Proceedings of the 2015 Winter Simulation Conference
L. Yilmaz, W. K. V. Chan, I. Moon, T. M. K. Roeder, C. Macal, and M. D. Rossetti, eds.

COMPARING MESSAGE PASSING INTERFACE AND MAPREDUCE FOR LARGE-SCALE PARALLEL RANKING AND SELECTION

Eric C. Ni

Operations Research and Information Engineering
Cornell University
Ithaca, NY 14853, USA

Dragos F. Ciocan

Technology and Operations Management
INSEAD
Fontainebleau 77305, FRANCE

Shane G. Henderson

Operations Research and Information Engineering
Cornell University
Ithaca, NY 14853, USA

Susan R. Hunter

School of Industrial Engineering
Purdue University
West Lafayette, IN 47907, USA

ABSTRACT

We compare two methods for implementing ranking and selection algorithms in large-scale parallel computing environments. The Message Passing Interface (MPI) provides the programmer with complete control over sending and receiving messages between cores, and is fragile with regard to core failures or messages going awry. In contrast, MapReduce handles all communication and is quite robust, but is more rigid in terms of how algorithms can be coded. As expected in a high-performance computing context, we find that MPI is the more efficient of the two environments, although MapReduce is a reasonable choice. Accordingly, MapReduce may be attractive in environments where cores can stall or fail, such as is possible in low-budget cloud computing.

1 INTRODUCTION

The simulation optimization (SO) problem is a nonlinear optimization problem in which the objective function can only be observed with error as output from a Monte Carlo simulation. That is, the SO problem can be expressed as

$$\text{Find:} \quad \arg\max_{x \in \mathscr{D}} \mathrm{E}[f(x,\xi)],$$

where the function f is embedded in a Monte Carlo simulation, ξ represents a source of randomness, and $\mathscr{D} \subset \mathbb{R}^d$ is some known feasible set. At each point $x \in \mathscr{D}$, we may only observe an estimate of the function value, $\hat{f}(x)$, as output from the simulation. When the feasible set $\mathscr{D}$ is finite, this SO problem is also called a *ranking and selection* (R&S) problem, and each feasible point is referred to as a "system." The system having the maximum objective function value is called the "best" system. That is, if there are k competing systems and each system has mean objective value $\mu_i := \mathrm{E}[f(x_i,\xi)]$ for all systems $i = 1,\ldots,k$ such that $\mu_1 \leq \mu_2 \leq \ldots \leq \mu_k$, system k is the best system (see, e.g., Bechhofer et al. 1995, Kim and Nelson 2006 for overviews of the R&S literature).

Since we only observe *estimates* of the systems' objective values, algorithms that solve the R&S problem often provide a probabilistic *correct selection* guarantee: with high probability, the solution returned upon algorithm termination is the best system, as long as the best system is at least some amount $\delta > 0$ away from the next-best system. That is, the probability of correct selection (PCS) guarantee ensures that, whenever $\mu_k - \mu_{k-1} \geq \delta$, $P\{\text{Selecting system } k\} \geq 1-\alpha$, for some (small) user-selected value of $\alpha > 0$. However, owing to their conservativeness, these procedures are often best suited for a small number of systems (Nelson et al. 2001). When the number of systems is large, the number of required simulation replications to terminate with the desired probabilistic guarantee often becomes prohibitive, at least on a serial processor — the platform for which most R&S algorithms are designed.

Therefore to solve large-scale R&S problems, we design an algorithm that simultaneously employs two variations on this popular R&S approach. First, we specifically design our algorithm to be implemented on a parallel computing platform, to harness the power of these recently ubiquitous computing resources — a nontrival task, owing to the non-trivial statistical issues that arise in parallel platforms (Heidelberger 1988, Glynn and Heidelberger 1990, Glynn and Heidelberger 1991). Second, instead of providing a PCS guarantee, we design our algorithm to provide a probability of *good selection* (PGS) guarantee (see, e.g., Nelson et al. 2001): with high probability, the solution returned upon algorithm termination is a system within δ of the best. Such a guarantee may be more practical when the number of systems is large, since systems near the best may be close together — for example, when a quadratic function is discretized with a fine mesh, many systems are close to the best system in objective function value. (For earlier work on implementing R&S algorithms in parallel, see Luo et al. 2000, Yoo et al. 2009, Chen 2005, Luo et al. 2013, Ni et al. 2013, Ni et al. 2014.)

