A Program for Sequential Allocation of Three Bernoulli Populations

Janis Hardwick Robert Oehmke Quentin F. Stout
University of Michigan
Ann Arbor, Michigan 48109 USA

Abstract

We describe a program for optimizing and analyzing sequential allocation problems involving three Bernoulli populations. Previous researchers had considered this problem computationally intractable, and we know of no prior exact optimizations for such problems, even for very small sample sizes. Despite this, our program is currently able to solve problems of size 200 or more by using a parallel computer, and problems of size 100 on a workstation. We describe the program and the techniques used to enable it to scale to large sample sizes. As an illustrative example, the program is used to create an adaptive sampling procedure that is the optimal solution to a 3-arm bandit problem. The bandit procedure is then compared to two other allocation procedures along various metrics. We also indicate extensions of the program that enable it to solve a variety of related problems.

Keywords and phrases: multi-arm bandit, parallel computing, dynamic programming, adaptive allocation, sequential sampling, clinical trial, load balancing, high-performance computing, recursive equations, design of experiments

1 Introduction

A *sequential* (or *adaptive*) *allocation* problem is one in which an investigator has the option to determine how to expend resources during the experiment based on the observations that have been obtained so far. This is in contrast to *fixed allocation* problems in which resources are assigned prior to the beginning the experiment. Here, we are interested in adaptive sampling problems in which the resources in question are the experimental units available and the investigator identifies the population from which to sample at each decision time. Note that the expressions *sampling from a population* and *allocating an experimental unit* refer to the same action.

An important class of sequential sampling problems are *bandit* problems in which each population has associated with it a reward. The goal of a bandit experiment is to sample from the available populations in such a way as to maximize the total reward at the termination of the experiment. The term "bandit" is a reference to a slot-machine with a single arm that has an unknown probability of paying off. Each time a coin is put into the machine, a random outcome is observed. One can extend this process to the case in which α machines are available, and coins are dropped in the different machines in an attempt to locate the machine that delivers the most money on average. Thus, an α-arm bandit is a model for problems in which

- there are α populations with unknown reward structures,
- sampling (or *arm pulling*) takes place sequentially, and
- decisions are made with the intent to optimize the total payoff.

Bandit models are typically *fully sequential*, in that each outcome is observed before a decision is made as to the next population to be sampled. They are used to model a variety of optimization and learning problems. In particular, bandits arise in the design of ethical clinical trials in which the goal is to minimize patient failures that occur during the trial. See [4] for an in-depth discussion of bandit problems.

In many situations, the performance of an adaptive design can be dramatically superior to that of a fixed allocation design in which all sampling decisions have been made in advance. An example of a fixed design for a clinical trial is one in which $1/\alpha$ of the subjects are assigned to each of the α therapies under consideration. If, during the trial, one of the treatment groups appears to be faring far less well than the others, a fixed design provides no mechanism for adjusting the sampling ratios. Through adaptation, however, one can significantly reduce costs or fatalities without sacrificing statistical objectives such as maximizing the probability of determining the best or better therapies.

Adaptive statistical designs have many applications and are highly flexible. Nevertheless, such designs are rarely used. One reason for this is that many statisticians are unfamiliar with sequential designs. Further, most practitioners are aware that examining data during an experiment can introduce biases and error during the analysis phase of a study. Analyzing sequential data requires new approaches. The "rules" are somewhat different and thus a bit controversial. Another important impediment to the adoption of adaptive designs has

been their analytic and computational intractability. With regard to the computational complexity, for example, typical comments include "the computation involved is prohibitive except for trivially small horizons" [1] and "In theory the optimal strategies can always be found by dynamic programming but the computation required is prohibitive" [21]. Note that the "horizon" of an experiment refers to the number of experimental units that are available for the experiment. For our purposes here, the horizon is simply the sample size, n, of the experiment.

In this paper, we pursue the three-armed problem to illustrate the progress that has been made in generating solutions to these demanding problems. We have implemented an algorithm for the optimization and analysis of sequential allocation problems involving three Bernoulli populations. For *optimization*, we assume there is some objective function V, and that the goal is to produce the sequential allocation procedure that minimizes (or maximizes, as appropriate) the value of V. For example, in a clinical trial, V may be the expected number of failures, while for an estimation problem it may be the mean squared error of an estimator. This optimization requires a Bayesian framework.

By *analysis* we mean that given an arbitrary adaptive allocation procedure, we can evaluate its expected behavior for various criteria. For example, one may want to determine the expected value of V for a simple adaptive procedure and compare it to the value attained using the optimal sampling procedure. As another example, one may have determined the optimal procedure with respect to an objective function V, but then ask for the expected value of this procedure with respect to other objectives. In the analysis phase for these procedures, one can use either a Bayesian or frequentist point of view. Note that the analysis framework need not be the same as the framework used to design the allocation procedure. This allows investigators to approach their research with added flexibility.

1.1 Previous Work

While there is extensive literature on sequential sampling, almost all of it is concerned with asymptotic behavior. Very little exact optimization or evaluation has been done.

