. Conn, Katya Scheinberg, and Luis N. Vicente. *Introduction to Derivative-Free Optimization*. SIAM and MPS, Philadelphia, PA, 2009.

[4] I. Dobson, B. A. Carreras, V. E. Lynch, and D. E. Newman. An initial model for complex dynamics in electric power system blackouts. In *34th Hawaii International Conference on System Science*, pages 705,709, January 2001.

[5] Ian Dobson, Benjamin A. Carreras, Vickie E. Lynch, and David E. Newman. Complex systems analysis of series of blackouts: Cascading failure, critical points, and self-organization. *Chaos: An Interdisciplinary Journal of Nonlinear Science*, 17(2):–, 2007.

[6] D.H.J. Epema, M. Livny, R. van Dantzig, X. Evers, and J. Pruyne. A worldwide flock of Condors: Load sharing among workstation clusters. *Future Generation Computer Systems*, 12:53–65, 1996.

[7] M. B. Freimer, D. J. Thomas, and J. T. Linderoth. The impact of sampling methods on bias and variance in stochastic linear programs. *Computational Optimization and Applications*, 51:51–75, 2012.

[8] Paul Hines, Jay Apt, and Sarosh Talukday. Large blackouts in North America: Historical trends and policy implications. *Energy Policy*, 37:5249–5259, 2009.

[9] T. Kolda, R. Lewis, and V. Torczon. Optimization by direct search: New perspectives on some classical and modern methods. *SIAM Review*, 45(3):385–482, 2003.

[10] Averill M. Law. *Simulation Modeling and Analysis*. McGraw-Hill, New York, NY, 2007.

[11] J. T. Linderoth and S. J. Wright. Implementing a decomposition algorithm for stochastic programming on a computational grid. *Computational Optimization and Applications*, 24:207–250, 2003. Special Issue on Stochastic Programming.

[12] J. T. Linderoth and S. J. Wright. Computational grids for stochastic programming. In S. Wallace and W. Ziemba, editors, *Applications of Stochastic Programming*, SIAM Mathematical Series on Optimization, pages 61–77. SIAM, 2005.

[13] M. J. Litzkow, M. Livny, and M. W. Mutka. Condor—A hunter of idle workstations. In *Proceedings of the 8th International Conference on Distributed Computing Systems*, pages 104–111, 1988.

[14] F. Niu, B. Recht, C. Re, and S. J. Wright. HOGWILD!: A lock-free approach to parallelizing stochastic gradient descent. In *Advances in Neural Information Processing Systems (NIPS)*, pages 693–701, 2011.

[15] R. Raman, M. Livny, and M. Solomon. Matchmaking: Distributed resource management for high throughput computing. In *Proceedings of the Seventh IEEE International Symposium on High Performance Distributed Computing (HPDC7)*, Chicago, IL, 1998.

[16] Douglas Thain, Todd Tannenbaum, and Miron Livny. Distributed computing in practice: the condor experience. *Concurrency - Practice and Experience*, 17(2-4):323–356, 2005.