Submitted to *Operations Research*
manuscript (Please, provide the manuscript number!)

Efficient Ranking and Selection in Parallel Computing Environments

Eric C. Ni[†], Dragos F. Ciocan[‡], Shane G. Henderson[†], Susan R. Hunter[*]

[†]School of Operations Research and Information Engineering, Cornell University, Ithaca, NY 14853
cn254, sgh9@cornell.edu

[‡]Technology and Operations Management, INSEAD, Fontainebleau, France 77300
florin.ciocan@insead.edu

[*]School of Industrial Engineering, Purdue University, West Lafayette, IN 47907-2023
susanhunter@purdue.edu

The goal of ranking and selection (R&S) procedures is to identify the best stochastic system from among a finite set of competing alternatives. Such procedures require constructing estimates of each system's performance, which can be obtained simultaneously by running multiple independent replications on a parallel computing platform. Nontrivial statistical and implementation issues arise when designing R&S procedures for a parallel computing environment. We propose several design principles for parallel R&S procedures that preserve statistical validity and maximize core utilization, especially when large numbers of alternatives or cores are involved. These principles are followed closely by our parallel Good Selection Procedure (GSP), which, under the assumption of normally distributed output, (i) guarantees to select a system in the indifference zone with high probability, (ii) in tests on up to 1,024 parallel cores runs efficiently, and (iii) in an example uses smaller sample sizes compared to existing parallel procedures, particularly for large problems (over 10^6 alternatives). In our computational study we discuss 3 methods for implementing GSP on parallel computers, namely the Message-Passing Interface (MPI), Hadoop MapReduce, and Spark, and show that Spark provides a good compromise between the efficiency of MPI and robustness to core failures.

Key words: ranking and selection; parallel computing

History: revised September 30, 2016.

1. Introduction

The simulation optimization (SO) problem is a nonlinear optimization problem in which the objective function is defined implicitly through a Monte Carlo simulation, and thus can only be observed with error. Such problems are common in a variety of applications including transportation, public health, and supply chain management; for these and other examples, see `SimOpt.org` (Henderson and Pasupathy 2014). Solution methods are characterized by the nature of the feasible set; for overviews of methods to solve the SO problem, see, e.g., Fu (1994), Andradóttir (1998), Fu et al. (2005), Pasupathy and Ghosh (2013), Fu (2015).

We consider the case of SO on finite sets, in which the decision variables can be categorical, integer-ordered and finite, or a finite "grid" constructed from a continuous space. Formally, the SO problem on finite sets can be written as

$$\max_{i \in \mathcal{S}} \ \mu_i = E[X(i;\xi)] \tag{1}$$

where $\mathcal{S} = \{1, \ldots, k\}$ is a finite set of design points or "systems" indexed by i, and ξ is a random element used to model the stochastic nature of simulation experiments. (In the remainder of the paper we assume that $\mu_1 \leq \mu_2 \leq \cdots \leq \mu_k$, so that, unbeknown to the analyst, System k is the best.) The objective function $\mu : \mathcal{S} \to \mathbb{R}$ cannot be computed exactly, but can be estimated using output from a stochastic simulation represented by $X(\cdot;\xi)$. While the feasible space $\mathcal{S}$ may have topology, as in the finite but integer-ordered case, we consider methods to solve the SO problem in (1) that do not exploit such topology and that apply when the computational budget permits at least *some* simulation of *every* system in consideration. Such methods are called *ranking and selection* (R&S) procedures.

R&S procedures are frequently used on spaces with topology when structural properties, such as convexity, are difficult to verify or do not hold. They can also be used in conjunction with heuristic search procedures in a variety of ways (Pichitlamken et al. 2006, Boesel et al. 2003) to provide probabilistic guarantees on some subset of the solutions in the search space. See Kim and Nelson

(2006a) for an introduction to, and overview of, R&S procedures. R&S problems are closely related to best-arm problems (e.g., Jamieson and Nowak 2014, Bubeck and Cesa-Bianchi 2012), but there are several differences between these bodies of literature. Almost always, the algorithms developed in the best-arm literature assume that one system is simulated at a time and that simulation outputs are bounded, or that all variances have a known bound.

R&S procedures are designed to offer one of several types of probabilistic guarantee, and can be Bayesian or frequentist in nature. Bayesian procedures offer guarantees related to a loss function associated with a non-optimal choice; see Branke et al. (2007) and Chen et al. (2015). Frequentist procedures typically offer one of two statistical guarantees; in defining these guarantees, let $\delta > 0$ be a known constant and let $\alpha \in (0, 1)$ be a parameter selected by the user. The *Probability of Correct Selection* (PCS) guarantee is a guarantee that, whenever $\mu_k - \mu_{k-1} \geq \delta$, the probability of selecting the best system k when the procedure terminates is at least $1 - \alpha$. Henceforth, the assumption that $\mu_k - \mu_{k-1} \geq \delta$ will be called the *PCS assumption*; if $\mu_k - \mu_{k-1} < \delta$ then a PCS guarantee does not hold. In contrast, the *Probability of Good Selection* (PGS) guarantee is a guarantee that the probability of selecting a system with objective value within δ of the best is at least $1 - \alpha$, i.e., that PGS $= \mathrm{P}[\text{Select a system } K \text{ such that } \mu_k - \mu_K \leq \delta] \geq 1 - \alpha$. A PGS guarantee makes no assumption about the configuration of the means and is the same as the "probably approximately correct" guarantee in best-arm literature.

The computational effort required to solve R&S problems depends heavily on the computational cost of obtaining simulation replications. Even if this cost is very modest, R&S problems have traditionally been quite limited in the number of systems k that they can successfully handle, e.g., $k \leq 100$, due to the methods used to construct validity proofs. The constants in such methods often scale poorly with the number of systems, particularly in the worst-case configurations assumed in validity proofs, leading to extremely large required simulation runlengths. The advent of screening, i.e., discarding clearly inferior alternatives early on (Nelson et al. 2001, Kim and Nelson 2006b, Hong 2006), has allowed R&S to be applied to larger problems, say $k \leq 500$. Exploiting parallel computing

is a natural next step as argued in, e.g., Fu (2002). By employing parallel cores, simulation output can be generated at a higher rate, and a parallel R&S procedure should complete in a smaller amount of time than its sequential equivalent, allowing larger problems to be solved.

Heidelberger (1988), Glynn and Heidelberger (1990, 1991) explored the use of parallel computers to construct valid simulation estimators, but R&S procedures that exploit parallel computing have emerged only recently. Luo et al. (2000) and Yoo et al. (2009) employ a web-based computing environment and present a parallel procedure under the optimal computing budget allocation (OCBA) framework. (OCBA has impressive empirical performance, but does not offer PCS or PGS guarantees.) Chen (2005) tests a sequential pairwise hypothesis testing approach on a local network of computers. Luo et al. (2015) is the best-known and most recent existing method for parallel ranking and selection. It provides an asymptotic (as $\delta \to 0$) PCS guarantee.

In this paper, we (i) identify opportunities and challenges that arise from adopting a parallel computing environment to solve large-scale R&S problems, (ii) propose a Good Selection Procedure (GSP) that solves R&S problems on parallel computers, and (iii) implement our procedure in three different parallel-computing frameworks. We make the following contributions.

Theoretical contributions. We propose a number of design principles that promote efficiency and validity in such an environment, and demonstrate them in a new parallel GSP. GSP showcases the power of these design principles in that it greatly extends the boundary on the size of solvable R&S problems. While the method of Luo et al. (2015) can solve on the order of 10^4 systems, one of our implementations of GSP is capable of solving R&S problems with more than 10^6 systems. Our computational results include such a problem, which we solve in under 6 minutes on 10^3 cores. Another important theoretical contribution of this paper is the redesigned screening method in GSP which, unlike many fully-sequential procedures (Kim and Nelson 2001, Hong 2006), does not rely on the PCS assumption. Accordingly, many systems can lie within the indifference-zone, i.e., have an objective function value within δ of that of System k, as will usually be the case when the number of systems is very large. Our procedure then provides the same PGS guarantee as existing indifference-zone procedures like Nelson et al. (2001) but with far smaller sample sizes.

Practical contributions. The GSP procedure discussed in this paper is intended for any parallel, shared or non-shared memory platform where cores can communicate with each other. As long as no core fails during execution, it should deliver expected results regardless of the hardware specification. The procedure is also amenable to a range of existing parallel computing frameworks. We offer implementations of GSP based on MPI (Message-Passing Interface), Hadoop MapReduce, and Spark, and show how they differ in construction and in performance. The reasons for our choice of implementation frameworks are twofold:

- MPI, MapReduce and Spark are among the most popular and mature platforms for deploying parallel code on a wide range of systems ranging from high performance supercomputers to commodity clusters such as Amazon EC2.
- The three frameworks provide points of comparison between two extreme parallel design philosophies. Broadly speaking, the first philosophy entails low-level tailoring and optimization in the implementation of a parallel procedure (MPI), while the second philosophy delegates as much of the implementation complexity as possible to an underlying structure, sacrificing some performance for ease of implementation (MapReduce & Spark).

As we shall see, MPI is the most efficient of the three, achieving speed and utilization gains of around an order of magnitude over MapReduce and Spark. On the other hand, MapReduce and Spark both offer acceptable performance for large scale problems, and are more robust to reliability issues that may arise in cloud-computing environments where parallel tasks may fail to complete due to unresponsive cores. Of these two, Spark is more efficient.

The remainder of the paper is organized as follows. §2 discusses the design principles followed in creating GSP to promote efficiency and ensure the procedure's validity. §3 describes our multi-stage parallel R&S procedure GSP, and establishes the PGS guarantee. Computational studies in §4 support our assertions on the quality of GSP and its parallel implementations, and point to open-access repositories where the code can be obtained. An electronic companion contains more proof detail, and further information on the MPI, MapReduce and Spark implementations. This paper is a considerable outgrowth of the conference papers Ni et al. (2013, 2014, 2015).

2. Design Principles for Parallel R&S Procedures

R&S procedures are essentially made up of three computational tasks: (1) deciding what simulations to run next, (2) running simulations, and (3) screening (computing statistical estimators and determining which systems are inferior). On a single-core computer, these tasks are repeatedly performed in a certain order until a termination criterion is met. On a parallel platform, multiple cores can simultaneously perform one or several of these tasks.

In this section, we discuss various issues that arise when a R&S procedure is designed for and implemented on parallel platforms to solve large-scale R&S problems. We argue that failing to consider these issues may result in impractically expensive or invalid procedures. We recommend strategies by which these issues can be addressed, and illustrate how we incorporate them in our procedure presented in §3, which iteratively runs Tasks (1) through (3) in multiple stages.

For discussing the design principles for parallel R&S procedures in this section, we consider a parallel computing environment that satisfies the following properties.

Assumption 1. *(Core Independence) A fixed number of processing units ("cores") are employed to execute the parallel procedure. Each core is capable of performing its own set of computations without interfering with other cores unless instructed to do so. Each core has its own memory and does not access the memory of other cores.*

Assumption 2. *(Message-passing) The cores are capable of communicating through sending and receiving messages of common data types and arbitrary lengths.*

Assumption 3. *(Reliability) Cores do not "fail" or suddenly become unavailable. Messages are never "lost".*

Many parallel computer platforms satisfy the first two assumptions, but some are subject to the risk of core failure, which may interrupt the computation in various ways. For clarity, we work under the reliability assumption and defer the design of failure-proof procedures to §4 where we discuss Hadoop MapReduce and Spark. Similar to Luo et al. (2015) and Ni et al. (2013), we

consider a master-worker framework, using a uniquely executed "master" process (typically run on a dedicated "master" core) to coordinate the parallel procedure, and letting other cores (the "workers") work according to the master's instructions. To the extent possible we want to avoid synchronization delays, where one core cannot continue until another core completes its task, as we will see in §4.2.

2.1. Implications of Random Completion Times

Consider the simplest case where only Task (2), running simulations, is run in parallel, and each replication completes in a random amount of time. To construct estimators for a single system simulated by multiple cores, one can either collect a fixed number of replications in a random completion time, or a random number of replications in a fixed completion time (Heidelberger 1988). Heidelberger (1988) and Glynn and Heidelberger (1990, 1991) discuss unbiased estimators of each type. Because a random number of replications collected after a fixed amount of time may not be i.i.d. with the desired distribution upon which much of the screening theory depends (Heidelberger 1988, Glynn and Heidelberger 1991, Ni et al. 2013, Luo et al. 2015), we restrict attention to estimators that produce a *fixed* number of replications in a *random* completion time.

Using estimators that produce a fixed number of replications in a random completion time for parallel R&S places a restriction on the manner in which replications can validly be farmed out to and collected from the workers. Consider the case where more than one core simulates the same system, and replications generated in parallel are aggregated to produce a single estimator. If replications are collected from cores in the order in which replications are completed, then bias can arise from statistical dependencies between completion time and the value of the simulation result, making it hard to establish provable statistical guarantees; see, e.g., Ni et al. (2013, §3.1). In contrast, a valid method is to place the finished replications in a predetermined order and use them as if they are generated following that order, to avoid "re-ordering" of the simulation replications caused by random completion time.

Under this principle, our GSP in §3 ensures that the simulation results generated in parallel are initiated, collected, assembled, and used by the screening routine in an ordered manner. Specifically, in Stage 2 of GSP, when the master instructs a worker to simulate system i for a batch of replications (Step 5(c)), the batch index is also received by the worker. The worker completes the entire batch, i.e., completes a *fixed* number of replications in a *random* amount of time, and sends the statistics back to the master alongside the batch index (Step 5(c)), which signals the batch's pre-determined position in the assembled batch sequence on the master. This step ensures the batch statistics sent to workers for screening (Step 5(d)) follow the exact order in which they were initiated, and constructed estimators are unbiased with the correct distribution. Luo et al. (2015) discuss a similar approach, which they refer to as "vector-filling".

2.2. Allocating Tasks to the Master and Workers

Previous work on parallel R&S procedures (Chen et al. 2000, Yoo et al. 2009, Luo and Hong 2011, Luo et al. 2015) focuses almost exclusively on pushing Task (2), running simulations, to parallel cores. In those procedures, usually the master is solely responsible for Tasks (1) and (3), deciding what simulations to run next and screening, and the workers perform Task (2) in parallel. In this setting, the benefit of using a parallel computing platform is entirely attributed to distributing simulation across parallel cores, hence reducing the total amount of time required by Task (2).

However, the master could potentially become a bottleneck in a number of ways. First, as noted by Luo and Hong (2011), the master can be overwhelmed with messages. Second, for the master to keep track of all simulation results requires a large amount of memory, especially when the number of systems is large (Luo et al. 2015). Finally, when the number of systems is large and simulation output is generated by many workers concurrently, running Tasks (1) and (3) on the master alone may become relatively slow, resulting in a waste of core hours on workers waiting for the master's further instructions. Therefore, a truly scalable parallel R&S procedure should allow its users a simple way to control the level of communication, use the memory efficiently, and distribute as many tasks as possible across parallel cores. In addition, it should perform some form of load-balancing to minimize idling on workers.

2.2.1. Batching to Reduce Communication Load One way to reduce the number of messages handled by the master is to control communication frequency by having the workers run simulation replications in batches and only communicate once after each batch is finished.

Since R&S procedures typically use summary statistics rather than individual observations when screening systems, it may even suffice for the worker to compute and report batch statistics instead of point observations from every single replication. Indeed, a useful property of our statistic for screening systems i and j is that it is updated using only the sample means over the entirety of the most recent batch r, instead of requiring the collection of individual replication outcomes. These sample means can be independently computed on the worker(s) running the rth batch of systems i and j, and the amount of communication needed in reporting them to the master is constant and does not grow with the batch size.

