EFFICIENT RANKING AND SELECTION IN PARALLEL COMPUTING ENVIRONMENTS

A Dissertation

Presented to the Faculty of the Graduate School

of Cornell University

in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy

by

Cao Ni

February 2016

EFFICIENT RANKING AND SELECTION IN PARALLEL COMPUTING ENVIRONMENTS

Cao Ni, Ph.D.

Cornell University 2016

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. However, nontrivial statistical and implementation issues arise when designing R&S procedures for a parallel computing environment. This dissertation develops efficient parallel R&S procedures.

In this dissertation, several design principles are proposed 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 the three parallel R&S procedures analyzed, each of which features a unique sampling and screening approach, and a specific statistical guarantee on the quality of the final solution. Finally, in our computational study we discuss three methods for implementing R&S procedures on parallel computers, namely the Message-Passing Interface (MPI), Hadoop MapReduce, and Apache Spark, and show that MPI performs the best while Spark provides good protection against core failures at the expense of a moderate drop in core utilization.

BIOGRAPHICAL SKETCH

Cao Ni (Eric) grew up in Hangzhou, an old city in China which many consider as the beginning of the Silk Road. His primary school is most famous for a badminton team and his high school is dedicated to training ambassadors, but Eric somehow developed an interest in numbers and logic as a boy. After high school he attended National University of Singapore and graduated with first-class honors degrees in engineering and economics.

During the four years spent in Ithaca, New York, Eric goes to the cinema and visits museums regularly, and enjoys his spring breaks in the Caribbeans. He has had fond memories attending ballroom dancing, golfing, and wine tasting courses at Cornell, and served as a teaching assistant for multiple courses. Upon graduating from Cornell, Eric will move to London, United Kingdom where he will begin his job as quantitative strategist working on equity derivatives for Goldman Sachs.

To my parents.

ACKNOWLEDGEMENTS

First and foremost, I wish to express my sincere gratitude to my advisor Shane Henderson for his tremendous guidance and support over the years. His patience, humor, and immense knowledge have been a constant source of motivation and I could not wish for a better or friendlier advisor.

Much of this thesis is joint work with Susan Hunter of Purdue University and Dragos Florin Ciocan of INSEAD, to whom I owe my special appreciation. Besides brilliant research ideas, I benefited enormously from their rigorous work attitude and writing styles.

I thank Peter Frazier and José Martínez for sitting on my research committee and offering helpful comments. I also thank the rest of the faculty and staff at the School of Operations Research and Information Engineering, especially Gennady Samorodnitsky, Dawn Woodard, and Mark Lewis, for whom I have had the great pleasure to work as teaching assistant.

It is a privilege to work among an outstanding and most easy-going group of Ph.D. students at Cornell. I praise the valuable insights into many aspects of graduate student life given by Jiawei Qian, Yi Shen and Chao Ding when I first came here. Thank you to my office mates Nanjing Jian, Yi Xuan Zhao, and Jiayi Guo, who never refuse to enjoy a moment of fellowship with me, be it in our office, in the movie theaters, or during week-long trips on the road.

I thank Jiayang Gao for placing her trust in me and being my best friend. Last but not the least, I am grateful to parents Peizhen Xu and Guoliang Ni for their undivided attention and love.

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

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

1.1 Background

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` [22]. For overviews of methods to solve the SO problem, see, e.g., [14, 1, 16, 48].

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

$$\max_{i \in \mathcal{S}} \ \mu_i = E[X(i;\xi)] \tag{1.1}$$

where $\mathcal{S} = \{1, 2, \ldots, k\}$ 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, unbeknownst to the selection procedure, that $\mu_1 \leq \mu_2 \leq \cdots \leq \mu_k$, and will refer to system k as "the best", albeit multiple best systems may exist. 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 only methods to solve the SO problem in (1.1) that (i) do not exploit such topology or structural properties of the function, and that (ii) apply when the computational budget permits at least

some simulation of *every* system. Such methods are called *ranking and selection* (R&S) procedures.

R&S procedures are frequently used in simulation studies because structural properties, such as convexity, are difficult to verify for simulation models and rarely hold. They can also be used in conjunction with heuristic search procedures in a variety of ways [49, 3], making them useful even if not all systems can be simulated. See [27] for an excellent introduction to, and overview of, R&S procedures. R&S problems are closely related to best-arm problems, but there are several differences between these bodies of literature. Almost always, the algorithms developed in the best-arm literature assume that only one system is simulated at a time see, e.g., [24, 5] and that simulation outputs are bounded, or are normally distributed and all variances have a known bound.

R&S procedures are designed to offer one of several types of probabilistic guarantees, and can be Bayesian or frequentist in nature. Bayesian procedures offer guarantees related to a loss function associated with a non-optimal choice; see [4] and [7]. 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 greater than $1 - \alpha$. 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 greater than $1 - \alpha$. That is, the PGS guarantee implies PGS = P[Select a system K 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 [37].

Traditionally, R&S procedures were limited to problems with a modest number of systems k, say $k \leq 100$, due to the need to assume worst-case mean configurations to construct validity proofs. The advent of screening, i.e., discarding clearly inferior alternatives early on [40, 29, 23], 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., [15]. 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.

[21, 17, 18] explored the use of parallel computers to construct valid simulation estimators, but R&S procedures that exploit parallel computing have emerged only recently. [36] and [54] 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.) [9] tests a sequential pairwise hypothesis testing approach on a local network of computers. More recently, [34] develop a parallel adaptation of a fully-sequential R&S procedure that provides an asymptotic (as $\delta \to 0$) PCS guarantee. [34] is the best known existing method for parallel ranking and selection that provides a form of PCS guarantee on the returned solution, and is an outgrowth of [35].

1.2 Contributions

In this thesis, we (i) identify opportunities and challenges that arise from adopting a parallel computing environment to solve large-scale R&S problems, (ii) propose a number of procedures that solve R&S problems on parallel computers, and (iii) implement and test our procedures 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 the construction of our parallel procedures. Our procedures showcase the power of these design principles in that they greatly extend the boundary on the size of solvable R&S problems. While the method of [34] can solve on the order of 10^4 systems, one of our implementations of Good Selection Procedure (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 thesis is the redesigned screening method in GSP which, unlike many fully-sequential procedures [28, 23], 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. GSP then provides the same PGS guarantee as existing indifference-zone procedures like [40] but with far smaller sample sizes.

Practical contributions. The parallel procedures discussed in this thesis are 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,

they should deliver expected results regardless of the hardware specification. The procedures are also amenable to a range of existing parallel computing frameworks. For instance, we offer implementations of GSP based on MPI (Message-Passing Interface), Apache Hadoop MapReduce, and Apache Spark, and show how the implementations differ in construction and in performance. The reasons for our choice of implementation frameworks are twofold:

- Both MPI and MapReduce 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. Spark is a fast-growing parallel computing framework that has become increasing popular within the data analytics community thanks to its remarkable performance improvement over MapReduce.
- MPI and MapReduce/Spark provide points of comparison between two different parallel design philosophies. Broadly speaking, the former enables low level tailoring and optimization in the implementation of a parallel procedure, while the latter is more of a "one-size-fits-all" framework that delegates as much of the implementation complexity as possible to the MapReduce or Spark packages themselves.

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. On the other hand, MapReduce and Spark 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 the two, Spark is more efficient.

The remainder of the thesis is organized as follows. Chapter 2 discusses the

design principles followed in creating GSP to promote efficiency and ensure the procedure's validity. The contents of Chapter 2 are contained in [45] which has been submitted for publication. Chapters 3, 4, and 5 each describes a parallel R&S procedures and establishes its statistical guarantee. Initial versions of these procedures have appeared in a series of conference papers [47, 46, 44]. Computational studies in Chapter 6 support our assertions on the quality of GSP and its parallel implementations, and point to open-access repositories where the code can be obtained. A portion of the computational studies are presented in [45].

CHAPTER 2

DESIGN PRINCIPLES FOR RANKING AND SELECTION ALGORITHMS IN HIGH PERFORMANCE COMPUTING ENVIRONMENTS

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 chapter, 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.

For discussing the design principles for parallel R&S procedures in this chapter, 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 §6.3 where we discuss Hadoop MapReduce and Apache Spark.

Similar to [34] and [47], 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.

2.1 Implications of Random Completion Times

Consider the simplest case where only Task (2), running simulations, is run in parallel, and each simulation 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 [21]. [21, 17, 18] 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 [21, 18, 47, 34], we confine our attention to estimators that produce a fixed number of replications in a random completion time. (The cause of this difficulty can be traced to dependence between the estimated objective function and computational 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. A naïve way is to collect replications from any core following the order in which they are generated, but as demonstrated by the following example, the estimators may be biased, making it hard to establish provable statistical guarantees.

Example 1. *Suppose each worker $j = 1, 2$ can independently generate iid replications $X_{j1}, X_{j2}, \ldots$ of the same system, with associated generation times $T_{j1}, T_{j2}, \ldots$. Such realizations may be obtained through the use of a random number generator with many streams and substreams, as discussed in §2.3.*

Suppose that the first replication from Worker 1 has the same distribution as the first replication from Worker 2, as would arise if we used the same code on identical cores. Let the joint distribution of the first replication from Worker j, (X_{j1}, T_{j1}), be such that X_{j1} is (marginally) normal$(0, 1)$, and let

$$T_{j1} = \begin{cases} 1 & \text{if } X_{j1} < 0, \\ 2 & \text{if } X_{j1} \geq 0, \end{cases}$$

for $j = 1, 2$. Hence it takes twice as long to generate larger values as smaller values. Let T_{*1} be the time at which the master receives the first replication, or replications in the event of simultaneous arrivals. Due to the marginal normality of X_{j1}, $j = 1, 2$, we have

$$\underbrace{P(X_{11} < 0, X_{21} < 0)}_{T_{*1}=1} = \underbrace{P(X_{11} < 0, X_{21} \geq 0)}_{T_{*1}=1} = \underbrace{P(X_{11} \geq 0, X_{21} < 0)}_{T_{*1}=1} = \underbrace{P(X_{11} \geq 0, X_{21} \geq 0)}_{T_{*1}=2} = 1/4. \tag{2.1}$$

Now consider the expected value of the first replication(s) received by the master. Let N^- and N^+ be random variables whose distribution is the same as $X_{j1} || X_{j1} < 0$ and

$X_{j1}||X_{j1} \geq 0$, *respectively,* $j = 1, 2$. *In all cases except the last in expression* (2.1), *the first replication(s) to report will be* N^- *because they are computed in only one time unit. Thus, the first communication received at the master is*

- *two iid replications of* N^- *after 1 time unit with probability 1/4,*
- *one replication of* N^- *after 1 time unit with probability 1/2, or*
- *two iid replications of* N^+ *after 2 time units with probability 1/4.*

The expected value of the first communication received at the master (where this value is assumed to be the average of the values of two replications if they are received simultaneously) is therefore

$$\frac{3}{4}E(N^-) + \frac{1}{4}E(N^+) = \frac{1}{2}E(N^-) < 0,$$

reflecting a negative bias, so that the first replication received is not *distributed as* X_{11}. *A similar problem arises if we average the replications that are received after any deterministic amount of time. For example, if we wait two time units and average the results received, we obtain an average of* $\frac{1}{12}E(N^-) < 0$. □

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 parallel procedures in subsequent chapters are constructed such that the simulation results generated in parallel are initiated, collected, assembled and used by the screening routine in an ordered manner. Specifically, in the iterative screening stages of both NHH and GSP, when the master instructs a worker to simulate system i for a batch of replications, the

batch index is also received by the worker. When the batch is completed, its statistics are sent back to the master alongside the batch index, which signals its pre-determined position in the assembled batch sequence on the master. This ensures that the batch statistics sent to workers for screening follow the exact order in which they were initiated, and constructed estimators are unbiased with the correct distribution. A similar approach is discussed in [34] and is referred to as "vector-filling".

2.2 Allocating Tasks to the Master and Workers

Previous work on parallel R&S procedures [8, 54, 35, 34] 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 [35], 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 [34]. 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 pro-

cedure 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 we have shown 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.

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 [23] as well as NHH and 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, [23] 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 sys-

tem 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

$$\frac{n_i T_i}{n_j T_j} = \frac{\sigma_i \sqrt{T_i}}{\sigma_j \sqrt{T_j}}. \tag{2.2}$$

This result is consistent with a conclusion in [19], 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).

2.2.3 Distributed screening

In fully sequential R&S procedures, e.g., [29, 23], 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 is proportional to the number of pairs of systems, which is $O(k^2)$.

In the serial R&S literature, the computational cost of screening is assumed to be negligible compared to that of simulation because the number of systems

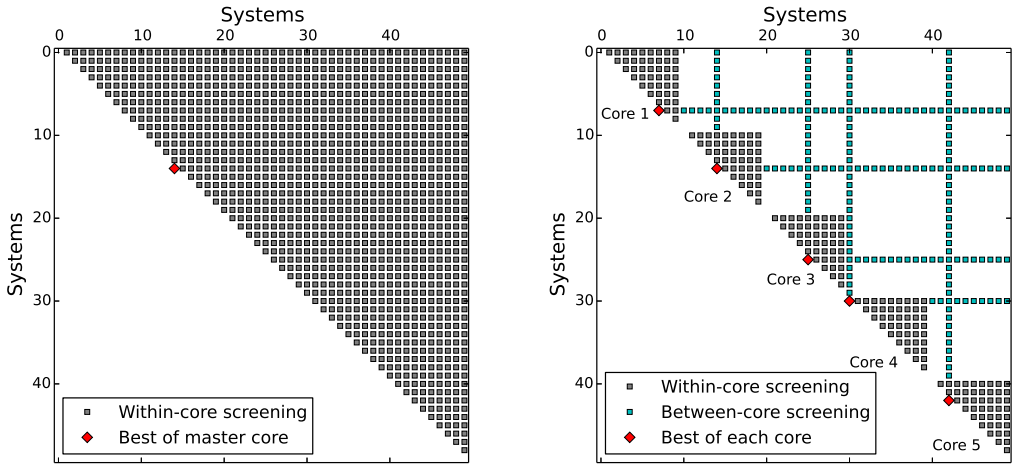

Figure 2.1: Comparison of screening methods applied on 50 systems. Each black or green dot represents a pair of systems to be screened. In the left panel, all pairs of screening is done on the master. In the right panel, each worker core gets 10 systems, screens between themselves, and screens its systems against one system from every other worker that has the highest sample mean.

k is usually quite small and each simulation replication may take orders of magnitude longer than $O(k^2)$ screening operations required in each iteration. Under this assumption, it is 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 amount of $O(ck^2)$ processing time is wasted, 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, reducing the benefits from a parallel implementation dramatically. 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 amount of $O(k^2)$ memory may easily 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 it screen among those systems only, as illustrated in Figure 2.1. 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. 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 2.1, for example, each core shares its own (estimated) 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 2.1, 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 statistical validity of some screening-based R&S procedures (e.g. [28, 23, 34]) requires screening to be performed once every replication (or batch of replications) is simulated. This implies that, when the identity of the estimated-best system(s) changes, the master has to communicate all previous replication results of the new estimated-best system(s) to the workers, so that they can perform all of the screening steps up to the current replication to ensure validity

of the screening. (If screening on a strict subsequence of replications, it may be sufficient to communicate summary statistics.) Such "catch-up" screening was used, for instance, in [49], in a different context. In chapter 3, we argue that catch-up screening is essential in providing the CS guarantee. In chapter 5, we employ a probabilistic bound that removes 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 NHH and 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, 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 experiments and simulation optimization procedures relies substantially on the quality and efficiency of (pseudo) random number generators. For a discussion of random number generators and their desirable properties, see [30].

