Studying Sabotage-Tolerance Mechanisms through Web-based Parallel Parametric Analysis and Monte Carlo Simulation

Luis F. G. Sarmenta[*]
Massachusetts Institute of Technology
Cambridge, MA 02139

Abstract

In this paper, we show how we have been able to use parallel parametric analysis and Monte Carlo simulations, running on a Java applet-based volunteer computing system, Bayanihan, to develop and study new mechanisms for addressing the problem of sabotage *by malicious volunteers in volunteer computing and Internet computing systems. We begin by describing the general-purpose framework we have developed for writing various parametric analysis and Monte Carlo applications on Bayanihan. Then, we give an overview of the sabotage-tolerance mechanisms we have developed, including voting, spot-checking, and credibility-based fault-tolerance, and present some examples of results we have gotten using our simulator. Finally, we present performance results that show how parallelizing our simulator has made our research possible by enabling us to do parametric analysis in much less time than possible with a single computer.*

1 Introduction

A very promising class of Internet computing systems are *volunteer computing systems* [11], which seek to make it possible to build very large parallel networks very quickly by enabling casual users on the Internet to share their computers' idle time through easy-to-use software. Probably the most popular examples of these are distributed.net, which gained fame in 1997 by solving the RSA RC5-56 challenge using thousands of volunteers' personal computers around the world [2], and SETI@home, which is currently employing hundreds of thousands of volunteer machines to search massive amounts of radio telescope data for signs of extraterrestrial intelligence [12]. A number of academic projects have also ventured to study and develop volunteer computing systems, including some, like our own Bayanihan [9, 11], that promote web-based systems using Java [1, 3]. Even the commercial sector has joined the fray, with a number of new startup companies seeking to put volunteer computing systems to commercial use, and pay volunteers for their computer time [4, 6, 7, 8].

Such volunteer computing systems, however, present a new and largely unstudied problem – if we allow anyone to join a computation, how do we prevent *malicious volunteers* from invalidating the computation by submitting bad results? Traditional *fault-tolerance* techniques that work well against *random* faults, such as using parity and checksum schemes, will not be effective in this case because they cannot protect against *intentional* attacks by malicious volunteers – or *saboteurs* – who can disassemble the code, and figure out how to produce valid checksums for bad data. Thus, there is a need for new *sabotage-tolerance* mechanisms that work in the presence of malicious saboteurs without depending on checksums or cryptographic techniques.

In earlier work [9], we presented preliminary experiments exploring the use of traditional techniques such as *voting* as well as a new technique called *spot-checking*. In these experiments, we ran a real application (Mandelbrot set image rendering) and measured error rates with different sabotage-tolerance mechanisms and different numbers of good workers and saboteurs. Although these experiments produced very interesting and promising results, they were limited by the number of worker machines we could use in a controlled environment (in this case, less than 20 machines). They were also very time-consuming since they required doing a real Mandelbrot rendering computation. Thus, we could not extrapolate our results to more general and more realistic cases where there may be hundreds or thousands of worker machines and thousands of work pieces to compute.

In this paper, we describe how we have gotten around this problem by using Bayanihan to run parallel parametric analysis and Monte Carlo simulations on a web-based volunteer computing system using Java applets. We begin by describing the general-purpose framework we have developed for writing various parametric analysis and

[*]The author is now at the Department of Information Systems and Computer Science, Loyola Schools, Ateneo de Manila University, P.O. Box 154, Manila, Philippines. Email: `lfgs@admu.edu.ph`, Web site: `http://www.cag.lcs.mit.edu/bayanihan/`

Monte Carlo applications on Bayanihan. Then, we give an overview of the sabotage-tolerance mechanisms we have developed, and present some examples of results we have gotten using our simulator. Finally, we present performance results showing how parallelizing our simulator has given us speedups that has made our research possible by allowing us to do parametric analysis in much less time than it would have taken with a single computer.