The distribution of batches in parallel must be handled with care. Most importantly, since using a random number of replications after a fixed run time may introduce bias (as discussed in §2.1), a valid procedure should employ a predetermined and fixed batch size for each system, which may vary across different systems. Batches generated in parallel for the same system should be assembled according to a predetermined order, following the same argument used in §2.1. Furthermore, if the procedure requires screening upon completion of every batch, then it is necessary to perform screening steps following the assembled order.

Batching also reduces the amount of switching between simulations of different systems, which may be computationally beneficial (Hong and Nelson 2005).

2.2.2. Allocating Simulation Time to Systems When multiple systems survive a round of screening, R&S procedures need to decide which system(s) to simulate next (possibly on multiple cores), and how many replications to take. While sequential procedures usually sample one replication from the chosen system(s), or multiple replications from a single system, it is natural for a parallel procedure to consider strategies that sample multiple replications from multiple systems. In doing so, the parallel procedure may adopt sampling strategies such that simulation resources are allocated to surviving systems in a most efficient manner.

The best practice in making such allocations depends on the specific screening method. For instance, in Hong (2006) as well as GSP, screening between systems i and j is based on a scaled Brownian motion $B([\sigma_i^2/n_i + \sigma_j^2/n_j]^{-1})$ where $B(\cdot)$ denotes a standard Brownian motion (with zero drift and unit volatility), n_i is the sample size and σ_i^2 is the variance of system i. To drive this Brownian motion rapidly with the fewest samples possible, which accelerates screening, Hong (2006) recommended that the ratio n_i/σ_i be kept equal across all surviving systems.

The above recommendation implicitly assumes that simulation completion time is fixed for all systems, and is suboptimal when completion time varies across systems. Suppose all workers are identical, and each replication of system i takes a fixed amount of time T_i to simulate on any worker. We can then formulate the problem of advancing the above Brownian motion as

$$\max\ [\sigma_i^2/n_i + \sigma_j^2/n_j]^{-1}$$

$$\text{s.t. } n_i T_i + n_j T_j = T$$

which yields the optimal computing time allocation

$$\frac{n_i T_i}{n_j T_j} = \frac{\sigma_i \sqrt{T_i}}{\sigma_j \sqrt{T_j}}. \tag{2}$$

This result is consistent with a conclusion in Glynn and Whitt (1992), that when simulation completion time T_i varies, an asymptotic measure of efficiency per replication is inversely proportional to $\sigma_i^2 E[T_i]$.

In practice, T_i is unknown and possibly random, so both $E[T_i]$ and σ^2 need to be estimated in a preliminary stage. Suppose they are estimated by some estimators $\bar{T}_i$ and S_i^2. Then we recommend setting the batch size for each system i proportional to $S_i/\sqrt{\bar{T}_i}$, following (2).

2.2.3. Distributed screening In fully sequential R&S procedures, e.g., Kim and Nelson (2006b), Hong (2006), each screening step typically involves doing a fixed amount of calculation between every pair of systems to decide if one system is better than another with a certain degree of statistical confidence. The amount of work in each round of screening therefore depends on

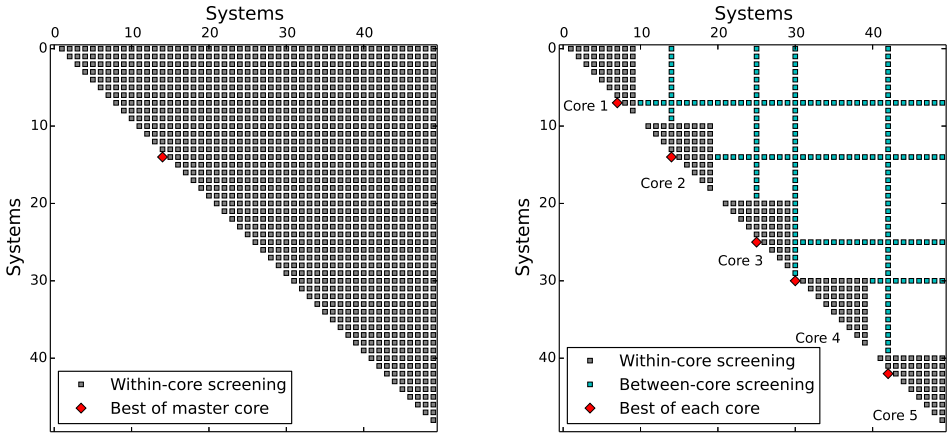
the number of pairs of surviving systems, which is $O(k^2)$, at least at the first stage of screening. The total work in screening over the course of an R&S procedure is typically greater than $O(k^2)$ because of the conservatism introduced in these procedures to establish a statistical guarantee. It is also difficult to quantify, so in this section we refer only to the amount of work in each round of screening, rather than the total amount of screening over the entire run of the R&S procedure.

In the serial R&S literature, the computational cost of each round of screening is assumed to be negligible compared to that of simulation because the number of systems k is usually quite small and each simulation replication may take orders of magnitude longer than the $O(k^2)$ screening operations required in each round of screening. It is therefore tempting to simply have the master handle all screening after the workers complete a simulation batch. This approach can easily be implemented and proven to be statistically valid. However, it may become computationally inefficient, because all workers stay idle while the master screens, so a total of $O(ck^2)$ processing time is wasted at each round of screening, where c is the number of workers. For a large problem with a million systems solved on a thousand cores, the wasted processing time per round of screening can easily amount to thousands of core hours, dramatically reducing the benefits from a parallel implementation. Moreover, if the procedure requires computing and storing in memory some quantities for each system pair (for instance, the variance of differences between systems), then the total memory required may be $O(k^2)$ and exceed the limit for a single core.

It is therefore worth considering strategies that distribute screening among workers. A natural strategy is to assign roughly k/c systems to each worker, and let the worker screen among those systems only, as illustrated in Figure 1. By doing so, each worker screens k/c systems, occupying only $O(k^2/c^2)$ memory, and performing $O(k^2/c^2)$ work in parallel at each round of screening. Hence the wall-clock time for each round of screening is reduced by a factor of c^2.

Under the distributed-screening scheme, not all pairs of systems are compared, so fewer systems may get eliminated. The reduction in effectiveness of screening can be compensated by sharing some good systems across workers. In Figure 1, for example, each core shares its own (estimated)

Figure 1 Comparison of screening methods applied on 50 systems. Each black or green dot represents a pair of systems to be screened. In the left panel, all pairs of screening is done on the master. In the right panel, each worker core gets 10 systems, screens between themselves, and screens its systems against one system from every other worker that has the highest sample mean.



best system with other cores, and each system is screened against other systems on the same core, as well as $O(c)$ good systems from other cores. This greatly improves the chance that each system is screened against a good one, despite the extra work to share those good systems. As illustrated in Figure 1, the additional number of pairs that need to be screened on each core is only $O(k)$ when the best system on each core is shared. Alternatively, the procedure may also choose to share only a smaller number $c' \ll c$ of good systems, so that the communication workload associated with this sharing does not increase as the number of workers increases. The procedure is statistically valid irrespective of the amount of distributed screening, since any eliminated systems would also be eliminated under full, pairwise screening, and our validity proof applies to full, pairwise screening.

The statistical validity of some screening-based R&S procedures (e.g. Kim and Nelson 2001, Hong 2006, Luo et al. 2015) relies on results for the first exit time of a Brownian motion from within a continuation region. Correct implementations of such procedures thus require determination of first exit times of the discrete-time version of a Brownian motion. For statistical validity, screening must then be performed once every replication (or batch of replications) is simulated, and we cannot skip screening after some replications because we may miss a first exit time. The Brownian motion

here is specific to the pair of systems under consideration. Therefore, when the identity of one of the systems within a screening pair changes, as arises at a worker performing screening when the identity of the best system(s) communicated from other workers changes, we need to determine the first exit time of a different set of Brownian motions from those considered thus far. The master then has to communicate all previous replication results of the new estimated-best system(s) to workers, so that they can perform all of the screening steps up to the current replication to ensure validity of the screening. Such "catch-up" screening was used, for instance, in Pichitlamken et al. (2006), in a different context. In §3, we employ a screening method based on a different principle that obviates the need for catch-up screening in GSP.

Besides core hours, distributing screening across workers also saves memory space on the master. In our implementation of GSP, the master keeps a complete copy of batch statistics only for a small number of systems that are estimated to be the best. For a system that is not among the best, the master acts as an intermediary, keeping statistics for only the most recent batches that have not been collected by a worker. Whenever some batch statistics are sent to a worker (for screening in Steps 3(c) or 5(d) of GSP), they can be deleted on the master. This helps to even out memory usage across cores, making the procedure capable of solving larger problems without the need to use slower forms of storage.

2.3. Random Number Stream Management

The validity and performance of simulation optimization procedures relies substantially on the quality and efficiency of (pseudo) random number generators (L'Ecuyer 2015).

To avoid unnecessary synchronization, each core may run its own random number generator independently of other cores. Some strategies for generating independent random numbers in parallel have been proposed in the literature. Mascagni and Srinivasan (2000) consider a class of random number generators that are parameterized so that each valid parametrization is assigned to one core. Karl et al. (2014) adopt L'Ecuyer et al. (2002)'s `RngStream` package, which supports streams and substreams, and demonstrated a way to distribute `RngStream` objects across parallel cores.

Both methods set up parallel random number generation in such a way that once initialized, each core will be able to generate a unique, statistically independent stream of pseudo random numbers, which we denote as U_w, for each $w = 1, 2, \ldots, c$. If a core has to switch between systems to simulate, one can partition U_w into substreams $\{U_w^i : i = 1, 2, \ldots, k\}$, simulating System i using U_w^i only. It follows that for any system i, U_w^i for different w are independent as they are substreams of independent U_w's, so simulation replicates generated in parallel with $\{U_w^i : w = 1, 2, \ldots, c\}$ are also i.i.d. Moreover, if it is desirable to separate sources of randomness in a simulation, it may help to further divide U_w^i into subsubstreams, each used by a single source of randomness.

In practice, one does not need to pre-compute and store all random numbers in a (sub)stream, as long as jumping ahead to the next (sub)stream and switching between different (sub)streams are fast. Such operations are easily achievable in constant computational cost; see L'Ecuyer et al. (2002) for an example.

Although our procedure does not support the use of common random numbers (CRN), it is worth noting that the above framework easily extends to accommodate CRN as follows. Begin by having one identical stream U_0 set up on all cores and partitioning it into substreams $\{U_0(\ell) : 1 \le \ell \le L\}$ for sufficiently large L. Let the master keep variables $\{\ell_i : i = 1, 2, \ldots, k\}$ that count the total number of replications already generated for system i over all workers. Each time the master initiates a new replication of System i on a worker, it instructs the worker to simulate System i using substream $\{U_0(\ell_i + 1)\}$ and adds 1 to ℓ_i. This ensures that for any $\ell > 0$, the ℓth replication of every system is generated by the same substream $\{U_0(\ell)\}$.

3. The Parallel Good Selection Procedure

In this section, we provide a R&S procedure GSP that incorporates the design principles from §2, and is implementable on a wide spectrum of parallel platforms. Our procedure applies to the general case in which the system mean and variance are both unknown and need to be estimated, and does not permit the use of common random numbers. An earlier version of the procedure for known variances is discussed in Ni et al. (2014). We prove that GSP offers a PGS guarantee for normally distributed observations.

3.1. The Setup

GSP consists of four broad stages. In an optional Stage 0, workers run n_0 simulation replications for each system in parallel to estimate completion times, which are subsequently used to try to balance the workload. As discussed in §2.2.2, Stage 0 samples are then dropped and not used to form estimators of μ_i's due to the potential dependence between simulation output and completion time. In Stage 1, a new sample of size n_1 is collected from each system to obtain variance estimates $S_i^2 = \sum_{\ell=1}^{n_1}(X_{i\ell} - \bar{X}_i(n_1))^2/(n_1-1)$, where $\bar{X}_i(n) = \sum_{l=1}^{n} X_{il}/n$. Prior to Stage 2, obviously inferior systems are screened. In Stage 2, the workers iteratively visit the remaining systems and run additional replications, exchange statistics, and independently perform screening over a subset of systems until all but one are eliminated, or a pre-specified limit on sample size is reached. The screening rule we use is an extension of one due to Hong (2006); see that paper for a lucid explanation of the method and its validity. Nevertheless, our development and proofs are self contained. The screening rule and the limit on sample size are jointly chosen so that inferior systems can be eliminated efficiently, while the best system k survives with high probability regardless of the configuration of true means $\mu_1, \ldots, \mu_k$. Finally, in Stage 3, all systems surviving Stage 2 enter a Rinott (1978) procedure where a maximum sample size is calculated, additional replications are simulated if necessary, and the system with the highest sample mean is selected as the best.

The sampling rules used in Stages 0, 1, and 3 are relatively straightforward, for they each require a fixed number of replications from each system. In Stage 2, where the procedure iteratively switches between simulation and screening, a sampling rule needs to be specified to fix the number of additional replications to take from each system before each round of screening. Prior to the start of Stage 2 we define increasing (in r) sequences $\{n_i(r) : i = 1, 2, \ldots, k, r = 0, 1, \ldots\}$ giving the total number of replications to be collected for System i by Batch r, and let $n_i(0) = n_1$ since we include the Stage 1 sample in mean estimation. Following the discussion in §2.2.2 where we recommend that the batch size for System i should be proportional to $S_i/\sqrt{\bar{T}_i}$ in order to efficiently allocate simulation budget across systems, we use

$$n_i(r) = n_1 + r\left\lceil \beta \left(\frac{S_i}{\sqrt{\bar{T}_i}}\right) \Big/ \left(\frac{1}{k}\sum_{j=1}^{k} \frac{S_j}{\sqrt{\bar{T}_j}}\right)\right\rceil \tag{3}$$

where $\bar{T}_i$ is an estimator for the simulation completion time of System i obtained in Stage 0 if available, and β is the average batch size and is specified by the user.

The parameters for the procedure are as follows. Before the procedure initiates, the user selects an overall confidence level $1-\alpha$, type-I error rates α_1, α_2 such that $\alpha = \alpha_1 + \alpha_2$, an indifference-zone parameter δ, Stage 0 and Stage 1 sample sizes $n_0, n_1 \geq 2$, and average Stage 2 batch size β. The user also chooses $\bar{r} > 0$ as the maximum number of iterations in Stage 2, which governs how much simulation budget to spend in iterative screening before moving to indifference-zone selection in Stage 3.

Typical choices for error rates are $\alpha_1 = \alpha_2 = 0.025$ for guaranteed PGS of 95%. Unequal splits of these error rates may yield improved performance, but we have not explored that possibility. The indifference-zone parameter δ is usually chosen within the context of the application, and is often referred to as the smallest difference worth detecting. The sample sizes n_0 and n_1 are typically chosen to be small multiples of 10, with the view that these give at least reasonable estimates of the runtime per replication and the variance per replication.

For non-normal simulation output, we recommend setting $\beta \geq 30$ to help ensure normally distributed batch means (see Assumption 4 in Section 3.3). The parameter β also helps to control communication frequency so as not to overwhelm the master with messages. Let T_{sim} be a crude estimate of the average simulation time (in seconds) per replication, perhaps obtained in a debugging phase. Then ideally the master communicates with a worker every $\beta T_{\text{sim}}/c$ seconds. If every communication takes T_{comm} seconds, the fraction of time the master is busy is $\rho = cT_{\text{comm}}/\beta T_{\text{sim}}$. We recommend setting β such that $\rho \leq 0.05$, in order to avoid significant waiting of workers.