With regard to bandit problems, there is a highly prominent result that determines optimal procedures for a class of multi-armed bandit problems with infinite horizons (see Gittins [9] for this result and related work). However, as is the case with the present problem, the computations required to generate the procedure are very difficult. Further, even if the procedure were readily available, the results would apply exactly only if the horizon were infinite, and geometric discounting was used for a bandit objective function. Most of the other work on bandit problems involving exact optimizations has focused on 1- or 2-arm Bernoulli bandit designs with very small sample sizes. Because of this past focus on Bernoulli bandits, we use the term *bandit problem* to mean Bernoulli bandit problem from this point on.

A 1-arm bandit represents a model in which there are two populations and the reward structure for one of them is known. Such problems are stopping rule problems and do not require the treatment described here. However, with the addition of a second random arm, we get the 2-armed bandit which requires a dynamic programming solution. For 2-armed bandits, a typical optimization was carried out by Jones [16], who solved a problem of size

$n = 25$, and noted the difficulties of solving larger problems. Kulkarni and Kulkarni [17] also noted that the computation required for 2-armed bandits make it "impractical to compute the decision even for moderate values of $n \geq 50$". It is our understanding that, prior to our work, the largest 2-armed bandit problem that had been solved appeared in the paper of Barry and Eick [3], which reports on work done around 1987. Utilizing a Cray 2 supercomputer, they were able to handle a sample size of $n = 200$. In early 1991, we began working on improving algorithms for 2-armed bandits. Soon thereafter, we were able to run problems of sizes greater than $n = 400$ and with more complex evaluations on our office workstations. Until now, however, problems requiring exact solutions to 3-arm bandits have been considered infeasible.

It is useful to recall that a variety of approaches have been taken with the more general problem of sequential sampling from three or more arms. For example, Siegmund [19] and Coad [8] have applied repeated significance testing to the case where several arms are available and the outcome variables have normal distributions. Betensky [5, 6], also working with normal outcome variables, has used other hypothesis testing approaches to tackle the 3-arm problem. Bather and Coad [2] addressed the multi-armed problem with Bernoulli outcomes, and in this work, they emphasize locating procedures that work well along several criteria but do not attempt to optimize on any given criterion. A number of researchers have also examined multi-arm problems using nonparametric ranking and selection methods. The primary goal of such designs is to select either the best of several arms or a best subset of arms. See [10] and [7] for examples of this approach.

Whether it's a testing problem or a selection problem, most multi-armed designs incorporate a mechanism for removing obviously poor arms during the experiment. While we do not explicitly include such an option here, the optimal sampling procedures we generate effectively remove poor arms, in the sense that they do not continue to sample from them. Further, our analysis routines can evaluate arbitrary 3-arm sequential procedures, including ones that eliminate arms.

We noted that most of the procedures that have been suggested for multi-arm problems have been derived from asymptotic arguments. To ascertain the behavior of the procedures for practical sample sizes, simulation studies are typically used. Many excellent procedures have been developed in this manner. Still, we know of only one for which exact optimality has been obtained. Palmer [18] optimized a 3-arm knock-out tournament in which one samples equally often from each population, sampling until 1 arm can be eliminated. The experiment continues with equal allocation from the remaining 2 arms. This is not an optimal solution to the problem of identifying the best arm, since the allocation strategy is partially fixed. Palmer's solutions also required that fairly restrictive conditions be imposed on the prior distributions for the parameters representing the success probabilities of the arms. Overall, then, we know of no prior work addressing exact, optimal solutions for fully sequential, finite horizon, 3-arm problems.

1.2 Model Used

We are interested in α-arm sequential allocation problems in which the arms represent populations of Bernoulli random variables, indexed as $1, \ldots, \alpha$. The outcomes are viewed as *successes* and *failures*, where the success probability on arm i is P_i for $i = 1, \ldots, \alpha$.

The arms and pulls are assumed to be independent. At each stage, $m = 0, 1, \ldots$, of an experiment, we select a population and observe the response. At stage m, let (s_i, f_i) represent the number of successes and failures respectively from arm i. Then $m = \sum(s_i + f_i)$ and $\langle s_1, f_1, \ldots, s_\alpha, f_\alpha \rangle$ is a vector of sufficient statistics for this problem. This vector will be called a *state*. We utilize a Bayesian approach, in which the population parameters, P_i, have a prior distribution which is the product of α independent distributions. In our program these distributions are beta, but other distributions may be used as well.

We assume that there are exactly n observations available, the *fixed horizon* model. This assumption merely provides a uniform framework for comparison, and can be easily relaxed to allow for optional stopping.

We assume that there is an objective function V. For optimization the goal is to minimize $\mathbf{E}(V)$, while for analysis the goal is merely to determine $\mathbf{E}(V)$. We require that V can be computed by merely knowing the terminal state reached (and the priors). While this includes the majority of objective functions of interest, it does exclude some. For example, if one wanted to determine the expected length of the longest run (consecutive pulls on the same arm) during the experiment, somewhat different programs would be needed. In such a situation, one would expand the state space to include the information needed to determine V.

2 Computational Issues

To describe the time and space requirements of algorithms, we use "generalized O-notation" from computer science, in which O and o have the same meanings as in statistical use; and in which we say a function $f = \Theta(g)$ or $f(n) = \Theta(g(n))$ if there exist positive constants C, D, N such that $C \cdot g(n) \leq f(n) \leq D \cdot g(n)$ for all $n \geq N$.