2 The Bayanihan Framework for Parametric Analysis and Monte Carlo Simulation

Parametric analysis applications are ones wherein we run a large number of independent computationally intensive *sequential* computations, each with a slightly different set of input parameters. They are among the most promising classes of applications for volunteer computing today not only because they are typically coarse-grain and easily parallelizable, but even more so because they form a natural extension of the simulators that most programmers write today. Since most programmers today do not have access to parallel machines, many simulators today are still written as sequential programs. Programmers use these by feeding them different parameter combinations and taking note of the results for each combination. Parallel parametric analysis offers these programmers a straightforward way to save time by enabling them to run their programs on many machines with many different parameter combinations at the same time. Since no change in the sequential code is needed, a programmer can start benefitting from volunteer computing right away without having to learn complex parallel programming techniques.

A particularly promising class of such applications are *Monte Carlo simulations*. These simulations compute certain properties of a hard-to-analyze system by simulating its behavior given many different *random* sequences of events, and analyzing the results. Note that Monte Carlo simulations are simply parametric analysis computations where the parameter that is varied is the random number generator seed that generates the random events within the sequential simulator. Thus, they can be run on any system that supports parametric analysis, and are particularly appropriate for volunteer computing systems.

2.1 Writing Parametric Analysis Applications

Bayanihan provides a simple but flexible framework for writing parametric analysis applications. To write an application, one needs to define four main types of objects: the work, result, configuration, and program objects. The *work* object contains the sequential code for doing one run of the computation with a specific set of parameters. It takes a *configuration* object representing the parameters, and returns a *result* object representing the output of the computation for that particular configuration. The *program* object is responsible for creating a list of parameter configurations that the programmer wants to try, and then adding work objects with each of these configurations to the work pool so that they can be processed in parallel by the volunteer workers. After the parallel step, control returns to the program object, which then processes the results that have been collected, and writes them into a file, if desired. After processing these results, the program can proceed to try more configurations if desired, possibly depending on the results of the previous batch.

2.2 Writing Monte Carlo Applications

In addition to the basic framework for parametric analysis, we also provide a framework for Monte Carlo simulations. This framework provides the following features:

- **Random number generator classes.** These classes, adapted from the Colt project [5], provide random number generators that have better statistical properties and are more well-suited to Monte Carlo simulations than the built-in Java `Random` class. One of the classes taken from Colt is a random seed generator that can be used to generate sequences of non-correlated seeds to be given as initial parameters to each work object. This is crucial in *parallel* Monte Carlo simulators since these simulators require not only that the random number generators within each work object have good pseudo-random properties, but also that the random numbers generated in each work object are independent from those in other objects.

- **Statistics summary class.** This class, `DStat`, has an `addSample()` method that takes samples in the form of a `double` or another `DStat` object, and efficiently computes running values of statistics such as the mean, standard deviation, and maximum and minimum values as each sample is added. This makes collecting and analyzing the results from each work object simple. The standard deviation statistic produced by `DStat` can also be used to measure the precision of the results so far, and determine whether the results are good enough or more samples need to be taken.

- **Parallel methods.** Monte Carlo programs can make use of the `doBatch()` method, which takes a work class object and a configuration object, and allows us to run many different instances of the work class in parallel (using the same parameter configuration but with different random number seeds), and collect the results through a *collector* object that extracts information from result objects generated by work classes and places them in corresponding `DStat` objects.

The sample code in the appendix shows how some of these features are used.

3 Sabotage-Tolerance Mechanisms

By enabling us to observe the performance of different mechanisms under different conditions represented by different parameter configurations, the Bayanihan framework has allowed us to develop and study different sabotage-tolerance mechanisms. We give an overview of these mechanisms in this section, and discuss how we have simulated them in the next section.

3.1 Model and Parameters

Our mechanisms assume a *work pool based master-worker* model of computation. Here, a *computation* is divided into a sequence of *batches*, each of which consists of N independent *work objects*. At the start of each batch, these work objects are placed in a *work pool* by the *master* node, and are then distributed to P different *worker* nodes who execute them in parallel and return their *results* to the master. When the master has collected the results for all the work objects, it generates the next batch of work objects and repeats the whole process until the computation is done.

To simulate sabotage, we assume that up to a certain *faulty fraction* f of the P workers are *saboteurs*. Each saboteur is assumed to be a Bernoulli process with a probability s of submitting a bad result, known as the *sabotage rate*. Without the use of sabotage-tolerance mechanisms, these bad results eventually get accepted at the end of each batch, and become *errors*. The average fraction of final accepted results that are errors is defined as the *error rate* (ε). This fraction is also equal to the probability of each *individual* final result being bad.

