ELSEVIER Computational Statistics & Data Analysis 31 (1999) 397–416

COMPUTATIONAL STATISTICS & DATA ANALYSIS

www.elsevier.com/locate/csda

A program for sequential allocation of three Bernoulli populations

Janis Hardwick [a,*], Robert Oehmke [b], Quentin F. Stout [b]

[a] *Purdue University, West Lafayette, IN 47907, USA*
[b] *University of Michigan, Ann Arbor, MI 48109, USA*

Received 1 October 1998; received in revised form 1 March 1999; accepted 1 April 1999

Abstract

A program for optimizing and analyzing sequential allocation problems involving three Bernoulli populations and a general objective function is described. Previous researchers had considered this problem computationally intractable, and there appears to be no prior exact optimizations for such problems, even for very small sample sizes. This paper contains a description of the program, along with the techniques used to scale it to large sample sizes. The program currently handles problems of size 200 or more by using a modest parallel computer, and problems of size 100 on a workstation. As an illustration, 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 Bayesian and frequentist metrics. Extensions enabling the program to solve a variety of related problems are discussed.

Keywords: Sequential sampling; Multi-arm bandit; Parallel computing; Dynamic programming; Adaptive allocation; 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 of the experiment. Here, the focus is on 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 this is a design problem and

* Corresponding author.

0167-9473/99/$ - see front matter
PII: S 0167-9473(99)00039-0

that the expressions *sampling from a population* and *allocating an experimental unit* refer to the same action.

Bandit problems, in which each population has associated with it a random reward, form an important class of sequential sampling problems. The goal of a bandit experiment is to sample from the available populations in such a way as to maximize the expected total reward at the termination of the experiment. The term "one-arm bandit" is a reference to a slot-machine with a single lever 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 the number of available machines is α, 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 Berry and Fristedt (1985) 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 them. Further, most practitioners are aware that examining data during an experiment can introduce bias 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, Armitage (1985), for example, comments "the computation involved is prohibitive except for trivially small horizons". Wang (1991) reiterates that "In theory the optimal strategies can always be found by dynamic programming but the computation required is prohibitive". 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, the three-armed problem will be used to illustrate the progress that has been made in generating solutions to these demanding problems. An algorithm

has been implemented for the optimization and analysis of sequential allocation problems involving three Bernoulli populations. For *optimization*, there is some objective function V, and 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. Note that such estimators may involve nonlinear functions of the observations on the different arms, whereas bandit problems involve simpler linear combinations. Optimization, as opposed to evaluation, requires a Bayesian framework.

By *analysis* we mean that the expected behavior or operating characteristics of an arbitrary adaptive allocation procedure can be evaluated. For example, it may be of interest 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, an investigator 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. Either a Bayesian or frequentist point of view can be taken in the analysis phase. 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.

In the rest of this section, previous work is reviewed and the problem and model are specified. In Section 2, the general computational approach to the 3-arm problem is described, and in Section 3 space requirements are discussed. In Section 4, parallelization is considered with an emphasis on load balancing and scalability issues. An example in which the algorithms have been applied to a multi-criteria problem is presented in Section 5. Finally, in Section 6, extensions to the basic 3-arm problem are described and future work is discussed.

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, 1989 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 were being 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. Bernoulli bandits are also used here, so to simplify the discussion, the term *bandit problem* is used to mean *Bernoulli bandit problem*.

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, the problem requires a dynamic programming solution. For 2-armed bandits, a typical optimization was carried out by Jones (1992), who solved a problem of size $n = 25$, and noted the difficulties of solving larger problems. Kulkarni and Kulkarni (1987) 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 the 1991 work described in Hardwick and Stout (1993), the largest 2-armed bandit problem to be solved was described in the paper of Berry and Eick (1995), which reports on work done around 1987. Utilizing a Cray 2 supercomputer, the authors were able to handle a sample size of $n = 200$. In Hardwick and Stout (1999), improved algorithms for 2-armed bandits were proposed. This made it feasible for workstations to solve problems involving sample sizes greater than $n = 400$. Not only were solutions to these larger bandit problems obtained, but extensive evaluations of the optimal procedures' operating characteristics were also carried out. More recently, the authors began work on generating solutions to 3-arm bandit problems. A move to parallel architectures was necessary to go beyond non-trivial sample sizes. A preliminary description of the initial work, which corresponds to Algorithm 3 in Section 4 applied to the bandit objective function, appears in Hardwick et al. (1997).