For a sequential allocation problem of horizon n involving α Bernoulli arms, there are

$$\binom{n + 2\alpha}{2\alpha} \approx \Theta(n^{2\alpha}/(2\alpha)!)$$

states. (Our Θ-analyses assumes $\alpha \ll n$, and for most purposes we will fix $\alpha = 3$.) To optimize such problems, one typically uses a *dynamic programming* approach. One first computes the value of each terminal state (those with n observations), and then the optimal solution is found for all states with m observations based on the optimal solutions for all states with $m+1$ observations, for m ranging from $n-1$ down to 0. To determine the optimal solution at a state, one determines the expected value of each option available (a pull on an arm), and selects the best one. The relevant recursive equations are given in Figure 1.

While Figure 1 shows the algorithm for optimization, with only a small change it is also an algorithm for evaluation of an arbitrary adaptive design $\mathcal{A}$. If, instead of choosing the arm

```
{1_i^s, 1_i^f:  one success, failure on arm i}
{si, fi:  number of successes, failures arm i}
{m:  number of observations so far}
{n:  total number of observations}
{|σ|:  number of observations at state σ}
{V: the function being optimized, where V(0) is the answer}
{pi(si,fi):  prob of success on arm i, if si successes and fi failures
   have been observed}

For all states σ with |σ|=n, {i.e., for all terminal states}
   Initialize V(σ)

For m=n-1 downto 0 do {compute for all states of size m}
   For s3=0 to m do
   For f3=0 to m-s3 do
   For s2=0 to m-s3-f3 do
   For f2=0 to m-s3-f3-s2 do
   For s1=0 to m-s3-f3-s2-f2 do
      f1 = m-s3-f3-s2-f2-s1
      σ = ⟨s1,f1,s2,f2,s3,f3⟩
      V(σ) = min{
         (p1(s1,f1)·V(σ + 1_1^s) + (1-p1(s1,f1))·V(σ + 1_1^f)) ,
         (p2(s2,f2)·V(σ + 1_2^s) + (1-p2(s2,f2))·V(σ + 1_2^f)) ,
         (p3(s3,f3)·V(σ + 1_3^s) + (1-p3(s3,f3))·V(σ + 1_3^f)) }
```

Figure 1: Serial Algorithm for Determining Optimal Adaptive 3-Arm Allocation

---

that gives the optimal value of V, one uses the value of V corresponding to the arm chosen by $\mathcal{A}$, then the program determines the expected value of V obtained using procedure $\mathcal{A}$. This computational approach is known as *backward induction*.

For dynamic programming, it takes a constant amount of time to evaluate each arm, and thus the total amount of time required to optimize an α-arm allocation problem is $\Theta(n^{2\alpha}/(2\alpha-1)!)$. The time for backward induction can be a factor of α faster, if the determination of which arm $\mathcal{A}$ uses can be done in constant time per state, as opposed to the $\Theta(\alpha)$ time per state needed by dynamic programming. For a 3-arm problem, either dynamic programming or backward induction have the rather formidable growth rate of $\Theta(n^6)$.

Because of this growth rate, a parallel computer was needed to allow us to solve 3 population adaptive sampling problems of useful size in a feasible amount of time. Our goals were to write parallel code that is portable, maintainable, and flexible. In addition, we needed to maintain the serial efficiencies that had previously been exploited for 2-arm bandit-

like problems (see [14]). The parallel code is written in Fortran 77, with MPI (Message-Passing Interface) for the communication among the processors. These standard languages are available on most parallel computers, and also on distributed systems such as networks of workstations, so the program is quite portable.

In extending the previous 2-arm serial algorithms to a 3-arm parallel algorithm, the major new computational issues were:

1. Space reduction (useful for both serial and parallel execution)
2. Load balancing among the processors (for efficient parallel execution)

Space, rather than time, has become the limiting factor in solving fully sequential allocation problems. This is due, in part, to prior developments that allow for more efficient computations. Even with the space reductions described below, the ratio of computation time to RAM space for an α-arm model grows only as $\Theta(n^{1+1/(2\alpha-1)})$, i.e., it is nearly linear, and the ratio of time to disk space is linear.

3 Space

There are two different space issues that need to be addressed. One is the need to reduce the space utilized to store the `V` array, which is typically stored in RAM. The second is the amount of storage needed to keep track of the arm chosen at each state, and this can be kept on disk.

3.1 RAM Space

The program seems to imply that a 6-dimensional array is needed to store values of `V`, since the state space is 6-dimensional and `V` is computed at each state. However, this array can be compressed to a 5-dimensional array by reusing memory. This can be accomplished by recognizing that, given `m`, the value of `f1` is determined by knowing the values of `s3`, `f3`, `s2`, `f2`, and `s1`. Thus, `f1` can be omitted as an index, which means that array entries will be overwritten. It is simple to verify that if each of the inner loops is increasing, then an array entry for a specific `m` value is overwritten (by the corresponding entry for `m-1`) after all reads of the value corresponding to `m` have occurred (see [14] for a further discussion of this point). This is a well-known space compression technique for sequential allocation and other dynamic programming problems. Because the array locations are being reused, for high performance it is best if they are in RAM, although the sequential access patterns allows disk storage to achieve reasonable efficiency.