The goal of our sabotage-tolerance mechanisms, therefore, is to reduce the error rate to an acceptably small value. For some applications, a relatively high error rate would suffice. In image rendering, for example, a few scattered erroneous pixels would not be noticeable to the human eye. Other applications, on the other hand, cannot tolerate even a single error in a batch. In these cases, the target error rate must be made very small in order to make the probability of getting *any* error at all acceptably small. Fortunately, as we will show, we can shrink the error rate exponentially with only a small linear increase in redundancy.

3.2 Mechanisms

Some of the sabotage-tolerance mechanisms we have developed and studied include the following (for a more detailed description and discussion, please see [10] and [11]):

Voting. Here, we wait until we receive at least m *matching* results for the same work object. The error rate in this case is *exponential* in form, and can be shown to be roughly $\varepsilon_{\mathrm{majv}} \approx (cf)^m$ where c is between 1.7 and 4. Voting is good when f is small, but is very inefficient for relatively large values of f, requiring a large value of m for even modest error reductions. Also, it requires that we do *all* the work objects at least twice, and so has a minimum redundancy and slowdown of 2.

Spot-checking. Here, we do not redo all work objects, but instead randomly check workers' results with a probability q known as the *spot-check rate*. If a worker is caught, then we *backtrack* through all its submitted results, invalidating them and forcing them to be redone. We also remove the worker from the system, and, if possible, *blacklist* it such that it cannot submit further results anymore. Spot-checking is more efficient than voting and only slows down the computation $1/(1-q)$ times instead of m times as in voting. If we have blacklisting, then the maximum average error rate (which occurs if a saboteur chooses some optimal sabotage rate s between 0 and 1) can be shown to be bounded by roughly $(1/qne)(f/(1-f))$, where n is the amount of work given to a worker in a batch (i.e., N/P plus spot-checks and other redundancies), and e is the natural logarithmic base. If blacklisting cannot be enforced, then a saboteur can maximize damage by only doing $l < n$ works and then leaving and rejoining under a new identity. In this case, the average error rate is bounded by roughly f/ql. In either case, note that the error rate is reduced by a factor that is *linear* in the amount of work done by a worker, and thus linear in time. If our target error rates are relatively large (e.g., around 1%), then spot-checking performs better than voting, especially for large values of f (e.g., greater than around 5%).

Voting and spot-checking combined. Although useful alone, voting and spot-checking can also be combined. If blacklisting is enforced, such a hybrid approach effectively takes the linearly-reduced error rate due to spot-checking and then exponentially reduces it by voting. This achieves much lower error rates than either voting or spot-checking alone for the same slowdown.

Credibility-based fault-tolerance. In this new and highly generalizable technique, the key idea is to compute the *credibility* of each tentative result as the conditional probability that the result is correct, based on voting (i.e., the more workers agreeing on a result, the higher its credibility), spot-checking (i.e., the higher the number of spot-checks passed by the workers who produced these results, the higher the credibility of the workers and the results), and other factors (e.g., human knowledge that some worker machines are more trustworthy than others). While the credibility of a result is below a certain *credibility threshold* ϑ, we continue to have it recomputed by other workers, and continue to spot-check workers. When the credibility threshold is reached (which can happen either because we gather enough matching results, or the workers pass enough spot-checks, or both), then we accept the result as final. By waiting for the threshold to be reached in this way, we guarantee that on average, the error rate will not exceed $1-\vartheta$. At the same time, we allow the system to automatically combine voting and spot-checking, and efficiently trade-off redundancy for more reliability, using only as much redundancy as necessary to reach the threshold. The result is mathematically guaranteeable correctness with "minimal" (i.e., given what we can know and control) slowdown.

4 Simulating the Mechanisms

4.1 The Simulator

To study all these mechanisms, we have developed a sabotage-tolerance simulator with Bayanihan that is both a parametric and Monte Carlo simulator. As a Monte Carlo simulator, it computes the average expected error rate of a volunteer computing system given a sabotage-tolerance mechanism (e.g., voting, spot-checking, or credibility-based fault-tolerance) and specific values for the different parameters (e.g., the faulty fraction f, sabotage rate s, credibility threshold ϑ, etc.), by running multiple instances of the simulator with the same parameters but different random number seeds. At the same time, it allows us to do parametric studies by varying the different parameters and observing the effect on the error rate.