It is useful to recall that a variety of approaches have been utilized for the more general problem of sequential sampling from three or more arms. For example, Siegmund (1993) and Coad (1995) have applied repeated significance testing to the case where several arms are available and the outcome variables have normal distributions. Betensky (1996,1997), also working with normal outcomes, has used various hypothesis testing approaches to tackle the 3-arm problem. In particular, Betensky (1997) examines a 3-arm problem with censored outcomes. Bather and Coad (1992) addressed the multi-armed problem with Bernoulli outcomes, and in this work, the authors 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, where the primary goal of such designs is to select either the best of several arms or a best subset of arms. See Buringer et al. (1980) and Gupta and Liang (1989) for examples of this approach.

Whether it is a testing problem or a selection problem, most multi-armed designs incorporate a mechanism for removing obviously poor arms during the experiment. When this occurs in the early stages of an experiment, the complexity of the procedure being used is often significantly reduced. While no such option is explicit in the procedures presented here, the optimal sampling procedures generated effectively remove poor arms, in the sense that they do not continue to sample from them. Further, the proposed analysis routines can easily evaluate arbitrary 3-arm sequential procedures, including those that eliminate arms.

As noted, 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. However, it is difficult

to ascertain the quality of such rules without having access to the optimal solution. Thus, even if exact optimizations are not required in a given setting, the determination of optimal solutions is important to the evaluation process.

With regard to other research in which exactly optimal solutions to 3-arm problems have been obtained, we are familiar only with the work of Palmer (1993). Palmer 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. While this is an optimal solution to the problem stated by the author, it is not equivalent 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.

1.2. Model used

The focus is on 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, one selects a population and observes the response. At stage m, let (s_i, f_i) represent the number of successes and failures, respectively, observed on 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*. A Bayesian approach is utilized, in which the population parameters, P_i, have a prior distribution which is the product of α independent distributions. In our examples these distributions are beta, but in general one could utilize arbitrary distributions.

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

It is assumed that there is an objective function V. For optimization the goal is to minimize $\boldsymbol{E}(V)$, while for analysis the goal is merely to determine $\boldsymbol{E}(V)$. It is assumed that V can be computed by knowing merely 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, "generalized O-notation" from computer science will be used, in which O and o have the same meanings as in statistical use; and in which one says 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 assume $\alpha \ll n$, and for most purposes α will be fixed to be 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 Algorithm 1.

Algorithm 1. Serial Algorithm for Optimal Adaptive 3-Arm Allocation

```
{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 do {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)) }
```

While Algorithm 1 shows the algorithm for optimization, with only a small change it is also an algorithm for evaluation of an arbitrary adaptive design A. If, instead of choosing the arm that gives the optimal value of V, one uses the value of V corresponding to the arm chosen by A, then the program determines the expected

value of V obtained using procedure 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 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 solve three 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, the serial efficiencies that had been exploited previously for 2-arm bandit-like problems needed to be maintained (Hardwick and Stout, 1993). The code was implemented assuming a distributed memory model. As distributed memory code can easily be adapted to run on a shared memory machine, but not necessarily vice versa, this gave the greatest flexibility. Shared memory systems are discussed further in Section 6. The parallel code is written in Fortran 77, with MPI (Message-Passing Interface) for the communication among the processors. As Fortran 77 and MPI are commonly available on parallel computers, as well as on distributed systems such as networks of workstations, the present program is quite portable. See Gropp et al. (1995) for a discussion of parallel programming using MPI.

In extending the previous 2-arm serial algorithms to a 3-arm parallel algorithm, the major new computational issues were *space reduction* (useful for both serial and parallel execution) and *load balancing* and communication minimization among the processors (for efficient parallel execution). Space, rather than time, has become the limiting factor in solving many 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 six-dimensional array is needed to store values of `V`, since the state space is six-dimensional and `V` is computed at each state. However, this array can be compressed to a five-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 Hardwick and Stout, 1993 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 allow 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 five-dimensional array is actually used. The corner occupied is only approximately $\frac{1}{5}! = \frac{1}{120}$ 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 direct 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 fifth 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, `T` was decomposed into a sum of offsets, i.e.,

$$T = T_5(m, s3) + T_4(m - s3, f3) + T_3(m - s3 - f3, s2) + T_2(m - s3 - f3 - s2, f2) + T_1(m - s3 - f3 - s2 - f2, s1).$$