The next observation is that, due to the constraint that `s3+f3+s2+f2+s1+f1` $\leq n$, only a corner of the 5-dimensional array is actually used. The corner occupied is only approximately $1/5! = 1/120$ of the total array, so one can map this corner into a linear array and translate all array references. The algebra is straightforward but tedious, and algebraic manipulation packages can be used to help. The most straightforward way to implement this translation is to write a function, say `T(s1,f1,s2,f2,s3,f3)`, which computes the positions in the linear

array to which states get mapped. Using this, each reference to `V(`σ`)` is replaced by `V(T(`σ`))`. While this is, indeed, straightforward, it also has the unfortunate effect of dramatically increasing the computational time. This is because `T` is a somewhat complicated 5th degree polynomial and relatively little computation is done per array position accessed. As a result, far more time would be spent on determining array positions than on using their contents.

To alleviate this problem, while still mapping into a linear array, we instead decomposed `T` into a series of offsets. At each level of nesting of the loops, the offset of that level is added to that of the previous one. We used a mapping such that the inner levels add the smallest offset. Thus, at the innermost level, the final position calculation is merely to add `s1` to the offset from the preceding level. This makes the position calculations quite efficient, although it has the regrettable effect of making the code harder to read and maintain. Note that such a mapping also maintains good cache utilization, since the innermost loop accesses array positions in consecutive order.

3.2 Disk Space

Unfortunately, in many cases `V` is not the only array required, because one needs to know more than just `V(0)`. For example, one may want to utilize the allocation procedure for an experiment, or evaluate it along additional criteria. In such settings, one needs to keep a record, at each state σ, of which arm to pull to achieve the optimal value. It is a common property of dynamic programming that one must add additional storage to record the decisions which achieve the optimal value.

Since the decisions must be retained for all states, the array used to store this information cannot overwrite values. As a result, the decision array remains 6-dimensional, although it too can be collapsed into a corner, which allows one to reduce its space requirements by approximately $1/6! = 1/720$. This array needs only 3 bits per state (we allow for the possibility of ties, so there are 7 possible outcomes), although for convenience we used one byte per state. Thus the total space for the decision array grows approximately as $n^6/720$ bytes. In some cases one can use monotonicity properties of the optimal decisions to reduce the space needed (see [14]), but since we wanted to develop a general purpose algorithm suitable for arbitrary objective functions this approach was not used.

Fortunately, the decision array is written to once, and in later analyses is only read once. This is because nearly all analyses can be accomplished via *path induction* [15], which, after a single initialization pass through the decision array, reduces each evaluation to a computation over the final states. The path induction can be written to visit the states in the reverse order they were visited for the dynamic programming, and hence the decision array can be read in reverse of the order in which it was written. Thus a simple serial write/read mechanism can be used, which allows us to store the decision array on disk with relatively little loss of efficiency. A serial implementation of path induction is given in Figure 2.

Note that the path induction algorithm allows for a random choice of arm. This permits one to evaluate various biased coin or urn-based allocation schemes, or optimal allocation schemes when ties result in randomizing among the arms achieving the optimal `V` value.

Since users often want to evaluate a design on a variety of secondary criteria (such as

```
{path(σ): number of paths reaching state σ}
{probi(si,ni): probability that ni pulls of arm i have exactly si
successes}
{prob(σ): prob1(s1,s1+f1)·prob2(s2,s2+f2)·prob3(s3,s3+f3)}
{p(σ,i): probability that arm i selected at state σ}

{during computation of V(σ), the arm(s) that were selected are written to
disk}

{initialization phase}
path(0)=1
For m=0 to n-1 do
    For s3=m downto 0
    For f3=m-s3 downto 0 do
    For s2=m-s3-f3 downto 0 do
    For f2=m-s3-f3-s2 downto 0 do
    For s1=m-s3-f3-s2-f2 downto 0 to
        Read arms used for σ = ⟨s1,f1,s2,f2,s3,f3⟩
        For all arms i used for σ
            path(σ + 1_i^s) = path(σ + 1_i^s) + p(σ,i)·path(σ)
            path(σ + 1_i^f) = path(σ + 1_i^f) + p(σ,i)·path(σ)

{evaluation phase}
For all post-analysis parameters π
    W(π) = ∑ {path(σ)·prob(σ)·W(π,σ) : σ a terminal state}
```

Figure 2: Serial Implementation of Path Induction

---

robustness), the use of path induction is an important step in making such high-dimensional designs practical. Secondary criteria play a particularly important role in the design of clinical trials since researchers may need to optimize competing factors. Typically, to do this requires repeated reevaluation of the design, and this ultimately becomes the most time-consuming part of the entire computational process. Prior to our introduction of path induction, each such evaluation was evaluated via backward induction (see, for example, [3]). For 3 populations, this requires $\Theta(n^6)$ time per evaluation. With path induction, there is still an initialization step that requires $\Theta(n^6)$ time. However, each subsequent evaluation occurs only over the final states, requiring only $\Theta(n^5)$ time. As for the space requirements, beyond the decision array one needs the `path` array to keep the path counts. However, this array can re-occupy the space used by `V`, utilizing the same compression techniques. The access patterns of the `path` array are the exact reverse of those occurring for `V`.

```
{P_j:  processor j}
{start_s3(j,m), end_s3(j,m):  range of s3 values assigned to P_j for this m
value,
   with start_s3(j+1,m)=end_s3(j,m)+1 }

{For all processors P_j simultaneously, do}

For all states σ assigned to P_j with |σ|=n,
   Initialize V(σ)

For m=n-1 downto 0 do {compute for all states of size m}
   For s3=start_s3(j,m) to end_s3(j,m) do
      For f3, s2, f2, s1 as before do
         compute V as before
      Send needed V values to P_{j-1}
      Receive V values from P_{j+1}
      Send needed V values to P_{j+1}
      Receive V values from P_{j-1}
```