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 litera-

ture. [38] consider a class of random number generators which are parametrized so that each valid parametrization is assigned to one core. [26] adopt [31]'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 [31] for an example.

Although the procedures discussed in this paper do 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 $\{U_0(\ell) : 1 \leq \ell \leq L\}$ for sufficiently large L. Let the master keep variables $\{\ell_i : i = 1, 2, \ldots, k\}$ which 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 $\{U_0(\ell)\}$.

2.4 Synchronization and Load-Balancing

To the extent possible, we should avoid synchronization delays, where one core cannot continue until another core completes its task. In an asynchronous procedure, each worker communicates with the master whenever its current task is completed, regardless of the status of other workers. During each communication phase, the master performs a small amount of work determining the next task (either screening or simulation) for the free worker, then assigns the task to the worker. Because point-to-point communication is fast relative to a simulation or screening task, the master is almost always idle and ready for the next communication, and free workers spend very little time waiting for the master.

Whereas synchronized procedures may require evenly distributing workloads across multiple workers between synchronizations to minimize idling time, asynchronous procedures achieve this automatically, as any idle worker shall receive a task from the master almost instantly. Hence, it is essential to ensure that communication size and frequency are carefully controlled, as discussed in §2.2.1, so the master remains idle most of the time.

For procedures that iteratively simulate systems and screen out inferior ones, asynchronism also helps to naturally adapt to system elimination and automatically balances screening and simulation. At the beginning, there may still be a large number of systems remaining so screening may be run on many workers.

As more systems are eliminated, workers spend relatively less time on screening. Eventually most workers will cease to perform screening because all systems assigned to them will be eliminated, and those workers will spend their remaining computing time running new simulations only.

CHAPTER 3

THE NHH PARALLEL R&S PROCEDURE WITH CORRECT SELECTION GUARANTEE

3.1 Introduction

An important class of traditional R&S procedures are fully sequential in nature. A fully sequential procedure maintains a sample of simulation results for each system, iteratively grows the sample size by running additional simulation replications, and periodically screens out inferior systems by running certain statistical tests on available simulation results. By design, such procedures tend to allocate more computation budget towards systems with higher expected performances, by gradually increasing the sample size of each system and eliminating a system as soon as there is sufficient statistical evidence that its performance is not the best. For examples of fully-sequential R&S procedures, see [28, 23].

In this chapter, we focus on the issue of adjusting a fully-sequential R&S procedure, namely the unknown variance procedure (UVP) in [23], in such a way that it runs on a parallel computer efficiently and delivers the same statistical guarantee. To achieve this, we launch small simulation and screening jobs independently on parallel workers, avoid screening all system pairs, and carefully manage the sequence in which simulation results are collected and screened. The resulting parallel procedure, called NHH, first appeared in [47]. NHH applies to the general case in which the system means and variances are both unknown and need to be estimated, and does not permit the use of common random numbers. Under the PCS assumption $\mu_k - \mu_{k-1} \geq \delta$, the NHH

procedure provides a guarantee on PCS for normally distributed systems. The method of parallelism in NHH was partly motivated by [33].

The NHH procedure includes an (optional) initial stage, Stage 0, where 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. Stage 0 samples are then dropped and not used to form estimators of μ_i's due to the potential correlation 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 simulation statistics and independently perform screening over a subset of systems until all but one system is eliminated.

The sampling rules used in Stages 0 and 1 are relatively straight forward, 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 (see §2.2.1). Prior to the start of the overall selection procedure 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 batch size for system i be proportional to $S_i/\sqrt{\bar{T}_i}$ in order to efficiently

allocate simulation budget across systems, we use

$$n_i(r) = n_1 + r\left[\beta\left(\frac{S_i}{\sqrt{\bar{T}_i}}\right) \Big/ \left(\frac{1}{k}\sum_{j=1}^{k}\frac{S_j}{\sqrt{\bar{T}_j}}\right)\right], \tag{3.1}$$

where $\bar{T}_i$ is an estimator for 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 $1 - \alpha$, 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.

A typical choice for the error rate is $\alpha = 0.05$ for guaranteed PCS of 95%. 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. However, note that NHH offers a PCS guarantee which depends on the CS assumption $\mu_k - \mu_{k-1} \geq \delta$, and a large δ that violates the assumption may render the PCS invalid. 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.

For non-normal simulation output, we recommend setting $\beta \geq 30$ to ensure normally distributed batch means. 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, where c is the number of workers employed. 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.

Finally, for any two systems $i \neq j$, define

$$t_{ij}(r) = \left[\frac{\sigma_i^2}{n_i(r)} + \frac{\sigma_j^2}{n_j(r)}\right]^{-1}, \qquad Z_{ij}(r) = t_{ij}(r)[\bar{X}_i(n_i(r)) - \bar{X}_j(n_j(r))],$$
$$\tau_{ij}(r) = \left[\frac{S_i^2}{n_i(r)} + \frac{S_j^2}{n_j(r)}\right]^{-1}, \qquad Y_{ij}(r) = \tau_{ij}(r)[\bar{X}_i(n_i(r)) - \bar{X}_j(n_j(r))].$$

We will show later that the statistics $Z_{ij}(r)$ and $Y_{ij}(r)$ can be related to some time-scaled Brownian motions observed at times $t_{ij}(r)$ and $\tau_{ij}(r)$, respectively. This relationship will allow us to leverage existing theories on Brownian motion to establish the probability guarantees for our procedures.

3.2 Procedure NHH

In this section, we present NHH in full details. We start by outlining the algorithmic structure of the procedure as follows.

1. Select overall confidence level $1 - \alpha$, practically significant difference δ, Stage 0 sample size $n_0 \geq 2$, Stage 1 sample size $n_1 \geq 2$, and number of systems k. Let $\lambda = \delta/2$ and a be the solution to

$$E\left[\frac{1}{2}\exp\left(-\frac{a\delta}{n_1 - 1}R\right)\right] = 1 - (1 - \alpha_1)^{\frac{1}{k-1}}, \tag{3.2}$$

   where R is the minimum of two i.i.d. χ^2 random variables, each with $n_1 - 1$ degrees of freedom. Let the distribution function and density of such a χ^2 random variable be denoted $F_{\chi^2_{n_1-1}}(x)$ and $f_{\chi^2_{n_1-1}}(x)$, respectively. Hence R has density $f_R(x) = 2[1 - F_{\chi^2_{n_1-1}}(x)]f_{\chi^2_{n_1-1}}(x)$.

2. **(Stage 0, optional)** Master sends an approximately equal number of systems to each worker. Each system i is simulated for n_0 replications and its average completion time $\bar{T}_i$ is reported to the master.

3. **(Stage 1)** Master assigns systems to load-balanced simulation groups G_1^w for $w = 1, \ldots, c$ where c is the total number of workers (using information from Stage 0, if completed).

4. 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 if there exists a system $j \in G_1^w : j \neq i$ such that $Y_{ij}(r) < \min[0, -a + \lambda\tau_{ij}(r)]$.

   (d) Report survivors, together with their Stage 1 sample means and variances, to the master.

5. **(Stage 2)** Master assigns surviving systems $\mathcal{S}$ to approximately equal-sized screening groups G_w^2 for $w = 1, \ldots, m$. Master determines a sampling rule $\{n_i(r) : i \in \mathcal{S}, r = 1, 2, \ldots\}$ where each $n_i(r)$ represents the total number of replications to be collected for system i by iteration r. A recommended choice is $n_i(r) = n_1 + r\lceil\beta S_i\rceil$ where β is a constant and a large batch size $\lceil\beta S_i\rceil$ reduces communication.

6. For $w = 1, 2, \ldots, m$ in parallel on workers (this step entails some communication with the master in steps (6b) through (6e), the details of which are omitted):

   (a) Set $r_w \leftarrow 1$. Repeat steps (6b) through (6f) until $|\mathcal{S}| = 1$:

(b) If the r_wth iteration has completed for all systems $i \in G_w^2$ and $|G_w^2| > 1$ then go to step (6d), otherwise go to step (6c).

(c) (Following the Master's instructions) Simulate the next system i in $\mathcal{S}$ (not necessarily G_w^2) for $\lceil \beta S_i \rceil$ replications and go to step (6b).

(d) Screen within group G_w^2: system i is eliminated if there exists system $j \in G_w^2 : j \neq i$ such that $Y_{ij}(r_w) < \min[0, -a + \lambda\tau_{ij}(r_w)]$.

(e) 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_w^2.

(f) Remove any eliminated system from G_w^2 and $\mathcal{S}$. Let $r_w \leftarrow r_w + 1$ and go to step (6b).

7. Report the single surviving system in $\mathcal{S}$ as the best.

In Figures 3.1 through 3.2 we present NHH in greater detail, highlighting the various design principles discussed in §2. In NHH the master core allocates and distributes systems in batches, random number streams are created and distributed together with the assigned systems to ensure independent sampling, and batch statistics obtained in parallel are collected and used in the same order in which they were initiated, in order to form valid estimators for distributed screening. Furthermore, Stage 2 of NHH is completely asynchronous as each worker receives simulation or screening tasks directly from the master, independent of the progress of other workers.

The following notation for some subroutines are used:

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

Master Core Routine

Input: List of systems $\mathcal{S}$; Average Stage 2 batch size β; Parameters $\delta, \alpha, n_0, n_1, a, \lambda$ 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 $\delta, \alpha, n_0, n_1, a, \lambda$.

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 3.1: Stages 0 and 1, Procedure NHH: Master (left) and workers (right) routines

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.

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. Sim-

begin Stage 2: Iterative screening

$G_1 \leftarrow$ systems that survived Stage 1;
$\{G_2^w : w = 1, 2, \ldots, c\} \leftarrow$`Partition`$(G_1, 2)$;
$\mathcal{S} \leftarrow G_1$;
foreach *worker* $w = 1, 2, \ldots, c$ **do**
  Send G_1, G_2^w to Worker w;
  foreach *system* $i \in G_1$ **do**
    Send S_i^2 from Stage 1 to Worker w;
  end
  foreach *system* $i \in G_2^w$ **do**
    Send $\texttt{Stat}_{i,0}$ to worker w;
  end
end
$b_i \leftarrow$ `BatchSize`(i, β), $q_i \leftarrow 1$ for all $i \in G_1$;
$r_w^{\text{sent}} \leftarrow 0$, $r_w^{\text{received}} \leftarrow 0$, $r_w^{\text{screened}} \leftarrow 0$,
$\texttt{flag}_w \leftarrow 0$ for all $w = 1, 2, \ldots, c$;
while $|\mathcal{S}| > 1$ **do**
  Wait for the next worker w to call `Communicate()`;
  if $\texttt{flag}_w = 1$ **then**
    /* Send screening task to worker w */
    $\{i, q_i, \texttt{Stat}_{i,q_i}\} \leftarrow$`RecvOutput`$(w)$;
  else if $\texttt{flag}_w = 2$ **then**
    /* Send simulation task to worker w */
    $\{\mathcal{S}, G_2^w, r_w^{\text{screened}}\} \leftarrow$`RecvScreen`$(w)$;
    $\{i_w^*, r_w^{\text{received}}, \{\texttt{Stat}_{i_w^*,r} : r \le r_w^{\text{received}}\}\} \leftarrow$`RecvBest`$(w)$;
  end
  if $|\mathcal{S}| > 1$ **then**
    $r^{\text{current}} \leftarrow$`CountBatch`$(w)$;
    if $r^{current} > r_w^{sent}$ **then**
      $\texttt{flag}_w \leftarrow 2$;
      `SendAction`$(w, \texttt{flag}_w)$;
      `SendStats`$(w, r_w^{\text{sent}}, r^{\text{current}})$;
      `SendBestStats`(w);
      $r_w^{\text{sent}} \leftarrow r^{\text{current}}$;
    else
      $\texttt{flag}_w \leftarrow 1$;
      `SendAction`$(w, \texttt{flag}_w)$;
      Select next $i \in \mathcal{S}$ such that $q_i = q_{\text{Global}}$;
      `SendSim`(w, i, q_i, b_i);
      $q_i \leftarrow q_i + 1$;
      if $q_i > q_{Global}$ *for all* $i \in \mathcal{S}$ **then**
        $q_{\text{Global}} \leftarrow q_{\text{Global}} + 1$;
      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* $i \in G_1$ **do**
  Receive S_i^2 collected in Stage 1;
end
foreach *system* $i \in G_2^w$ **do**
  Receive $\texttt{Stat}_{i,0}$ from the master;
end
$r_w \leftarrow 0$;
`Communicate`();
while *No termination instruction received* **do**
  $\texttt{flag}_w \leftarrow$`RecvAction`();
  if $\texttt{flag}_w = 2$ **then**
    $\left\{r^{\text{new}}, \{\texttt{Stat}_{i,r} : i \in G_2^w, r_w + 1 \le r \le r^{\text{new}}\}\right\}$
    $\leftarrow$`RecvStats`();
    $\{\mathcal{W}, \{r_{w'}^{received} : w' \in \mathcal{W}\}, \{\texttt{Stat}_{i_{w'}^*,r} : w' \in \mathcal{W}, r \le r_{w'}^{received}\}\}$
    $\leftarrow$`RecvBestStats`();
    $G_2^w \leftarrow$`Screen`$(G_2^w, r_w + 1, r^{\text{new}}, true)$;
    $r_w \leftarrow r^{\text{new}}$;
    `Communicate`();
    `SendScreen`(G_2^w, r_w); `SendBest`(r);
  else
    $\{i, q_i, b_i\} \leftarrow$`RecvSim`();
    `Simulate`$(i, b_i, \texttt{Stat}_{i,q_i})$;
    `Communicate`();
    `SendOutput`$(i, q_i, \texttt{Stat}_{i,q_i})$
  end
end
end

Figure 3.2: Stage 2, Procedure NHH: Master (left) and workers (right) routines

ulation of a system is no longer dedicated to a particular worker, and G_2^w is the set of systems that worker w 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 §3.2.

`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

$$b_i = \left\lceil \frac{S_i \sqrt{T_i}}{\frac{1}{|\mathcal{S}|} \sum_{j \in \mathcal{S}} S_j \sqrt{T_j}} \beta \right\rceil \tag{3.3}$$

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 a system $j \in G_2^w : j \neq i$ such that $Y_{ij}(r') < \min[0, -a + \lambda\tau_{ij}(r')]$ where Y_{ij} and a_{ij} are defined in §3.2.

In addition, if *useothers*= *true*, then for each worker $w' \neq w$ the worker w also screens its systems in G^w against system $i^*_{w'}$, the best system from worker w', using batch statistics $\{\texttt{Stat}_{i^*_{w'},r'} : r' = 1, 2, \ldots\}$ up to batch $\min\{r_{w'}^{\text{received}}, r_1\}$.

`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 $\mathcal{W} = \{w' \ne w : |G_{w'}| > 0)\}$ 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.

3.3 Analysis of Computational Complexity

Many R&S procedures guarantee to select the best system upon termination, subject to a user-specified probability of selection error. Among procedures with the same level of statistical guarantee, an efficient procedure should terminate in as few simulation replications as possible. The most efficient procedure may vary from one R&S problem to another depending on the configuration (distribution of system means and variances) of the systems in consideration. In addition, user-specified parameters such as the indifference-zone parameter δ have a significant impact on the efficiency of R&S procedures.

To assess and predict the efficiency of the NHH procedure under various configurations, we provide approximations for its expected number of replications needed upon termination. To simplify our analysis, we assume that an inferior system can only be eliminated by the best system, system k, and that system k eliminates all other systems. Strictly speaking this assumption does not apply to NHH because screening is distributed across workers and so not every system will necessarily be compared with system k. However, NHH shares statistics from systems with high means across cores during screening in Step 6e, so that a near-optimal system will be compared to inferior systems with high probability. Therefore, the total number of replications required by the procedure can be approximated by summing the number of replications needed for system k to eliminate all others. Although in practice the best sys-

tem is unknown and an inferior system may eliminate another before system k does, an inferior system i is most likely eliminated by system k because the difference between μ_k and μ_i is the largest.