In this paper, we outline our parallel R&S procedure and demonstrate its performance using two popular parallel computing approaches: Message Passing Interface (MPI) and MapReduce. MPI is a message passing system by which cores on a parallel computer can communicate with each other; MPI is a highly flexible and customizable way of implementing a parallel R&S algorithm. MapReduce is more of a “one-size-fits-all” approach, which we implement by creating “map” procedures that obtain simulation replications and summary statistics for the systems, while the “reduce” procedures perform screening. While our MPI implementation can be highly asynchronous, the MapReduce implementation requires a certain amount of synchronization across cores. (We provide more detail about MPI and MapReduce in §2.) Because MPI is customizable, a priori, we expect that our MPI implementation will outperform our MapReduce implementation. Thus the interesting research question is not which parallel framework is faster, but rather whether MapReduce can offer most of the speed of MPI, along with its inherent advantages in terms of programming simplicity and its portability across different computing infrastructures.

To obtain a fair comparison of these frameworks, we implement both versions of our procedure on a reliable high-performance computing (HPC) platform. Broadly, our results indicate that

- as expected, Apache Hadoop (the particular implementation of MapReduce we employ) exhibits significantly slower wall-clock times than MPI across all of our trials, with MPI always being at least a factor of 6 times more efficient than Hadoop, perhaps owing to the fixed overhead incurred by Hadoop;
- the gap between MPI and Hadoop reduces as the system-to-core ratio or the simulation effort per batch-to-core ratio increases;
- in line with the wall-clock results, Hadoop exhibits less core efficiency than MPI, with the gap closing as the system-to-core ratio or the simulation effort per batch-to-core ratio increases;
- either implementation of our procedure is robust to random completion times, likely because we obtain simulation replications in batches, leading to a summing effect that reduces the variance of simulation run times relative to their mean;
- the Hadoop implementation requires a series of phases, each of which consists of one map and one reduce phase, and this may necessitate substantial data transfer at the start of each phase as simulation runs are initialized with the data necessary to complete simulation replications.

In the remainder of this paper we will mostly use the terms Hadoop and MapReduce as synonyms, although they are not the same thing. Indeed, Apache Hadoop is *one* implementation of the MapReduce framework.

The remainder of the paper is organized as follows. In §2, we outline the selection procedure and our implementations of MPI and Hadoop. In §3, we provide our computational results, with a discussion in

§4. For all implementation details and computational results, including a proof that our procedure provides a PGS guarantee, we refer the reader to Ni et al. (2015).

2 THE SELECTION PROCEDURE AND IMPLEMENTATIONS

In order to be able to contrast an MPI implementation with a MapReduce implementation, we first need to understand the overall structure of our parallel procedure that guarantees a probability of good selection (henceforth referred to as good-selection procedure, or GSP). We will subsequently describe how that procedure can be implemented within MPI and MapReduce frameworks.

2.1 A Parallel Procedure with Good Selection Guarantee

Our good-selection procedure proceeds in several stages.

