e-companion

EC.1. Proof of Theorem 1.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

□

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

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

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

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

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

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

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

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

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

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

thereby completing the proof. □

EC.2. On computing the parameter η

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

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

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

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

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

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

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

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

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

EC.3. Full Description of the MPI implementation

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

end

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

end

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

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

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

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

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

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

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

where β is a pre-determined average batch size.

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

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

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

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

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

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

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

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

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

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

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

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

EC.4. Full Description of the Hadoop MapReduce implementation

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

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

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

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

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

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

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

Step 1.

- **Map**: Estimate S_i^2

  Input $[i]$

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

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

- **Reduce**

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

  Operation Calculate $\sum_i S_i$.

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

Step 2.

- **Map**: Calculate batch size

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

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

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

- **Reduce**: Screen within a group

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

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

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

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

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

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

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

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

Operation (2) Output to all groups.

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

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

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

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

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

Step 4. • **Map**: Simulation

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

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

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

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

(Same as Step 2 Reduce.)

Step 5. Screen against best systems from other groups.

(Same as Step 3.)

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

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

Operation Output to Reducer.

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

• **Reduce**

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

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

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

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

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

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

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

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

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

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

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

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

Input (This step has only one Reducer)

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

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

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

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

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

Table EC.1 A comparison of two implementations of GSP using parameters $\delta = 0.1$, $n_0 = 50$, $\alpha_1 = \alpha_2 = 2.5\%$, $\bar{r} = 1000/\beta$. "Total time" is summed over all cores. (Results to 2 significant figures)

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

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

EC.5. Full Description of the Spark implementation

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

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

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

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

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

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

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

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

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

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

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

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

Figure EC.4 Spark Implementation

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

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

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

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

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

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

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

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

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

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

References

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

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

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

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

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