In the remainder of this section we assume that system k has mean $\mu_k = 0$ and variance σ_k^2, and an inferior system i has mean $\mu_i < 0$ and variance σ_i^2. We let $\mu_{ki} := \mu_k - \mu_i > 0$.

The NHH procedure uses a triangular continuation region $C = C(a, \lambda) = \{(t, x) : 0 \leq t \leq a/\lambda, |x| \leq a - \lambda(t)\}$ and a test statistic $Z_{ij}(r) = [S_i^2/n_i(r) + S_j^2/n_j(r)]^{-1}[\bar{X}_i(n_i(r)) - \bar{X}_j(n_j(r))]$. System i is eliminated by system k when $([S_i^2/n_i(r)+S_k^2/n_k(r)]^{-1}, Z_{ki}(r))$ exits C for the first time. Using the result that $Z_{ki}(r)$ is equal in distribution to $B_{\mu_k-\mu_i}([\sigma_i^2/n_i(r) + \sigma_k^2/n_k(r)]^{-1})$ where $B_\Delta(\cdot)$ denotes a Brownian motion with drift Δ and volatility 1, we approximate the expected number of replications from system $i \neq k$ by

$$N_i^{\text{NHH}} \approx E[n_i(\inf\{r : B_{\mu_{ki}}([\sigma_i^2/n_i(r) + \sigma_k^2/n_k(r)]^{-1}) \notin C\})] \approx \sigma_i(\sigma_i + \sigma_k)E[\inf\{t : B_{\mu_{ki}}(t) \notin C\}] \tag{3.4}$$

assuming, as recommended in [23] and [47], $n_i(r) \approx S_i r \approx \sigma_i r$. The last expectation is the expected time that a Brownian motion with drift μ_{ki} exits the triangular region C, and is given in [20] by

$$E[\inf\{t : B_{\mu_{ki}}(t) \notin C\}] = \sum_{\mu \in \{\mu_{ki}, -\mu_{ki}\}} \sum_{j=0}^{\infty} (-1)^j \frac{a(2j+1)}{\mu + \lambda} \cdot e^{2aj(\lambda j - \mu)} \cdot \left[\bar{\Phi}\left(\frac{(2j + 1 - (\mu + \lambda)/\lambda)a}{\sqrt{a/\lambda}}\right) - e^{2a(2j+1)(\mu+\lambda)}\bar{\Phi}\left(\frac{(2j + 1 + (\mu + \lambda)/\lambda)a}{\sqrt{a/\lambda}}\right)\right] \tag{3.5}$$

where $\bar{\Phi}(x)$ is the tail probability of the standard normal distribution and can be approximated by $x^{-1}e^{-x^2/2}/\sqrt{2\pi}$ for large x.

Equation (3.5) is complicated, so we approximate it by focusing on one parameter at a time. First, the terms in the infinite sum rapidly approach zero as

j increases, so the $j = 0$ term dominates. Second, when system i is significantly worst than system k, μ_{ki} is large, and then (3.5) is dominated by the $a/(\mu_{ki} + \lambda)$ term, which is of order $O(\mu_{ki}^{-1})$ and we expect NHH to eliminate system i in very few replications. Third, (3.5) does not involve σ_i^2 and by (3.4) the cost of eliminating system i should be approximately proportional to $\sigma_i^2 + \sigma_i\sigma_k$.

Moreover, since the indifference-zone parameters a and λ are typically chosen such that $a \propto \delta^{-1}$ and $\lambda \propto \delta$, we may analyze the expectation in (3.5) in terms of δ. After some simplification we see that as $\delta \downarrow 0$, (3.5) is dominated by the $a(2j + 1)/(\mu_{ki} + \lambda)$ term which is $O(\delta^{-1})$ when $\mu_{ki} \geq \delta$ and $O(\delta^{-2})$ when $\mu_{ki} \ll \delta$.

In conclusion, the expected number of replications needed to eliminate system i is on the order of $O((\sigma_i^2 + \sigma_i\sigma_k)\mu_{ki}^{-1}\delta^{-1})$ for sufficiently large μ_{ki} and $O((\sigma_i^2 + \sigma_i\sigma_k)\delta^{-2})$ when $\mu_{ki} \ll \delta$. This result agrees with intuition: high variances require larger samples to achieve the same level of statistical confidence, a large difference in system means helps to eliminate inferior systems early, and a more tolerant level of practical significance requires lower precision, hence a smaller sample.

3.4 Guaranteeing Correct Selection for NHH

In this section, we state and prove the probability of correct selection guarantee provided by NHH. As NHH is a parallel extension of the sequential R&S procedure in [23], its proof is closely related to that of its sequential predecessor. Nevertheless, we will highlight here some key features of the parallel procedure and how PCS is preserved.

As is common to the sequential R&S literature, the probabilistic guarantee on the final solution relies on the following assumption on the distribution of simulation output.

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.*

Remark. In a parallel environment, multiple workers may simulate the same system i and their simulation output are periodically assembled to form the sequence $\{X_{i\ell}, \ell = 1, 2, \ldots\}$, which is used for screening. Even if simulation replications generated by each worker are marginally normal and independent, to maintain the normality of the assembled sequence $\{X_{i\ell}, \ell = 1, 2, \ldots\}$, in NHH we (1) use a unique and independent random number stream on each worker so that simulation results from multiple workers are mutually i.i.d. as discussed in §2.3, and (2) assemble (batched) simulation output in a predetermined order, following the conditions given in §2.1.

We now formally state the correct selection guarantee.

Theorem 1. *Under Assumption 4, the NHH procedure selects the best system k with probability at least $1 - \alpha$ whenever $\mu_k - \mu_{k-1} \geq \delta$.*

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 1. *([23, 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 $Z(m, n) := \left[\sigma_i^2/m + \sigma_j^2/n\right]^{-1} [\bar{X}_i(m) - \bar{X}_j(n)]$ 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.*

Remark. In Lemma 1, the functions $m(r)$ and $n(r)$ represent the sampling rules (the number of simulation replications to be collected by batch r) for systems i and j, respectively. The Lemma holds if the sampling rules are deterministic, but does not generally hold if the sampling rules depend on the simulation output $\bar{X}_i(m(r))$, $\bar{X}_j(n(r))$, as subsequent estimators $\bar{X}_i(m(r+1))$, $\bar{X}_j(n(r+1))$ may lose normality [21, 18]. For NHH, note that the sampling rule for a system i depends on the Stage 0 estimate of completion time $\bar{T}_i$ and Stage 1 estimate of variance S_i^2. It is well known [6, 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 $\{\bar{X}_i(n_i(r)) : r = 0, 1, 2, \ldots\}$ sequence, and indeed we may apply Lemma 1 to relate the statistic $Z(m(r), n(r))$ to a Brownian motion.

Lemma 2. *([25, Appendix 3]) For any symmetric continuation region C and any $\Delta \geq 0$, consider two processes: a Brownian motion $B_\Delta(\cdot)$, and a discrete process obtained by observing $B_\Delta(\cdot)$ at a random, increasing sequence of times $\{t_i : i = 0, 1, 2, \ldots\}$ where $t_0 = 0$, the value of t_i for $i > 0$ depends on $B_\Delta(\cdot)$ only through its value in the period $[0, t_{i-1}]$, and the conditional distribution of $t_i|B_\Delta(t) = b(t), t \geq 0$ is the same as $t_i|B_\Delta(t) = -b(t), t \geq 0$. Define $\tau^C = \inf\{t' > 0 : B_\Delta(t') \notin C\}$ and $\tau^D = \inf\{t_i : B_\Delta(t_i) \notin C\}$. Then* $\Pr[B_\Delta(\tau^D) < 0] \leq \Pr[B_\Delta(\tau^C) < 0]$.

Remark. For NHH, the sequence of times where we observe the Brownian motion, $\{t_i : i = 0, 1, \ldots\}$, is determined by the sampling rule, which depends on the Stage 0 completion time estimate $\bar{T}_i$, Stage 1 variance estimate S_i^2 and user-specified average batch size parameter β. These quantities are independent of

the sequence $\{Z(m(r), n(r)) : r = 0, 1, \ldots\}$ and the discretely-observed Brownian motion $\{Z'(m(r), n(r)) : r = 0, 1, \ldots\}$ associated to it in Lemma 1. Therefore the conditions on $\{t_i\}$ in Lemma 2 are satisfied. Furthermore, as τ^D is defined as the *first* discrete time t_i at which B_Δ exits C, it is necessary to inspect $B_\Delta(t_i)$ sequentially for every $i = 0, 1, 2, \ldots$ in order to invoke Lemma 2. Therefore, if two systems are being screened, then screening needs to be performed at every batch. This implies that the "catch-up" screening discussed in §2.2.3 is required whenever two systems are screened against each other.

Lemma 3. *Let $i \neq j$ be any two systems. Define $a'_{ij} := \min\{S_i^2/\sigma_i^2, S_j^2/\sigma_j^2\}a$. It can be shown [23] 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 $a'_{ij} \leq t_{ij}(r)a/\tau_{ij}(r)$ regardless of the sampling rules $n_i(\cdot)$ and $n_j(\cdot)$ for all $r \geq 0$.*

Lemma 4. *([23, 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 5 ([13])**.** *Suppose $a > 0, \Delta > 0, \lambda = \Delta/2$. Let $C = \{(t, x) : 0 \leq t \leq a/\lambda, -a+\lambda t \leq x \leq a - \lambda t\}$ denote a triangular continuation region and $\tau = \inf\{t' > 0 : B_\Delta(t') \notin C\}$ be the (random) time that a Brownian motion with drift Δ exits C for the first time. Then $P[B_\Delta(\tau) < 0] = \frac{1}{2}e^{-a\Delta}$.*

Lemma 6. *([51]) Let $V_1, V_2, \ldots, V_k$ be independent random variables, and let $G^j(v_1, v_2, \ldots, v_k)$, $j = 1, 2, \ldots, p$, be non-negative, real-valued functions, each one non-decreasing in each of its arguments. Then*

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

Proof of Theorem 1. For any two systems i and j, let KO_{ij} be the event that system i eliminates system j, R_{ij} be the first batch r at which $Y_{ij}(r) \notin [-a + \lambda\tau_{ij}(r), a - \lambda\tau_{ij}(r)]$, T^1_{ij} be the first time t that $B_{\mu_i-\mu_k}(t) \notin [-\frac{t_{ki}(R_{ki})}{\tau_{ki}(R_{ki})}a + \lambda t, \frac{t_{ki}(R_{ki})}{\tau_{ki}(R_{ki})}a - \lambda t]$, T^2_{ij} be the first time t that $B_{\mu_i-\mu_k}(t) \notin [-a'_{ij} + \lambda t, a'_{ij} - \lambda t]$, and T^3_{ij} be the first time t that $B_\delta(t) \notin [-a'_{ij} + \lambda t, a'_{ij} - \lambda t]$ where a'_{ij} is defined in Lemma 3. It then follows that

$\Pr[KO_{ik}]$

$=E[\Pr[KO_{ik}|S^2_k, S^2_i]]$

$\leq E[\Pr[Y_{ki}(\tau_{ki}(R_{ki})) < -a + \lambda\tau_{ki}(R_{ki})|S^2_k, S^2_i]]$

since system i could be eliminated by some other system before eliminating system k

$=E[\Pr[Z_{ki}(t_{ki}(R_{ki})) < -\frac{t_{ki}(R_{ki})}{\tau_{ki}(R_{ki})}a + \lambda t_{ki}(R_{ki})|S^2_k, S^2_i]]$

$=E[\Pr[B_{\mu_k-\mu_i}(t_{ki}(R_{ki})) < -\frac{t_{ki}(R_{ki})}{\tau_{ki}(R_{ki})}a + \lambda t_{ki}(R_{ki})|S^2_k, S^2_i]]$ by Lemma 1

$\leq E[\Pr[B_{\mu_k-\mu_i}(T^1_{ki}) < 0|S^2_k, S^2_i]]$ by Lemma 2

$\leq E[\Pr[B_{\mu_k-\mu_i}(T^2_{ki}) < 0|S^2_k, S^2_i]]$ by Lemmas 3 and 4

$\leq E[\Pr[B_\delta(T^3_{ki}) < 0|S^2_k, S^2_i]]$ since $\mu_k - \mu_i \geq \delta$

$\leq E\left[\frac{1}{2}\exp(-a'_{ki}\delta)\right]$ by Lemma 5

$=E\left[\frac{1}{2}\exp\left(-\frac{a\delta}{n_1 - 1}\min\left\{\frac{(n_1-1)S^2_i}{\sigma^2_i}, \frac{(n_1-1)S^2_k}{\sigma^2_k}\right\}\right)\right]$

$=1 - (1-\alpha)^{\frac{1}{k-1}}$ by (3.2),

since $(n_1 - 1)S^2_i/\sigma^2_i$ and $(n_1 - 1)S^2_k/\sigma^2_k$ are i.i.d. $\chi^2_{n_1-1}$ random variables.

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

$$\Pr[\text{system } k \text{ is selected}] \geq \Pr\left\{\bigcap_{i=1}^{k-1}\overline{KO}_{ik}\right\}$$

as system k may not get screened against all systems

$$
\begin{aligned}
&= E\left[\Pr\left\{\bigcap_{i=1}^{k-1}\overline{KO}_{ik}\middle|X_{k1}, X_{k2}, \ldots\right\}\right] \\
&= E\left[\prod_{i=1}^{k-1}\Pr\left\{\overline{KO}_{ik}\middle|X_{k1}, X_{k2}, \ldots\right\}\right] \\
&\geq \prod_{i=1}^{k-1} E\left[\Pr\left\{\overline{KO}_{ik}\middle|X_{k1}, X_{k2}, \ldots\right\}\right] \text{ by Lemma 6} \\
&= \prod_{i=1}^{k-1}\Pr\left[\overline{KO}_{ik}\right] \geq \prod_{i=1}^{k-1}\left[1-\left(1-(1-\alpha)^{\frac{1}{k-1}}\right)\right] = 1-\alpha.
\end{aligned}
$$

□

CHAPTER 4

THE NSGS$_P$ PARALLEL R&S PROCEDURE WITH GOOD SELECTION GUARANTEE

4.1 Introduction

In this chapter we present a modified, parallelized version of the NSGS procedure [40], which we call NSGS$_p$. NSGS is a two-stage procedure that is straightforward: the first stage estimates system variances and screens out some systems which are clearly inferior, whereas the second stage generates additional simulation replications and selects the one with the highest overall sample mean as the best.

Unlike fully-sequential procedures which actively seek to eliminate poor systems by eliminating them as early as possible with iterative screening, NSGS screens only once and tends to consume more computation budget. Nevertheless, NSGS$_p$ is easier to implement on a parallel platform compared to NHH because the sample sizes are pre-computed, simulation is naively parallel, and only one round of screening is required in each stage.

Another important advantage of NSGS$_p$ over NHH is that its second stage sample can be used to construct simultaneous confidence intervals on $\{\mu_k - \mu_i, \forall i\}$, which can then be used to prove that NSGS$_p$ guarantees a probability of good selection, as we will see in §4.4. The idea of establishing PGS via simultaneous confidence intervals will inspire us to construct a more sophisticated procedure in Chapter 5 that continues to guarantee good-selection and is more budget-efficient.