To choose $\bar{r}$, note that small $\bar{r}$ implies insufficient screening, whereas a large $\bar{r}$ may be too conservative. We are exploring this tradeoff, which is highly involved, in work not reported here. In the interim, we have found that good results are obtained by choosing $\bar{r}$ such that a fair amount of simulation budget (no more than 20% of the sum of Rinott sample sizes) is spent in the iterative screening stages. This rule of thumb is admittedly ad-hoc, but the numerical experiments suggest that it is effective.

Under these general principles, our choices of $(\beta = 100, \bar{r} = 10)$ and $(\beta = 200, \bar{r} = 5)$ in the experiments in §4 work reasonably well on our testing platform, but it is conceivable that other values could improve performance.

Finally, we define some quantities used in the iterative screening stages. Let η be the solution to

$$E\left[2\bar{\Phi}\left(\eta\sqrt{R}\right)\right] = 1 - (1-\alpha_1)^{\frac{1}{k-1}}, \tag{4}$$

where $\bar{\Phi}(\cdot)$ denotes the complementary standard normal distribution function, and R is the minimum of two i.i.d. χ^2 random variables, each with $n_1 - 1$ degrees of freedom. Denote the distribution function and density of such a χ^2 random variable by $F_{\chi^2_{n_1-1}}(x)$ and $f_{\chi^2_{n_1-1}}(x)$, respectively. Then R has density $f_R(x) = 2[1 - F_{\chi^2_{n_1-1}}(x)]f_{\chi^2_{n_1-1}}(x)$. Also, for any two systems $i \neq j$, define

$$\begin{aligned}
t_{ij}(r) &= \left[\frac{\sigma_i^2}{n_i(r)} + \frac{\sigma_j^2}{n_j(r)}\right]^{-1}, & Z_{ij}(r) &= t_{ij}(r)[\bar{X}_i(n_i(r)) - \bar{X}_j(n_j(r))], \\
\tau_{ij}(r) &= \left[\frac{S_i^2}{n_i(r)} + \frac{S_j^2}{n_j(r)}\right]^{-1}, & Y_{ij}(r) &= \tau_{ij}(r)[\bar{X}_i(n_i(r)) - \bar{X}_j(n_j(r))], \\
a_{ij}(\bar{r}) &= \eta\sqrt{(n_1-1)\tau_{ij}(\bar{r})}.
\end{aligned}$$

3.2. Good Selection Procedure under Unknown Variances

We dedicate subsets of systems to cores for *screening* purposes, but the *simulation* of a given system can be performed on any worker. We do this because simulation is far more computationally expensive than screening, so by distributing the simulation effort we ensure high core utilization. In the process of screening in Stage 2, let s_i be the number of batches of System i that have been *simulated*. Also, let r_w be the number of batches that have been *screened* on worker w. Since systems are dedicated to workers for the purpose of screening, this value is the same for all systems assigned to worker w, and is thus indexed by w, rather than by systems.

1. **(Stage 0), optional** Master assigns systems to workers, so that each system i is simulated for n_0 replications and the average simulation completion time $\bar{T}_i$ is reported to the master.
2. **(Stage 1)** Master assigns systems to load-balanced (using $\bar{T}_i$ if available) simulation groups G_1^w for $w = 1, \ldots, c$. Let $\mathcal{I} \leftarrow \mathcal{S}$ be the set of surviving systems.

3. For $w = 1, 2, \ldots, c$ in parallel on workers:
   (a) Sample $X_{i\ell}$, $\ell = 1, 2, \ldots, n_1$ for all $i \in G_1^w$.
   (b) Compute Stage 1 sample means and variances $\bar{X}_i(n_1)$ and S_i^2 for $i \in G_1^w$.
   (c) Screen within Group G_1^w: System i is eliminated (and removed from $\mathcal{I}$) if there exists a system $j \in G_1^w, j \neq i$ such that $Y_{ij}(0) < -a_{ij}(\bar{r})$.
   (d) Report survivors, together with their Stage 1 sample means $\bar{X}_i(n_i(0))$ and variances S_i^2, to the master.
4. **(Stage 2)** Let $G_1 \leftarrow \mathcal{I}$ be the set of systems surviving Stage 1. Master computes sampling rule (3) using S_i^2 obtained in Stage 1, and divides G_1 into approximately load-balanced screening groups G_2^w for $w = 1, \ldots, c$. Set $s_i \leftarrow 0, i \in G_1$, i.e., set the count of the number of simulated batches in Stage 2 for System i to be 0. Also set $r_w \leftarrow 0$ for all workers w, i.e., set the count of the number of screened batches in Stage 2 for all workers to be 0.
5. For $w = 1, 2, \ldots, c$ in parallel on workers, iteratively switch between simulation and screening as follows (this step entails some communication with the master, the details of which are omitted):
   (a) Check termination criteria with the master: if $|\mathcal{I}| = 1$ (only one system remains) or $r_{\tilde{w}} \geq \bar{r}$ for all workers $\tilde{w}$ (each worker has screened up to $\bar{r}$, the maximum number of batches allowed), go to Step 6 (Stage 3); otherwise continue to Step 5(b).
   (b) Decide to either simulate more replications or perform screening based on available results: check with the master whether the $(r_w + 1)$th batch has completed for all systems $i \in G_2^w$ and $|G_2^w| > 1$, if so, go to Step 5(d), otherwise go to Step 5(c).
   (c) Retrieve the next system i in G_1, which may not belong to G_2^w, from the master and simulate its $(s_i + 1)$th batch, i.e., simulate System i for an additional $n_i(s_i + 1) - n_i(s_i)$ replications. Set $s_i \leftarrow s_i + 1$. Report simulation results to the master. Go to Step 5(a).
   (d) Screen within G_2^w as follows. Retrieve necessary statistics for systems in G_2^w from the master (recall that a system in G_2^w is not necessarily simulated by worker w). Let $r_w \leftarrow$

$r_w + 1$. System $i \in G_2^w$ is eliminated if $r_w \leq \bar{r}$ and there exists a system $j \in G_2^w, j \neq i$ such that $Y_{ij}(r_w) < -a_{ij}(\bar{r})$. Also use a subset of systems from other workers, e.g., those with the highest sample mean from each worker, to eliminate systems in G_2^w. Remove any eliminated system from G_2^w and $\mathcal{I}$. Go to Step 5(a).

6. **(Stage 3)** Let $G_2 \leftarrow \mathcal{I}$ be the set of systems surviving Stage 2. If $|G_2| = 1$, select the single system in G_2 as the best. Otherwise, set $h = h(1-\alpha_2, n_1, k)$, where the function $h(\cdot)$ gives Rinott's constant (see, e.g., Bechhofer et al. 1995, Chapter 2.8). For each remaining system $i \in G_2$, compute $N_i = \max\{n_i(\bar{r}), \lceil (hS_i/\delta)^2 \rceil\}$, and take an additional $\max\{N_i - n_i(\bar{r}), 0\}$ sample observations in parallel. Once a total of N_i replications have been collected in Stages 1 through 3 for each $i \in G_2$, select the system K with the highest $\bar{X}(N_K)$ as the best.

3.3. Guaranteeing Good Selection

The probabilistic guarantee relies on the following assumption on the distribution of simulation output, which is common in the sequential R&S literature.

ASSUMPTION 4. *For each system $i = 1, 2, \ldots, k$, the simulation output random variables $\{X_{i\ell}, \ell = 1, 2, \ldots\}$ are i.i.d. replicates of a random variable X_i having a normal distribution with finite mean μ_i and finite variance σ_i^2, and are mutually independent for different i.*

In §2.1 we gave conditions under which the simulation output generated by parallel cores satisfies this assumption.

Our proof of good selection also relies, among other things, on properties of Rinott's procedure. The eventual sample sizes for surviving systems in Stage 3 may be larger than those employed in Rinott's procedure because our screening stage may run too long. Accordingly, we write our proof for a slightly modified version of GSP, called GSP′. Procedure GSP′ is identical to GSP except in Stage 3, the runlength N_i is computed as $\max\{n_1, \lceil (hS_i/\delta)^2 \rceil\}$, where n_1 is the sample size used in Stage 1. This ensures exact agreement between the sample sizes used in GSP′ and in Rinott's procedure for systems surviving to Stage 3. We view this imperfection in the result as a technicality,

and we conjecture that the GSP procedure provides the same good-selection guarantee as described in Theorem 1. Our discussion after the presentation of Theorem 1 assumes this conjecture is true.

THEOREM 1. *Under Assumption 4, the procedure* GSP′ *selects a system* K *that satisfies* $\mu_k - \mu_K \leq \delta$ *with probability at least* $1-\alpha$.

Sketch of Proof See §EC.1 for the full proof. First, we show that the best system, System k, survives the iterative screening in Stages 1 and 2 with probability at least $1-\alpha_1$. Indeed, conditioning on the Stage 1 variance estimates, we can, for any system $i \neq k$, relate the batch statistics $Z_{ki}(r) : r = 0, 1, \ldots, \bar{r}$ to a properly scaled Brownian motion with drift $\mu_k - \mu_i \geq 0$. Then, using the reflection principle of Brownian motion, we can upper-bound the probability that the scaled Brownian motion traverses the threshold $-a$ before some time t, and thence the probability that $Y_{ki}(r)$ falls below $-a_{ki}(\bar{r})$ in some rth iteration where $r \leq \bar{r}$, which is the criterion used to eliminate System k in the iterative screening stages. The construction of the continuation region parameter η ensures that the unconditional probability of eliminating k is no greater than α_1.

Second, Stage 3 is closely related to the Rinott (1978) procedure with confidence level $1-\alpha_2$. A coupling argument that assumes that *all* systems, and not just those that survive to Stage 3, receive Rinott sample sizes, together with Theorem 1 of Nelson and Matejcik (1995), implies that with high probability, Stage 3 delivers a good selection amongst the surviving systems when System k survives to Stage 3, completing the proof. □

The key difference between the screening methods used in GSP and the $\mathcal{KN}$ family of procedures (Kim and Nelson 2001, Hong and Nelson 2005, Hong 2006) is that the $\mathcal{KN}$ family relies on the PCS assumption ($\mu_k - \mu_i \geq \delta > 0$ for all $i \neq k$) to guarantee PCS, whereas our approach does not. Therefore, GSP works for any indifference-zone parameter $\delta > 0$, and when there exist multiple systems i such that $\mu_i \geq \mu_k - \delta$, GSP is guaranteed to select one such system with probability at least $1-\alpha$.

In §EC.2, we show that the parameter η, as defined in (4), grows roughly linearly with the number of systems. η controls how quickly we eliminate inferior systems with the screenings in Stage 2: thus our bound on η suggests that the total cost of the GSP structure does not become prohibitively expensive as the alternative space become large.

4. Computational Study

In this section, we discuss our parallel computing environments, parallel implementations of GSP, our test problem, and the results of our numerical experiments.

4.1. Parallel Computing Environment

We implement GSP in two parallel computing environments.

HPC Cluster The bulk of our experiments are conducted on Extreme Science and Engineering Discovery Environment (XSEDE)'s Wrangler high-performance cluster. Each node in this cluster has two 12-core Intel Xeon E5-2680 v3 processors and 128 GB of memory and the cluster runs a Linux CentOS 6.3 operating system (Texas Advanced Computing Center 2016). The processors on Wrangler are highly reliable; we have never observed a core failure.

Cloud Computing We use Amazon's EC2 service, which is a lower cost, freely available alternative to an HPC cluster. The possible configurations vary; see Amazon (2016). We used m3.medium and m3.2xlarge instances, consisting of Intel Xeon E5-2670 v2 processors with between 3.75 and 30 GB of memory. Core failures are possible, but we did not see any in our experiments.

4.2. Parallel Implementations of GSP

We discuss three implementations of GSP using three distributed computing frameworks that span the gamut of trade-offs between customizability and ease-of-use. Although we primarily test the implementations on Wrangler, all three can be configured to run on a wide range of parallel platforms from multi-core personal computers to the Amazon EC2 cloud.

4.2.1. MPI Message-Passing Interface (MPI) is a distributed-memory parallel-programming framework with libraries available in C/C++ and Fortran and is the de-facto standard for parallel programming on many high-performance parallel clusters including Wrangler. Using MPI, programs operate in an environment where Assumptions 1 and 2 hold. The method by which parallel cores independently execute instructions and communicate through message-passing can be highly customized, allowing us a great deal of flexibility in controlling the flow of jobs to workers. Our

MPI implementation, which is fully described in §EC.3, uses the mvapich2 library. Its source code and documentation are hosted in the open-access repository Ni (2015b).

Our MPI implementation is designed primarily for high-performance clusters like Wrangler and does not detect and manage core failures. Therefore, for cheap and less reliable parallel platforms, the MPI implementation needs to be augmented with a "fault-tolerant" mechanism in order to allow the procedure to continue even if some task fails on a core. This motivates us to seek alternative programming tools such as MapReduce or Spark that handle failures automatically.

4.2.2. Hadoop MapReduce and Apache Spark Both MapReduce (Dean and Ghemawat 2008) and Apache Spark (Zaharia et al. 2010) are parallel computing frameworks that offer a relatively simple and standardized functional language for writing parallel code. Hadoop is an open source system implementing the MapReduce parallel computing architecture. Hadoop and Spark have become mainstream for processing large amounts of data due to the following advantages.

- **Simplicity.** They allow users to solely focus on mapping the parallel program to the framework's functional programming language, without explicitly handling the complex details of message-passing and distributing workload to cores, a task that is completely handled by the framework itself. Our GSP implementation can be expressed with Spark in less than 400 lines, roughly an order of magnitude less than with MPI.
- **Portability.** They can be easily deployed with minimal changes on a wide range of parallel computer platforms.
- **Resilience to core failures.** On less reliable hardware where there is a non-negligible probability of core failures, both systems can automatically reload any failed task on another worker to guide the parallel job towards completion.

A disadvantage of these frameworks is that they lack the flexibility of MPI, where users can customize virtually every aspect of the parallel computation. Indeed, they exhibit limitations that may potentially reduce their efficiency for highly iterative algorithms such as parallel R&S procedures.

- **Synchronization.** Hadoop and Spark constrain the implementations to more synchronization steps than, in principle, needed for R&S procedures. If load-balancing is difficult, for instance

Table 1 Major differences between GSP implementations

Task	MPI	Hadoop MapReduce	Spark
Master	Explicitly coded	Automated	Automated
Message-passing	Explicitly coded	Automated	Automated
Synchronization	Once after each stage	Four times in every Stage 2 iteration	Twice in every Stage 2 iteration
Simulation	Each worker simulates one system per iteration	Each worker simulates multiple systems per iteration	
Load-balancing	Via asynchronous communications between the master and a single worker	By assigning approximately equal number of systems (Mappers) to each worker in each synchronized iteration	
Batch statistics and random number seeds	Always stored in workers' memory	Written to hard disk after each iteration	Either, optimized to size of data
Protection against core failures	Possible but hard	Automated	

as a result of random simulation completion times, then this unneccessary synchronization can waste core hours.

- **Load Balancing.** Load-balancing is handled by these frameworks in a nearly black-box way; the user only specifies the number of workers to be deployed and the system attempts to distribute tasks evenly across them. With MPI, we are able to define a custom load balancing policy that is specifically optimized for a given R&S procedure.