In the plots in Sect. 4.2, for example, each point represents the result of 100 Monte Carlo simulations run in parallel using the same combination of parameters, and gives us the expected performance (i.e., error rate and slowdown) of the system for a particular combination of f, s, and ϑ, averaged over these simulations. At the same time, the plots themselves represent the result of running *many* such *sets* of parallel Monte Carlo simulations, and show how the error rate and slowdown vary with these different parameters.

The appendix shows sample code for a sabotage-tolerance experiment. Here, each call to `doBatch()` in `doCredScan()` runs a parallel Monte Carlo simulation using a particular work class and configuration. After the call to `doBatch()` the collector object contains statistics such as the error rate and slowdown, which we use to plot one point in the plots in Sect. 4.2. To get the whole plot, we run many such parallel Monte Carlo simulations with different parameters, and record the results in a file.

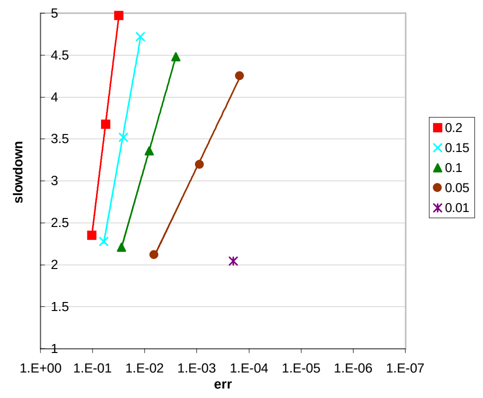

Figure 1. Majority voting. Slowdown (relative to a system without sabotage-tolerance) vs. maximum final error rate at various values of f and $m = \{2, 3, 4\}$.

4.2 Results

Figures 1 to 3 show the experimental results we get from running our Monte Carlo simulator. For our experiments, we ran 100 runs of simulated computation per point, each consisting of a sequence of 10 batches of $N = 10000$ work objects each, done by $P = 200$ workers. These numbers were chosen to be small enough to be simulatable in a reasonable amount of time, but large enough to provide good precision (i.e., the smallest measurable error rate is 1×10^{-7}) and to prevent blacklisting from killing all the saboteurs too early. In addition, the work-per-worker ratio, $N/P = 50$, was chosen to be large enough to show the effects of spot-checking, while still being representative of potential real applications. Also, having the computation go through 10 batches allows us to see the benefits of letting good workers gain higher credibility over time.

Figure 1 plots the resulting slowdown and error rate from majority voting given different values of the initial faulty fraction f (assuming a sabotage rate of 1). As shown, when f is large, majority voting requires a lot of redundancy to achieve even relatively large error rates. Extending the line for $f = 0.2$ theoretically, we find that it would take a slowdown of more than 32 to achieve a final error rate of 1×10^{-6}. Note, however, that the slope becomes less steep as f becomes smaller. (The other points for $f = 0.01$ resulted in no errors in this experiment, and are not shown.)

Figure 2 shows the results of using credibility-based fault-tolerance with voting and spot-checking with blacklisting. Here, each group of points corresponding to a credibility level is divided into three curves corresponding to $f = 0.2$, 0.1 and 0.05, respectively. This plot shows that, as guaranteed, the average error rate does not go above $1-\vartheta$, regardless of s and f.

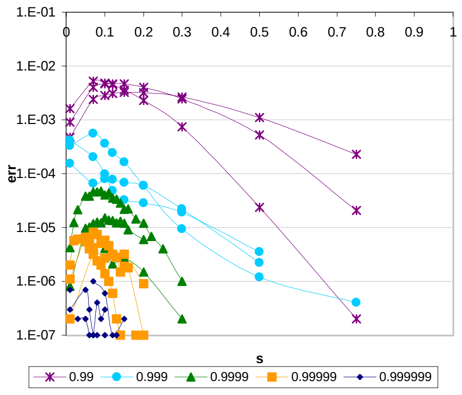

Figure 2. Credibility-based voting with spot-checking and blacklisting. Error rate vs. s at $f = \{0.2, 0.1, 0.05\}$.