The parameters to the offset at each loop level depend only on the values of the outer loops and the value of the current level. At each level the offset of that level is added to that of the previous one, and the eventual sum inside the innermost loop is `T`. In general, T_i is an ith degree polynomial. Here,

$$T_1(t,x) = x,$$

$$T_2(t,x) = [-x^2 + (2t+3)x]/2,$$

$$T_3(t,x) = [x^3 - (3t+6)x^2 + (3t^2+12t+11)x]/6,$$

$$T_4(t,x) = [-x^4 + (4t+10)x^3 - (6t^2+30t+35)x^2 + (4t^3+30t^2+70t+50)x]/24,$$

$$T_5(t,x) = [x^5 - (5t+15)x^4 + (10t^2+60t+85)x^3 - (10t^3+90t^2+255t+225)x^2 + (5t^4+60t^3+255t^2+450t+274)x]/120.$$

This mapping is such that the offset calculations range from most complex at the outermost loop to simplest at the innermost, and thus the most complex calculations are done the fewest times. This method of calculating `T` is fairly efficient, but it has the regrettable effect of making the code harder to read and maintain.

One useful aspect of this system of offsets is that the same functions are used even when the number of loop levels changes. For example, for a 2-arm problem there are only three loops within the `m` loop, and the mapping would be $T_3(m, s2) + T_2(m - s2, f2) + T_1(m - s2 - f2, s1)$.

Note that the innermost loop accesses array positions in consecutive order. This is important for maintaining high processor efficiency because most accesses will be to data in cache. The *cache* is a relatively small portion of the memory that can be accessed much faster than the computer's primary RAM. When the contents of a memory location are retrieved from main memory, the contents of adjacent memory locations are retrieved as well and stored in the cache. Thus a program which accesses memory sequentially will often be able to retrieve memory contents out of the cache instead of having to access the much slower RAM. Haphazard array access patterns, so that few references are found in cache, are a common source of performance degradation.

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 $\frac{1}{6}! = \frac{1}{720}$. This array needs only 3 bits per state (we allow for the possibility of ties, so there are seven possible outcomes), although for convenience one byte per state was used. 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 Hardwick and Stout, 1993), but this was not exploited here because the goal was to develop a general purpose algorithm suitable for arbitrary objective functions.

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* (see Hardwick and Stout, 1999). Using path induction, after a single initialization pass through the decision array, each evaluation is reduced to a computation over the final states. This algorithm 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 one to store the decision array on disk with relatively little loss of efficiency. A serial implementation of path induction is given in Algorithm 2.

Algorithm 2. Serial Implementation of Path Induction

{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 **do**
    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 **do**
            Read arms used for $\sigma = \langle$s1,f1,s2,f2,s3,f3$\rangle$
            for all arms i used for σ **do**
              path($\sigma + \mathbf{1}_i^s$)=path($\sigma + \mathbf{1}_i^s$)+ p(σ,i)·path(σ)
              path($\sigma + \mathbf{1}_i^f$)=path($\sigma + \mathbf{1}_i^f$)+ p(σ,i)·path(σ)
{evaluation phase}
for all post-analysis parameters π **do**
  $W(\pi) = \sum\{\texttt{path}(\sigma) \cdot \texttt{prob}(\sigma) \cdot W(\pi, \sigma) : \sigma$ `a terminal state`$\}$

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 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, for example, 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 the introduction of path induction, each such evaluation was carried out via backward induction (see, for example, Berry and Eick, 1995). For three 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`.

4. Parallelization

To develop an efficient parallel program from a serial one, there are numerous, sometimes competing, factors that must be taken into account. The problem must be partitioned into pieces executed by the different processors, with the goal being to subdivide the work evenly. This partitioning is called *load-balancing*, and its most important aspect is that the most heavily loaded processor should be as close to the average as possible (since the running time is determined by the last processor to finish). The partitioning also induces communication, since in general processors need to utilize values computed by other processors. This communication (which in the present program is performed by MPI) can be very time-consuming, and thus should be minimized. *Computational dependencies* determine which calculations can be performed concurrently by different processors, as opposed to those that must be done sequentially with the communication of one result needed for the calculation of the next. The communication and load-balancing are further tied together in the sense that, typically, calculation and communication alternate in rounds. If one processor takes longer to finish a calculation round, it may force others to wait for its communication before they can proceed on their next round. Thus it is important to load-balance each round, and not merely to balance the aggregate calculations. Load-balancing, communications, computational dependencies, and other aspects of efficient parallel algorithm development are discussed in Atallah (1999).