Table 1 summarizes some of the critical differences between MPI, MapReduce and Spark. §EC.4 and §EC.5 provide more details of our MapReduce and respectively, Spark, implementations.

Given the observations above, the a priori expectation of our computational experiments is that, for iterative procedures, a highly optimized MPI approach will outperform a Hadoop or Spark one; thus the question we seek to answer is not which is fastest, but whether these frameworks can offer most of the speed of MPI along with being simpler to use and deploy in practice.

In our experiments Spark strictly dominates Hadoop in terms of performance. This is not surprising as Spark is designed as an improved sequel to Hadoop MapReduce. We thus focus on the more interesting comparison between MPI and Spark, but report additional experiments with Hadoop in Section EC.4.

4.3. Test Problem

We test R&S procedures on a throughput-maximization problem taken from `SimOpt.org` (Henderson and Pasupathy 2014). In this problem, we solve

$$\max_x E[g(x;\xi)] \tag{5}$$

$$\text{s.t. } r_1 + r_2 + r_3 = R$$

$$b_2 + b_3 = B$$

$$x = (r_1, r_2, r_3, b_2, b_3) \in \{1, 2, \ldots\}^5$$

where the function $g(x;\xi)$ represents the random throughput of a three-station flow line with finite buffer storage in front of Stations 2 and 3, denoted by b_2 and b_3 respectively, and an infinite number of jobs in front of Station 1. The processing times of each job at stations 1, 2, and 3 are independently exponentially distributed with service rates r_1, r_2 and r_3, respectively. The overall objective is to maximize expected steady-state throughput by finding an optimal (integer-valued) allocation of buffer and service rate.

For each choice of the problem parameters $R, B \in \mathbb{Z}^+$, the number of feasible allocations is finite and can be easily computed. We consider three problem instances with very different sizes presented in Table 2. Since the service times are all exponential, we can analytically compute the expected throughput of each feasible allocation by modeling the system as a continuous-time Markov chain. Furthermore, it can be shown that $E[g(r_1, r_2, r_3, b_2, b_3; \xi)] = E[g(r_3, r_2, r_1, b_3, b_2; \xi)]$ for any feasible allocation $(r_1, r_2, r_3, b_2, b_3)$, so the problem may have multiple optimal solutions. Therefore, this is a problem for which the PCS assumption $\mu_k - \mu_{k-1} \geq \delta > 0$ can be violated and R&S procedures that only guarantee correct selection might be viewed as heuristics.

In each simulation replication we warm up the system for 2,000 released jobs starting from an empty system before observing the simulated throughput to release the next 50 jobs. This is not an efficient way to estimate steady-state throughput compared to taking batch means from a single long run, but it suits our purpose which is to obtain i.i.d. random replicates in parallel. Due to the fixed number of jobs, the wall-clock time for each simulation replication exhibits low variability.

Table 2 Summary of three instances of the throughput maximization problem.

(R, B)	Number of systems k	Highest mean μ_k	pth percentile of system means $p=75$	$p=50$	$p=25$	No. of systems in $[\mu_k - \delta, \mu_k]$ $\delta=0.01$	$\delta=0.1$	$\delta=1$
$(20, 20)$	3,249	5.78	3.52	2.00	1.00	6	21	256
$(50, 50)$	57,624	15.70	8.47	5.00	3.00	12	43	552
$(128, 128)$	1,016,127	41.66	21.9	13.2	6.15	28	97	866

4.4. Numerical Experiments

We now explore the performance of GSP on the throughput maximization test problem through an ensemble of experiments that vary across several dimensions: R&S procedure used, computing environment, parallelization framework, problem size and length of simulation time per replication. We do not attempt a "full factorial" experiment design, instead pursuing three main directions:

- Comparing the performance of GSP versus existing parallelizable R&S procedures on a variety of platform, framework and problem-parameter configurations.
- Comparing the efficiency (in terms of wall-clock time and utilization) of the MPI and Spark implementations of GSP running on the same computational environment.
- Investigating the effects of longer and more variable simulation times on the efficiency of both MPI and Spark.

4.4.1. GSP vs Existing Parallel Procedures GSP is motivated by an earlier computational study by Ni et al. (2014), which compares the performance of two parallel procedures, NHH (Ni et al. 2013) and NSGS$_p$ (Ni et al. 2014). NHH is a parallel procedure that adopts the fully-sequential serial procedure proposed in Hong (2006), and only provides a PCS guarantee. It can be viewed as a variant of GSP using a different screening method and without a Rinott-like Stage 3. NSGS$_p$ is a parallel implementation of the NSGS procedure (Nelson et al. 2001), and is a simplification of GSP without the iterative screening Stage 2.

Comparison of MPI Implementations on Wrangler We implement all three procedures using MPI and test them on different instances of the throughput maximization problem using our HPC cluster. We first run 100 macro replications of the small test problem with $k = 3,249$ to test the quality of the solutions returned by the three procedures. Figures 2 and 3 plot the histogram of

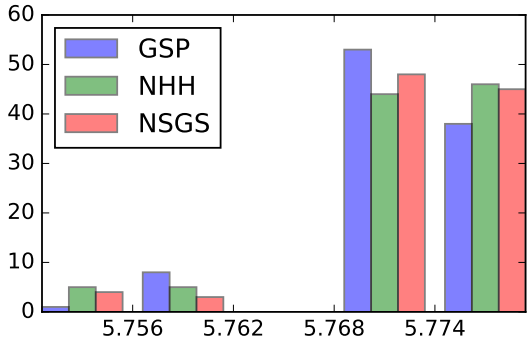


Figure 2 Histogram of 100 macro replications of the small problem instance with $k = 3,249$ on 64 cores, using parameters $\alpha_1 = \alpha_2 = 2.5\%$, $\beta = 100$, $\bar{r} = 10$, $\delta = 0.1$.

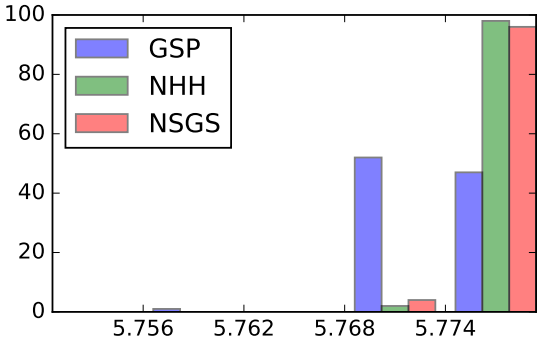


Figure 3 Histogram of 100 macro replications of the small problem instance with $k = 3,249$ on 64 cores, using parameters $\alpha_1 = \alpha_2 = 2.5\%$, $\beta = 100$, $\bar{r} = 10$, $\delta = 0.01$.

the returned solution's mean for an optimality gap tolerance of $\delta = 0.1$ and respectively $\delta = 0.01$. It appears that GSP has a higher chance to return lower-quality solutions that are still consistent with the PGS guarantee, reflecting greater efficiency than the more conservative NSGS and NHH.

We report the performance of these procedures in terms of total wall-clock time and simulation replications required to find a solution in Table 3. Our previous runs on the smaller test problems suggest that the variation in these two measures between multiple runs of the entire selection procedure is limited, so we only present results from a single replication to save core hours.

Ni et al. (2014) argue that NHH tends to devote excessive simulation effort to systems with means that are very close to the best, whereas NSGS_p has a weaker screening mechanism but its Rinott stage can be effective when used with a large δ, which is associated with higher tolerance of an optimality gap. GSP, by design, combines iterative screening with a Rinott stage. Like NSGS_p, we expect that GSP will cost less with a large δ as the Rinott sample size is $O(1/\delta^2)$, but its improved screening method should eliminate more systems than NSGS_p before entering the Rinott stage. Therefore, we expect GSP to work particularly well when a large number of systems exist both inside and outside the indifference zone. This intuition is supported by the outcomes of the medium and large test cases with $\delta = 0.1$, when GSP outperforms both NHH and NSGS_p.

Table 3 A comparison of procedure costs using parameters $n_0 = 20$, $n_1 = 50$, $\alpha_1 = \alpha_2 = 2.5\%$, $\beta = 100$, $\bar{r} = 10$. (Results to 2 significant figures). We used our MPI implementation and deployed it on Wrangler.

Configuration	δ	Procedure	Wall-clock time (sec)	Total number of simulation replications ($\times 10^6$)
3,249 systems on 144 cores	0.01	GSP	14	4.5
		NHH	9.3	2.7
		NSGS$_p$	180	18
	0.1	GSP	2.0	0.55
		NHH	1.8	0.47
		NSGS$_p$	2.5	0.35
57,624 systems on 144 cores	0.01	GSP	960	320
		NHH	300	93
		NSGS$_p$	7,900	2,000
	0.1	GSP	35	11
		NHH	41	12
		NSGS$_p$	120	26
1,016,127 systems on 960 cores	0.1	GSP	310	420
		NHH	1,600	430
		NSGS$_p$	6,700	4,300

Comparison of Spark Implementations on EC2 It is of practical importance to see whether our procedure performs favorably versus incumbent procedures in cloud computing environments, which are more widely available than HPC clusters. Whereas MPI often requires cluster-specific software configuration, Apache Hadoop or Apache Spark are more natural choices for parallel computing on the cloud, as they both provide simple tools for the user to conveniently launch clusters on EC2 that are automatically configured for cloud computing. Here we run our Spark implementation of GSP and NSGS$_p$ procedures on EC2. The results are given in Table 4. For the largest problem instance, NSGS$_p$ demands an excessive number of replications, so we terminate once the required Rinott sample size is computed, and do not actually run the simulation replications.

We measure how efficiently the implementations use available cores to generate simulation replications through *core utilization* as

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

The higher the utilization, the less overhead the procedure spends on communication and screening.

Table 4 shows that GSP is a favourable choice with cloud computing, where it offers cost savings compared to a naïvely parallel procedure like NSGS$_p$. GSP solves all 3 instances of the test problem

Table 4 A comparison of GSP and NSGS on Amazon EC2 with Spark 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. (Results to 2 significant figures). For the examples with 3249 systems and 57,624 systems, we used 32 m3.medium instances, while for 1,016,127 systems we used 60 m3.2xlarge instances; see Amazon (2016) for details of these configurations.

Configuration	Version	β	Number of replications ($\times 10^6$)	Wall-clock time (sec)	Total time Simulation ($\times 10^3$ sec)	Utilization %
3,249 systems	GSP	100	0.45	76	0.44	9.0
on 64 cores		200	0.58	70	0.56	13
	$NSGS_p$	100	0.62	52	0.58	17
		200	0.62	53	0.58	17
57,624 systems	GSP	100	9.9	410	8.6	33
on 64 cores		200	13	460	11	38
	$NSGS_p$	100	46	1500	39	40
		200	46	1400	39	42
1,016,127 systems	GSP	100	350	740	190	52
on 480 cores		200	410	840	220	54
	$NSGS_p$	100	7600	Did not finish		
		200	7600	Did not finish		

Table 5 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. (Results to 2 significant figures)

Configuration	β	Version	Number of replications ($\times 10^6$)	Wall-clock time (sec)	Total time Simulation ($\times 10^3$ sec)	Utilization %
3,249 systems	100	Spark	0.45	35	0.30	5.9
on 144 cores		MPI	0.56	2.4	0.24	68
	200	Spark	0.58	32	0.37	7.9
		MPI	0.74	3.1	0.31	69
57,624 systems	100	Spark	9.9	110	5.5	36
on 144 cores		MPI	11	35	4.7	94
	200	Spark	13	120	7.1	41
		MPI	15	45	6.2	95
1,016,127	100	Spark	350	700	190	56
systems		MPI	440	460	190	85
on 480 cores	200	Spark	410	810	220	57
		MPI	520	490	220	94

with a much smaller sample size than $NSGS_p$. Although $NSGS_p$ achieves higher core utilization as it does not feature iterative screening and hence requires less synchronization, GSP's significant sample-size advantage allows it to more quickly solve the medium and large instances.

4.4.2. A Comparison of MPI and Spark Versions of GSP We now focus on GSP and test its implementations discussed in §4.2. By default, Spark does not dedicate each worker to a single task. It may simultaneously launch several tasks on a single worker, or even use workers that have different hardware configurations. In any of theses cases, simulation completion times may be highly variable and time-varying. Therefore, Stage 0 of GSP (estimation of simulation run time) is dropped from our Spark (and Hadoop) implementations. We also remove Stage 0 from the MPI version to ensure a fair comparison.

In our initial experiments, we observed that the master could become overwhelmed by communication with the workers in the screening stages of our MPI implementation. We fixed this problem by screening using only the 20 best systems from other cores in MPI, versus the best systems from *all* other cores in Spark. While less screening is not a negligible effect, our results suggest that it is dominated by the time spent in simulation.

In Table 5 we report the number of simulation replications, wall-clock time, and utilization for each of the GSP implementations. The MPI implementation takes less wall-clock time than Spark to solve every problem instance, although it requires slightly more replications due to its asynchronous and distributed screening. The gap in wall clock times narrows to within a factor of 2 as the batch size β and/or the system-to-core ratio are increased. Similarly, the MPI implementation also yields much higher utilization. Compared to MPI, the Spark version utilizes core hours less efficiently but again its utilization significantly improves as we double batch size and increase the system-to-core ratio. This behavior can mainly be attributed to the synchronization overhead incurred by the Spark implementation. While it does not seem to increase proportionally with problem sizes, it can still harm the performance, particularly for small or medium problems where it becomes a larger share of the total computation.

Figure 4 plots how wall-clock times scale versus the number of machines used by our MPI and Spark experiments. We are interested in deviations from the "ideal", linearly decreasing scaling. From this perspective, Spark scales significantly more poorly than MPI as we add more machines.

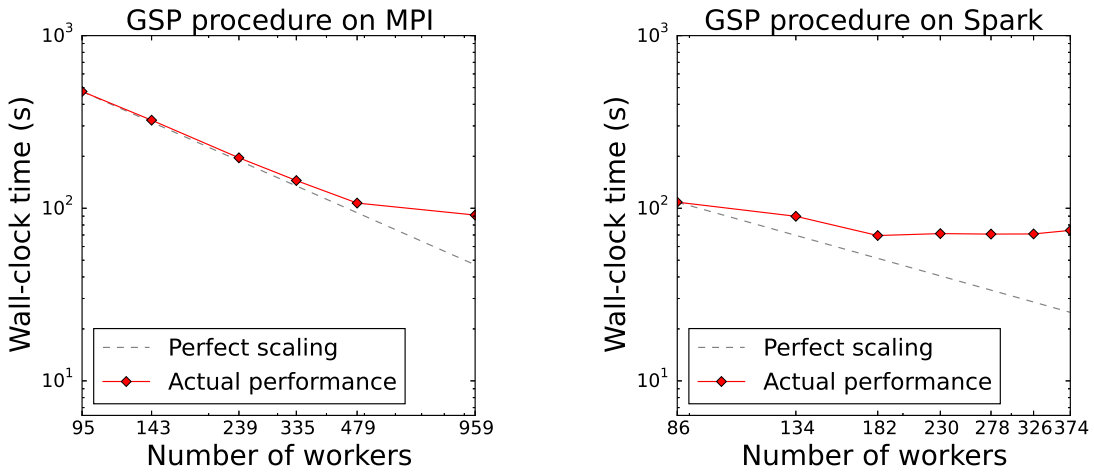


Figure 4 Scaling performances of GSP implementations on medium instance with $k = 57,624$, using parameters $\alpha_1 = \alpha_2 = 2.5\%$, $\delta = 0.1$, $\beta = 100$, $\bar{r} = 10$, and warm-up period of each job equal to 2,000.