- **Stage 0:** In this optional stage we simulate each system a fixed and limited number of times in order to estimate (wall clock) running times for a single replication for each system. This information can subsequently be used to approximately load balance the workload on cores, as described below.
- **Stage 1:** We next obtain a sample of size n_0 from every system. (This is a standard step in many ranking and selection algorithms.) The sample is used to obtain sample means and variances for all systems. The estimated run times for systems obtained in Stage 0 may be used to determine how to do this so that all cores are approximately equally loaded.
- **Stage 2:** We then perform an asynchronous sampling-and-screening step. Typically in screening, all systems are compared with all others, but that requires on the order of k^2 work, which is prohibitive when the number of systems k is large. Accordingly, we do not screen all systems against one another, but rather divide the k systems into subgroups, and screen within subgroups. We limit the screening *across* subgroups to simply sharing the apparent "best" within each subgroup with the others, to attempt to rapidly "knock out" systems.
  For user-selectable maximum number of batches $\bar{r}$ and average batch size β (see Ni et al. 2015 for suggested values), Stage 2 proceeds by gathering up to a maximum of $\bar{r}$ batches of observations from each system. The batch sizes are chosen to ensure both statistical and computational efficiency, and have average size β.
  Our screening stage has two key properties:
  1. First and foremost, the probability that the truly best system (or at least one of the best systems if there are ties) is retained throughout the screening stage is at least $1-\alpha_1$, where α_1 is on the order of 0.025 (for a 0.95 probability of good selection PGS). This property holds regardless of the value of $\mu_k - \mu_{k-1}$ and δ.
  2. The screening does not necessarily continue until only one system remains. Rather, we proceed up to a maximum number of batches $\bar{r}$ that is specified by the user, and so there will be a random number of systems remaining at the conclusion of the screening stage.

  The screening stage is designed to quickly eliminate clearly inferior systems, and to not expend too much effort in distinguishing high-quality systems that are therefore close in objective function value.
- **Stage 3:** At the outset of this stage a random number of systems remain. We now perform a Rinott (1978) final stage on these remaining systems with confidence level $1-\alpha_2$, where α_2 is on the order of 0.025 (for a 0.95 PGS). In more detail, we halt screening, determine a residual simulation runlength for each remaining system, and obtain that many replications for each remaining system. We then compute the surviving system with the largest sample mean over the replications obtained throughout Stages 1 through 3, and select that system as the best. In this stage we rely on the indifference zone parameter δ in selecting the residual runlength, rather than on the likely small true difference in mean performance

of each surviving system, so that the computational effort in this stage is primarily determined by the number of surviving systems and δ.
The Rinott procedure provides a "multiple comparison with the best" guarantee that we can then leverage (Nelson and Matejcik 1995, Ni et al. 2015) to ensure that with probability at least $1-\alpha_2$ we select a system with objective value that is within δ of the best of the systems that reached Stage 3.

The overall procedure has a PGS of $1-\alpha_1-\alpha_2$, since the procedure can only fail if the truly best system is eliminated in Stage 2 or a system that is not within δ of the best surviving system is selected in Stage 3. A union bound on these two "bad" events yields the PGS guarantee.

2.2 An MPI Implementation

We employ a master-worker framework whereby a single core is designated as the master, and controls the entire procedure. Our implementation assumes reliable cores, and would not work well in an environment where cores can stall or fail, in which case the parallel program may crash and no interim results can be obtained. This lack of resilience arises from MPI itself and not from our implementation within MPI.

In our MPI implementation, simulation and screening tasks are broken down and allocated to workers in an asynchronous fashion as they complete their tasks and signal the master that they are available. Our algorithm design limits the amount of synchronization in an attempt to obtain an overall procedure that scales well as the number of cores and/or systems gets large. The only synchronization arises at the end of each of the 4 stages (Stages 0 through 3), so is limited.

The master has direct control over the simulation effort of workers, and accordingly can employ predictions of system-specific wall-clock running times of replications from Stage 0 in order to attempt to balance loads across cores.

Screening is performed within subgroups by workers, so at any given time a worker may be screening within its subgroup or obtaining simulation replications.

2.3 A MapReduce Implementation

The philosophy of MapReduce-based algorithms is very different from that of algorithms based on MPI. With MPI, all communication must be carefully designed and explicitly coded. With MapReduce, all communication is handled by the MapReduce implementation (we employ Apache Hadoop, which is a standard implementation of MapReduce). The focus of algorithm design instead turns to designing the "mapper" and "reducer" steps without regard to the "system" level issues of how to control the cores.

We employ a series of mapper and reducer phases in our MapReduce implementation. In broad terms, simulation replications are obtained in batches within mapping phases, and screening is performed in reducer phases. Each mapper phase must complete before the ensuing reducer phase can begin, and the following MapReduce phase can only begin once the previous reducer phase completes. This introduces (undesirable) synchronization steps within Stage 2 that reduce the efficiency of the overall procedure relative to MPI.