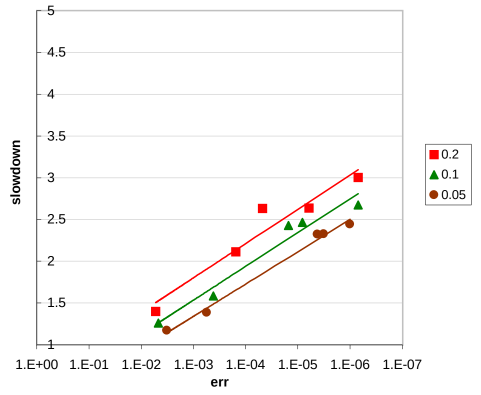

Figure 3. Credibility-based voting with spot-checking and blacklisting. Slowdown vs. maximum final error rate at $\vartheta = 0.99, \ldots, 0.99999$ at various values of f.

Figure 3 shows the automatic trade-off between redundancy and correctness. Here, we plot the slowdown incurred in achieving the maximum error rate for each combination of f and ϑ. Note how the slopes of the lines here are much better than those in simple majority voting. For example, whereas majority voting would have required a slowdown of more than 32 to achieve an error rate of 1×10^{-6} for $f = 0.2$, here we only need around 3. Also note that in some cases, spot-checking can be enough to reduce f down to the threshold $1 - \vartheta$ without requiring voting, as shown by the points with slowdown less than 2.

Several other results are presented in [10] and [11]. In these, we show results from other variations of the techniques presented here, such as spot-checking without blacklisting, and using voting itself as a spot-checking mechanism. Using the latter technique, we show that we can achieve an average error rate of less than 1×10^{6} at $f = 0.2$

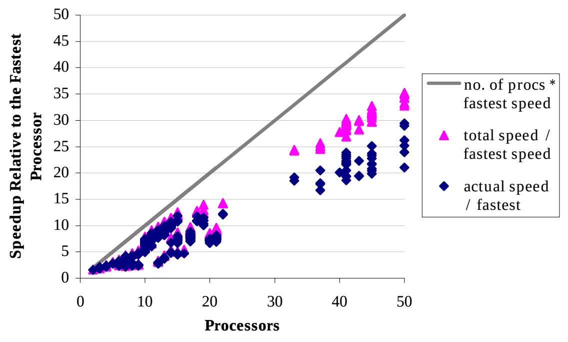

Figure 4. Speedup of the simulator relative to the fastest machine.

with a slowdown just little over a 2.5, even without blacklisting. This error rate is almost 10^5 times smaller than that of majority voting for the same slowdown.

4.3 Performance

Since each point in these plots requires 100 simulations of 10 batches of 10,000 work objects each, it can take hours to compute a point with a single machine. Thus, parallelizing the application was really necessary in this case. To generate these plots, we used a variety of machines within the MIT campus network running on different operating systems. These included Windows PCs (a 166 MHz Pentium Pro, 180 MHz and 200 MHz dual-Pentium machines, 350 MHz Pentium machines, and others), Linux PCs (800 MHz Pentium III machines), and Sun workstations (Sun Sparcstation 5 and Ultra 5 workstations). Bayanihan's use of web-based Java applets allowed us to run the parallel simulator on these machines with minimal effort.

Figure 4 shows the speedup of the system *relative to the fastest machine*. Because the machines had different speeds, speedup in this case was not clearly defined. To measure speedup, we measured the speed of each individual machine by having it report the amount of local computation time required (not counting communication time) and the number of operations done for each work object. At the end of the batch, the server goes through all the results submitted by the worker and computes the total number of operations and the total time. Dividing these gives us the average speed of each machine for that batch. Finding the fastest machine and comparing its speed to the actual throughput at the server (measured as the total number of operations for the batch over "wall clock" time at the server), we get the speedups indicated by diamonds in Fig. 4.

As shown, we get more speed as we add more processors. Apart from that, however, it is difficult to draw any conclusions about the performance of the system. Since

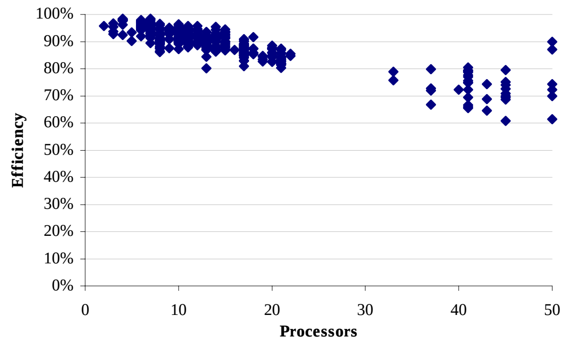

Figure 5. Efficiency of the simulator.