This shows that removing the need to synchronize and balancing the load between workers, which is achieved by our MPI implementation, can make a real difference if one wishes to solve a problem faster by using more workers.

4.4.3. Robustness to Unequal and Random Run Times We observe from our experiments that both Hadoop MapReduce and Spark allocate approximately equal numbers of simulation replications to each worker. Together with the fact that simulation run times per replication are nearly constant for our test problems, we see that the computational workload in an iteration should be fairly balanced. However, since both require frequent synchronization, we would expect synchronization delays that would greatly reduce utilization if the simulation run times exhibit enough variation that one worker's load of replications takes much longer than the others.

To verify this conjecture, we design two additional computational experiments. In the first, we increase the warm-up period for each system to $20,000$ to artificially lengthen the running time of a replication. The results are reported in Table 6. Compared to the original array of experiments in Table 5 with a $2,000$ burn-in time, we see that utilization increases significantly, while wall clock time scales up by the expected factor of 10. We attribute the increase in utilization to the fact that as we uniformly increase burn-in times, cores can spend a larger portion of their time efficiently simulating rather than communicating or waiting for synchronization.

Table 6 Performance of GSP under a longer burn-in period, 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. (Results to 2 significant figures).

Configuration	Version	β	Number of replications ($\times 10^6$)	Wall-clock time (sec)	Total time Simulation ($\times 10^3$ sec)	Utilization %
3,249 systems	Spark	100	0.44	90	2.3	18
on 144 cores		200	0.60	110	3.2	21
	MPI	100	0.62	20	2.3	81
		200	0.62	28	3.2	80
57,624 systems	Spark	100	10	1000	55	38
on 144 cores		200	14	1400	75	37
	MPI	100	10	310	46	97
		200	15	450	62	97

Table 7 A comparison of GSP implementations using a random number of warm-up job releases $\exp(X)$, where $X \sim N(\mu, \sigma^2)$. We use parameters $\delta = 0.1$, $n_0 = 50$, $\alpha_1 = \alpha_2 = 2.5\%$, $\beta = 200$, $\bar{r} = 5$. (Results to 2 significant figures)

Configuration	μ	σ^2	Version	Wall-clock time (sec)	Utilization %
3,249 systems	7.4	0.5	Spark	33	7.9
on 144 cores			MPI	5.4	55
	6.6	2.0	Spark	35	7.3
			MPI	4.7	60
57,624 systems	7.4	0.5	Spark	130	41
on 144 cores			MPI	68	82
	6.6	2.0	Spark	130	41
			MPI	64	82

In the second experiment we introduce variability in replication times by warming up each system for a random number of job releases W. We take W to be (rounded) log-normal, parameterized so that the average warm-up period is approximately 2,000, in the hope that the heavy tails of the log-normal distribution will lead to occasional large run times that might slow down the entire procedure. Parameters of the (rounded) log-normal distribution and the results of the experiment are given in Table 7.

We observe very similar wall-clock time and utilization in all instances compared to the base cases in Table 5 where we used fixed warm-up periods. Both implementations seem quite robust against the additional randomness in simulation times, despite our intuition that the Spark version might be noticeably impacted due to additional synchronization waste. A potential explanation is that as each core is allocated at least 50 systems and each system is simulated for an average of 200 replications in each step, the variation in single-replication completion times is averaged out.

Rather extreme variations would be required for Spark to suffer a sharp performance decrease. For problems with much longer simulation times and a lower systems-to-core ratio, the averaging effect might not completely cancel the variations across simulation run times.

5. Concluding Remarks

We develop a selection procedure that can solve R&S problems in parallel computing environments. We provide a PGS guarantee, as well as a computational study where we implement our procedure using modern distributed frameworks. We successfully solve instances with up to 10^6 alternatives, pushing the frontier of SO problems that can be solved by enumeration. In terms of followup research, we would like to explore whether we can provide a PGS guarantee given non-normal simulation output (see Glynn and Juneja (2015) for a discussion of the challenges involved in the absence of distributional assumptions, and Luo et al. (2015) for *asymptotic* PCS guarantees based on second moment assumptions), as well as provide theoretically sound guidelines for choosing algorithm parameters such β and $\bar{r}$. From an implementation standpoint, a future version of our code would attempt to distribute the coordination tasks to multiple, possibly tiered, masters to alleviate the possibility of congestion.

Acknowledgments

We thank David Eckman who pointed out a mistake in one of the proofs in an earlier version of this paper. 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

Amazon. 2016. Amazon ec2 instance types. URL `https://aws.amazon.com/ec2/instance-types/`.

Andradóttir, S. 1998. A review of simulation optimization techniques. D. J. Medieros, E. F. Watson, J. S. Carson, M. S. Manivannan, eds., *Proceedings of the 1998 Winter Simulation Conference*. Institute of Electrical and Electronics Engineers: Piscataway, New Jersey, 151–158.

Bechhofer, Robert E., Thomas J. Santner, David M. Goldsman. 1995. *Design and Analysis of Experiments for Statistical Selection, Screening, and Multiple Comparisons*. Wiley New York.

Boesel, J., B. L. Nelson, S.-H. Kim. 2003. Using ranking and selection to 'clean up' after simulation optimization. *Operations Research* **51**(5) 814–825.

Branke, J., S. E. Chick, C. Schmidt. 2007. Selecting a selection procedure. *Management Science* **53**(12) 1916–1932.

Bubeck, Sébastien, Nicolo Cesa-Bianchi. 2012. Regret analysis of stochastic and nonstochastic multi-armed bandit problems. *Foundations and Trends in Machine Learning* **5**(1) 1–122.

Casella, George, Roger L. Berger. 2002. *Statistical Inference*. Thomson Learning, Australia; Pacific Grove, CA.

Chen, C.-H., S. E. Chick, L. H. Lee. 2015. Ranking and selection: Efficient simulation budget allocation. Michael C. Fu, ed., *Handbook of Simulation Optimization*, *International Series in Operations Research & Management Science*, vol. 216, chap. 3. Springer, New York, 45–80.

Chen, C.-H., J. Lin, E. Yücesan, S. E. Chick. 2000. Simulation budget allocation for further enhancing the efficiency of ordinal optimization. *Discrete Event Dynamic Systems* **10**(3) 251–270.

Chen, E. Jack. 2005. Using parallel and distributed computing to increase the capability of selection procedures. M. E Kuhl, N. M. Steiger, F. B. Armstrong, J. A. Joines, eds., *Proceedings of the 2005 Winter Simulation Conference*. 723–731.

Dean, Jeffrey, Sanjay Ghemawat. 2008. MapReduce: simplified data processing on large clusters. *Communications of the ACM* **51**(1) 107–113.

Fu, M. 1994. Optimization via simulation: A review. *Annals of Operations Research* **53** 199–247.

Fu, M. C. 2002. Optimization for simulation: theory vs. practice. *INFORMS Journal on Computing* **14** 192–215.

Fu, M. C., F. W. Glover, J. April. 2005. Simulation optimization: a review, new developments, and applications. M. E. Kuhl, N. M. Steiger, F. B. Armstrong, J. A. Joines, eds., *Proc. of the 2005 Winter Simulation Conference*. Institute of Electrical and Electronics Engineers, Inc., Piscataway, NJ, 83–95.

Fu, M.C. 2015. *Handbook of Simulation Optimization*. International Series in Operations Research & Management Science, Springer New York.

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

Glynn, P. W., P. Heidelberger. 1991. Analysis of parallel replicated simulations under a completion time constraint. *ACM Transactions on Modeling and Computer Simulation* **1**(1) 3–23.

Glynn, P. W., W. Whitt. 1992. The asymptotic efficiency of simulation estimators. *Operations Research* **40** 505–520.

Glynn, Peter W., Sandeep Juneja. 2015. Ordinal optimization - empirical large deviations rate estimators, and stochastic multi-armed bandits. Draft available on arXiv:1507.04564.

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

Henderson, S. G., R. Pasupathy. 2014. Simulation optimization library. URL `http://www.simopt.org`.

Hong, L. J., B. L. Nelson. 2005. The tradeoff between sampling and switching: new sequential procedures for indifference-zone selection. *IIE Transactions* **37** 723–734.

Hong, L. Jeff. 2006. Fully sequential indifference-zone selection procedures with variance-dependent sampling. *Naval Research Logistics (NRL)* **53**(5) 464–476.

Jamieson, K., R. Nowak. 2014. Best-arm identification algorithms for multi-armed bandits in the fixed confidence setting. *Information Sciences and Systems (CISS), 2014 48th Annual Conference on*. 1–6.

Karl, Andrew T., Randy Eubank, Jelena Milovanovic, Mark Reiser, Dennis Young. 2014. Using RngStreams for parallel random number generation in C++ and R. *Computational Statistics* 1–20.

Kim, S.-H., B. L. Nelson. 2006a. Selecting the best system. S. G. Henderson, B. L. Nelson, eds., *Simulation*, *Handbooks in Operations Research and Management Science*, vol. 13. North-Holland Publishing, Amsterdam, 501–534.

Kim, Seong-Hee, Barry L. Nelson. 2001. A fully sequential procedure for indifference-zone selection in simulation. *ACM Transactions on Modeling and Computer Simulation* **11**(3) 251–273.

Kim, Seong-Hee, Barry L. Nelson. 2006b. On the asymptotic validity of fully sequential selection procedures for steady-state simulation. *Operations Research* **54**(3) 475–488.

L'Ecuyer, Pierre. 2015. Random number generation with multiple streams for sequential and parallel computing. L. Yilmaz, W. K. V. Chan, T. M. K. Roeder, C. Macal, M.D. Rosetti, eds., *Proceedings of the 2015 Winter Simulation Conference*. IEEE Press, Piscataway NJ, 31–44.

L'Ecuyer, Pierre, Richard Simard, E. Jack Chen, W. David Kelton. 2002. An object-oriented random-number package with many long streams and substreams. *Operations Research* **50**(6) 1073–1075.

Luo, Jun, Jeff L. Hong, Barry L. Nelson, Yang Wu. 2015. Fully sequential procedures for large-scale ranking-and-selection problems in parallel computing environments. *Operations Research* **63**(5) 1177–1194.

Luo, Jun, L. Jeff Hong. 2011. Large-scale ranking and selection using cloud computing. S. Jain, R.R. Creasey, J. Himmelspach, K.P. White, M. Fu, eds., *Proceedings of the 2011 Winter Simulation Conference*. 4051–4061.

Luo, Yuh-Chuyn, Chun-Hung Chen, E. Yucesan, Insup Lee. 2000. Distributed web-based simulation optimization. *Proceedings of the 2000 Winter Simulation Conference*, vol. 2. 1785–1793.

Mascagni, Michael, Ashok Srinivasan. 2000. Algorithm 806: Sprng: A scalable library for pseudorandom number generation. *ACM Trans.Math.Softw.* **26**(3) 436–461.

Nelson, B. L., F. J. Matejcik. 1995. Using common random numbers for indifference-zone selection and multiple comparisons in simulation. *Management Science* **41**(12) 1935–1945.

Nelson, Barry L., Julie Swann, David Goldsman, Wheyming Song. 2001. Simple procedures for selecting the best simulated system when the number of alternatives is large. *Operations Research* **49**(6) 950–963.

Ni, Eric C. 2015a. MapRedRnS: Parallel ranking and selection using MapReduce. URL `https://bitbucket.org/ericni/mapredrns`.

Ni, Eric C. 2015b. mpirns: Parallel ranking and selection using MPI. URL `https://bitbucket.org/ericni/mpirns`.

Ni, Eric C. 2015c. SparkRnS: Parallel ranking and selection using Spark. URL `https://bitbucket.org/ericni/sparkrns`.

Ni, Eric C., Dragos F. Ciocan, Shane G. Henderson, Susan R. Hunter. 2015. Comparing Message Passing Interface and MapReduce for large-scale parallel ranking and selection. L. Yilmaz, W. K. V. Chan, I. Moon, T. M. K. Roeder, C. Macal, M. D. Rossetti, eds., *Proceedings of the 2015 Winter Simulation Conference*. Submitted.

Ni, Eric C., Shane G. Henderson, Susan R. Hunter. 2014. A comparison of two parallel ranking and selection procedures. A. Tolk, S. D. Diallo, I. O. Ryzhov, L. Yilmaz, S. Buckley, J. A. Miller, eds., *Proceedings of the 2014 Winter Simulation Conference*. 3761–3772.

Ni, Eric C., Susan R. Hunter, Shane G. Henderson. 2013. Ranking and selection in a high performance computing environment. R. Pasupathy, S.-H. Kim, A. Tolk, R. Hill, M. E. Kuhl, eds., *Proceedings of the 2013 Winter Simulation Conference*. 833–845.

Pasupathy, R., S. Ghosh. 2013. Simulation optimization: A concise overview and implementation guide. H. Topaloglu, ed., *TutORials in Operations Research*, chap. 7. INFORMS, 122–150. doi:10.1287/educ.2013.0118.

Pichitlamken, Juta, Barry L. Nelson, L. Jeff Hong. 2006. A sequential procedure for neighborhood selection-of-the-best in optimization via simulation. *European Journal of Operational Research* **173**(1) 283–298.

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

Tamhane, Ajit C. 1977. Multiple comparisons in model I one-way ANOVA with unequal variances. *Communications in Statistics - Theory and Methods* **6**(1) 15–32.

Texas Advanced Computing Center. 2016. What is wrangler (TACC/IU)? Retrieved April 9, 2016, `https://www.tacc.utexas.edu/user-services/user-guides/stampede-user-guide`.

Yoo, Taejong, Hyunbo Cho, Enver Yücesan. 2009. Web Services-Based Parallel Replicated Discrete Event Simulation for Large-Scale Simulation Optimization. *Simulation* **85**(7) 461–475.

Zaharia, Matei, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, Ion Stoica. 2010. Spark: Cluster Computing with Working Sets. *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing*. HotCloud'10, USENIX Association, Berkeley, CA, USA, 10. URL `http://dl.acm.org/citation.cfm?id=1863103.1863113`.

e-companion

EC.1. Proof of Theorem 1.

Proving Theorem 1 requires the following lemmas, where we use $B_\Delta(\cdot)$ to denote a Brownian motion with drift Δ and volatility one.