Figure 3: Initial Parallel Algorithm

4 Load Balancing

The goal of load-balancing is to minimize the amount of time taken by the most heavily loaded processor in a parallel computer. This is a critical concern for achieving efficient parallel performance. Load-balancing high-dimensional dynamic programming codes such as those in Figure 1 is a non-trivial problem, even though all of the load information can be determined in advance. When examining the dependencies, one sees that the code is similar to a simple PDE solver over a regular grid, where the outermost loop (on `m`, the number of pulls) acts like a time variable, and hence cannot be parallelized. At the same time, the inner loops act like space variables which are amenable to parallelization, despite the dependencies among neighbors. However, unlike a PDE solver, the "space" rapidly shrinks with `m`, and is a high-dimensional simplex.

4.1 Initial Parallel Version

A natural first approach is to parallelize at the outermost loop possible, which is the `s3` loop, assigning each processor an interval of values. Using an interval of values both reduces the amount of communication, and makes the parallel code as similar as possible to the serial version. This is an important consideration when one is trying to maintain and extend serial and parallel solutions to the same problem. For a given value of `m`, the range of `s3` intervals assigned to processor $\mathcal{P}_j$ is `start_s3(j,m)...end_s3(j,m)`. A very simplified pseudo-code version of the parallel code is given in Figure 3.

Once the iterations are assigned, the message-passing required to send neighbor information is straightforward, based on the recurrence equation for `V`. If the `start_s3` and `end_s3` values did not change with `m`, then the only messages needed would be for processor $\mathcal{P}_j$ to send processor $\mathcal{P}_{j-1}$ a copy of the `V` values corresponding to `s3 = start_s3(j,m)`. This message-passing is the first send-receive pair in Figure 3. These would need to be sent at the end of each iteration of `m`. However, because the iterations assigned to a processor can change with each `m` value, the value of `start_s3(j,m-1)` can be smaller than `start_s3(j,m)`. In this case, processor $\mathcal{P}_j$ needs the values of `V` corresponding to `s3` in the range `start_s3(j,m-1)...start_s3(j,m)`. These are obtained from processor $\mathcal{P}_{j-1}$, in the second send-receive pair in Figure 3.

Unfortunately the `s3` iterations do not represent uniform load since the amount of work grows like $(\texttt{m-s3})^4$. As a result, one cannot merely assign each processor the same number of iterations. Determining the partitioning for optimal load balance can be accomplished via dynamic programming, taking $\Theta(mp)$ time for p processors. To reduce the overhead, we developed a simple yet effective heuristic which emphasizes careful assignment of the iterations with the most work (i.e., those with small `s3` value). This is important because misplacing a single iteration can create a significant imbalance if the iteration requires a large amount of work. This heuristic runs in $\Theta(m)$ time and is given in [13], along with comparisons to the optimal subdivision of the `s3` loops. We refer to this algorithm as the *initial* parallel algorithm, and it is the one that was used for the work reported in [11].

The changes needed for the initialization phase of path induction are the same as those needed for dynamic programming, since it performs nearly identical calculations in the reverse order. For the evaluation phase, the only modification needed is a reduction operation to combine the values from the individual processors into a single value. Reduction operations are explicitly provided in MPI and other parallel programming systems. This communication is quite efficient, especially when compared to the message passing required at each stage of dynamic programming or backward induction. Since we often find it useful to have nearly 100 reevaluations of a design with n in the range of a few hundred, the use of a path induction algorithm results in substantial savings for both serial and parallel implementations.

4.2 Improved Scalability

The initial parallel algorithm is suitable only for a small to moderate number of processors because the algorithm's load balance is imperfect. For a given n, this imbalance worsens as the number of processors increases, because the work per `s3` iteration varies so greatly. This holds even when the `start_s3` and `end_s3` functions are determined optimally. As one might expect, because the computational and space requirements grow rapidly with the sample size, n, more processors are needed to solve problems with large sample sizes in an acceptable amount of time. Further, if the goal is to try to solve problems that barely fit into the total space available, then the imbalance of the initial parallel algorithm may render it unsuitable even with few processors.