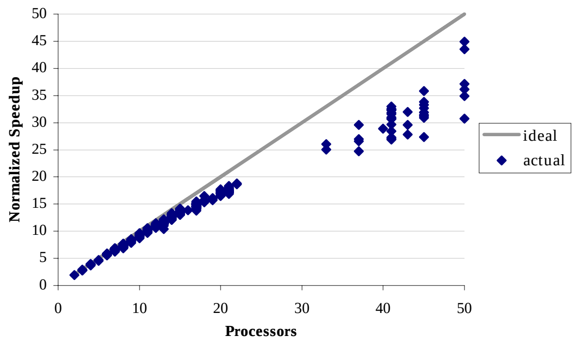

Figure 6. Normalized speedup of the simulator.

some of the machines are slower, the ideal speedup is less than the ideal speedup of P for P processors in a homogeneous system. To get a better idea of how much speedup we can expect, we took the total of the individual speeds of the machines at the end of the batch. As shown by the triangles in Fig. 4, this was less than the number of processors, as we expected, but more than the actual net speed measured at the server. Given this total speed, we can get a better measure of the performance of the system by calculating its *efficiency* as the actual speed over the total speed. Figure 5 shows a scatterplot of the efficiencies we observed across different batches and runs of the simulator with a varying combination of machines, while Fig. 6 shows an equivalent plot with the *normalized speedup*, computed by multiplying the efficiency by the number of processors.

As shown, we get reasonably good efficiency, even at 50 processors. Interestingly, the losses in efficiency in this case are *not* due to communication and server-side overhead, as we might at first suspect. The simulations shown here were fairly coarse-grain computations – i.e., the average time taken by the fastest computer for each work object (i.e., sequential Monte Carlo run) was between 2 and 200 seconds with a median of around 20 seconds (the times varied because some parameter combinations resulted in longer or shorter simulations). We suspect that most of the inefficiencies are instead due to a limitation of Bayanihan's *eager scheduling* system [9]. Generally, eager scheduling is a good feature that makes sure that slow processors do not slow down the faster processors (i.e., in the worst case, all the work will be done in the time it would take the faster processors to do the job by themselves, as if the slower processors were not there). However, in this case, we had only 100 work objects distributed among up to 50 workers. This caused fast workers to quickly run out of unassigned work to do, and start doing work already assigned to slower workers – possibly finishing the work before they do. This effectively wasted the processing power of the slower processors.

In general, the way to get around this problem and get more efficiency is to have longer batches. In this case, one solution might be to provide a `doMultiBatch()` method in the Monte Carlo API that would take multiple pairs of configuration and collector objects, and allow Monte Carlo simulations with more than one set of parameter values to run in parallel in the same batch. Suppose for example, that we want to see the effects of changing s from 0.1 to 0.2 to 0.3. Instead of having to run a sequence of three batches of 100 work objects each as we do now, we may run a single batch with 300 work objects instead, and thus improve efficiency.

In any case, however, even with its slightly suboptimal efficiency, the sabotage-tolerance simulator has proven itself an invaluable *enabling tool* that demonstrates the ability of volunteer computing to empower people to conduct previously impossible research.

5 Conclusion

In this paper, we have presented two main results. The first is that it is possible to protect volunteer computing systems from computational sabotage attempts by malicious volunteers without too much redundancy. We have shown, through simulation results, that by combining old techniques such as voting, with new ones such as spot-checking, and credibility-based fault-tolerance, we can reduce error rates by several orders of magnitude, needing only 2 or 3 times more time than a system without sabotage-tolerance. As future work, it would be interesting to implement these techniques in real volunteer computing and Internet computing systems, and see how they fare in real life, as well as how they can be improved.