In slightly more detail, in Stage 2 each mapper is associated with a single system. The mapper takes as input the summary statistics for the system to date, performs an additional batch of simulation replications, and outputs the updated summary statistics. These statistics are then fed to reducers that perform the screening. Those systems that survive the screening are then output to the next phase of simulation and screening. In Stage 3 there is one more map and reduce phase in which the Rinott samples are obtained and the estimated best system identified. Additional mapper and reducer phases are used in the course of the algorithm to perform additional minor steps, but the ones explicitly discussed above are the most important.

3 COMPUTATIONAL RESULTS

In this section, we present our computational results for the MPI and MapReduce implementations.

3.1 Computing Environment

To establish a common platform for comparison between MPI and MapReduce, we constrain our experiments to a fixed computational infrastructure, specifically the Extreme Science and Engineering Discovery Environment (XSEDE) Stampede high performance computing cluster. Stampede consists of approximately 6400 server nodes with two 8 core Intel Xeon E5 processors and 32GB of memory, networked together using FDR InfiniBand technology. There is no shared memory between distinct nodes. Stampede offers mvapich2, which is an optimized version of the MPI standard. There is relatively less support for MapReduce – we managed to deploy our Hadoop 1.2.1 code using myhadoop, an experimental software package that provides the interface between Hadoop and the environment's batch scheduler. Both versions of the algorithm can be found in the repository (Ni 2015).

While our experiments are restricted to the Stampede cluster, the code can be ported to other distributed systems, including commercially available "cloud computing" services such as Amazon's EC2. However, for EC2-like environments, our MPI implementation needs to be augmented with fault tolerance functionality, which should enable the procedure to continue in the event that one of the tasks distributed across the cluster fails to complete. One relatively common example of how to implement this is periodically backing up the state of the computation; thus, upon a failure the procedure reverts to this state rather than starting from scratch. While working on the Stampede environment, we have never seen a node failure or any other reliability issues, so we wrote our code under the assumption that all tasks complete successfully. This assumption may be violated on other clusters, either because a core may physically fail, or because our task running on that core may get pre-empted by a higher priority task that requires 100% of the core's time. MPI leaves the user the responsibility to write their own procedures handling these issues. On the other hand, Hadoop is native to failure-prone computing environments and comes with in-built fault tolerance (if a core does not respond within a certain period of time, its current task will automatically be replicated and executed on another core), making an "out-of-the-box" port from one computing platform to another much easier.

Although it is "easier to use" because it handles all communication and core control, MapReduce has several disadvantages over MPI.

1. Computation and communication are synchronized with MapReduce: before moving from a map phase to a reduce phase, all mappers must finish, even if the bulk of the cores are idling for large periods of time waiting for the slowest one to finish.
2. Hadoop maintains no state between two different MapReduce rounds, which leads to unnecessary reading and writing to disk at every round, as problem state is transferred from one round to another via a distributed file system; additionally, the same holds for problem data that may not change over Hadoop's execution. This disadvantage is specific to the MapReduce implementation (such as Hadoop in our case), rather than being an inherent drawback of the MapReduce concept.
3. MapReduce incurs fault tolerance overhead that is unneeded on Stampede and we are not aware of options to turn it off.

All these disadvantages are trade-offs associated with using a "one size fits all" framework in the case of MapReduce, versus one such as MPI that allows the implementation's designer high levels of flexibility in specifying how tasks and communication are scheduled across the compute nodes. As noted in the introduction, the question we seek to answer with our experiments is whether an easy to use framework like MapReduce can be competitive in terms of speed with a tailored MPI implementation.

There are some variants of MapReduce that attempt to address some of the limitations above, e.g., Elgohary (2012), but we do not exploit them because they are not available for the platform we employ.

Table 1: A comparison of two implementations of GSP using parameters $\delta = 0.1$, $n_0 = 50$, $\alpha_1 = \alpha_2 = 2.5\%$, $\bar{r} = 1000/\beta$. "Total time" is summed over all cores.