LEMMA EC.1. *(Hong 2006, Theorem 1) Let $m(r)$ and $n(r)$ be arbitrary nondecreasing integer-valued functions of $r = 0, 1, \ldots$ and i, j be any two systems. Define $Z(m,n) := \left[\sigma_i^2/m + \sigma_j^2/n\right]^{-1}\left[\bar{X}_i(m) - \bar{X}_j(n)\right]$ and $Z'(m,n) := B_{\mu_i-\mu_j}([\sigma_i^2/m + \sigma_j^2/n]^{-1})$. Then the random sequences $\{Z(m(r),n(r)) : r = 0,1,\ldots\}$ and $\{Z'(m(r),n(r)) : r = 0,1,\ldots\}$ have the same joint distribution.*

LEMMA EC.2. *Let $i \neq j$ be any two systems. Define $\tilde{a}_{ij}(\bar{r}) := \min\{S_i^2/\sigma_i^2, S_j^2/\sigma_j^2\}a_{ij}(\bar{r})$ and $\tilde{t}_{ij}(\bar{r}) := \min\{S_i^2/\sigma_i^2, S_j^2/\sigma_j^2\}\tau_{ij}(\bar{r})$. It can be shown (Hong 2006) that $\min\{S_i^2/\sigma_i^2, S_j^2/\sigma_j^2\} \leq t_{ij}(r)/\tau_{ij}(r)$ for all $r \geq 0$ regardless of the sampling rules $n_i(\cdot)$ and $n_j(\cdot)$. Therefore $\tilde{a}_{ij}(\bar{r}) \leq t_{ij}(r)a_{ij}(\bar{r})/\tau_{ij}(r)$ and $\tilde{t}_{ij}(\bar{r}) \leq t_{ij}(r)\tau_{ij}(\bar{r})/\tau_{ij}(r)$ regardless of the sampling rules $n_i(\cdot)$ and $n_j(\cdot)$ for all $r \geq 0$.*

LEMMA EC.3. *(Hong 2006, Lemma 4) Let $g_1(\cdot)$, $g_2(\cdot)$ be two non-negative-valued functions such that $g_2(t') \geq g_1(t')$ for all $t' \geq 0$. Define symmetric continuations $C_m := \{(t',x) : -g_m(t') \leq x \leq g_m(t')\}$ and let $T_m := \inf\{t' : B_\Delta(t') \notin C_m\}$ for $m = 1,2$. If $\Delta \geq 0$, then $P[B_\Delta(T_1) < 0] \geq P[B_\Delta(T_2) < 0]$.*

LEMMA EC.4. *By the reflection principle of Brownian motion, $P[\min_{0\leq t'\leq t} B_0(t') < -a] = 2P[B_0(t) < -a] = 2\bar{\Phi}(a/\sqrt{t})$ for all $a, t > 0$.*

LEMMA EC.5. *(Tamhane 1977) Let $V_1, V_2, \ldots, V_k$ be independent random variables, and let $G_j^w(v_1, v_2, \ldots, v_k)$, $j = 1,2,\ldots,p$, be non-negative, real-valued functions, each one nondecreasing in each of its arguments. Then*

$$E\left[\prod_{j=1}^{p} G_j^w(V_1, V_2, \ldots, V_k)\right] \geq \prod_{j=1}^{p} E[G_j^w(V_1, V_2, \ldots, V_k)].$$

LEMMA EC.6. *Under the conditions of Theorem 1, System k survives through to Stage 3 with probability at least $1-\alpha_1$.*

Proof. For any system i, it is well known (Casella and Berger 2002, page 218) that $\bar{X}_i(n_1)|S_i^2$ is normally distributed and $X_{i\ell}$ is independent of S_i^2 for all $\ell > n_1$. Furthermore, $\bar{T}_i$ is obtained in Stage 0 independently of all $X_{i\ell}$'s. Therefore, choosing the sampling rule based on $\bar{T}_i$ and S_i^2 does not affect the normality of the $\{\bar{X}_i(n_i(r)) : r = 0, 1, \ldots, \bar{r}\}$ sequence.

For any two systems i and j, let KO_{ij} be the event that system i eliminates system j in Stages 1 or 2. It then follows that

$\Pr[KO_{ik}$ in Stages 1 or 2]

$=E[\Pr[KO_{ik}$ in Stages 1 or $2|S_k^2, S_i^2]]$

$\leq E[\Pr[Y_{ki}(\tau_{ki}(r)) < -a_{ij}(\bar{r})$ for some $r \leq \bar{r}|S_k^2, S_i^2]]$

since system i could be eliminated by some other system before it can eliminate system k

$=E[\Pr[Y_{ki}(\tau_{ki}(r)) < -a_{ij}(\bar{r})$ and $\tau_{ki}(r) \leq \tau_{ki}(\bar{r})$ for some $r|S_k^2, S_i^2]]$

$=E[\Pr[Z_{ki}(t_{ki}(r)) < -\dfrac{t_{ki}(r)}{\tau_{ki}(r)} a_{ij}(\bar{r})$ and $t_{ki}(r) \leq \dfrac{t_{ki}(r)}{\tau_{ki}(r)} \tau_{ki}(\bar{r})$ for some $r|S_k^2, S_i^2]]$

$=E[\Pr[B_{\mu_k-\mu_i}(t_{ki}(r)) < -\dfrac{t_{ki}(r)}{\tau_{ki}(r)} a_{ij}(\bar{r})$ and $t_{ki}(r) \leq \dfrac{t_{ki}(r)}{\tau_{ki}(r)} \tau_{ki}(\bar{r})$ for some $r|S_k^2, S_i^2]]$ by Lemma EC.1

$\leq E[\Pr[B_{\mu_k-\mu_i}(t_{ki}(r)) < -\tilde{a}_{ij}(\bar{r})$ and $t_{ki}(r) \leq \tilde{t}_{ij}(\bar{r})$ for some $r|S_k^2, S_i^2]]$ by Lemmas EC.2 and EC.3

$\leq E[\Pr[B_{\mu_k-\mu_i}(t) < -\tilde{a}_{ij}(\bar{r})$ for some $t \leq \tilde{t}_{ij}(\bar{r})|S_k^2, S_i^2]]$

$\leq E[\Pr[B_0(t) < -\tilde{a}_{ij}(\bar{r})$ for some $t \leq \tilde{t}_{ij}(\bar{r})|S_k^2, S_i^2]]$ since $\mu_k \geq \mu_i$

$= E\left[2\bar{\Phi}\left(\dfrac{\tilde{a}_{ij}(\bar{r})}{\sqrt{\tilde{t}_{ij}(\bar{r})}}\right)\right]$ by Lemma EC.4

$= E\left[2\bar{\Phi}\left(\dfrac{a_{ij}(\bar{r})}{\sqrt{\tau_{ij}(\bar{r})(n_1-1)}}\sqrt{\min\left\{\dfrac{(n_1-1)S_i^2}{\sigma_i^2}, \dfrac{(n_1-1)S_k^2}{\sigma_k^2}\right\}}\right)\right]$

$= E\left[2\bar{\Phi}\left(\eta\sqrt{\min\left\{\dfrac{(n_1-1)S_i^2}{\sigma_i^2}, \dfrac{(n_1-1)S_k^2}{\sigma_k^2}\right\}}\right)\right]$ by choice of $a_{ij}(\bar{r})$

$= 1-(1-\alpha_1)^{\frac{1}{k-1}}$ by (4), since $(n_1-1)S_i^2/\sigma_i^2$ and $(n_1-1)S_k^2/\sigma_k^2$ are i.i.d. $\chi^2_{n_1-1}$ random variables.

Then, noting that simulation results from different systems are mutually independent, we have

$$\begin{aligned}
\Pr[\text{system } k \in G_2] &= E\left[\Pr\left\{\bigcap_{i=1}^{k-1} \overline{KO}_{ik} | X_{k1}, X_{k2}, \ldots\right\}\right] \\
&= E\left[\prod_{i=1}^{k-1} \Pr\left\{\overline{KO}_{ik} | X_{k1}, X_{k2}, \ldots\right\}\right] \\
&\geq \prod_{i=1}^{k-1} E\left[\Pr\left\{\overline{KO}_{ik} | X_{k1}, X_{k2}, \ldots\right\}\right] \text{ by Lemma EC.5} \\
&= \prod_{i=1}^{k-1} \Pr\left[\overline{KO}_{ik}\right] \geq \prod_{i=1}^{k-1} \left[1 - \left(1 - (1-\alpha_1)^{\frac{1}{k-1}}\right)\right] = 1 - \alpha_1.
\end{aligned}$$

□

Thus far we have shown that System k survives to Stage 3 with high probability. We now want to show that in Stage 3 we obtain a good selection amongst the surviving systems with high probability, at least when System k is amongst the survivors. To this end, let $S \subseteq \mathcal{S}$ be an arbitrary subset of systems, and let $k(S)$ be the index of the system with largest true mean in S, i.e., $k(S) = \arg\max_{j \in S} \mu_j$, breaking any ties by selecting the larger index. Let $\bar{X}_j(N_j)$ be our estimator of the performance of System j. Central to our proof of good selection is the condition that

$$\bar{X}_{k(S)}(N_{k(S)}) - \bar{X}_j(N_j) - (\mu_{k(S)} - \mu_j) > -\delta \quad \forall j \in S. \tag{EC.1}$$

LEMMA EC.7. *If G_2, the random set of systems surviving to Stage 3, includes a best system, and if* (EC.1) *holds for $S = G_2$, then the procedure* GSP′ selects a good system.

Proof. Recall that GSP′ selects the system K in Stage 3 with the highest sample mean. From (EC.1) with $S = G_2$ and for the specific choice of $j = K$,

$$\mu_{k(S)} - \mu_K < \bar{X}_{k(S)}(N_{k(S)}) - \bar{X}_K(N_K) + \delta.$$

But $\mu_{k(S)} = \mu_k$ because the set of survivors $S = G_2$ contains a best system. Also, since K is the system with the highest sample mean, the difference of sample means on the right hand side is at most 0. Thus $\mu_k - \mu_K < \delta$ as required. □

Proof of Theorem 1 Let A_1 be the event that System k survives to Stage 3. By Lemma EC.6, $\Pr(A_1) \geq 1-\alpha_1$. To complete the proof we employ a form of coupling. To this end, imagine that Rinott sample sizes are taken of *all* systems, and not just for the set G_2 of those that survive to Stage 3. Rinott's procedure offers a correct-selection guarantee (Rinott 1978) over all k systems provided that the best system is at least δ better than the second best, and by Theorem 1 of Nelson and Matejcik (1995) and the commentary immediately afterward, this conclusion can be strengthened to the assertion that $\Pr(A_2) \geq 1-\alpha_2$, where A_2 is the event that (EC.1) holds for $S = \{1, 2, \ldots, k\}$, i.e.,

$$\bar{X}_k(N_k) - \bar{X}_j(N_j) - (\mu_k - \mu_j) > -\delta \quad \forall j = 1, 2, \ldots, k.$$

Therefore, on the event $A_1 \cap A_2$, the conditions of Lemma EC.7 hold and we obtain a good selection overall. But, as in the decomposition lemma of Nelson et al. (2001),

$$\Pr(A_1 \cap A_2) \geq \Pr(A_1) + \Pr(A_2) - 1 \geq 1 - \alpha_1 + 1 - \alpha_2 - 1,$$

thereby completing the proof. □

EC.2. On computing the parameter η

The parameter η, which is the solution to (4), determines the value of $a_{ij}(\bar{r})$, and hence determines how quickly an inferior system is eliminated in screening Steps 3(c) and 5(d). One way to compute η is by integrating the left-hand side (LHS) using Gauss-Laguerre quadrature and using bisection to find the root of (4). Alternatively, we may employ a bounding technique to approximate η as follows. The LHS of (4) is

$$\begin{aligned}
E\left[2\bar{\Phi}\left(\eta\sqrt{R}\right)\right] &= \int_{y=0}^{\infty} 2\bar{\Phi}(\eta\sqrt{y}) 2[1 - F_{\chi^2_{n_1-1}}(y)] f_{\chi^2_{n_1-1}}(y) \mathrm{d}y \\
&\leq \int_{y=0}^{\infty} 4\bar{\Phi}(\eta\sqrt{y}) f_{\chi^2_{n_1-1}}(y) \mathrm{d}y \qquad \text{(EC.2)} \\
&\leq \int_{y=0}^{\infty} 4\frac{e^{-\eta^2 y/2}}{\eta\sqrt{2\pi y}} \frac{y^{\frac{n_1-1}{2}-1} e^{-y/2}}{2^{\frac{n_1-1}{2}}\Gamma(\frac{n_1-1}{2})} \mathrm{d}y \qquad \text{(EC.3)} \\
&= \frac{4\Gamma(\frac{n_1-2}{2})(\frac{2}{\eta^2+1})^{\frac{n_1-2}{2}}}{\sqrt{2\pi}\eta 2^{\frac{n_1-1}{2}}\Gamma(\frac{n_1-1}{2})} \int_0^{\infty} \frac{(\frac{\eta^2+1}{2})^{\frac{n_1-2}{2}} y^{\frac{n_1-1}{2}-1-\frac{1}{2}} e^{-\frac{\eta^2+1}{2}y}}{\Gamma(\frac{n_1-2}{2})} \mathrm{d}y \qquad \text{(EC.4)} \\
&= \frac{2\Gamma(\frac{n_1-2}{2})}{\sqrt{\pi}\Gamma(\frac{n_1-1}{2})\eta(\eta^2+1)^{\frac{n_1-2}{2}}}, \qquad \text{(EC.5)}
\end{aligned}$$

where (EC.2) is inspired by a similar argument in Hong (2006) and holds because distribution functions are non-negative; (EC.3) follows from the fact that $\bar{\Phi}(x) \leq e^{-x^2/2}/(x\sqrt{2\pi})$ for all $x > 0$; and the integrand in (EC.4) is the pdf of a Gamma distribution with shape $(n_1 - 1)/2$ and scale $2/(\eta^2 + 1)$, and hence integrates to 1.

Note that (EC.5) is an upper-bound on the left-hand side of (4). Setting (EC.5) to $1 - (1 - \alpha_1)^{\frac{1}{k-1}}$ and solving for η yields an overestimate η', which is more conservative and does not reduce the PGS. Furthermore, as (EC.5) is strictly decreasing in η, η' can be easily determined using bisection.

Since the value of η directly impacts the effectiveness of the iterative screening, it is desirable that η does not grow dramatically as the problem gets bigger. Observe that (EC.5) can be further bounded by

$$\frac{2\Gamma(\frac{n_1-2}{2})}{\sqrt{\pi}\Gamma(\frac{n_1-1}{2})\eta^{n_1-1}} := C\eta^{1-n_1}. \tag{EC.6}$$

Setting (EC.6) to $1 - (1 - \alpha_1)^{\frac{1}{k-1}}$ implies that the right-hand side of (EC.6) must be small. After some further manipulations we have

$$\log(1 - \alpha_1) = (k - 1)\log\left(1 - C\eta^{1-n_1}\right) \approx (k - 1)(-C\eta^{1-n_1}) \tag{EC.7}$$

where the approximation holds because $\log(1 - \epsilon) \approx -\epsilon$ for small $\epsilon > 0$. It follows from (EC.7) that for fixed α_1, the parameter η grows very slowly with respect to k, at a rate of $k^{1/(n_1-1)}$. Therefore, the continuation region defined by η and $\bar{r}$ as well as the power of our iterative screening are not substantially weakened as the number of systems increases, especially when n_1 or k is large. In this regime, we should expect the total cost of this R&S procedure to grow approximately linearly with respect to the number of systems.

EC.3. Full Description of the MPI implementation

The purpose of this section is to provide additional insight into our parallel codes.