To make a truly scalable algorithm able to handle large problems near the limit of a machine's capacity, it is necessary to balance the work and space much more evenly. To do

```
{P_j:  processor j}
{start_σ(j,m), end_σ(j,m):  range of σ values assigned to P_j for this m
value,
   with start_σ(j+1,m)=end_σ(j,m)+1 }

{For all processors P_j simultaneously, do}

For σ=start_σ(j,n) to end_σ(j,n) do {initialize terminal states}
   Initialize V(σ)

For m=n-1 downto 0 do {compute for all states of size m}
   For σ=start_σ(j,m) to end_σ(j,m) do
      determine s1, f1, s2, f2, s3, f3 from σ
      compute V as before
   Send needed V values to other processors
   Receive V values from other processors
```

Figure 4: Scalable Parallel Algorithm

this, we note that the 1-dimensional `V` array can easily be subdivided evenly into intervals. However, because a given interval may start and end at arbitrary value of `s3`, `f3`, `s2`, `f2`, and `s1`, such a partitioning does not correspond to simple subdivisions of the control loops. Thus we converted to a control structure which worked on intervals of `V` entries, and determined corresponding values of `s3`, ..., `f1`. Changes were also needed to determine the index ranges and location of `V` entries needed from and by other processors, as processor $\mathcal{P}_j$ may now need to exchange values with processors other than $\mathcal{P}_{j\pm1}$. An overview of the improved algorithm is given in Figure 4. By utilizing techniques such as the offset calculations mentioned in Section 3.1, the overhead for determining `s3`, ..., `f1`, along with the locations of the `V` entries needed to determine `V(σ)`, can be kept at an acceptable level which is comparable to that needed in the serial algorithm. Further details can be found in [13], along with experimental analyses of the time and space needed by the improved algorithm.

It should be noted that such extensive changes come at a cost. Besides being tedious, they are more error-prone, and the resulting code is more difficult to understand and maintain. The greater the deviation from a simple serial description, the worse these problems become. If such changes could be made automatically, this situation would be greatly improved, but no current systems can do this. Space compression and nested loops with ranges that depend on outer loops are beyond current parallelization tools.

5 Application

To illustrate the use of the parallel algorithm in Figure 4, it was applied to the design and analysis of three sequential allocation procedures involving 3 arms. Our intent here is not

to promote any specific design, but rather to show that the algorithm provides heretofore unattainable exact evaluations of these procedures for useful sample sizes.

The procedures examined were:

Bandit The fully sequential design which maximizes the expected number of successes. It is determined via dynamic programming.

Myopic A fully sequential design which chooses, at each state, the arm that has the highest probability of producing a success.

Equal Allocation A commonly used fixed design approach, in which each arm receives $n/3$ pulls. This procedure is also referred to as *vector at a time* sampling.

As noted, to optimize the bandit procedure, a Bayesian approach is taken in the design phase. Similarly, myopic allocation is also determined via a Bayesian approach. Recall, however, that the procedures can be analyzed from either a Bayesian or frequentist perspective. To illustrate this, the allocation schemes were compared (analyzed) according to two criteria — one Bayesian and the other frequentist.

The first criterion is the *expected number of failures* given the prior distribution which, in the examples here, is the product of three uniforms. This is the criterion that the bandit optimizes. Myopic allocation, which always seeks to make the best decision based on taking only one more observation, is a commonly referenced ad hoc attempt to achieve similar performance. Note that dynamic programming is needed to determine the bandit allocation, and backward induction is needed to determine the expected value of myopic allocation. For equal allocation, the expected number of failures is just the sum, over all arms, of $n/3$ times the prior probability of failure.

The second criteria examined is the *probability of correct selection* or "P(CS)". Given an indifference tolerance δ (herein selected to be 0.1), the probability of correct selection is the minimum, over all arm probabilities p_1, p_2, p_3, of the probability that at the end of the experiment, the arm declared the winner has a success probability within δ of the success probability of the best arm. (By *winner* we mean the arm with the highest observed rate of success. In case of ties, the winner was selected randomly, as is standard.) For an arbitrary allocation algorithm it is not known which values of p_i yield the minimum, which indicates a search throughout the parameter space is needed to determine P(CS). However, it can be shown that the minimum occurs when one arm is exactly δ better than the other two so the dimension of the relevant search space is reduced. P(CS) is an example of a criterion for which an allocation algorithm needs to be evaluated multiple times. Because of these multiple evaluations, path induction was used to determine P(CS) for the bandit and myopic designs. For equal allocation much simpler approaches were employed.

In general one expects equal allocation to perform extremely well on the P(CS) criterion. For 2 arms, in fact, this procedure can easily be shown to be optimal. However, for 3 arms equal allocation is no longer optimal, as there exist adaptive procedures that are better. In general, for a given n, the design that has the optimal P(CS) is not known for 3 or more arms. Standard dynamic programming approaches cannot be used to solve this problem because of the nonlinear nature of the minimum operation in the definition of P(CS).

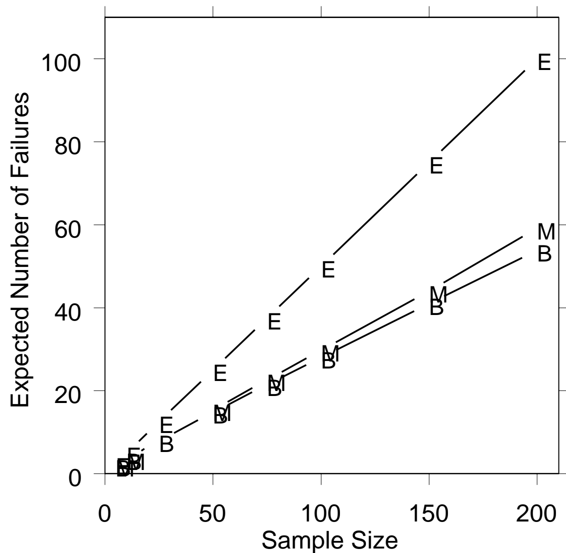

Figure 5: Sample Size vs. E(Failures)