Configuration	β	Version	Wall-clock time (sec)	Number of replications ($\times 10^6$)	Total time Simulation ($\times 10^3$ sec)	Total time Screening (sec)	Utilization %
3,249 systems on 64 cores	100	HADOOP	458	0.46	0.34	0.14	1.2
		MPI	3.01	0.50	0.18	0.01	94
	200	HADOOP	277	0.63	0.41	0.10	2.3
		MPI	4.11	0.69	0.25	0.01	95
57,624 systems on 64 cores	100	HADOOP	545	8.83	5.14	1.94	15
		MPI	52.7	9.07	3.29	0.89	98
	200	HADOOP	406	12.4	6.97	1.70	27
		MPI	75.0	12.9	4.69	0.83	98
1,016,127 systems on 1,024 cores	100	HADOOP	1250	275	158	120	12
		MPI	123	315	114	29.9	91
	200	HADOOP	812	338	194	88.5	23
		MPI	141	383	140	28.9	97

3.2 Experiments

We test our implementations on an ensemble of throughput maximization problems from `SimOpt.org` (Henderson and Pasupathy 2014); the specific task is to maximize the throughput in a 3 station line with an infinite number of jobs in front of station 1, finite buffers $(b_2, b_3) \in Z_+^2$ in front of stations 2 and 3 respectively and exponentially distributed service times $r_i \in Z_+$ for stations $i = 1, 2, 3$, subject to the constraints

$$b_2 + b_3 = B,$$
$$r_1 + r_2 + r_3 = R.$$

We obtain 3 instances with (i) 3,249, (ii) 57,624 and (iii) 1,016,127 systems by setting $(B, R) =$ (i) (20, 20), (ii) (50, 50) and (iii) $(128, 128)$. Each replication consists of a warm-up time of 2000 job releases, followed by estimating the steady-state throughput using the following 50 job releases. A consequence of this design is that our replication times will exhibit little variability.

Before proceeding to the computational comparisons, we should highlight two important differences between our MPI code and our Hadoop code.

1. The MPI code is in C++, while the Hadoop code is in Java, because these are the languages that are native to these environments.
2. We perform less screening in the MPI implementation than in Hadoop. More specifically, in our experiments we found that in the MPI implementation, communication at the master was becoming a bottleneck due to the passing of statistics on best systems between cores for screening. Hence, in the MPI implementation only, we screen on any core only between the systems on that core and the 20 best systems from other cores, as opposed to the best system from *every* other core in Hadoop. The difference in the amount of screening is not negligible, but it is modest relative to the effort in running simulation experiments as we see below.

In Table 1, we compare the performance of our MPI and Hadoop MapReduce implementations along two dimensions: wall-clock time and core utilization, defined as

$$\text{Utilization} = \frac{\text{total time spent on simulation}}{\text{wall-clock time} \times \text{number of cores}}.$$

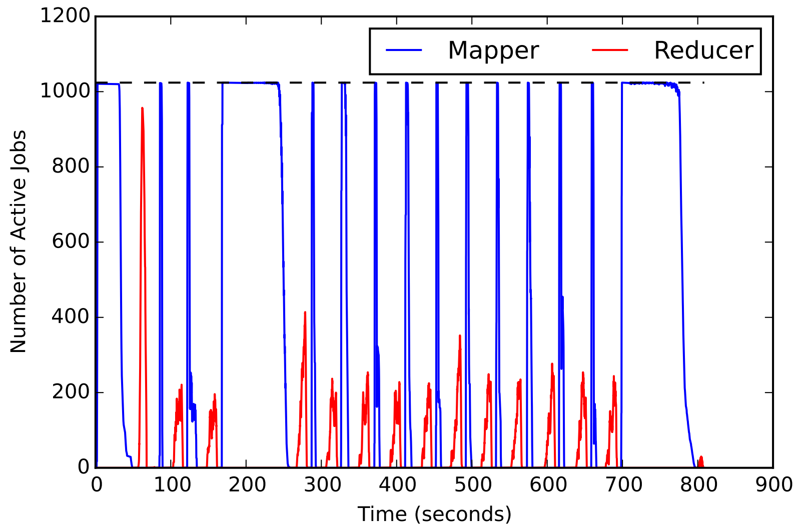

Figure 1: A profile of the MapReduce implementation solving 1,016,127 systems on 1,024 cores with average batch size $\beta = 200$. The first three MapReduce steps implement Stage 1, the next ten MapReduce steps constitute Stage 2, and the final two MapReduce steps complete Stage 3.