Our MPI implementation designates one core as the master and lets it control other worker cores. Communication is fast on Wrangler, the environment that MPI is tailored to, taking only

Master Core Routine
Input: List of systems $\mathcal{S}$; Average Stage 2 batch size β; Parameters $\delta, \alpha_1, \alpha_2, n_0, n_1, \bar{r}$ and a random number *seed*.

begin Preparation: Setting up random number streams
  Initialize random number generator using the *seed*;
  foreach *worker* $w = 1, 2, \ldots, c$ **do**
    Generate a new random number stream U_w;
    Send U_w to w;
  end
end

begin Stage 0: Estimating simulation completion time
  $\{G_w^0 : w = 1, 2, \ldots, c\} \leftarrow$`Partition`$(\mathcal{S}, 0)$;
  foreach *worker* $w = 1, 2, \ldots, c$ **do**
    Send G_w^0 to Worker w;
  end
  `Collect`$(\overline{T}_i)$;
end

begin Stage 1: Estimating sample variances
  $\{G_w^1 : w = 1, 2, \ldots, c\} \leftarrow$`Partition`$(\mathcal{S}, 1)$;
  foreach *worker* $w = 1, 2, \ldots, c$ **do**
    Send G_w^1 to Worker w;
  end
  `Collect`$(S_i^2$ and `Stat`$_{i,0})$;
  $\{\mathcal{S}, G_w^1\} \leftarrow$`RecvScreen`$(w)$;
end

Worker Core Routine
Input: List of systems $\mathcal{S}$; Parameters $\delta, \alpha_1, \alpha_2, n_0, n_1$.

begin Preparation: Setting up random number streams
  Receive random number stream U_w;
  Initialize random number generator using U_w;
end

begin Stage 0: Estimating simulation completion time
  Receive the set of systems to simulate, G_w^0;
  foreach *system* $i \in G_w^0$ **do**
    `Simulate`$(i, n_0,$ simulation time $\overline{T}_i)$;
  end
  Return $\{\overline{T}_i : i \in G_w^0\}$ to master;
end

begin Stage 1: Estimating sample variances
  Receive the set of systems to simulate, G_w^1;
  foreach *system* $i \in G_w^1$ **do**
    `Simulate`$(i, n_1, (S_i^2,$ `Stat`$_{i,0}))$;
  end
  Return $\{(S_i^2,$ `Stat`$_{i,0}) : i \in G_w\}$ to master;
  $G_w \leftarrow$`Screen`$(G_w^1, 0, 0, false)$;
  `SendScreen`(G_w^1);
end

Figure EC.1 Stages 0 and 1, MPI Implementation: Master (left) and workers (right) routines

10^{-6} to 10^{-4} seconds per message, depending on the message size. Therefore, with an appropriate choice of the batch-size parameter β, the master remains idle most of the time. Workers are then able to communicate with the master with little delay.

In Figures EC.1 through EC.3 we demonstrate in greater detail how the master core allocates and distributes systems, how random number streams are created and distributed together with the assigned systems to ensure independent sampling, and how simulation results are communicated between cores.

We use the following notation for some subroutines in Figures EC.1 through EC.3:

`Partition`($\mathcal{S}$, *Stage*) The master divides the set of systems $\mathcal{S}$ into disjoint partitions $\{G_{Stage}^w : w = 1, 2, \ldots, c\}$:

In *Stage 0*, all systems are simulated for n_0 replications to estimate simulation completion time. The master randomly permutes $\mathcal{S}$ (in case of long runtimes for some systems that are indexed closely) and assigns approximately equal numbers of systems to each G_0^w.

```
begin Stage 2: Iterative screening
    G_1 ← systems that survived Stage 1;
    {G_2^w : w = 1,2,...,c} ←Partition(G_1, 2);
    S ← G_1;
    foreach worker w = 1,2,...,c do
        Send G_1, G_2^w to Worker w;
        foreach system i ∈ G_1 do
            Send S_i^2 from Stage 1 to Worker w;
        end
        foreach system i ∈ G_2^w do
            Send Stat_{i,0} to worker w;
        end
    end
    b_i ← BatchSize(i, β), q_i ← 1 for all i ∈ G_1;
    r_w^sent ← 0, r_w^received ← 0, r_w^screened ← 0, flag_w ← 0 for
    all w = 1,2,...,c;
    while |S| > 1 and r_w^screened < r̄ for some w do
        Wait for the next worker w to call
            Communicate();
        if flag_w = 1 then
            /* Send screening task to worker w   */
            {i, q_i, Stat_{i,q_i}} ←RecvOutput(w);
        else if flag_w = 2 then
            /* Send simulation task to worker w  */
            {S, G_2^w, r_w^screened} ←RecvScreen(w);
            {i*_w, r_w^received, {Stat_{i*_w,r} : r ≤
            r_w^received}} ←RecvBest(w);
        end
        if |S| > 1 then
            r^current ←CountBatch(w);
            if r^current > r_w^sent then
                flag_w ← 2; SendAction(w,flag_w);
                SendStats(w, r_w^sent, r^current);
                SendBestStats(w);
                r_w^sent ← r^current;
            else
                flag_w ← 1; SendAction(w, flag_w);
                Select next i ∈ S such that q_i = q_Global;
                SendSim(w, i, q_i, b_i);
                q_i ← q_i + 1;
                if q_i > q_Global for all i ∈ S then
                    q_Global ← q_Global + 1;
                end
            end
        end
    end
    Send a termination instruction to all workers;
end
```

```
begin Stage 2: Iterative screening
    Receive the set of systems that survived, G_1;
    Receive the set of systems to screen, G_2^w;
    foreach System i ∈ G_1 do
        Receive S_i^2 collected in Stage 1;
    end
    foreach system i ∈ G_2^w do
        Receive Stat_{i,0} from the master;
    end
    r_w ← 0;
    Communicate();
    while No termination instruction received do
        flag_w ←RecvAction();
        if flag_w = 2 then
            {r^new, {Stat_{i,r} : i ∈ G_2^w, r_w + 1 ≤ r ≤ r^new}}
                ←RecvStats();
            {W, {r_{w'}^received : w' ∈ W},
                {Stat_{i*_{w'},r} : w' ∈ W, r ≤ r_{w'}^received}}
                ←RecvBestStats();
            G_2^w ←Screen(G_2^w, r_w + 1, r^new, true);
            r_w ← r^new;
            Communicate();
            SendScreen(G_2^w, r_w); SendBest(r);
        else
            {i, q_i, b_i} ←RecvSim();
            Simulate(i, b_i, Stat_{i,q_i});
            Communicate();
            SendOutput(i, q_i, Stat_{i,q_i})
        end
        for i = 1 to r do
            Simulate(i, n_i, X̄_i) for one batch;
        end
        Screen among r sys. using current batch stats.;
        for each batch k up to the current one do
            If batch k stats. available, screen the r
            systems simulated, against the best systems
            from the other workers, at batch k;
        end
        if Master sends a terminate instruction then
            continue← false;
        else if Master is ready to communicate then
            Report indexes of eliminated sys. to master;
            Report stats. of the best system for each
            batch simulated up to this point; Receive
            stats. of the best sys. from other workers;
        end
    end
end
```

Figure EC.2 Stage 2, MPI Implementation: Master (left) and workers (right) routines

In *Stage 1*, a fixed number n_1 of replications are required from each system. To balance the simulation work among workers, the master chooses G_1^w such that the estimated completion time $\sum_{i \in G_1^w} n_1 \bar{T}_i / n_0$ is approximately equal for all w.

In *Stage 2*, both simulation and screening are performed iteratively. Simulation of a system is no longer dedicated to a particular worker, and G_2^w is the set of systems that worker w

begin Stage 3: Rinott Stage
- $G_2 \leftarrow$ systems that survived Stage 2;
- **if** $|G_2| = 1$ **then**
  - Report the single surviving system as the best;
- **else**
  - $h \leftarrow h(1-\alpha_2, n_1, |G_2|)$;
  - **foreach** *system* $i \in G_2$ **do**
    - $N_i \leftarrow \max\{n_i(\bar{r}), \lceil (hS_i/\delta)^2 \rceil\}$;
    - $N_i^{\text{sent}} \leftarrow 0$; $N_i^{\text{received}} \leftarrow 0$;
  - **end**
  - $\texttt{flag}_w \leftarrow 0$ for all $w = 1, 2, \ldots, c$;
  - **while** $N_i^{received} < N_i - n_i(\bar{r})$ *for some* $i \in G_2$ **do**
    - Wait for the next worker w to call `Communicate()`;
    - **if** $\mathit{flag}_w = 1$ **then**
      - Receive i, b_i' and sample mean of the current batch;
      - Merge sample mean into $\bar{X}_i$;
      - $N_i^{\text{received}} \leftarrow N_i^{\text{received}} + b_i'$
    - **end**
    - **if** $N_i^{sent} < N_i - n_i(\bar{r})$ *for some* $i \in G_2$ **then**
      - Find an appropriate batch size $b_i' = \min\{b_i, N_i - n_i(\bar{r}) - N_i^{textsent}\}$ for system i;
      - Send system i and b_i' to worker w;
      - $N_i^{\text{sent}} \leftarrow N_i^{\text{sent}} + b_i'$;
      - $\texttt{flag}_w \leftarrow 1$;
    - **end**
  - **end**
  - Report the system $i^* = \arg\max_{i \in G_2} \bar{X}_i(N_i)$ as the best;
- **end**
- Send a termination instruction to all workers;

end

begin Stage 3: Rinott Stage
- `Communicate()`;
- **while** *No termination instruction received* **do**
  - Receive a system i and batch size $b'i$ from the master;
  - Simulate system i for b_i replications;
  - `Communicate()`;
  - Send i, b_i' and sample mean of the b_i' replications to the master;
- **end**

end

Figure EC.3 Stage 3, MPI Implementation: Master (left) and workers (right) routines

needs to screen. To load-balance the screening work, the master assigns approximately equal numbers of systems to each G_2^w.

`Collect`(*info*) The master collects *info* from all workers for all systems, in arbitrary order.

`Simulate`$(i, n,$ *info*$)$ Worker w simulates system i for n replications and records *info* using the next subsubstream in U_w^i.

$\texttt{Stat}_{i,r}$ The batch statistics for the rth batch of system i. This includes sample size $n_i(r)$ and sample mean $\bar{X}_i(n_i(r))$ as described in §3.

`BatchSize`(i, β) The master calculates batch size b_i system i used in Stage 2. Following the recommendation from §2.2.2, we let

$$b_i = \left\lceil \frac{S_i\sqrt{T_i}}{\frac{1}{|\mathcal{S}|}\sum_{j\in\mathcal{S}} S_j\sqrt{T_j}}\beta \right\rceil \tag{EC.8}$$

where β is a pre-determined average batch size.

`Screen`$(G^w, r_0, r_1,$ *useothers*$)$ Screen systems in G^w from batches r_0 through r_1 inclusive. It can be checked that worker w has received $\texttt{Stat}_{i,r}$ for all $i \in \mathcal{S}$, all $r \le r_1$ and stored the data in its memory.

A system $i \in G^w$ is eliminated if there exists system $j \in G_2^w : j \neq i$ and some $r' : r_0 \le r' \le r_1$ such that $r' \le \bar{r}$ and $Y_{ij}(r') < -a_{ij}(\bar{r})$ where Y_{ij} and a_{ij} are defined in §3.

In addition, if *useothers*$= true$ and $\mathcal{W} \neq \emptyset$, then for each $w' \in \mathcal{W}$ the worker also screens the systems in G^w against system $i^*_{w'}$, the best system from worker w', using batch statistics $\{\texttt{Stat}_{i^*_{w'},r'} : r' \le r_{w'}\}$ up to batch $\min\{r_{w'}, r_1\}$.

`SendScreen`(G^w, r_w) and `RecvScreen`(w) Worker w sends r_w and screening results (updated G^w) to the master, which then updates G^w and $\mathcal{S}$ on its own memory accordingly. The master also receives r_w and lets $r_w^{\text{screened}} \leftarrow r_w$.

`Communicate`() Worker sends a signal to master and waits for the master to receive the signal, before proceeding.

`SendSim`(w, i, q_i, b_i) and `RecvSim`() The master instructs worker w to simulate the q_ith batch of system i, for b_i replications. Worker w receives i, q_i, b_i from the master.

`SendOutput`$(i, q_i, \texttt{Stat}_{i,q_i})$ and `RecvOutput`(w) Worker w sends simulation output $\texttt{Stat}_{i,q_i}$ for the q_ith batch of system i to the master. The master stores $\texttt{Stat}_{i,q_i}$ in memory.

`SendBest`() and `RecvBest`(w) Worker w sends its estimated-best system i^*_w (the one in G^w with the highest batch mean) to the master, together with all batch statistics for system i^*_w, $\{\texttt{Stat}_{i^*_w,r} : r \le r_w\}$; the master receives r_w and lets $r_w^{\text{received}} \leftarrow r_w$.

`CountBatch`(w) The master finds the largest $r^{\text{current}} \ge r_w$ such that $\texttt{Stat}_{i,r}$ for all $i \in G^w$, $r_w < r \le r^{\text{current}}$ have been received by the master.

`SendAction`$(w, \texttt{flag}_w)$ and `RecvAction`() The master sends an indicator flag_w to worker w, where $\text{flag}_w = 1$ indicates "simulate a batch" and $\text{flag}_w = 2$ indicates "perform screening".

`SendStats`(w) and `RecvStats`() The master sends $\texttt{Stat}_{i,r}$ for all $i \in G^w$, $r_w < r \le r^{\text{current}}$ to worker w; the worker receives r^{current} and lets $r^{\text{new}} \leftarrow r^{\text{current}}$; the worker should have $\texttt{Stat}_{i,r}$ for all $i \in G^w$, $0 < r \le r^{\text{new}}$ upon completion.

`SendBestStats`(w) and `RecvBestStats`() The master computes $\mathcal{W} = \{w' \neq w : |G_{w'}| > 0)\}$ and sends $\mathcal{W}$ to worker w; the master then sends all available batch statistics for best systems, $\{\texttt{Stat}_{i^*_{w'},r} : w' \in \mathcal{W}, r \leq r_{w'}^{\text{received}}\}$, to worker w.

EC.4. Full Description of the Hadoop MapReduce implementation

We present in this section the full details of the MapReduce implementation of GSP. A MapReduce calculation consists of a *Map* phase where *data entries* are processed by "Mapper" functions in parallel, and a *Reduce* phase where Mapper outputs are grouped by *keys* and summarized using parallel "Reducer" functions. Any parallel algorithm written in Hadoop MapReduce must be expressed through these Map and Reduce functions.

Conceptually, each Mapper reads a comma-separated string of varied length, denoted by [value 1, value 2, ..., `$type`], where the last component `$type` is used to indicate the specific information captured in the string. A Mapper usually runs some simulation, updates batch statistics, and generates one or more `key`: {`value`} pairs. All pairs under the same `key` are sent to the same Reducer, which is typically responsible for screening. A Reducer may generate one or more comma-separated strings which become the input to the Mapper in the next iteration. Our MapReduce implementation is based on the native Java interface for MapReduce provided in Apache Hadoop 1.2.1. It is hosted in the open-access repository Ni (2015a).

We propose a variant of GSP using iterative MapReduce as follows. In each Mapper function, we treat each surviving system as a single data entry, obtain an additional batched sample, and output updated summary statistics such as sample sizes, means, and variances. Each output entry is associated with a key that represents the screening group to which it belongs. Once output entries of Mappers are grouped by their keys, each Reducer receives a group of systems, screens amongst them, and writes each surviving system as a new data entry that in turn is used as the input to the next Mapper.

Each system i is coupled with $\texttt{stream}_i$ which is used by some random number generator and updated each time a random number is generated. The coupling of systems and `streams` ensures

that the random numbers generated for each system in each iteration are all mutually independent. We also assume that each system i is preallocated to a particular screening group, as determined by the function `Group`(i).