4.2 Procedure NSGS$_p$

In NSGS$_p$, we use a master-worker framework in which, after an optional Stage 0 to estimate simulation replication completion time, the master assigns systems to load-balanced groups and deploys the groups to the workers for simulation of Stage 1. Upon completion of the required replications, the workers calculate the first stage statistics and the systems are screened — first within groups by the workers, and then across groups by the master, where only the systems that survive worker-level screening are reported to the master. The master then computes the second-stage sample sizes for the surviving systems using the Rinott function [50], and organizes the remaining required simulation replications into batches that are deployed across the workers for simulation of Stage 2. Upon completion of the required Stage 2 replications, the workers report the final sufficient statistics to the master, which compiles the results and returns the system with the largest sample mean to the user.

The procedure is detailed as follows.

1. Select overall confidence level $1 - \alpha$, type-I error rates α_1, α_2 such that $\alpha = \alpha_1 + \alpha_2$, practically significant difference δ, and first-stage sample size $n_1 \geq 2$. Let t be the upper α_1 point of Student's t distribution with $n_1 - 1$ degrees of freedom, and set $h = h(1 - \alpha_2, n_1, k)$, where h is Rinott's constant (see, e.g., [2]).
2. **(Stage 0, optional)** Conduct the sampling of Stage 0 as in Step 2 of NHH to estimate the simulation completion time for load-balancing.
3. **(Stage 1)** Master assigns systems to load-balanced groups G_w for $w = 1, \ldots, c$ where c is the total number of workers (using information from

Stage 0, if completed).

4. For $w = 1, 2, \ldots, c$ in parallel on workers:

   (a) Sample $X_{i\ell}$, $\ell = 1, 2, \ldots, n_1$ for all $i \in G_w$.

   (b) Compute first-stage sample means and variances $\bar{X}_i^{(1)} = \bar{X}_i(n_1)$ and S_i^2 for $i \in G_w$.

   (c) Screen within group G_w: For all $i \neq j$, $i, j \in G_w$, let $W_{ij} = t(S_i^2/n_1 + S_j^2/n_1)^{1/2}$. System i is eliminated if there exists system $j \in G_w : j \neq i$ such that $\bar{X}_j^{(1)} - \bar{X}_i^{(1)} > \max\{W_{ij} - \delta, 0\}$.

   (d) Report survivors to the master.

5. Master completes all remaining pairwise screening tasks according to step (4c).

6. **(Stage 2)** Let $\mathcal{I}$ be the set of systems surviving Stage 1. For each system $i \in \mathcal{I}$, the master computes the additional number of replications to be obtained in Stage 2

$$N_i^{(2)} = \max\{0, \lceil (hS_i/\delta)^2 \rceil - n_1\}. \tag{4.1}$$

7. Master load-balances the required remaining work into "batches" which are then completed by the workers in an efficient manner.

8. Master compiles all Stage 2 sample means $\bar{X}_i^{(2)} = \sum_{\ell=1}^{n_1+N_i^{(2)}} X_{i\ell}/(n_1 + N_i^{(2)})$ and selects the system with the largest sample mean as best once all sampling has been completed.

4.3 Analysis of Computational Complexity

The NSGS$_p$ procedure begins by taking a sample of n_1 replications from each system in the first stage, and uses the sample for variance estimation and a single round of screening. System i is eliminated by system k in this stage if $(t((S_i^2 + S_k^2)/n_1)^{1/2} - \delta)^+ < \bar{X}_k^{(1)} - \bar{X}_i^{(1)}$ (henceforth denoted as event A). If system i is not eliminated in the first stage, then a second-stage sample of size $N_i^{(2)}$ is taken per Equation (4.1). Dropping other systems and considering only systems i and k, we can approximate the expected number of replications from system i needed by NSGS$_p$ by $N_i^{\text{NSGS}_p} \approx n_1 + E[(1 - \mathbb{1}(A))N_i^{(2)}]$ where $\mathbb{1}(\cdot)$ is the indicator function. Replacing random quantities by their expectations, and using a deterministic approximation for the indicator-function term, we obtain the crude approximation

$$N_i^{\text{NSGS}_p} \approx n_1 + \mathbb{1}\{(t[((\sigma_i^2 + \sigma_k^2)/n_1)^{1/2} - \delta]^+ - \mu_{ki} > 0\}(\left\lceil (h\sigma_i/\delta)^2 \right\rceil - n_1)^+. \tag{4.2}$$

Like NHH, (4.2) shows that higher σ_i^2, σ_k^2, δ^{-1} or μ_{ki}^{-1} may lead to higher elimination cost. In addition, the dependence on Stage 1 sample size n_1 is somewhat ambiguous: small n_1 reduces the probability of elimination $P[A]$, whereas big n_1 may be wasteful. An optimal choice of n_1 depends on the problem configuration and is unknown from the user's perspective. Furthermore, it can be easily checked that for sufficiently large $n_1 (\geq 20)$, the constants t and h are very insensitive to n_1 and k [2].

The dependence of N_i^{NSGS} on μ_{ki} differs somewhat from that of N_i^{NHH}. For sufficiently large μ_{ki}, $P[A]$ is very low and the NSGS$_p$ procedure requires a minimum of n_1 replications for elimination. On the other hand, when μ_{ki} is small, the elimination cost is less sensitive to μ_{ki} as the Stage 2 sample size is bounded

from above. Moreover, for sufficiently large σ_i^2, σ_k^2, and δ^{-1}, the NSGS$_p$ procedure almost always enters Stage 2, and (4.2) implies that the elimination cost is then directly proportional to σ_i^2 and δ^{-2}.

To summarize, it takes $O(\sigma_i^2\delta^{-2})$ replications for the NSGS$_p$ procedure to eliminate an inferior system i. Compared to our previous analysis on NHH, this result suggests that NSGS$_p$ is less sensitive to σ_k^2, but more sensitive to small δ, and may spend too much simulation effort on inferior systems when μ_{ki} is large.

4.4 Guaranteeing Good Selection for NSGS$_p$

NSGS$_p$ preserves the statistical guarantee of NSGS because it simply farms out the simulation work to parallel processors while employing the same sampling and screening rules as the original NSGS. In this section, we show that NSGS$_p$ guarantees a probability of good selection (PGS), which is stronger than the probability of correct selection (PCS) guarantee provided by NHH and many other fully-sequential procedures. The PGS guarantee of NSGS$_p$ is formally stated as follows.

Theorem 2. *Under Assumption 4, NSGS$_p$ selects a system K that satisfies $\mu_k - \mu_K \leq 2\delta$ with probability at least $1 - \alpha$.*

The statistical guarantee of NSGS$_p$ does not rely on the PCS assumption $\mu_k - \mu_{k-1} \geq \delta$ which is required by NHH to establish a lower bound on the probability that the best system k is selected. For problems where the PCS assumption holds, Theorem 2 implies that NSGS$_p$ guarantees to select the best system k with probability $1 - \alpha$. If δ is chosen to be greater than $\mu_k - \mu_{k-1}$, the PCS probabilis-

tic guarantee may become invalid. In practice, the true difference $\mu_k - \mu_{k-1}$ is unknown and it is impractical to choose δ to safeguard against violation of the PCS guarantee. Moreover, as the experimenter may have a tolerance for a small deviation from the true best solution, the parameter δ is often interpreted as the smallest difference worth detecting [27] and a budget-savvy experimenter may have incentive to choose a large δ which often leads to a much smaller required sample size (§3.3, 4.3). Therefore, a procedure with PGS may find more practical relevance than a PCS procedure, especially for problems where many near-optimal systems exist.

In the remainder of this section, we discuss two important Lemmas which can be used to establish PGS for multi-stage procedures, particularly NSGS$_p$, in §4.4.1 and §4.4.2. We then complete the proof of Theorem 2 in §4.4.3.

4.4.1 An α-Splitting Lemma for Multi-Stage R&S Procedures

NSGS$_p$ essentially employs two distinct screening and selection methodologies in the process of determining the best system. The Stage 1 screening eliminates obviously inferior systems and the Rinott step in Stage 2 selects one system from those that survived Stage 1. Both stages are subject to Type-I errors with rates α_1 and α_2, respectively. To bound the overall Type-I error rate of the NSGS$_p$ procedure, we employ a decomposition Lemma which shows that $\alpha \leq \alpha_1 + \alpha_2$, i.e. the Type-I error rate is "split" between the two stages.

Here we state and prove a generalization of the decomposition Lemma proposed in [40] which applies to NSGS$_p$ and also will be used to prove PGS for the GSP procedure in Chapter 5. Let $\mathcal{P}$ be a procedure with two phases, $\mathcal{P}_1$ and

$\mathcal{P}_2$. $\mathcal{P}_1$ takes as input the set of systems $\mathcal{S}$ and outputs a subset $I \subseteq \mathcal{S}$ of systems, whereas $\mathcal{P}_2$ selects a system $K \in I$ as the best. Let B_1 be the event that $\mathcal{P}_1$ yields a "successful" selection in the sense that the subset I selected in phase 1 satisfies a specific criteria (for instance, letting $B_1 = \{k \in I\}$ means $\mathcal{P}_1$ is successful if and only if a best system k survives phase 1), and let $\mathcal{J}$ be the collection of all such subsets. Let $B_2(I)$ be the event that the system K selected in phase 2 satisfies a specific criterion when the input to $\mathcal{P}_2$ is some subset of systems I. Finally, denote $B_2 = \cap_{I \in \mathcal{J}} B_2(I)$.

Lemma 7. *Suppose that*

(a) $P[B_1] \geq 1 - \alpha_1$*, i.e. phase 1 has a high probability of success;*

(b) $P[B_2(I)] \geq 1 - \alpha_2$ *for any* $I \in \mathcal{J}$*, i.e. phase 2 has a high probability of success whenever phase 1 is successful;*

(c) $\mathcal{S} \in \mathcal{J}$*, in other words, the event that all systems survive* $\mathcal{P}_1$ *is considered a success; and*

(d) $B_2(\mathcal{S}) \subseteq B_2(I)$ *for all* $I \in \mathcal{J}$*, that is, for any outcome that yields a successful selection when* $\mathcal{P}_2$ *is applied to the entire set, then a successful selection would be made if* $\mathcal{P}_2$ *is applied to any subset* $I \in \mathcal{J}$*.*

Then $P[B_1 \cap B_2] \geq 1 - (\alpha_1 + \alpha_2)$*.*

Proof. Note that

$$\begin{aligned} P[B_1 \cap B_2] =& P[B_1] + P[B_2] - P[B_1 \cup B_2] \\ \geq& P[B_1] + P[B_2(\mathcal{S})] - P[B_1 \cup B_2] \\ \geq& (1 - \alpha_1) + (1 - \alpha_2) - 1, \end{aligned}$$

where the first inequality follows because $P[B_2] = P[\cap_{I \in \mathcal{J}} B_2(I)] \geq P[B_2(\mathcal{S})]$. □

4.4.2 Providing PGS for the Rinott Stage

The following Lemma proposed by [39] provides a recipe for PGS.

Lemma 8. *Let $I = \{1, 2, \ldots, k_I\}$ be a (sub)set of systems with expected output $\mu_1 \leq \mu_2 \leq \ldots \leq \mu_{k_I}$, and $\mathcal{R}$ be a procedure that constructs an estimator $\hat{\mu}_i$ for every system $i \in I$ and selects the system $K_I = \arg\max_i \hat{\mu}_i$.*

Furthermore, let $\zeta_1, \zeta_2, \ldots, \zeta_{k_I}$ be k_I systems whose output have the same distribution as systems $1, 2, \ldots, k_I$, respectively except that their expected output are $\theta_1, \theta_2, \ldots, \theta_{k_I}$ with $\theta_{k_I} = \mu_{k_I}$ and $\theta_i = \mu_{k_I} - \delta$ for all $i \neq k_I$. Let $\hat{\theta}_i$ be the estimator for system ζ_i obtained through applying $\mathcal{R}$ on I.

If $\mathcal{R}$ provides a PCS guarantee

$$P[\hat{\mu}_{k_I} > \hat{\mu}_i, \forall i \neq k_I | \mu_{k_I} - \mu_{k_I - 1} \geq \delta] \geq 1 - \alpha,$$

and if

$$\begin{pmatrix} \hat{\mu}_{k_I} \\ \hat{\mu}_{k_I - 1} + (\mu_{k_I} - \mu_{k_I - 1} - \delta) \\ \vdots \\ \hat{\mu}_1 + (\mu_{k_I} - \mu_1 - \delta) \end{pmatrix} \overset{\mathcal{D}}{=} \begin{pmatrix} \hat{\theta}_{k_I} \\ \hat{\theta}_{k_I - 1} \\ \vdots \\ \hat{\theta}_1 \end{pmatrix} \tag{4.3}$$

where $\overset{\mathcal{D}}{=}$ denotes equal in distribution, then $\mathcal{R}$ also guarantees PGS,

$$P[\mu_{K_I} \geq \mu_{k_I} - \delta] \geq 1 - \alpha.$$

Proof. Note that (4.3) implies that

$$P[\hat{\mu}_{k_I} - \hat{\mu}_i - (\mu_{k_I} - \mu_i) > -\delta, \forall i \neq k_I] = P[\hat{\theta}_{k_I} - \hat{\theta}_i > 0, \forall i \neq k_I] \geq 1 - \alpha$$

where the second inequality follows from the PCS guarantee of $\mathcal{R}$. The result then follows by noticing that if $K_I \neq k_I$, then $\hat{\mu}_{k_I} - \hat{\mu}_{K_I} \leq 0$ and therefore

$$\begin{aligned}\hat{\mu}_{k_I} - \hat{\mu}_i - (\mu_{k_I} - \mu_i) > -\delta, \forall i \neq k_I &\Rightarrow \hat{\mu}_{k_I} - \hat{\mu}_{K_I} - (\mu_{k_I} - \mu_{K_I}) > -\delta \\ &\Rightarrow \mu_{K_I} + (\hat{\mu}_{k_I} - \hat{\mu}_{K_I}) > \mu_{k_I} - \delta \\ &\Rightarrow \mu_{K_I} > \mu_{k_I} - \delta.\end{aligned}$$

□

Remark. Condition (4.3) requires that the estimators constructed by the procedure $\mathcal{R}$ to be invariant to a shift in expected output for all systems in I. The Rinott step used in Stage 2 of NSGS$_p$ is one such procedure that satisfies this condition. Indeed, for the Rinott procedure, the estimator for μ_i is a sample mean of system i with a size determined by the variance estimate S_i. Alternating μ_i to θ_i does not change the Rinott sample size for ζ_i, hence (4.3) holds. Coupling this fact with a PCS guarantee [50] yields a PGS for the Rinott procedure.

On the other hand, iterative screening procedures such as Stage 2 of NHH are typically designed in a way that the required sample size for a system i is random and generally decreases as the true difference between μ_i and μ_k increases. For such procedures, a shift in expected output from μ_i to θ_i will certainly affect the sample size and hence the distribution of their estimators, holding other things equal. Therefore condition (4.3) is violated and Lemma 8 does not apply.

4.4.3 Proof of Good Selection

To complete the proof of Theorem 2 we need the following Lemma which guarantees the quality of the results from Stage 1.

Lemma 9 (Section 4, [40]). *With probability at least* $1-\alpha_1$*, a system* $K_1 \in \mathcal{S}$ *satisfying* $\mu_{K_1} \geq \mu_k - \delta$ *survives Stage 1 of NSGS*$_p$.

Now we are in a position to finally prove Theorem 2.

Proof of Theorem 2. To prove the PGS guarantee for the two-stage NSGS$_p$ procedure, we will argue that

1. A system K_1 with $\mu_{K_1} \geq \mu_k - \delta$ survives Stage 1 with probability $1 - \alpha_1$ (Lemma 9).
2. Among the set of systems surviving Stage 1, Stage 2 selects a system within δ of the best (Lemma 8).
3. As a result, the overall procedure provides PGS at least $1 - \alpha_1 - \alpha_2$ (Lemma 7).