Load-balancing high-dimensional dynamic programming codes such as those in Algorithm 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, because `V` values corresponding to `m`+1 are used to determine `V` values corresponding to `m`. At the same time, the inner loops act like space variables which are amenable to parallelization because no value depends on any other. 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 P_j is start_s3(j,m)... end_s3(j,m). A very simplified pseudo-code version of the parallel code is given in Algorithm 3.

Algorithm 3. Initial Parallel Algorithm

```
{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 do {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=start_s3(j,m) to end_s3(j,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
            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}
end for m
```

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 P_j to send processor 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 Algorithm 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 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 P_{j-1}, in the second send–receive pair in Algorithm 3.

Unfortunately the s3 iterations do not represent a uniform load since the amount of work grows like $(m-s3)^4$. 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, a simple yet effective heuristic was developed, 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 Oehmke et al. (1999), along with comparisons to the optimal subdivision of the `s3` loops. This algorithm will be referred to as the *initial* parallel algorithm, and it is the one that was used for the work reported in Hardwick et al. (1997).

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 languages and tools. This communication is quite efficient, especially when compared to the message passing required at each stage of dynamic programming or backward induction. Since it is often useful to have nearly one hundred 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

While the initial parallel algorithm works moderately well, it suffers 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. Thus, for example, relatively little improvement was observed for $n = 50$ when the number of processors was increased from 8 to 16, even when the `start_s3` and `end_s3` functions were determined optimally. On the other hand, if the number of processors is fixed and the sample size increases, then the parallelization improves. This is a well-known scaling phenomenon.

In general, when there is a specific problem size of concern, researchers will refine a parallel program until adequate performance is achieved on the target machine. If the goal, however, is to solve problems far beyond current machine capabilities, then current boundaries will continually be pushed so that the largest possible problem can be solved. This is one of the reasons for migrating to parallel computers. Such machines allow investigators to wield computational power that would not be available on serial computers for many years. In the present case, the generation of optimal solutions to sequential allocation problems is in its infancy, and there is interest in expanding the frontier as far as possible. Further, for problems of this nature, with exponential growth in the number of arms, one cannot envisage a maximum size or complexity to solve that will address all problems of interest. In this

sense, developing algorithms to make the most of the largest available machines is the scenario we are trying to enable.

Algorithm 4. Scalable Parallel Algorithm

```
{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 {i.e. for all 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
```

To make a truly scalable algorithm able to efficiently use many processors to analyze large problems, it is necessary to balance the work much more evenly. Load-balancing becomes critical to reduce not only the time, but also the space requirements since they too grow rapidly with n. If one processor requires significantly more space than average, then it becomes the limiting factor. Towards this end, note first that the 1-dimensional `V` array can easily be subdivided evenly into intervals. Note also, however, that because a given interval may start and end at arbitrary values of `s3`, `f3`, `s2`, `f2`, and `s1`, such a partitioning does not correspond to simple subdivisions of the control loops. To address this, a conversion has been made to a control structure that 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 P_j may now need to exchange values with processors other than $P_{j\pm1}$. An overview of the improved algorithm is given in Algorithm 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 that is comparable to that needed in the serial algorithm. Further details can be found in Oehmke et al. (1999), 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. Aside from 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 Algorithm 4, it was applied to the design and analysis of three sequential allocation procedures involving 3 arms. The 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 sampling 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. Myopic allocation also utilizes 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*, "$E(F)$" 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 $E(F)$ for bandit allocation, and that backward induction is needed to determine the $E(F)$ for myopic allocation. For equal allocation, the $E(F)$ is simply the sum, over all arms, of $n/3$ times the prior probability of failure.

The second criterion examined is the *probability of correct selection*, "$P(\mathrm{CS})$". 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. This indicates that a search throughout the parameter space is needed to determine $P(\mathrm{CS})$. 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. $P(\mathrm{CS})$ 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 $P(\mathrm{CS})$ 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 $P(\mathrm{CS})$ criterion. For two arms, in fact, this procedure can easily be shown to be optimal. For three arms, however, equal allocation is no longer optimal, as there exist adaptive

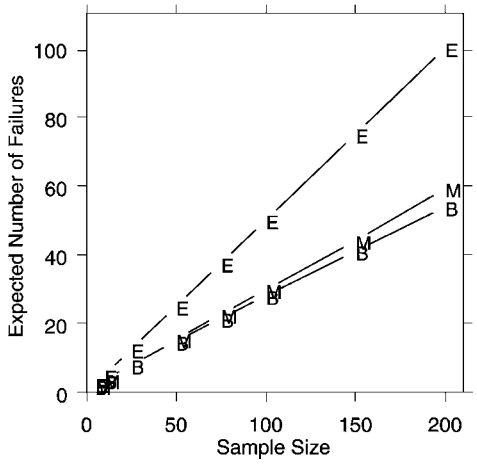

Fig. 1. Sample size vs. E(failures).