Our second result is the way itself by which we were able to arrive at our first result. By using our Bayanihan web-based volunteer computing system to parallelize our simulator, and thus allow us to get results in much less time than before, we have demonstrated that web-based volunteer computing systems and other Internet computing systems can indeed be used for real research, and not just for solving esoteric mathematical problems, as some may think. In particular, we have shown how volunteer computing can be used for parametric analysis and Monte Carlo simulations. These include a wide range of useful applications beyond our own sabotage-tolerance simulator, and thus offer researchers many opportunities to start benefitting from volunteer computing today.

Appendix: Sample Code

```
package bayanihan.apps.ftsim;

import java.io.*; import bayanihan.util.*;

public class CredScanProgBL extends FTSimProg
{  // ... some code omitted ...
   public CredScanProgBL()
   {  super();
      // this class specifies the simulator version
      // which contains formulas used for credibility
      // in this case, this class uses the credibility
      // metric for spot-checking with blacklisting
      this.simRunClass = SimRunWorkCredBL.class;
   }

   /*  run() method  */

   public void run()
   {  // ... some code omitted ...

      // the following are the parameter settings to try
      double credArr[] = { 0.999999, 0.99999,
                           0.9999, 0.999, 0.99 };
      double fArr[] = { 0.2, 0.1, 0.05, 0.01 };
      double sArr[] = { 0.01, 0.02, 0.03, 0.05, 0.07,
                        0.09, 0.1, 0.12, 0.14, 0.16,
                        0.18, 0.20, 0.22, 0.25, 0.3,
                        0.35, 0.5, 0.65, 0.8, 1.0 };

      // loop through different values of f
      for ( int i = 0; i < fArr.length; i++ )
      {  this.config.fractBad = fArr[i];
         this.config.cF = fArr[i];

         // loop through different values of
         // credibility threshold, and s
         doCredScan( sArr, credArr );
      }
   }

   /*  Parallel method  */

   protected void doCredScan( double[] sArr,
                              double[] credArr )
   {  // loop through different credibility thresholds
      for ( int c = 0; c < credArr.length; c++ )
      {  this.config.cThresh = credArr[c];

         // loop through different values of s
         for ( int i = 0; i < sArr.length; i++ )
         {  this.config.probSabotage = sArr[i];

            // doBatch runs a Monte Carlo simulator
            // in parallel, given the work class,
            // the config object, the number of runs,
            // and a collector object for result stats
            doBatch( this.sumRunClass, this.config,
                     this.numRuns, this.collector );

            writeCollector( collector );
            System.gc();
         }
      }
   }
}
```

References

[1] A. Baratloo, M. Karaul, Z. Kedem, and P. Wyckoff, Charlotte: Metacomputing on the Web, in Proc. 9th Intl. Conf. on Parallel and Distributed Computing Systems, 1996. http://cs.nyu.edu/milan/charlotte/

[2] distributed.net home page. http://www.distributed.net

[3] P. Cappello, B.O. Christiansen, M.F. Ionescu, M.O. Neary, K.E. Schauser, and D. Wu, Javelin: Internet-Based Parallel Computing Using Java, in *Proc. ACM Workshop on Java for Science and Engineering Computation*, Las Vegas, 1997. http://javelin.cs.ucsb.edu/

[4] Entropia home page. http://www.entropia.com/

[5] W. Hoscheck, Colt Web page. http://nicewww.cern.ch/~hoschek/colt/index.htm

[6] Parabon home page. URL: http://www.parabon.com/

[7] Popular Power home page. http://www.popularpower.com/

[8] Process Tree home page. http://www.processtree.com/

[9] L.F.G. Sarmenta and S. Hirano, Building and Studying Web-Based Volunteer Computing Systems Using Java. *Future Generation Computer Systems*, Special Issue on Metacomputing, 15(5-6), Elsevier, 1999. http://www.cag.lcs.mit.edu/bayanihan/papers/fgcs/

[10] L.F.G. Sarmenta, Sabotage-Tolerance Mechanisms for Volunteer Computing Systems, to appear in *CCGrid 2001*, Brisbane, Australia, May, 2001. http://www.cag.lcs.mit.edu/bayanihan/papers/ccgrid01/

[11] L.F.G. Sarmenta, *Volunteer Computing*, Ph.D. thesis, MIT, Cambridge, MA, March, 2001. http://www.cag.lcs.mit.edu/bayanihan/papers/sarmenta-phd/

[12] SETI@home home page. http://setiathome.ssl.berkeley.edu/