Apply Lemma 7 and define $B_1 = \{\exists K_1 \in I : \mu_{K_1} \geq \mu_k - \delta\}$ and $B_2(I) = \{\text{select system } K \in I : \mu_K - \max_{i \in I} \mu_i \geq -\delta\}$. By Lemmas 8 and 9 it is easy to verify that NSGS$_p$ satisfies conditions (a)-(d) in Lemma 7. It follows that

$$P[\text{NSGS}_p \text{ selects } K : \mu_K - \mu_k \geq 2\delta] \geq P[B_1 \cap B_2] \geq 1 - \alpha_1 - \alpha_2$$

where the first inequality follows from the fact that any outcome that satisfies both B_1 and B_2 results in a selection of system K that is at most 2δ worse than the best system k. □

CHAPTER 5

THE PARALLEL GOOD SELECTION PROCEDURE

5.1 Introduction

In previous chapters, we have seen that both NHH and NSGS$_p$ procedures have strengths and limitations. As suggested by the complexity analyses in §3.3 and §4.3, NHH utilizes a highly-efficient iterative screening method that efficiently eliminates inferior systems with a sample of size $O(\mu_k - \mu_i)^{-1}$, whereas NSGS$_p$ tends to over-sample those systems. However, the statistical validity of NHH relies on the PCS assumption $\mu_k - \mu_{k-1} \geq \delta$. In practice, $\mu_k - \mu_{k-1}$ is usually unknown and a large δ may be preferred under a tight simulation budget, making the PCS guarantee of NHH difficult to validate.

In this chapter, we take advantage of the "α-splitting" idea of §4.4.1 which allows the combination of two difference selection and screening methodologies to deliver an overall probabilistic guarantee, and propose a hybrid R&S procedure which combines NHH-like iterative screening with a Rinott [50] indifference-zone selection. The new procedure starts with sequential screening as the first phase, simulating and screening various systems until some switching criterion is met or only one system remains. If multiple systems survive after the first phase, then the procedure applies the [50] indifference-zone method on the surviving systems in a second phase similar to NSGS$_p$, taking $N_i^{(2)}$ additional samples from each system i as per (4.1). Finally, the procedure combines the first- and second-phase samples and selects the system with the highest sample mean as the best.

Similar to NHH, the first phase of our procedure is designed to efficiently eliminate clearly inferior systems. In addition, we choose the sequential screening parameters and specify a switching criterion for this phase such that the best system survives with high probability, *regardless of the actual distribution of system means*. Effectively, the first stage acts like a subset-selection procedure [27] where a subset of systems that are close to the best would survive and the rest are eliminated efficiently with $O(\mu_k - \mu_i)^{-1}$ sample size. Then, among the systems that survive the first stage, the Rinott procedure in the second stage will guarantee to select a "good" one, again with high probability. Since the probabilities of making an error in both stages are carefully bounded, the overall hybrid procedure will guarantee good selection following Lemma 7. For this reason, we name it the Good-Selection Procedure, or GSP in short.

5.2 The Setup

GSP consists of four broad stages. Its first two stages, Stages 0 and 1, are identical to those of NHH, where simulation completion times and variances are estimated for every system. Stage 2 of GSP is also very similar to NHH, with the workers iteratively switching between simulation and screening and eliminating system during the process. However, in GSP a different elimination rule is used and instead of screening until a single system remains, Stage 2 may also terminate when a pre-specified limit on sample size is reached. The screening rule and the limit on sample size are jointly chosen such that inferior systems can be eliminated efficiently, while the best system k survives this stage 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 [50] 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.

We employ the same sampling rule for GSP where $n_i(r)$, the total number of replications for system i up to the rth batch is given by (3.1).

The parameters for GSP 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%. 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 ensure approximately normally distributed batch means. 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.

We recommend choosing $\bar{r}$ such that a fair amount of simulation budget (no more than 20% of the sum of Rinott sample sizes) will be spent in the iterative screening stages. Note that a small $\bar{r}$ implies insufficient screening whereas a large $\bar{r}$ may be too conservative.

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

Finally, we let η be the solution to

$$E\left[2\bar{\Phi}\left(\eta\sqrt{R}\right)\right] = 1 - (1-\alpha_1)^{\frac{1}{k-1}}, \tag{5.1}$$

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. Let the distribution function and density of such a χ^2 random variable be denoted $F_{\chi^2_{n_1-1}}(x)$ and $f_{\chi^2_{n_1-1}}(x)$, respectively. Hence 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

$$a_{ij}(\bar{r}) = \eta\sqrt{(n_1-1)\tau_{ij}(\bar{r})}.$$

5.3 Good Selection Procedure under Unknown Variances

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 (3.1) using S_i^2 obtained in Stage 1, and divides G_1 to approximately load-balanced screening groups G_2^w for $w = 1, \ldots, c$. Let $s_i \leftarrow 0, i \in G_1$ be the count of the number of batches simulated in Stage 2 for system i.

5. For $w = 1, 2, \ldots, c$ in parallel on workers, let $r_w \leftarrow 0$ be the count of the number of batches screened on worker w (which is common to all systems in the screening), and 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 $r_w \geq \bar{r}$ for all w (each worker has screened up to $\bar{r}$, the maximum number of batches allowed), go to 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 if the $(r_w + 1)$th iteration 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 (not necessarily G_2^w) from the master and simulate it 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 $r_w \leftarrow r_w + 1$. A 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 $k' := |G_2| = 1$, 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. [2, 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.

The first two stages of GSP are almost identical to those of NHH with the exception that Stage 1 screening uses a different screening method (see Step 3(c) of GSP). Therefore we refer to Figure 3.1 for a full description of these stages.

begin Stage 2: Iterative screening
  $G_1 \leftarrow$ systems that survived Stage 1;
  $\{G_2^w : w = 1, 2, \ldots, c\} \leftarrow$ `Partition`$(G_1, 2)$;
  $\mathcal{S} \leftarrow G_1$;
  foreach *worker* $w = 1, 2, \ldots, c$ **do**
    Send G_1, G_2^w to Worker w;
    foreach *system* $i \in G_1$ **do**
      Send S_i^2 from Stage 1 to Worker w;
    end
    foreach *system* $i \in G_2^w$ **do**
      Send $\texttt{Stat}_{i,0}$ to worker w;
    end
  end
  $b_i \leftarrow$ `BatchSize`(i, β), $q_i \leftarrow 1$ for all $i \in G_1$;
  $r_w^{\text{sent}} \leftarrow 0$, $r_w^{\text{received}} \leftarrow 0$, $r_w^{\text{screened}} \leftarrow 0$,
  $\texttt{flag}_w \leftarrow 0$ for all $w = 1, 2, \ldots, c$;
  while $|\mathcal{S}| > 1$ *and* $r_w^{screened} < \bar{r}$ *for some* w **do**
    Wait for the next worker w to call
      `Communicate()`;
    if $\texttt{flag}_w = 1$ **then**
      /* Send screening task to worker w */
      $\{i, q_i, \texttt{Stat}_{i,q_i}\} \leftarrow$`RecvOutput`$(w)$;
    else if $\texttt{flag}_w = 2$ **then**
      /* Send simulation task to worker w */
      $\{\mathcal{S}, G_2^w, r_w^{\text{screened}}\} \leftarrow$`RecvScreen`$(w)$;
      $\{i_w^*, r_w^{\text{received}}, \{\texttt{Stat}_{i_w^*,r} : r \leq r_w^{\text{received}}\}\} \leftarrow$`RecvBest`$(w)$;
    end
    if $|\mathcal{S}| > 1$ **then**
      $r^{\text{current}} \leftarrow$`CountBatch`$(w)$;
      if $r^{current} > r_w^{sent}$ **then**
        $\texttt{flag}_w \leftarrow 2$;
        `SendAction`$(w, \texttt{flag}_w)$;
        `SendStats`$(w, r_w^{\text{sent}}, r^{\text{current}})$;
        `SendBestStats`(w);
        $r_w^{\text{sent}} \leftarrow r^{\text{current}}$;
      else
        $\texttt{flag}_w \leftarrow 1$;
        `SendAction`$(w, \texttt{flag}_w)$;
        Select next $i \in \mathcal{S}$ such that $q_i = q_{\text{Global}}$;
        `SendSim`(w, i, q_i, b_i);
        $q_i \leftarrow q_i + 1$;
        if $q_i > q_{Global}$ *for all* $i \in \mathcal{S}$ **then**
          $q_{\text{Global}} \leftarrow q_{\text{Global}} + 1$;
        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* $i \in G_1$ **do**
    Receive S_i^2 collected in Stage 1;
  end
  foreach *system* $i \in G_2^w$ **do**
    Receive $\texttt{Stat}_{i,0}$ from the master;
  end
  $r_w \leftarrow 0$;
  `Communicate`();
  while *No termination instruction received* **do**
    $\texttt{flag}_w \leftarrow$`RecvAction`();
    if $\texttt{flag}_w = 2$ **then**
      $\left\{r^{\text{new}}, \{\texttt{Stat}_{i,r} : i \in G_2^w, r_w + 1 \leq r \leq r^{\text{new}}\}\right\}$
        $\leftarrow$`RecvStats`();
      $\{\mathcal{W}, \{r_{w'}^{received} : w' \in \mathcal{W}\}$,
        $\{\texttt{Stat}_{i_{w'}^*,r} : w' \in \mathcal{W}, r \leq r_{w'}^{received}\}\}$
        $\leftarrow$`RecvBestStats`();
      $G_2^w \leftarrow$`Screen`$(G_2^w, r_w + 1, r^{\text{new}}, true)$;
      $r_w \leftarrow r^{\text{new}}$;
      `Communicate`();
      `SendScreen`(G_2^w, r_w); `SendBest`(r);
    else
      $\{i, q_i, b_i\} \leftarrow$`RecvSim`();
      `Simulate`$(i, b_i, \texttt{Stat}_{i,q_i})$;
      `Communicate`();
      `SendOutput`$(i, q_i, \texttt{Stat}_{i,q_i})$
    end
  end
end

Figure 5.1: Stage 2, GSP: Master (left) and workers (right) routines

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**
    - $N_i \leftarrow \max\{n_i(\bar{r}), \left\lceil (hS_i/\delta)^2 \right\rceil\}$;
    - $N_i^{\text{sent}} \leftarrow 0$; $N_i^{\text{received}} \leftarrow 0$;
  - **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** $\texttt{flag}_w = 1$ **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 5.2: Stage 3, GSP: Master (left) and workers (right) routines

Stage 2 of GSP differs slightly from NHH in that the master does not simulate or screen beyond the $\bar{r}$th batch, as illustrated in Figure 5.1. Finally, Figure 5.2 describes Stage 3 of GSP.

5.3.1 Choice of parameter η

One way to compute η, the solution to (5.1), is by integrating the LHS using Gauss-Laguerre quadrature and using bisection to find the root of (5.1). Alternatively, we may employ a bounding technique to approximate η as follows. The LHS of (5.1) is

$$
\begin{aligned}
E\left[2\bar{\Phi}\left(\eta\sqrt{R}\right)\right] &= \int_{y=0}^{\infty} 2\bar{\Phi}(\eta\sqrt{y})2[1-F_{\chi^2_{n_1-1}}(y)]f_{\chi^2_{n_1-1}}(y)\mathrm{d}y \\
&\leq \int_{y=0}^{\infty} 4\bar{\Phi}(\eta\sqrt{y})f_{\chi^2_{n_1-1}}(y)\mathrm{d}y \qquad (5.2)\\
&\leq \int_{y=0}^{\infty} 4\frac{e^{-\eta^2 y/2}}{\eta\sqrt{2\pi y}}\frac{y^{\frac{n_1-1}{2}-1}e^{-y/2}}{2^{\frac{n_1-1}{2}}\Gamma(\frac{n_1-1}{2})}\mathrm{d}y \qquad (5.3)\\
&= \frac{4\Gamma(\frac{n_1-2}{2})(\frac{2}{\eta^2+1})^{\frac{n_1-2}{2}}}{\sqrt{2\pi}\eta 2^{\frac{n_1-1}{2}}\Gamma(\frac{n_1-1}{2})}\int_0^{\infty}\frac{(\frac{\eta^2+1}{2})^{\frac{n_1-2}{2}}y^{\frac{n_1-1}{2}-1-\frac{1}{2}}e^{-\frac{\eta^2+1}{2}y}}{\Gamma(\frac{n_1-2}{2})}\mathrm{d}y \qquad (5.4)\\
&= \frac{2\Gamma(\frac{n_1-2}{2})}{\sqrt{\pi}\Gamma(\frac{n_1-1}{2})\eta(\eta^2+1)^{\frac{n_1-2}{2}}}, \qquad (5.5)
\end{aligned}
$$

where (5.2) holds because distribution functions are non-negative and is inspired by a similar argument in [23], (5.3) 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 (5.4) 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 (5.5) is an upper-bound on the left-hand side of (5.1). Setting (5.5) 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 (5.5) is strictly decreasing in η, η' can be easily determined using bisection.

The parameter η determines the value of $a_{ij}(\bar{r})$, and hence how quickly an inferior system is eliminated in screening Steps 3(c) and 5(d). The value of η therefore directly impacts the effectiveness of the iterative screening. Hence, it is desirable that η does not grow dramatically as the problem gets bigger.

Observe that (5.5) can be further bounded by

$$\frac{2\Gamma(\frac{n_1-2}{2})}{\sqrt{\pi}\Gamma(\frac{n_1-1}{2})\eta^{n_1-1}} := C\eta^{1-n_1}. \tag{5.6}$$

Setting (5.6) to $1-(1-\alpha_1)^{\frac{1}{k-1}}$ implies that the right-hand side of (5.6) must be small. After some further manipulations we have

$$\log(1-\alpha_1) = (k-1)\log\left(1-C\eta^{1-n_1}\right) \approx (k-1)(-C\eta^{1-n_1}) \tag{5.7}$$

where the approximation holds because $\log(1-\epsilon) \approx -\epsilon$ for small $\epsilon > 0$. It follows from (5.7) 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.

5.3.2 Choice of parameter $\bar{r}$

GSP employs a free parameter to govern the maximum number of batches to be simulated in Stage 1 before moving on to Stage 2. In this section, we discuss the effect of $\bar{r}$ on the efficiency of the procedure and outline some ideas to determine the optimal choice of $\bar{r}$. These ideas may form the basis of a method of automatically selecting $\bar{r}$, but they are not yet fully implemented.

We begin by considering a simplified problem with $k = 2$ systems with expected output μ_1 and μ_2 such that $\mu_{21} := \mu_2 - \mu_1 > 0$. Assume known and equal system variances $\sigma_1^2 = \sigma_2^2 = 1/2$, so Stage 1 (variance estimation) can be skipped. Furthermore, let $\beta = 1$ so $n_i(r) = r$ for $i = 1, 2$. In this simplified setting, GSP

screens using the test statistic $Z_{21}(r) = r[\bar{X}_2(r) - \bar{X}_1(r)]$ and eliminates a system if $|Z_{21}(r)| > a_{21}(\bar{r}) = O(\sqrt{\bar{r}})$ for some $r = 1, 2, \ldots, \bar{r}$.