To fully implement GSP, MapReduce is run for several iterations. The first iteration implements Stage 1, where both $\bar{X}_i(n_1)$ and S_i^2 are collected. Then, a maximum number of $\bar{r}$ subsequent iterations are needed for Stage 2, with only $n_i(r)$ and $\bar{X}_i(n_i(r))$ being updated in each iteration. (Additional MapReduce iterations can be run where the best system from each group is shared for additional between-group screening.) The same Reducer can be applied in both Stages 1 and 2, as the screening logic is the same. Finally, a Stage 3 MapReduce features a Mapper that calculates the additional Rinott sample size, simulates the required replications, and a different Reducer that simply selects the system with the highest sample mean at the end.

The procedure begins with Steps 1-3 which implements Stage 1, then enters Stage 2 where Steps 4 and 5 are run repeatedly for a maximum of $\bar{r}$ iterations. If multiple systems survive Stage 2, the procedure runs Steps 6 and 7 to finish Stage 3.

Step 1.

- **Map**: Estimate S_i^2

  Input $[i]$

  Operation Initialize $\texttt{stream}_i$ with seed i; Simulate system i for n_1 replications to obtain $\bar{X}_i(n_1)$ and S_i^2.

  Output i: $\{\bar{X}_i(n_1),\ S_i^2,\ \texttt{stream}_i,\ \$S0\}$

- **Reduce**

  Input i: $\{\bar{X}_i(n_1),\ S_i^2,\ \texttt{stream}_i,\ \$S0\}$

  Operation Calculate $\sum_i S_i$.

  Output $[i,\ \bar{X}_i(n_1),\ S_i^2,\ \text{stream}_i,\ \$S0]$

Step 2.

- **Map**: Calculate batch size

  Input $[i,\ \bar{X}_i(n_1)\ S_i^2,\ \texttt{stream}_i\ \$S0]$

  Operation Calculate batch size b_i using $b_i = \beta S_i/(\sum_i S_i/k)$.

Output `Group`(i): $\{i, \bar{X}_i(n_1), n_1, b_i, S_i^2,$ `stream`$_i$, \$Sim$\}$

- **Reduce**: Screen within a group

Input `Group`: $\{i, \bar{X}_i(n_i), n_i, b_i, S_i^2, \text{stream}_i,$ \$Sim$\}$ for all i in the `Group`

Operation Screen all systems in the `Group` and find the one i^* with the highest mean.

Output $[i, \bar{X}_i(n_i), n_i, b_i, S_i^2,$ `stream`$_i$, \$Sim$]$ for each surviving system i, and $[i^*, \bar{X}_{i^*}(n_{i^*}), n_{i^*}, b_{i^*}, S_{i^*}^2,$ \$Best$]$ for the best system i^*

Step 3. • **Map**: Share best systems between groups

Input (1) $[i, \bar{X}_i(n_i), n_i, b_i, S_i^2,$ `stream`$_i$, \$Sim$]$

Operation (1) Simply output to `Group`(i).

Output (1) `Group`(i): $\{i, \bar{X}_i(n_i), n_i, b_i, S_i^2,$ `stream`$_i$, \$Sim$\}$

Input (2) $[i^*, \bar{X}_{i^*}(n_{i^*}), n_{i^*}, b_{i^*}, S_{i^*}^2,$ \$Best$]$

Operation (2) Output to all groups.

Output (2) `Group`: $\{i^*, \bar{X}_{i^*}(n_{i^*}), n_{i^*}, b_{i^*}, S_{i^*}^2,$ \$Best$\}$ for every `Group`

- **Reduce**: Screen against the best systems from other groups

Input `Group`: $\{i, \bar{X}_i(n_i), n_i, b_i, S_i^2, \text{stream}_i,$ \$Sim$\}$ for all i in the `Group`, and `Group`: $\{i^*, \bar{X}_{i^*}(n_{i^*}), n_{i^*}, b_{i^*}, S_{i^*}^2,$ \$Best$\}$ from every other `Group`

Operation Screen all systems in `Group` against the best systems from other groups.

Output $[i, \bar{X}_i(n_i), n_i, b_i, S_i^2,$ `stream`$_i$, \$Sim$]$ for each surviving system i

Step 4. • **Map**: Simulation

Input $[i, \bar{X}_i(n_i), n_i, b_i, S_i^2,$ `stream`$_i$, \$Sim$]$

Operation Simulate system i for additional b_i replications, update n_i, $\bar{X}_i(n_i)$, and `stream`$_i$.

Output `Group`(i): $\{i, \bar{X}_i(n_1), n_1, b_i, S_i^2,$ `stream`$_i$, \$Sim$\}$

- **Reduce**: Screen within a group.

(Same as Step 2 Reduce.)

Step 5. Screen against best systems from other groups.

(Same as Step 3.)

Step 6. • **Map**: Determine Rinott sample sizes

Input $[i, \bar{X}_i(n_i), n_i, b_i, S_i^2, \texttt{stream}_i, \$\text{Sim}]$

Operation Output to Reducer.

Output i: $\{\bar{X}_i(n_i), n_i, S_i^2, \texttt{stream}_i, \$\text{Sim}\}$

• **Reduce**

Input i: $\{\bar{X}_i(n_i), n_i, S_i^2, \texttt{stream}_i, \$\text{Sim}\}$

Operation Calculate Rinott sample size and divide the additional sample into batches. For each batch j, generate a substream $\texttt{stream}_i^j$ using $\texttt{steam}_i$.

Output $[i, \bar{X}_i(n_i), n_i, \$\text{S2}]$, and

for each batch j: $[i, \texttt{stream}_i^j, (\text{size of batch } j), \$\text{S3}]$

Step 7. • **Map**: Simulate additional batches

Input (1) $[i, \bar{X}_i(n_i), n_i, \$\text{S2}]$

Operation (1) Output to Reducer, since this is the batch statistics generated in Stage 2.

Output (1) 1: $\{i, \bar{X}_i(n_i), n_i, \$\text{S2}\}$

Input (2) $[i, \texttt{stream}_i^j, (\text{size of batch } j), \$\text{S3}]$

Operation (2) Simulate batch j of system i for the given batch size using $\texttt{stream}_i^j$, calculate batch sample mean $\bar{X}_i^j$.

Output (2) 1: $\{i, \bar{X}_i^j, (\text{size of batch } j), \$\text{S3}\ \}$

• **Reduce**: Merge batches and find the best system

Input (This step has only one Reducer)

1: $\{i, \bar{X}_i(n_i), n_i, \$\text{S2}\}$ and

1: $\{i, \bar{X}_i^j, (\text{size of batch } j), \$\text{S3}\}$ for all system i and all batch j

Operation For each system i, merge all batches (including the one from Stage 2) to form a single sample mean.

Output Report the system i^* that has the highest sample mean.

Hadoop MapReduce Performance Here we report the performance of our MapReduce experiments omitted in Section 4. These experiments are run on Stampede, a separate XSEDE high-performance cluster. Table EC.1 summarizes the Stampede performance of our Hadoop MapReduce

Table EC.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. (Results to 2 significant figures)

Configuration	β	Version	Number of replications ($\times 10^6$)	Wall-clock time (sec)	Total time Simulation ($\times 10^3$ sec)	Total time Screening (sec)	Utilization %
3,249 systems	100	HADOOP	0.46	460	0.34	0.14	1.2
on 64 cores		MPI	0.50	3.0	0.18	0.01	94
	200	HADOOP	0.63	280	0.41	0.10	2.3
		MPI	0.69	4.1	0.25	0.01	95
57,624 systems	100	HADOOP	8.8	550	5.1	1.9	15
on 64 cores		MPI	9.1	53	3.3	0.89	98
	200	HADOOP	12	410	7.0	1.7	27
		MPI	13	75	4.7	0.83	98
1,016,127	100	HADOOP	280	1300	160	120	12
systems		MPI	320	120	110	30	91
on 1,024 cores	200	HADOOP	340	810	190	89	23
		MPI	380	140	140	29	97

GSP implementation versus our MPI implementation with the Stage 0 removed to allow for fair comparison. The gap, in terms of either wall-clock time and utilization, is noticeably larger versus MPI than in the case of Spark. However, as with Spark, this gap narrows as the problem size or batch size increase.

EC.5. Full Description of the Spark implementation

Apache Spark (Zaharia et al. 2010) inherits the portability and fault-tolerance features from Hadoop MapReduce, and is designed to provide a significant improvement in performance in the following aspects.

- **In-memory Resilient Distributed Datasets (RDDs).** In Spark, parallel computing tasks are defined as a sequence of operations on Resilient Distributed Datasets (RDDs). RDDs are data objects stored in a distributed fashion and protected against core failures. By default, Spark stores moderately-sized RDDs in memory and only large RDDs are spilled to the disk. For a R&S procedure implemented in Spark that frequently updates a small amount of summary statistics for each system, storing the results in-memory drastically reduces disk read-write overhead.
- **More flexible computing models.** In addition to map and reduce, Spark supports a rich set of parallelizable operations on RDDs such as filter, join, and union. With these operations,

R&S procedures can be implemented in a more intuitive and effective style. For instance, screening against a subset of best systems in Spark can be implemented as a simple filter operation, rather than a complete MapReduce step as is the case with Hadoop (Step 3, §EC.4). Not only does the flexible API eliminate the expensive auxiliary MapReduce steps, it also makes the code significantly shorter: our Spark code for GSP, written in Scala, spans less than 400 lines, less than one eighth of the length of the MPI implementation in C++.

- **Lazy evaluation of transformations.** The majority of RDD operations are defined as transformations whose actual evaluations can be delayed until their results are needed, a strategy commonly known as lazy evaluation. Upon actual evaluation, Spark actively seeks to combine sequences of transformations into an independent computing stage that is then partitioned and evaluated independently across workers, thus communication and synchronization overhead is greatly reduced.

We implement GSP based on the native Scala interface of Apache Spark 1.2.0. The implementation is hosted in the open-access repository Ni (2015c).

Spark works in a functional programming style, performing parallel operations on data sets which themselves can be partitioned and distributed in parallel. An algorithm implemented in Spark is made up of a sequence of such parallel operations. In Figure EC.4 we briefly illustrate the Spark implementation of GSP. For simplicity, we skip the estimation of simulation run time in Stage 0.

In our implementation, we use the following parallel operations supported by Spark:

`Map`($\mathcal{X};\mathcal{Y}$) Similar to the Map operation in Hadoop, performs a certain operation on every element of set $\mathcal{X}$ given (optional) input $\mathcal{Y}$.

`Reduce`($\mathcal{X}$) Similar to the Reduce operation in Hadoop, reduces the data in $\mathcal{X}$ to a smaller set.

`Combine`($\mathcal{X}$) The Combine operation takes a set of [`key`, `value`] pairs and reduces the `values` under each `key` to a smaller set.

`Filter`($\mathcal{X};\mathcal{Y}$) Performs a certain true/false check on every element of set $\mathcal{X}$ given (optional) input $\mathcal{Y}$, and only retains those for which the check returns true.

Input: List of systems $\mathcal{S}$;
begin Stage 1: Estimating sample variances
  $\mathcal{O}_1 := \{(i, \bar{X}_i(n_1), S_i^2) : i \in \mathcal{S}\} \leftarrow \texttt{Map}_1(\mathcal{S})$;
  $\bar{S} \leftarrow \texttt{Reduce}_1(\mathcal{O}_1)$;
  $\mathcal{O}^b := \{[g(i), (i, \bar{X}_i(n_1), S_i^2, b_i)] : i \in \mathcal{S}\} \leftarrow \texttt{Map}_1^s(\mathcal{O}_1, \bar{S})$;
  $\mathcal{O}^s := \{[g(i), (i, \bar{X}_i(n_1), S_i^2, b_i)] : i$ survives screening $\} \leftarrow \texttt{Combine}(\mathcal{O}^b)$;
  $\mathcal{B} := \{(i, \bar{X}_i(n_1), S_i^2, b_i) : i$ with the highest sample mean in each group $\} \leftarrow \texttt{Map}^b(\mathcal{O}^s)$;
  $O^f := \{(i, \bar{X}_i(n_1), S_i^2, b_i) : i$ survives screening against each system in $\mathcal{B}\} \leftarrow \texttt{Filter}(\mathcal{O}^s, \mathcal{B})$;
  $r \leftarrow 1$;
  while $|O^f| > 1$ *and* $r <= \bar{r}$ **do**
    $O^b := \{[g(i), (i, \bar{X}_i(n_i), S_i^2, b_i)] : i \in O^f\} \leftarrow \texttt{Map}^s(O^f)$;
    $O^s := \{[g(i), (i, \bar{X}_i(n_i), S_i^2, b_i)] : i$ survives screening $\} \leftarrow \texttt{Combine}(O^b)$;
    $B := \{(i, \bar{X}_i(n_i), S_i^2, b_i) : i$ with the highest sample mean in each group $\} \leftarrow \texttt{Map}^b(O^s)$;
    $O^f := \{(i, \bar{X}_i(n_i), S_i^2, b_i) : i$ survives screening against each system in $B\} \leftarrow \texttt{Filter}(O^s, B)$;
    $r \leftarrow r + 1$;
  end
  if $|O^f| > 1$ **then**
    $h \leftarrow \texttt{Rinott}(k, \alpha_1, n_1 - 1)$;
    $O^R := \{(i, \bar{X}_i(n_i^2)) : i \in O^f\} \leftarrow \texttt{Map}^R(O^f, h)$;
    $i^* \leftarrow \texttt{Reduce}_2(O^R)$;
  else
    $i^* \leftarrow$ the remaining system in O^f;
  end
  Report i^* as the best system;
end

Figure EC.4 Spark Implementation

We now describe each parallel operation shown in Figure EC.4 as follows.

$\texttt{Map}_1(\mathcal{S})$ maps each system in $\mathcal{S}$ to an $(i, \bar{X}_i(n_1), S_i^2)$ data point containing the first-stage sample mean $\bar{X}_i(n_1)$ and variance S_i^2, by simulating it for n_1 replications. Parallelization is trivial as the data points can be mapped independently.

$\texttt{Reduce}_1(\mathcal{O}_1)$ computes the average sample standard deviation $\bar{S} := \sum_{i \in \mathcal{S}} S_i / |\mathcal{S}|$ using the dataset $\mathcal{O}_1$. In particular, the summation can be performed in parallel by distributing to workers.

$\texttt{Map}_1(\mathcal{O}, \bar{S})$ calculates the batch size b_i and the screening group $g(i)$ for each system. The results are in a [`key`, `value`] form where the `key` is the group assignment $g(i)$ and the `value` is the Stage 1 statistics and batch size b_i.

$\texttt{Combine}(\mathcal{O}^b)$ performs screening in each individual group (with the same key), dropping systems that are eliminated. For parallelization, each small-group screening can be sent to a worker.

$\texttt{Map}^b(\mathcal{O}^s)$ maps each group to a single data point containing the identity of the best system in the group and its sample statistics.

$\texttt{Filter}^b(\mathcal{O}^s, \mathcal{B})$ screens each system i in $\mathcal{O}^s$ against the set of best systems $\mathcal{B}$. Only those systems surviving all comparisons with members of $\mathcal{B}$ survive. Again, parallelization is trivial as each system can be compared against $\mathcal{B}$ independently.

$\texttt{Map}^s(\mathcal{O}^f)$ runs an additional b_i simulation replications for each system i and updates the statistics.

$\texttt{Map}^R(\mathcal{O}^f, h)$ calculates the Stage 2 sample size using the Rinott constant h, runs any additional replications, and updates statistics. (This step can be optimized further by breaking the Stage 2 samples into small workloads of approximately equal size which are run separately, but we omit the details here for simplicity.)

$\texttt{Reduce}_2(\mathcal{O}^R)$ simply finds the system with the highest sample mean.