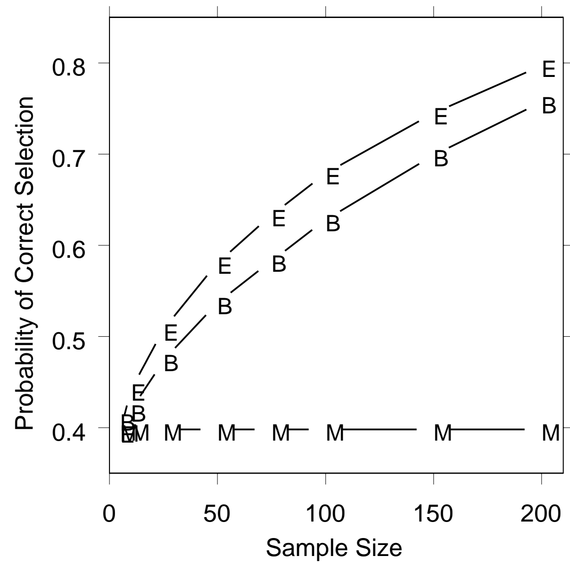

Figure 6: Sample Size vs. P(CS), $\delta = 0.1$

All arms had uniform priors.

In Figure 5 we plot expected failures for the three procedures as a function of the sample size. Similarly, in Figure 6 we plot P(CS) as a function of sample size. Uniform priors were used throughout merely for comparison purposes. The program currently handles arbitrary beta priors, and can easily be adapted to handle other prior distributions.

Note that, for these priors, the bandit allocation comes very close to achieving the P(CS) of equal allocation, while incurring far fewer failures. Myopic allocation also incurs few failures, but has a very poor ability to correctly locate the best arm. For the indifference region of $\delta = 0.1$, the minimum P(CS) for myopic allocation occurs when one arm has a success probability of 1 and the others have probability 0.9. In this situation, there is a probability > 0.6 that it will never try the superior arm. The asymptotic P(CS) of myopic allocation, as n tends to infinity, is less than the P(CS) of equal allocation using $n = 6$. There are simple ways to alter myopic allocation so that the P(CS) significantly improves with very little increase in failures, but that is beyond the scope of this work.

6 Extensions and Future Work

To summarize the current standing of this project, the serial (Figures 1 and 2), initial parallel (Figure 3), and improved parallel (Figure 4) algorithms have all been implemented, with fully operational dynamic programming/backward induction and path induction. These programs can be applied with very general evaluation criteria V which typically will be application specific. While we illustrated only a bandit objective here, minimizing total failures, with trivial changes the program can be applied to estimation problems. One can also add sampling costs, optional stopping, etc. As was illustrated, the dynamic programming can be

used to produce optimal Bayesian designs, and the backward and path induction can be used to evaluate arbitrary designs with respect to very general criteria which can be either Bayesian or frequentist. All optimizations and evaluations are exact.

At present the scalable algorithm of Figure 4 can handle sample sizes as great as $n = 200$ using only 16 processors of an IBM SP2, where each processor has 1GB of RAM and a local disk. Using only one processor, we are able to handle sample sizes greater than 100. We are currently conducting studies for larger values of n and more processors, analyzing both the time and space aspects of scalability. The results will be reported in [13].

From a statistical vantage point, we plan to evaluate optimal and sub-optimal strategies along multiple criteria and also to examine the operating characteristics of all procedures under consideration. As noted earlier, very little is known about the behavior of 3-arm strategies, especially optimal strategies. Further, as highlighted in the discussion of equal allocation and P(CS) in Section 5, even problems that are well-understood for 2 arms may be quite complex for 3 arms.

We also plan to extend the design features of the algorithm. For example, the parallel program for the 3-arm bandit can be trivially adapted to solve 2-arm bandits with trichotomous responses. This is because the natural states are of the form $\langle o_1^1, o_1^2, o_1^3, o_2^1, o_2^2, o_3^3 \rangle$, where o_j^i indicates the number of outcomes of type i on arm j. The recurrences for this problem would be of the form

$$V(\sigma) = F(V(\sigma + \mathbf{1}_1^1), V(\sigma + \mathbf{1}_1^2), V(\sigma + \mathbf{1}_1^3), V(\sigma + \mathbf{1}_2^1), V(\sigma + \mathbf{1}_2^2), V(\sigma + \mathbf{1}_2^3)),$$

where $\mathbf{1}_j^i$ denotes a single observation of outcome i on arm j. Note that this recurrence has the same dependency structure as the recurrence in Figure 1, and hence all of the communication requirements are identical. A special case of the trichotomous response problem is a 2-arm sequential allocation problem with censored outcomes. A *censored observation* is one in which the outcome cannot be observed (e.g., a patient dies of causes unrelated to the treatment being studied). If the censoring mechanism is independent of the arm selected, then a 2-arm fully sequential allocation problem can be optimized via a 5-dimensional dynamic programming approach. However, if censoring is not independent of the arm, then 6-dimensional dynamic programming is needed. In either case, the recurrences remain near-neighbor recurrences, as in Figure 1. As a result, the present work can be easily applied to them.