By Lemma 1, the random sequence $\{Z_{21}(r) : r = 1, 2, \ldots\}$ has the same distribution as $\left\{B_{\mu_{21}}(r) : r = 1, 2, \ldots\right\}$, a Brownian motion with drift μ_{21} observed at (discrete time steps) r. Therefore, we approximate $\arg\min\{r = 1, 2, \ldots : |Z_{21}(r)| > a_{21}(\bar{r})\}$, the first time $Z_{21}(\cdot)$ exists two-sided boundaries $\pm a_{21}(\bar{r})$, by $T_{21} := \arg\min\{t > 0 : B_{\mu_{21}}(t) > a_{21}(\bar{r})\}$, the first time $B_{\mu_{21}}(\cdot)$ exits the one-sided boundary $a_{21}(\bar{r})$, for $a_{21}(\bar{r})$ is chosen such that the probability of $B_{\mu_{21}}(\cdot)$ exiting $\pm a_{21}(\bar{r})$ from below is small. Observe that the latter is the time for a Brownian motion with drift μ_{21} to hit the positive barrier a_{21}, and is known to follow an Inverse Gaussian distribution with density

$$f_{T_{21}}(t) = \frac{a_{21}}{\sqrt{2\pi t^3}} \exp\left[\frac{-(\mu_{21}t - a_{21})^2}{2t}\right]$$

and expectation $E[T_{21}] = a_{21}/\mu_{21}$ (see, e.g., [10]). It then follows that the expected sample size for GSP is approximately

$$E[T_{21}\mathbb{1}_{T_{21}\leq\bar{r}}] + \max\{\bar{r}, (h/\delta)^2\}P[T_{21} > \bar{r}]. \tag{5.8}$$

Our objective is therefore to minimize (5.8) by changing $\bar{r}$. Intuitively, we expect (5.8) to be unimodal in $\bar{r}$ because a small $\bar{r}$ means insufficient screening and a large sample size of $(h/\delta)^2$, whereas a large $\bar{r}$ may be wasteful.

In practice, the number of systems $k \gg 2$, but for any system i, we consider minimizing (5.8) for the Brownian motion $B_{\mu_{ki}}$ between the best system k and system i, as it has the highest drift among all $k - 1$ Brownian motions between i and other systems. Solving the minimization problem for system i then yields an optimal Stage 1 batch size $\bar{r}_i^*$ which can be incorporated in a revised screening method as follows. If system i survives by the time $\bar{r}_i^*$ replications have been

simulated in Stage 2, we exclude the system from further simulation and screening in Stage 2, wait until all other systems either get eliminated or reach their optimal level of $\bar{r}$, and then run Stage 3 on all surviving systems.

The true drift of the Brownian motion, μ_{ki}, is unknown to the procedure and needs to be estimated in order to establish the density function $f_{T_{ki}}$ used to optimize $\bar{r}$ for each individual system. We may employ a preliminary round of simulation, from which any statistic(s) including mean, variance, and simulation completion time estimates can be used to calculate the optimal level of $\bar{r}$. Note, however, that in order to maintain the normality of the screening statistic, the sample from the preliminary round has to be excluded from the $Z_{ki}(\cdot)$ sequence.

5.4 Guaranteeing Good Selection

In this section we state and prove the PGS guarantee of the GSP procedure.

Theorem 3. *Under Assumption 4, GSP selects a system K that satisfies $\mu_k - \mu_K \leq \delta$ with probability at least $1 - \alpha$.*

To prove Theorem 3 we introduce some additional Lemmas, as follows.

Lemma 10. *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 [23] 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 11. *By the reflection principle of Brownian motion,* $P[\min_{0\le t'\le t} B_0(t') < -a] = 2P[B_0(t) < -a] = 2\bar{\Phi}(a/\sqrt{t})$ *for all* $a, t > 0$.

Remark. Lemma 11 plays a similar role as Lemma 2, in the sense that they both bound the probability of making an error in a screening procedure, when a Brownian motion with drift exits a symmetric continuation region from the opposite side. However, Lemma 2 looks specifically at the probability of error when the Brownian motion leaves the continuation region *for the first time*, whereas Lemma 11 bounds the probability of *ever* leaving from the opposite side *within a specified time frame*. The latter is an upper-bound on the probability of making an error *at some (random) time steps* within the same time frame, which implies screening does not necessarily have to be performed at every batch. Therefore, catch-up screening is not required for GSP, reducing the need for the procedure to retain past batch statistics on either the master or the workers.

Proof of Theorem 3. 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

$$
\begin{aligned}
&\Pr[KO_{ik} \text{ in Stages 1 or 2}] \\
=&E[\Pr[KO_{ik} \text{ in Stages 1 or 2}|S_k^2, S_i^2]] \\
\le&E[\Pr[Y_{ki}(\tau_{ki}(r)) < -a_{ij}(\bar{r}) \text{ for some } r \le \bar{r}|S_k^2, S_i^2]] \\
&\quad \text{since system } i \text{ could be eliminated by some other system} \\
&\quad \text{before eliminating system } k \\
=&E[\Pr[Y_{ki}(\tau_{ki}(r)) < -a_{ij}(\bar{r}) \text{ and } \tau_{ki}(r) \le \tau_{ki}(\bar{r}) \text{ for some } r|S_k^2, S_i^2]] \\
=&E[\Pr[Z_{ki}(t_{ki}(r)) < -\frac{t_{ki}(r)}{\tau_{ki}(r)} a_{ij}(\bar{r}) \text{ and } t_{ki}(r) \le \frac{t_{ki}(r)}{\tau_{ki}(r)} \tau_{ki}(\bar{r}) \text{ for some } r|S_k^2, S_i^2]] \\
=&E[\Pr[B_{\mu_k-\mu_i}(t_{ki}(r)) < -\frac{t_{ki}(r)}{\tau_{ki}(r)} a_{ij}(\bar{r}) \text{ and } t_{ki}(r) \le \frac{t_{ki}(r)}{\tau_{ki}(r)} \tau_{ki}(\bar{r}) \text{ for some } r|S_k^2, S_i^2]] \\
&\quad \text{by Lemma 1}
\end{aligned}
$$

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

by Lemmas 10 and 4

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

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

$$= E\left[2\bar{\Phi}\left(\frac{\tilde{a}_{ij}(\bar{r})}{\sqrt{\tilde{t}_{ij}(\bar{r})}}\right)\right] \text{ by Lemma 11}$$

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

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

$= 1 - (1-\alpha_1)^{\frac{1}{k-1}}$ by (5.1),

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

$$\begin{aligned}
&\Pr[\text{system } k \text{ survives Stages 1 and 2}] \\
&\geq \Pr\left\{\bigcap_{i=1}^{k-1} \overline{KO}_{ik}\right\} \\
&\quad \text{as system } k \text{ may not get screened against all systems} \\
&= E\left[\Pr\left\{\bigcap_{i=1}^{k-1} \overline{KO}_{ik} \middle| X_{k1}, X_{k2}, \ldots\right\}\right] \\
&= E\left[\prod_{i=1}^{k-1} \Pr\left\{\overline{KO}_{ik} \middle| X_{k1}, X_{k2}, \ldots\right\}\right] \\
&\geq \prod_{i=1}^{k-1} E\left[\Pr\left\{\overline{KO}_{ik} \middle| X_{k1}, X_{k2}, \ldots\right\}\right] \text{ by Lemma 6} \\
&= \prod_{i=1}^{k-1} \Pr\left[\overline{KO}_{ik}\right] \geq \prod_{i=1}^{k-1}\left[1 - \left(1-(1-\alpha_1)^{\frac{1}{k-1}}\right)\right] = 1-\alpha_1.
\end{aligned}$$

Next, recall that we have shown in §4.4.2 that the Stage 3 Rinott step sat-

isfies P[select system K in Stage 3 : $\mu_K - \mu_k \geq \delta | k$ survives Stages 1 and 2] $\geq 1 - \alpha_2$. Now invoke Lemma 7 as follows. Let $\mathcal{P}_1$ be Stages 1 and 2, $\mathcal{P}_2$ be Stage 3 of GSP, and define $B_1 = \{k \in I\}$, $\mathcal{J} = \{I \subseteq \mathcal{S} : k \in I\}$ and $B_2(I) = \{\mathcal{P}_2$ selects system K in Stage 3 : $\mu_K - \mu_k \geq \delta |$ set I survives Stage 1 and 2$\}$, we have

$$
\begin{aligned}
& P[\text{GSP selects } K : \mu_K - \mu_k \geq \delta] \\
\geq & P[B_1 \cap B_2] \\
= & P[k \in I \text{ and } \mathcal{P}_2 \text{ selects system } K \text{ in Stage 3} : \mu_K - \mu_k \geq \delta \text{ for all } I \in \mathcal{J}] \\
\geq & 1 - \alpha_1 - \alpha_2
\end{aligned}
$$

where the first inequality follows from the fact that any outcome that satisfies both B_1 and B_2 results in a good selection. □

CHAPTER 6

IMPLEMENTATIONS AND NUMERICAL COMPARISONS OF PARALLEL R&S PROCEDURES

In this chapter, we discuss our parallel computing environment and test problem, a number of parallel implementations of the procedures proposed in previous chapters, and the results of our numerical experiments. The primary purpose of this study is to demonstrate the capability of the various parallel procedures discussed in preceding chapters, implemented using the right software tools, to harness large amounts of parallel computing resources and solve large-scale R&S problems. To validate the procedures' scalability we select test problems that can be parameterized to have up to 10^6 systems in the solution space, and show that the proposed procedures can solve such large problem instances with low parallel overhead. This result significantly expands the boundary of the size of solvable R&S problems, as traditional sequential methods typically handle no more than 10^3 systems.

6.1 Test Problems

Two problems from `SimOpt.org` [22] are selected to test the R&S procedures. In the first problem entitled throughput-maximization, we solve

$$\max_x \; E[g(x;\xi)] \tag{6.1}$$

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

$$b_2 + b_3 = B$$

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

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.

We choose this test problem primarily because it can be easily parameterized to obtain different problem instances with varying difficulty. For each choice of 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 6.1. In addition to that, 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.

By default, 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 may not be the most 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 from the $g(x;\xi)$ distribution in parallel. Due to the fixed number of jobs, the wall-clock

Table 6.1: Summary of three instances of the throughput maximization problem.

(R, B)	Number of systems k	Highest mean μ_k	pth percentile of means			No. of systems in $[\mu_k - \delta, \mu_k]$		
			$p = 75$	$p = 50$	$p = 25$	$\delta = 0.01$	$\delta = 0.1$	$\delta = 1$
$(20, 20)$	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

Table 6.2: Summary of two instances of the container freight minimization problem.

M	Number of systems k	Lowest mean μ_k	pth percentile of means			No. of systems in $[\mu_k - \delta, \mu_k]$		
			$p = 75$	$p = 50$	$p = 25$	$\delta = 0.01$	$\delta = 0.1$	$\delta = 1$
115	680	65.76	75.42	101.25	266.28	1	2	11
155	29,260	60.39	61.25	63.88	76.21	71	1278	8045

Table 6.3: Parameter values of the container freight problem.

i	λ_i (/minute)	$1/\mu_i$ (minutes)
1	52.8/60	67
2	11.7/60	46
3	13.0/60	92
4	22.5/60	34

time for each simulation replication exhibits low variability.

The second problem is entitled container freight optimization. In this test problem, we solve

$$\begin{aligned} &\max_x \; E[g(x;\xi)] && && (6.2)\\ \text{s.t.} \; &\sum_{i=1}^{4} x_i = M && \\ &\lambda_i/\mu_i x_i < 1, && i = 1,2,3,4 \\ &x_i \in \{1,2,\ldots\}, && i = 1,2,3,4 \end{aligned}$$

where the function $g(x;\xi)$ represents the steady-state waiting time of a job entering one of four $M/M/x_i$ queues each according to arrival rate λ_i, service rate μ_i, and capacity x_i. The parameters λ_i and μ_i for $i = 1,2,3,4$ are given in Table 6.3. The overall objective is to minimize steady-state waiting time by finding an optimal (integer-valued) allocation of servers.

Like with the first test problem, it is possible to iterate over all feasible allocations. For each choice of parameter $M \in \mathbb{Z}^+$, the number of feasible allocations is finite and can be easily computed. It is also possible to compute average waiting

time theoretically by

$$\frac{\sum_{i=1}^{4} \lambda_i w(\lambda_i, \mu_i, x_i)}{\sum_{i=1}^{4} \lambda_i}$$

where $w(\lambda, \mu, x)$ is the average steady-state waiting time in an $M/M/x_i$ queue with arrival rate λ and service rate μ, for which an analytical formula is available. We consider two problem instances as presented in Table 6.2.

For the container freight problem, we warm up the system for 100 simulated hours and take the average waiting time for the next 500 hours as one replication.

6.2 Parallel Computing Environment

Our numerical experiments are conducted on Extreme Science and Engineering Discovery Environment (XSEDE)'s Stampede and Wrangler clusters. The Stampede cluster contains over 6,400 computer nodes, each equipped with two 8-core Intel Xeon E5 processors and 32 GB of memory and runs a Linux Centos 6.3 operating system [52].

The Wrangler cluster contains 96 data analytics server nodes each with two 12-core Intel Haswell E5-2680-v3 CPUs and 128 GB of memory [53]. The cluster is designed and optimized primarily for high-speed, high-volume data analytics, and has a wide range of software systems installed for this purpose.

Both XSEDE clusters are typical examples of "high-performance" platforms offering massive computing, data storage and network capacities for large-scale, computationally intensive jobs for which parallelism can be exploited to achieve significant speedup. These resources are shared by a large group of

users, who may purchase allocations of core hours as well as storage quota on those systems. Parallel programs are submitted through the Simple Linux Utility for Resource Management (SLURM) batch environment, which schedules jobs according to the user-specified number of cores and maximum period of execution required for each job. On XSEDE, users are only charged with the actual number of core hours used, so the cost of solving a R&S problem can be quantified by the core hours (wallclock time $\times$ number of cores used) consumed by our parallel R&S procedures.

We have never seen a core failure on XSEDE clusters as a result of the following factors:

- The high-performance processors on Stampede are highly reliable.
- The system does not re-assign cores allocated to us to higher priority tasks.
- Our R&S procedures and test problems do not have a heavy memory footprint, nor do they require excessive usage of disk storage or communication network beyond the hardware limits.

Despite using XSEDE's high-performance clusters as our main test venue, our procedures run on other parallel computing architectures with varying degree of success. On a small-scale parallel computer such as the 2013 Macbook Pro laptop equipped with a dual-core, four-thread Intel Core i5 processor, for example, our implementations of GSP can utilize the multi-core processor and solve the smallest instance of the throughput-maximization problem in less than a minute, but as the laptop clearly lacks the ability to scale up to employ more cores for large problems, we do not exploit it further.

Another important and commercially available type of parallel platform is

the cloud, such as Amazon's EC2 service. Cloud providers host a diverse range of computing resources and allow users to launch virtual machine instances with customizable qualities and capacities at very affordable prices.

We have successfully solved test problem instances on EC2, but do not conduct an extensive numerical study there mostly because of the difficulty to set up the software environment. Whereas the XSEDE clusters have many scientific, parallel computing software packages installed and optimized for the specific hardware, the cloud, being a flexible and general-purpose platform, does not provide that software by default and it is down to the user to install, manage and optimize the software before being able to fully exploit the requested resources, which can be time-consuming.

6.3 Parallel Programming Engines and their Applications in R&S

A parallel programming engine is a piece of computer software that allows its user to define the roles of parallel cores, manage communications, and ultimately implement parallel algorithms that require various levels of coordination and distribution. In this section, we discuss three parallel programming engines, namely Message-Passing Interface (MPI), Hadoop MapReduce, and Apache Spark, that are suitable for implementing parallel R&S procedures thanks to a number of common features:

- All three engines offer application programming interfaces (APIs) that give the user the opportunity to define high-level parallel operations with-

out working directly with the hardware.