Utilization is a measure of the efficiency of our implementations, as it measures the fraction of CPU time spent running the simulation, rather than communicating data between cluster nodes or handling MPI or Hadoop overhead. This measure ignores the time needed for screening, but that is relatively negligible as we shall see in the results below.

3.2.1 Wall-clock Times

As expected, MPI exhibits faster wall-clock times than Hadoop across all of our trials. The gap in wall clock times narrows as the batch size and/or the system-to-core ratio are increased. This is consistent with our intuition that MapReduce's synchronization incurs rather large fixed overhead costs at each iteration, while the MPI procedure eliminates these costs as much as possible. Hence, increasing the per round computational cost of the simulations spreads out MapReduce's overhead, diluting the advantage of MPI. Indeed, as we increased the system-to-core ratio and the batch size, the gap between Hadoop and MPI narrowed down to roughly a factor of 6, a twenty-fold improvement over the gap for the smallest problems. We conjecture that as the problem and batch sizes are further increased, the gap in wall-clock times could be further reduced.

3.2.2 Utilization Rates

The MPI implementation also consistently achieves higher efficiency (up to $91-98\%$) compared to the Hadoop one, but as with wall-clock times, the gap closes the larger the system-to-core ratio and the batch size.

Figure 1 empirically highlights some of our previous intuition regarding the inefficiencies that are inherent to a MapReduce implementation.

1. Often the bulk of the cores are idle in both the map and reduce stages waiting for the slower cores to finish. With MPI, the master could use idle cores more efficiently by allocating them new tasks asynchronously.

Table 2: A comparison of two implementations of GSP using a random number of warm-up job releases distributed like $\min\{\exp(X), 20,000\}$, where $X \sim N(\mu, \sigma^2)$. We use parameters $\delta = 0.1$, $n_0 = 50$, $\alpha_1 = \alpha_2 = 2.5\%$, $\bar{r} = 1000/\beta$.

Configuration	β	Version	μ	σ^2	Wall-clock time (sec)	Utilization %
1,016,127 systems on 1,024 cores	200	HADOOP	7.35	0.5	151	25
		MPI			851	97
1,016,127 systems on 1,024 cores	200	HADOOP	6.60	2.0	853	22
		MPI			146	97

2. Hadoop spends additional overhead between the map and reduce phases performing an intermediary step called a "shuffle", which sorts and distributes the output of the mappers to the reducers. Consequently, the shuffle and reducer steps form significant gaps between the map steps and a large amount of computing time is wasted. While some of this coordination is unavoidable, Hadoop does not offer flexibility to streamline it to our setting.

3.2.3 Uneven Simulation Times

Intuitively, the fact that our simulation times are not highly variable relative to their means is why MapReduce is competitive with MPI, since this presumably reduces the impact of synchronization delays. On the other hand, for settings where the simulation times exhibit enough variation that one simulation takes much longer than the others, one would expect that the synchronization-free design of our MPI procedure would become a decisive advantage.

We attempted to verify this conjecture with a different suite of experiments, where we artificially injected variance into our run times by randomizing the number of job releases in the warm-up stage (previously, this was deterministically set to 2000). We chose a truncated log-normal distribution from which to sample, as its heavy tails should capture the scenario where a single simulation's very large run time slows down the entire system. The truncation of the log-normal was necessary due to the fact that Hadoop has an in-built time-out value: after a mapper calculation has exceeded this large time value, it is declared a time-out by the Hadoop master. Parameters of the truncated lognormal distribution are given in Table 2.