With more substantive changes, one can also address problems involving 2-arm sequential allocation with delayed responses. In *delayed response* problems, one doesn't necessarily know the outcome of past decisions before new ones must be made. In a cancer therapy study, for example, the ideal response is that the subject lives for a long period of time. In the interim, however, other patients are admitted for treatment. During an experiment, then, an investigator typically has observed some outcomes and has also started several patients for whom the outcomes are not yet known. Nevertheless, the investigator retains the goal of making the best possible assignment for each new patient given the data that is actually known at the time. Clearly, the state space for this problem is larger than it is for the problem in which each outcome is known before the next subject needs to be assigned. However, some models of the 2-arm allocation problem with delayed response can

be handled by a 6-dimensional recurrence only somewhat more complex than that which appears in Figure 1. Such models can be solved by modifying the calculations and message-passing of our current program.

Of course, one can always pursue an extension involving an increase in the number of arms or the number of responses per arm. However, because the running time and space grow exponentially in the total number of arm responses, this rapidly limits the sample size that can be optimized or evaluated.

In closing, to our knowledge, no non-trivial optimal solutions have been produced for any of the problems just described, with the exception of our preliminary work on the 2-arm model with censoring independent of the arm [12]. We are especially interested in making progress on the censored data and delayed response problems because these extensions address important real-world considerations that have long obstructed adaptation of adaptive experimental designs.

Acknowledgments

Research supported in part by National Science Foundation grants DMS-9157715 and DMS-9504980. Computational support was provided by the Center for Parallel Computing at the University of Michigan.

References

[1] Armitage, P. (1985), "The search for optimality in clinical trials", *Int'l. Statist. Rev.* **53**, pp. 15–24.

[2] Bather, J.A. and Coad, D.S. (1992), "Sequential procedures for comparing several medical treatments", *Sequential Anal.* **11**, pp. 339–376.

[3] Berry, D.A. and Eick, S.G. (1995), "Adaptive assignment versus balanced randomization in clinical trials — a decision-analysis", *Stat. in Medicine* **14**, pp. 231–246.

[4] Berry, D.A. and Fristedt, B. (1985), *Bandit Problems: Sequential Allocation of Experiments*, Chapman and Hall.

[5] Betensky, R.A. (1992), *A Study of Sequential Procedures for Comparing Three Treatments*, Ph.D. Thesis, Stanford University.

[6] Betensky, R.A. (1996), "An O'Brien-Fleming sequential trial for comparing three treatments", *Annals of Statistics* **24**, pp. 1765–1791.

[7] Buringer, H., Martin, H. and Schriever, K. (1980), *Nonparametric Sequential Selection Procedures*, Birkhauser.

[8] Coad, D.S. (1995), "Sequential allocation rules for multi-armed clinical trials", *J. Statist. Comput. Simul.*, **52**, pp. 239–251.

[9] Gittins, J.C. (1989), *Multi-Armed Bandit Allocation Indices*, Wiley.

[10] Gupta, S.S. and Liang, T. (1989), "Selecting the best binomial population: parametric empirical Bayes approach", *J. Statistical Planning and Inference* **23**, pp. 21–31.

[11] Hardwick, J., Oehmke, R. and Stout, Q.F. (1997), "A parallel program for 3-arm bandits", *Computing Science and Statistics* **29**, pp. 390–395.

[12] Hardwick, J., Oehmke, R. and Stout, Q.F. (1998), "Adaptive allocation in the presence of censoring", *Computing Science and Statistics* **30**.

[13] Hardwick, J., Oehmke, R. and Stout, Q.F., "Scalable parallel implementation of high-dimensional dynamic programming", in preparation.

[14] Hardwick, J. and Stout, Q.F. (1993), "Exact computational analyses for adaptive designs", *Adaptive Designs* (N. Flournoy & W.F. Rosenberger, ed.'s), Institute Math. Stat. Lec. Notes **25**, pp. 223–237.

[15] Hardwick, J. and Stout, Q.F. (1999), "Path induction for evaluating sequential allocation procedures", *SIAM J. Scientific Computing*, to appear.

[16] Jones, P. (1992), "Multiobjective Bayesian Bandits", *Bayesian Statistics 4: Proc.* 4th *Valencia Int'l Meeting*, pp. 689–695.

[17] Kulkarni, R. and Kulkarni, V. (1987), "Optimal Bayes procedures for selecting the better of two Bernoulli populations", *J. Statistical Planning and Inference* **15**, pp. 311–330.

[18] Palmer, C. (1993), "Selecting the best of k treatments", *Adaptive Designs* (N. Flournoy & W.F. Rosenberger, ed.'s), Institute Math. Stat. Lec. Notes **25**, pp. 110–123.

[19] Siegmund, D. (1993), "A sequential clinical trial for comparing three treatments", *Annals of Statistics* **21**, pp. 464–483.

[20] Simon, R. (1977), "Adaptive treatment assignment methods and clinical trials", *Biometrics* **33**, pp. 743–744.

[21] Wang, Y.-G. (1991), "Sequential allocation in clinical trials", *Comm. in Statistics: Theory and Methods* **20**, pp. 791–805.