- All three engines support certain popular programming languages (C/C++/Fortran for MPI, Java for Hadoop MapReduce, Java/Scala/Python for Apache Spark), allowing the users to import and utilize existing math, statistics, and random-number generation packages and simulation code.
- All of them can be configured to run on a wide range of parallel platforms from multi-core personal computers to the Amazon EC2 cloud. Specifically on XSEDE clusters, MPI runs on both Stampede and Wrangler whereas MapReduce and Spark are supported on Wrangler.

In the remainder of this section we will illustrate how these engines differ significantly in terms of level of flexibility, robustness to core failures, ease of use, and amount of memory and disk footprint. As MPI is the most general-purpose of all three parallel engines, we implement all three R&S procedures (NHH,NSGS$_p$, and GSP) using MPI. We then extend GSP to Hadoop MapReduce and Apache Spark to showcase the key advantages and limitations of these two engines.

6.3.1 MPI

Message-Passing Interface (MPI) is a popular distributed-memory parallel programming framework with native libraries available in C/C++ and Fortran. MPI is the de-facto standard for parallel programming on many high-performance parallel clusters including Stampede and Wrangler, both of which offers an extensive set of MPI compilers, debuggers, and profilers. Using MPI, programs operate in an environment where cores are independent and commu-

nicate through sending and receiving messages. The method by which parallel cores independently execute instructions and communicate through message-passing can be highly customized to serve different purposes. Conceptually, there is no limit to the type of parallel algorithms implementable in MPI.

The NHH, NSGS_p and GSP procedures have all been implemented using C++ and MPI. We designate one core as the master and let it control other worker cores. We observe that communication is fast on Stampede, taking only 10^{-6} to 10^{-4} seconds each time depending on the size of the message. Therefore, with an appropriate choice of the batch-size parameter β, the master remains idle most of the time so the workers are usually able to communicate with the master without much delay.

Our MPI implementations are designed primarily for high-performance clusters like Stampede and do not actively detect and manage core failures. As simulation output is distributed across parallel cores without backup, the MPI implementation is vulnerable to core failures which may cause loss of data and break the program. 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 cores fail. This motivates us to seek alternative programming tools such as MapReduce that handle core failures automatically.

By default, our MPI implementations use the mvapich2 library but it is also compatible with other MPI versions such as impi (Intel MPI) and OpenMPI. The source code and documentation of our MPI procedures are hosted in the open-access repository [42].

6.3.2 Hadoop MapReduce

MapReduce [11] is a distributed computing model typically used to process large amounts of data. Conceptually, each MapReduce instance 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. MapReduce has become a popular choice for data intensive applications such as PageRank and TeraSort, thanks to the following advantages.

- **Simplicity.** The MapReduce programming model allows its users to solely focus on designing meaningful Mappers and Reducers that define the parallel program, without explicitly handling the complex details of the message-passing and the distribution of workload to cores, a task which is completely delegated to the MapReduce package.
- **Portability.** Apache Hadoop is a highly popular and portable MapReduce system that can be easily deployed with minimal changes on a wide range of parallel computer platforms such as the Amazon EC2 cloud.
- **Resilience to core failures.** On less reliable hardware where there is a non-negligible probability of core failure, the Apache Hadoop system can automatically reload any failed Mapper or Reducer job on another worker to guide the parallel job towards completion.

Despite these advantages, the use of MapReduce for computationally intensive and highly iterative applications, such as simulation and simulation optimization, is less documented. Moreover, most popular MapReduce implementations such as Apache Hadoop have limitations that may potentially reduce

the efficiency of highly iterative algorithms such as parallel R&S procedures.

- **Synchronization.** By design, each Reduce phase cannot start before the previous Map phase finishes and each new MapReduce iteration cannot be initiated unless the previous one shuts down completely. Hence, a R&S procedure using MapReduce has to be made fully synchronous. If load-balancing is difficult, for instance as a result of random simulation completion times, then core hours could be wasted due to frequent synchronization.
- **Absence of Cache.** In Apache Hadoop, workers are not allowed to cache any information between Map and Reduce phases. As a result, the outputs generated by Mappers and Reducers are often written to a distributed file system (which are usually located on hard drives) before they are read in the next iteration. Compared to the MPI implementation where all intermediate variables are stored in memory, the MapReduce version could be slowed down by repeated data I/O, i.e. the read/write operations involving slow forms of storage such as the hard disk. Moreover, the lack of cache requires the simulation program, including any static data and/or random number generators, to be initialized on workers before every MapReduce iteration.
- **Nonidentical Run Times.** By default, Apache Hadoop does not dedicate each worker to a single task. It may simultaneously launch several Mappers and Reducers on a single worker, run multiple MapReduce jobs that share workers on the same cluster, 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

Table 6.4: Major differences between MPI and Hadoop MapReduce implementations of GSP

Task	MPI	Hadoop MapReduce
Master	Explicitly coded	Automated
Message-passing	Explicitly coded	Automated
Synchronization	Once after each stage	More frequent: required in every iteration of Stage 2
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

(estimation of simulation run time) is dropped from our MapReduce implementation.

Although there are specialized MapReduce variants such as "stateful MapReduce" that attempt to address these limitations [12], we do not explore them as they are less available for general parallel platforms, at least at present. However, some of these limitations (such as the lack of caching across multiple MapReduce rounds) are idiosyncratic to specific packages like Hadoop rather than the framework itself. Nevertheless, the a priori expectation is that, for a highly iterative procedure like ours, a highly optimized MPI approach will outperform a Hadoop one; thus our question is not which is fastest, but whether

MapReduce can offer most of the speed of MPI along with its advantages discussed above.

Implementing GSP using Hadoop MapReduce

To use Apache Hadoop MapReduce as a parallel engine for driving R&S procedures, we must fit the desired procedure into the MapReduce paradigm. More specifically, the procedure needs to be defined and implemented as one or multiple MapReduce iterations, each of which contains a mapper and a reducer. Hence 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 which 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 which in turn is used as the input to the next Mapper.

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.

Our MapReduce implementation is based on the native Java interface for MapReduce provided in Apache Hadoop 2.4.0. It is hosted in the open-access repository [41]. Table 6.4 summarizes some of the major differences between the MPI and Hadoop implementations.

We now present the full details of the MapReduce implementation of GSP.

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.

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 `stream`s 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).

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

• **Reduce**: Screen within a group

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

Operation Screen all systems in the $\texttt{Group}$ and find the one i^* with the highest mean.

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

Operation (1) Simply output to $\texttt{Group}(i)$.

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

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

Operation (2) Output to all groups.

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

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

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

  `Group`: $\{i^*, \bar{X}_{i^*}(n_{i^*}), n_{i^*}, b_{i^*}, S^2_{i^*}$, \$Best$\}$ from every other `Group`

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

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

Step 4.

- **Map**: Simulation

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

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

  Output `Group`(i): $\{i, \bar{X}_i(n_1), n_1, b_i, S^2_i$, $\texttt{stream}_i$, \$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 [$i, \bar{X}_i(n_i), n_i, b_i, S^2_i$, $\texttt{stream}_i$, \$Sim]

  Operation Output to Reducer.

  Output i: $\{\bar{X}_i(n_i), n_i, S^2_i$, $\texttt{stream}_i$, \$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.

6.3.3 Apache Spark

Apache Spark [55] is a modern programming engine for parallel computing. It 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 which 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 much 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, §6.3.2). 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 un-

til 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 which 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 [43].

6.4 Numerical Experiments

We now demonstrate the practical performance of the various parallel R&S procedures by using them to solve the test problems.

6.4.1 Comparing Parallel Procedures on MPI

We test MPI implementations of all three procedures on different instances of the throughput maximization and container freight test problems. We measure the performance of these procedures on the XSEDE high-performance clusters in terms of total wall-clock time and simulation replications required to find a solution, and report them in Tables 6.5 and 6.6. Preliminary runs on smaller test problems suggest that the variation in these two measures between multiple runs of the entire selection procedure are limited. Therefore we only present results from a single replication to save core hours.

[46] argue that NHH tends to devote excessive simulation effort to systems

Table 6.5: A comparison of procedure costs using parameters $n_0 = 20$, $n_1 = 50$, $\alpha_1 = \alpha_2 = 2.5\%$, $\beta = 100$, $\bar{r} = 10$ on throughput maximization problem. Platform: XSEDE Stampede. (Results to 2 significant figures)

Configuration	δ	Procedure	Wall-clock time (sec)	Total number of simulation replications ($\times 10^6$)
3,249 systems	0.01	GSP	14	2.3
on 64 cores		NHH	14	2.5
		NSGS_p	120	13
	0.1	GSP	3.4	0.57
		NHH	2.6	0.44
		NSGS_p	3.4	0.48
57,624 systems	0.01	GSP	720	130
on 64 cores		NHH	520	89
		NSGS_p	11,000	1600
	0.1	GSP	60	10
		NHH	71	12
		NSGS_p	150	23
1,016,127 systems	0.1	GSP	260	320
on 1,024 cores		NHH	1,000	430
		NSGS_p	1,400	1900

Table 6.6: A comparison of procedure costs using parameters $n_0 = 20$, $n_1 = 50$, $\alpha_1 = \alpha_2 = 2.5\%$, $\beta = 100$, $\bar{r} = 10$ on container freight problem. Platform: XSEDE Wrangler. (Results to 2 significant figures)

Configuration	δ	Procedure	Wall-clock time (sec)	Total number of simulation replications ($\times 10^6$)
680 systems	0.01	GSP	710	2.7
on 144 cores		NHH	2,700	10
		NSGS_p	$> 14,000$	> 8.2
			(did not finish)	(did not finish)
	0.1	GSP	81	0.20
		NHH	320	1.2
		NSGS_p	610	0.16
29,260 systems	0.1	GSP	540	6.3
on 480 cores		NHH	2,100	25
		NSGS_p	$> 14,000$	> 4.8
			(did not finish)	(did not finish)

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 of the throughput maximization problem with $\delta = 0.1$ as well as all test cases of the container freight problem, when GSP outperforms both NHH and NSGS$_p$.

6.4.2 Comparing MPI and Hadoop Versions of GSP

We now focus on GSP and compare its MPI and Hadoop MapReduce implementations discussed in §6.3. Since Stage 0 is not included in the MapReduce implementation, we also remove it from the MPI version to have a fair comparison. Both procedures are tested on Stampede. While the cluster features highly optimized C++ compilers and MPI implementations, it provides relatively less support for MapReduce. Our MapReduce jobs are deployed using the myhadoop software [32], which sets up an experimental Hadoop environment on Stampede.

Another difference is that we perform less screening in MPI than in Hadoop. In our initial experiments, we observed that the master could become overwhelmed by communication with the workers in the screening stages, and we fixed this problem by screening using only the 20 best systems from other cores, versus the best systems from *all* other cores in Hadoop. While less screening is not a non-negligible effect, it will be apparent in our results that it is dominated by the time spent with simulation.

Before we proceed to the results, we define core utilization, an important

measure of interest, as

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

Utilization measures how efficiently the implementations use the available cores to generate simulation replications. The higher the utilization, the less overhead the procedure spends on communication and screening.

In Table 6.7 we report the number of simulation replications, wall-clock time, and utilization for each of the GSP implementations. The MPI implementation takes substantially less wall-clock time than MapReduce 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 as the batch size β and/or the system-to-core ratio are increased. Similarly, the MPI implementation also yields much higher utilization, spending more than 90% of the total computation time on simulation runs in all problem instances. Compared to the MPI implementation, the MapReduce version utilizes core hours less efficiently but again its utilization significantly improves as we double batch size and increase the system-to-core ratio.

To further understand the low utilization, we give the number of active Mapper and Reducer jobs over an entire MapReduce run in Figure 6.1. The plot reveals a number of reasons for low utilization. First, there are non-negligible gaps between Map and Reduce phases, which are due to an intermediary "Shuffle" step that collects and sorts the output of the Mappers and allocates it to the Reducers. Second, as the amount of data shuffled is likely to vary, the Reducers start and finish at different times. Third, owing to the varying amount of computing required for different systems, some Mappers take longer than others. In all, the strictly synchronized design of Hadoop causes some amount of core idle-

Table 6.7: A comparison of MPI and Hadoop MapReduce implementations of GSP using parameters $\delta = 0.1$, $n_1 = 50$, $\alpha_1 = \alpha_2 = 2.5\%$, $\bar{r} = 1000/\beta$. "Total time" is summed over all cores. Platform: XSEDE Stampede. (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

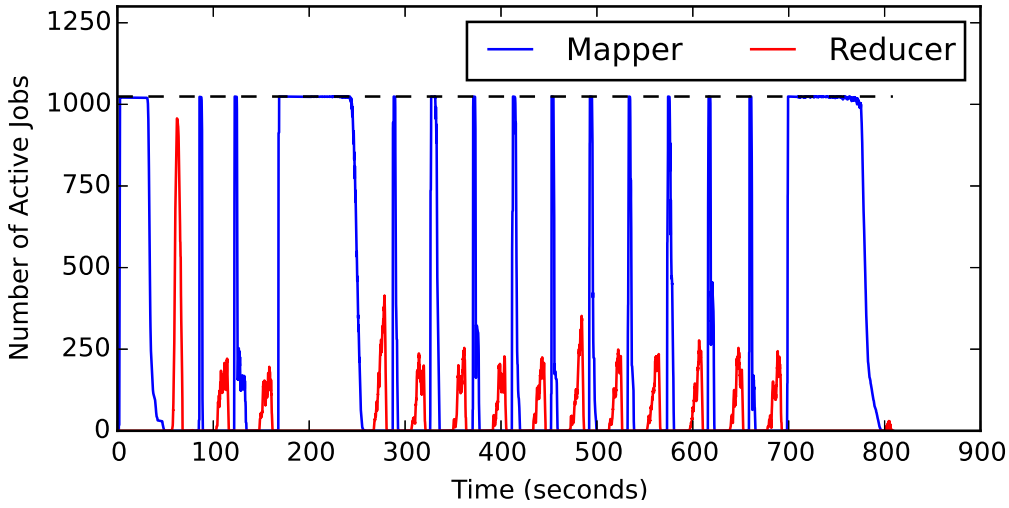

Figure 6.1: A profile of a MapReduce run solving the largest problem instance with $k = 1,016,127$ on 1024 cores, using parameters $\alpha_1 = \alpha_2 = 2.5\%$, $\delta = 0.1$, $\beta = 200$, $\bar{r} = 5$.

ness that is perhaps inherent in the methodology, and therefore unavoidable. Nevertheless, the fact that utilization increases as average batch size β or the system-to-core ratio increases suggests that the Hadoop overhead becomes less pronounced as the amount of computation work per Mapper increases. Therefore we expect utilization to also improve and become increasingly competitive with that of MPI for problems that feature a larger solution space or longer simulation runs.

6.4.3 Robustness to Unequal and Random Run Times

The MapReduce implementation allocates approximately equal numbers of simulation replications to each Mapper and the simulation run times per replication are nearly constant for our test problem, so the computational workload in each MapReduce iteration should be fairly balanced. Indeed, in Figure 6.1

Table 6.8: A comparison of GSP implementations using a random number of warm-up job releases distributed like $\min\{\exp(X), 20,000\}$, where $X \sim N(\mu, \sigma^2)$. We use parameters $\delta = 0.1$, $n_0 = 50$, $\alpha_1 = \alpha_2 = 2.5\%$, $\beta = 200$, $\bar{r} = 5$. (Results to 2 significant figures)

Configuration	μ	σ^2	Version	Wall-clock time (sec)	Utilization %
3,249 systems	7.4	0.5	HADOOP	280	2.3
on 64 cores			MPI	4.2	94
	6.6	2.0	HADOOP	280	2.0
			MPI	4.0	93
57,624 systems	7.4	0.5	HADOOP	400	27
on 64 cores			MPI	74	98
	6.6	2.0	HADOOP	400	26
			MPI	70	98
1,016,127 systems	7.4	0.5	HADOOP	850	25
on 1,024 cores			MPI	150	97
	6.6	2.0	HADOOP	850	22
			MPI	150	97

we observe that Mapper jobs terminate nearly simultaneously, which suggests that load-balancing works well. However, if the simulation run times exhibit enough variation that one Mapper takes much longer than the others, then we would expect synchronization delays that would greatly reduce utilization.