Contrary to our intuition, we were not able to observe a stronger divergence between MPI and MapReduce than our base case, as both wall-clock times and utilizations remained largely the same (see Table 2). A potential explanation is that batching averages out the variations across individual replications, and indeed we have observed that increasing the variance of the sampling distribution has little effect in terms of slowing down MapReduce. This suggests that, in fact, our procedure is quite robust to uneven simulation run times and that rather extreme variations would be required for MapReduce to suffer a sharp performance decrease.

4 DISCUSSION

The computational results confirm our expectation that MPI would outperform MapReduce in a high-performance parallel computing environment. As evidenced by its high core utilization, the MPI implementation efficiently uses parallel cores to generate simulation replications and incurs relatively little overhead. The core utilization figures are much lower for the MapReduce implementation.

One factor that may further deteriorate the performance of the MapReduce implementation is that simulations may take a significant amount of time to initialize. For example, vehicle routing simulations often preload a large amount of static data representing the "map" of the simulated area, which can be used repeatedly for multiple systems and replications. With MPI, such static data can be stored in memory and so the simulation runs only need to be initialized once. However, because Apache Hadoop does not allow cores to keep any data in memory between MapReduce iterations, the static data has to be reloaded repeatedly.

Despite showing relatively poor performance of the MapReduce implementation, the computational study does point to some regimes where it may work better. As the number of systems increases from 3,249 to 57,624 while keeping the number of cores constant, the utilization of MapReduce significantly increases, indicating that the procedure overhead does not increase much with respect to the number of systems. Further, as we reduce the number of phases in Stage 2 by increasing the average batch size parameter β (which is directly proportional to the number of simulation replications collected from each system in every iteration) from 100 to 200, we notice an increase in utilization as the result of less frequent screening and less shuffling overhead. Hence, for problems that feature a large solution space or require long simulation runs, we would expect the MapReduce overhead to be less pronounced and the overall utilization to improve. Furthermore, the observed overhead from full synchronization and data shuffling is largely due to the restrictions of the Apache Hadoop implementation rather than the MapReduce framework, and the performance of MapReduce-based procedures has much upward potential as more sophisticated MapReduce implementations are developed.

"Cloud computing" services like Amazon EC2 are a more accessible and affordable alternative to high-performance clusters such as Stampede. These services often offer a variety of computing resources with different processing speed and reliability. On less robust hardware, the MapReduce implementation is a more compatible choice, for Apache Hadoop offers built-in protection against core failures by having the master core periodically detect the status of worker cores and re-launch failed map or reduce tasks. The Hadoop distributed file system (HDFS), which is used by Apache Hadoop to keep MapReduce input and output, also maintains replicates of data blocks to ensure that no data is lost as a result of a single hardware failure. The increased overhead of those fault tolerance mechanisms is often offset by the decrease in cost of using cheaper resources. The current MPI implementation, on the other hand, does not monitor core failures and distributes data across parallel cores without backup, so a core failure that causes loss of simulation results may break the procedure. Therefore, to use MPI on a cloud environment where hardware failure is a norm, we need to either introduce additional protection or design a new procedure which is robust against random loss of simulation statistics.

ACKNOWLEDGMENTS

This work was partially supported by NSF grant CMMI-1200315, and used the Extreme Science and Engineering Discovery Environment (XSEDE), which is supported by National Science Foundation grant number ACI-1053575.

REFERENCES

Bechhofer, R. E., T. J. Santner, and D. Goldsman. 1995. *Design and Analysis of Experiments for Statistical Selection, Screening and Multiple Comparisons*. New York: John Wiley and Sons.

Chen, E. J. 2005. "Using Parallel and Distributed Computing to Increase the Capability of Selection Procedures". In *Proceedings of the 2005 Winter Simulation Conference*, 723–731.

Elgohary, A. 2012. "Stateful MapReduce".

Glynn, P. W., and P. Heidelberger. 1990. "Bias properties of budget constrained simulations". *Operations Research* 38:801–814.