To verify this conjecture, we design additional computational experiments where variability in simulation run times is introduced by warming up each

system for a random number W of job releases (by default, we use a fixed 2,000 job releases in the warm-up stage). 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. We also truncate the log-normal distributions from above at 20,000 job releases to avoid exceeding a built-in timeout limit in Hadoop. Parameters of the truncated log-normal distribution and the results of the experiment are given in Table 6.8.

We observe very similar wall-clock time and utilization in all instances compared to the base cases in Table 6.7 where we used fixed warm-up periods. Both implementations seem quite robust against the additional randomness in simulation times, despite our intuition that the MapReduce 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 MapReduce 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.

6.4.4 Comparing MPI and Spark Versions of GSP

Next, we compare the empirical performances of the MPI and Spark implementations of GSP. This test is conducted on XSEDE Wrangler, because the cluster

supports both MPI and Spark engines on the same hardware architecture. We also run the Hadoop MapReduce implementation on Wrangler so that all three implementations are directly comparable.

One noticeable difference between the two engines on Wrangler is that Spark is run on a "cluster" mode under which a single node (containing 48 cores) is designated to be the master, whereas our MPI program always uses a single core as the master. As a result, the MPI implementation running on 3 nodes (144 cores) is able to use $144 - 1 = 143$ worker cores, but the Spark version under the same allocation only has 2 nodes $\times$ 48 cores/node = 96 worker cores available. To account for the discrepancy, we define adjusted utilization as

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

Recall that Spark is designed to deliver performance enhancement over MapReduce by reducing synchronization and disk I/O. For computationally-intensive applications that do not require a huge amount of data transfer such as R&S, we expect the new features of Spark to provide significant speedup. Indeed, although Table 6.9 suggests that MPI is still the more efficient of the two implementations as measured by a shorter wall-clock time and higher utilization in all test cases, by comparing Table 6.9 with Table 6.7 we see that the performance gap between MPI and Spark is significantly smaller compared to the gap between MPI and MapReduce. For the larger test cases, the Spark implementation can utilize more than 40% of available workers, and is nearly half as efficient as the MPI version in terms of wall-clock time. Based on this evidence, we conclude that our Spark implementation is an efficient and robust alternative to the MPI version that offers some extra portability and fault-tolerance without a huge loss in performance.

Table 6.9: A comparison of MPI, Hadoop MapReduce and Spark implementations of GSP using parameters $\delta = 0.1$, $n_1 = 50$, $\alpha_1 = \alpha_2 = 2.5\%$, $\bar{r} = 1000/\beta$. "Total time" is summed over all cores. Platform: XSEDE Wrangler. (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 %	Adjusted Utilization %
3,249 systems	100	Spark	0.47	31	0.27	6.0	9.0
on 144 cores		MPI	0.58	2.3	0.25	73	73
		Hadoop	0.46	870	0.25	0.32	0.48
	200	Spark	0.64	32	0.36	7.8	12
		MPI	0.71	2.6	0.30	82	82
		Hadoop	0.61	560	0.31	0.63	0.94
57,624 systems	100	Spark	9.1	120	4.7	28	41
on 144 cores		MPI	9.9	31	4.2	94	94
		Hadoop	8.9	1600	4.7	3.3	4.9
	200	Spark	12	160	6.5	29	43
		MPI	13	42	5.7	95	95
		Hadoop	12	1200	6.4	5.7	8.5
1,016,127	100	Spark	240	660	120	39	43
systems		MPI	280	290	120	86	87
on 480 cores		Hadoop	280	3800	150	9.8	11
	200	Spark	300	810	160	40	45
		MPI	350	330	150	95	95
		Hadoop	350	3200	190	14	16

6.4.5 Discussions on Parallel Overhead

Ideally, a parallel procedure that provides a speedup through employing multiple processors should consume the same amount of total computing resources as its sequential equivalent. In practice, parallel speedup comes at the expense of some additional computing overhead cost, which is incurred as a consequence of the algorithmic design, the software implementation, the architectural specifics of the parallel computing hardware, and often the interaction of these different layers. In this section, we discuss the various factors that cause parallel overhead in the ranking and selection setting.

Overhead Caused by Parallel Algorithm Design

To adapt to the parallel environment where multiple processors can run simulation replications and some decision making (e.g. screening) independently in parallel, R&S procedures have to make some algorithmic changes that inevitably lead to some overhead, regardless of the actual software/hardware environment.

- **Synchronization.** It is difficult and often inefficient to assign the same amount of work to workers and different strategies can be taken to address this difficulty. Our parallel procedures are designed such that workers are allowed to communicate with the master independently without having to wait for other workers (Section 2.4) and the idea is fully implemented in the MPI version. Using this strategy, a free worker gets its next task from the master almost immediately (unless the master is communicating with other workers), so core utilization is high. One slight inefficiency, however,

is that this strategy may end up running more simulation replications than necessary, for it is possible for a master to initiate the $(r+1)$st batch for a system i on a free worker while i is being eliminated in the rth batch on another worker and the decision has not been returned to the master. As a result, we can observe from Table 6.9 that the asynchronous MPI version generates a larger number of replications than the synchronized Hadoop or Spark algorithms, but this loss is often outweighed by the improved core utilization thanks to the asynchronism.

Iterative screening can also be implemented using a number of fully synchronized simulation/screening steps, as evidenced in our MapReduce and Spark implementations (Section 6.3.2). To balance the load in a synchronized procedure, we need to balance the number of systems assigned to each worker, which is difficult especially in later iterations when the number of surviving systems can be much smaller than the number of available workers.

- **Distributed screening**. As discussed in Section 2.2.3, we do not perform the full $O(k^2)$ pairs of screening and instead assign roughly k/c systems to each worker which screens within the small group. Screening on workers speeds up the otherwise expensive operation, but inevitably weakens the screening and exposes some systems to additional simulation batches. Nevertheless, the negative effect is likely a minor one as we also share some good systems across all workers.

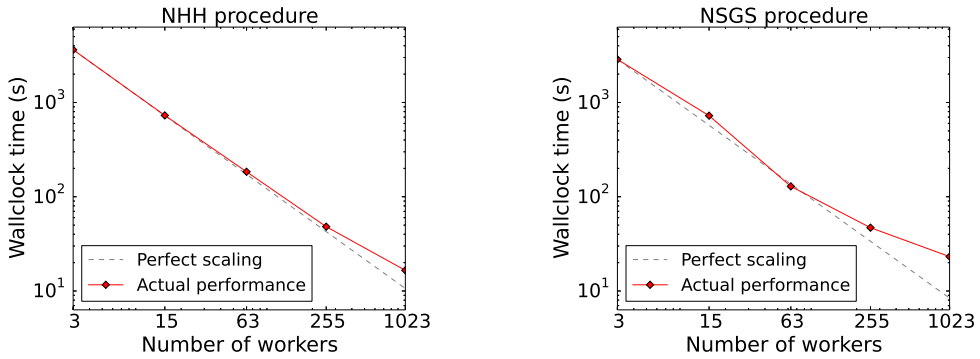

Figure 6.2: Scaling result of the MPI implementation on 57,624 systems with $\delta = 0.1$.

Overhead Associated with Parallel Software

Parallel engines may place specific restrictions on how a procedure may be implemented. They may also differ in the way in which intermediate results such as batch statistics and random number seeds are stored. In this regard, MPI is the best option as it offers a high degree of flexibility, allowing the programmer full control of communication and data storage. As shown in Figure 6.2, parallel overhead is kept at a minimum level by our MPI implementations, as they deliver fairly strong scaling performance.

Compared to the MPI version, a parallel procedure implemented in MapReduce or Spark has to be based on synchronized parallel operations. However, the high level of parallel overhead by MapReduce and Spark is not caused by synchronization loss alone. For example, our MapReduce implementation consists of a large number of MapReduce operations, each of which is launched as an independent MapReduce job and costs some time to setup virtual machines on workers. Virtual machines are containers that receive instructions from the master, execute mappers and reducers locally, and periodically update the worker's status with the master. In addition, as discussed in Section 6.3.2,

the output from each MapReduce operation is written to a distributed file system called HDFS and read from HDFS in the next operation. This incurs some disk I/O overhead which might be avoided by caching the data in memory. Furthermore, between each map and the reduce phase that follows, the mapper output is sorted and sent to specific reducers according to the keys, a step known as "shuffling" which often involves disk access as well.

Although we do not have a way to precisely measure these setup and disk I/O costs, Figure 6.1 offers some evidence that they contribute significantly to the parallel overhead. Note that the map phases are generally synchronized well as we do not observe any extended period of time where only a fraction of cores run mappers. In addition, the fraction of time spent on screening is extremely low (below 0.1%) across all cases, so the majority of the visible gaps between the various map phases are in fact caused by shuffling and disk access. These fixed, per-iteration costs are so high that if an average batch size β is increased from 100 to 200, which weakens screening, increases the number of simulation replications but reduces the number of iterations from 10 to 5, the MapReduce implementation actually finishes faster.

Compared to MapReduce, Spark eliminates disk I/O almost entirely and has the ability to group multiple operations into a single synchronized stage, which explains its better performance relative to MapReduce.

Overhead Related to Parallel Hardware

Inter-processor communication can be orders of magnitude slower than memory access. Particularly in a master-worker framework, a single master core

communicates with thousands of workers, sometimes simultaneously. Profiling results suggest that the loss in utilization in the MPI implementation (Table 6.9) is almost exclusively due to the master being a bottleneck and freed workers having to join a queue to communicate with the master. Our effort to limit this type of parallel overhead in our implementations involves running simulation replications in batches to control the frequency of master-worker communication. As evidenced in Tables 6.7 and 6.9, a larger batch size does improve utilization across all cases. However, a larger batch size also leads to lower screening frequency and more simulation replications. The optimal batch size, therefore, depends heavily on the actual communication speed supported by the hardware.

Another cause of parallel overhead for MapReduce and Spark implementations is the engines' built-in protection against core failures. Both engines replicate intermediate data across workers and relaunch any failed task on another worker. On XSEDE clusters, we rarely observe any core failure so the actual cost from re-running failed jobs is negligible, but the active replication of distributed dataset by both MapReduce and Spark adds another layer of hidden parallel overhead cost.

BIBLIOGRAPHY

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

[2] Robert E. Bechhofer, Thomas J. Santner, and David M. Goldsman. *Design and analysis of experiments for statistical selection, screening, and multiple comparisons*. Wiley New York, 1995.

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

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

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

[6] George Casella and Roger L. Berger. *Statistical inference*. Thomson Learning, Australia; Pacific Grove, CA, 2002.

[7] Chun-Hung Chen, Stephen E. Chick, Loo Hay Lee, and Nugroho A. Pujowidianto. Ranking and selection: Efficient simulation budget allocation. In Michael C Fu, editor, *Handbook of Simulation Optimization*, volume 216 of *International Series in Operations Research & Management Science*, pages 45–80. Springer New York, 2015.

[8] Chun-Hung Chen, Jianwu Lin, Enver Yücesan, and Stephen E. Chick. Simulation budget allocation for further enhancing the efficiency of ordinal optimization. *Discrete Event Dynamic Systems*, 10(3):251–270, 2000.

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

[10] D.R. Cox and H.D. Miller. *The Theory of Stochastic Processes*. Science paperbacks. Taylor & Francis, 1977.

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

[12] Ahmed Elgohary. Stateful MapReduce. Retrieved May 15, 2015, `http://bbs.chinacloud.cn/attachment.aspx?attachmentid=4762`, 2012.

[13] Vaclav Fabian. Note on Anderson's sequential procedures with triangular boundary. *The Annals of Statistics*, 2(1):170–176, 1974.

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

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

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

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

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

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

[20] W. J. Hall. The distribution of Brownian motion on linear stopping boundaries. *Sequential Analysis*, 16(4):345–352, 1997.

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

[22] S. G. Henderson and R. Pasupathy. Simulation optimization library, 2014.

[23] L. Jeff Hong. Fully sequential indifference-zone selection procedures with

variance-dependent sampling. *Naval Research Logistics (NRL)*, 53(5):464–476, 2006.

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

[25] C. Jennison, I.M. Johnston, and B.W.Turnbull. Asymptotically optimal procedures for sequential adaptive selection of the best of several normal means. *Technical Report, Department of Operations Research and Industrial Engineering, Cornell University*, 1980.

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

[27] S.-H. Kim and B. L. Nelson. Selecting the best system. In S. G. Henderson and B. L. Nelson, editors, *Simulation*, Handbooks in Operations Research and Management Science, pages 501–534. North-Holland Publishing, Amsterdam, Amsterdam, 2006.

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

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

[30] Pierre L'Ecuyer. Uniform random number generation. In S. G. Henderson and B. L. Nelson, editors, *Simulation*, Handbooks in Operations Research and Management Science, Volume 13, pages 55–81. Elsevier, 2006.

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

[32] G. K. Lockwood. myHadoop, 2014. `https://github.com/glennklockwood/myhadoop`.

[33] J. Luo and L. J. Hong. Large-scale ranking and selection using cloud computing. In S. Jain, R. R. Creasey, J. Himmelspach, K. P. White, and M. Fu,

editors, *Proc. of the 2011 Winter Simulation Conference*, pages 4051–4061, Piscataway, NJ, 2011. Institute of Electrical and Electronics Engineers, Inc.

[34] Jun Luo, Jeff L. Hong, Barry L. Nelson, and Yang Wu. Fully sequential procedures for large-scale ranking-and-selection problems in parallel computing environments. *Working Paper*, 2013.

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

[36] Yuh-Chuyn Luo, Chun-Hung Chen, E. Yucesan, and Insup Lee. Distributed web-based simulation optimization. In *Proceedings of the 2000 Winter Simulation Conference*, volume 2, pages 1785–1793, 2000.

[37] Shie Mannor, John N. Tsitsiklis, Kristin Bennett, and Nicol Cesa-bianchi. The sample complexity of exploration in the multi-armed bandit problem. *Journal of Machine Learning Research*, 5:2004, 2004.

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

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

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

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

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

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

[44] Eric C. Ni, Dragos F. Ciocan, Shane G. Henderson, and Susan R. Hunter.

Comparing Message Passing Interface and MapReduce for large-scale parallel ranking and selection. In L. Yilmaz, W. K. V. Chan, I. Moon, T. M. K. Roeder, C. Macal, and M. D. Rossetti, editors, *Proceedings of the 2015 Winter Simulation Conference*, page Submitted, 2015.

[45] Eric C. Ni, Dragos F. Ciocan, Shane G. Henderson, and Susan R. Hunter. Efficient ranking and selection in parallel computing environments. *Working Paper*, 2015.

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

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

[48] R. Pasupathy and S. Ghosh. Simulation optimization: A concise overview and implementation guide. In H. Topaloglu, editor, *TutORials in Operations Research*, chapter 7, pages 122–150. INFORMS, 2013.

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

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

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

[52] Texas Advanced Computing Center. TACC stampede user guide. Retrieved May 11, 2014, `https://www.tacc.utexas.edu/user-services/user-guides/stampede-user-guide`, 2014.

[53] Texas Advanced Computing Center. TACC wrangler user guide. Retrieved July 11, 2015, `https://portal.tacc.utexas.edu/user-guides/wrangler`, 2015.

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

[55] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster Computing with Working Sets. In *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'10, page 10, Berkeley, CA, USA, 2010. USENIX Association.