Glynn, P. W., and P. Heidelberger. 1991. "Analysis of Parallel Replicated Simulations Under a Completion Time Constraint". *ACM Transactions on Modeling and Computer Simulation* 1 (1): 3–23.

Heidelberger, P. 1988. "Discrete event simulations and parallel processing: statistical properties". *Siam J. Stat. Comput.* 9 (6): 1114–1132.

S. G. Henderson and R. Pasupathy 2014. "Simulation Optimization Library".

Kim, S.-H., and B. L. Nelson. 2006. “Selecting the best system”. In *Simulation*, edited by S. G. Henderson and B. L. Nelson, Handbooks in Operations Research and Management Science, Volume 13, 501–534. Elsevier.

Luo, J., J. L. Hong, B. L. Nelson, and Y. Wu. 2013. “Fully Sequential Procedures for Large-Scale Ranking-and-Selection Problems in Parallel Computing Environments”. *Working Paper*.

Luo, Y.-C., C.-H. Chen, E. Yucesan, and I. Lee. 2000. “Distributed Web-based simulation optimization”. In *Proceedings of the 2000 Winter Simulation Conference*, Volume 2, 1785–1793.

Nelson, B. L., and F. J. Matejcik. 1995. “Using Common Random Numbers for Indifference-Zone Selection and Multiple Comparisons in Simulation”. *Management Science* 41 (12): 1935–1945.

Nelson, B. L., J. Swann, D. Goldsman, and W. Song. 2001. “Simple procedures for selecting the best simulated system when the number of alternatives is large”. *Operations Research* 49 (6): 950–963.

Eric C. Ni 2015. “mpiRnS: Parallel Ranking and Selection Using MPI”. https://bitbucket.org/ericni/mpirns.

Ni, E. C., D. F. Ciocan, S. G. Henderson, and S. R. Hunter. 2015. “Efficient Ranking and Selection in Parallel Computing Environments”. *Working Paper*. http://arxiv.org/abs/1506.04986.

Ni, E. C., S. G. Henderson, and S. R. Hunter. 2014. “A comparison of two parallel ranking and selection procedures”. In *Proceedings of the 2014 Winter Simulation Conference*.

Ni, E. C., S. R. Hunter, and S. G. Henderson. 2013. “Ranking and selection in a high performance computing environment”. In *Proceedings of the 2013 Winter Simulation Conference*, 833–845.

Rinott, Y. 1978. “On two-stage selection procedures and related probability-inequalities”. *Communications in Statistics - Theory and Methods* 7 (8): 799–811.

Yoo, T., H. Cho, and E. Yucesan. 2009, July 01. “Web Services-Based Parallel Replicated Discrete Event Simulation for Large-Scale Simulation Optimization”. *Simulation* 85 (7): 461–475.

AUTHOR BIOGRAPHIES

ERIC C. NI is a Ph.D. student in the School of Operations Research and Information Engineering at Cornell University. He received a B.Eng. in Industrial and Systems Engineering and a B.Soc.Sci. in Economics from the National University of Singapore in 2010. His research interests include simulation optimization, emergency services and queuing theory. His webpage is http://people.orie.cornell.edu/cn254/.

DRAGOS F. CIOCAN is an assistant professor of Technology and Operations Management at INSEAD. His research interests are in stochastic optimization, large scale distributed optimization and non traditional applications of revenue management. His email address is florin.ciocan@insead.edu, and his webpage is http://www.insead.edu/facultyresearch/faculty/profiles/dciocan/.

SHANE G. HENDERSON is a professor in the School of Operations Research and Information Engineering at Cornell University. His research interests include discrete-event simulation and simulation optimization, and he has worked for some time with emergency services and bike sharing applications. He co-edited the Proceedings of the 2007 Winter Simulation Conference. His web page is http://people.orie.cornell.edu/∼shane.

SUSAN R. HUNTER is an assistant professor in the School of Industrial Engineering at Purdue University. Her research interests include Monte Carlo methods and simulation optimization. Her email address is susanhunter@purdue.edu, and her webpage is http://web.ics.purdue.edu/∼hunter63/.