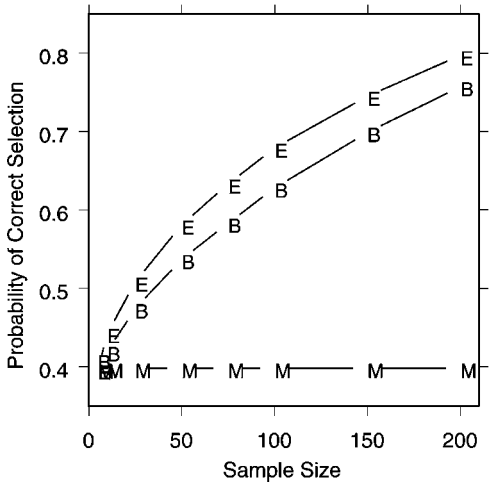

Fig. 2. Sample size vs. P(CS), $\delta = 0.1$.

procedures that are better. For the general case and for a specified n, the design that has the optimal P(CS) is not known for three 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).

In Fig. 1, $E(F)$ for each procedure is plotted as a function of the sample size. Similarly, P(CS) vs. sample size is presented in Fig. 2. In these figures, B = bandit, M = myopic and E = equal allocation. As noted, uniform priors have been used throughout. These were used mainly for convenience and because they are commonly used. Naturally, had other priors been used then the results would be somewhat different. The program can easily handle a wide range of prior distributions.

Note that the bandit allocation comes very close to achieving the $P(\mathrm{CS})$ 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 $P(\mathrm{CS})$ 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 greater than a 60% chance that the rule will never even try the superior arm. This is largely due to the prior assumption that the average success rates for the different arms are $\frac{1}{2}$. The myopic rule randomizes in the first stage and if a success is obtained, the parameter estimate for the arm selected is updated to $\frac{2}{3}$, while the other arm estimates remain at $\frac{1}{2}$. This procedure selects the next observation from the arm with the expected success rate of $\frac{2}{3}$. With the true parameter having a value ≥ 0.9, the outcome is again likely to be a success. This result inclines the rule even further in favor of the arm already sampled. There are simple ways to alter myopic allocation so that the $P(\mathrm{CS})$ significantly improves with very little increase in failures, however, a discussion of this is beyond the scope of the present paper.

6. Extensions and future work

To summarize the current standing of this work, the serial (Algorithms 1 and 2), initial parallel (Algorithm 3), and improved parallel (Algorithm 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 the example in Section 5 illustrates only a bandit objective, 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 Algorithm 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, it is 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 Oehmke et al. (1999).

One way of viewing the worth of parallel programming is to note that, even with Moore's law which predicts a doubling of serial processor speed roughly every 18 months, it will take approximately 6 years before a serial processor has the computing power of a current 16-processor parallel computer. Further, 6 yr from now a 16-processor system will still be 6 yr ahead of the serial system. Finally, even if the program is not especially efficient and is running, say, only 10 times faster than serial, instead of the hoped-for 16 times faster, this is still roughly 5 yr ahead of the serial capabilities.

As noted earlier, the program is designed for a distributed memory system, and is portable to a wide range of parallel computers, including shared memory ones. It would have been easier to develop an initial parallel version on a shared memory system, although the automatic parallelization, such systems provide, would have failed because they currently cannot analyze nested loops that depend on outer loops. However, an efficient, highly scalable shared memory code would be very similar to Algorithm 4, and would take similar amounts of time to develop and tune. This is because space compression, load-balancing, communication reduction, cache utilization, etc., remain important concerns on any parallel system.

From a statistical vantage point, we plan to extend this work 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 $P(\mathrm{CS})$ in Section 5, even problems that are well understood for two arms may be quite complex for three 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 $\langle o_1^1, o_1^2, o_1^3, o_2^1, o_2^2, o_2^3\rangle$, 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 Algorithm 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 five-dimensional dynamic programming approach. However, if censoring is not independent of the arm, then six-dimensional dynamic programming is needed. In either case, the recurrences remain near-neighbor recurrences, as in Algorithm 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 does not 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 are 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 Algorithm 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 (see Hardwick et al., 1998). 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.

Acknowledgements

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. We are grateful for the comments of three referees who reviewed this paper.

References

Armitage, P., 1985. The search for optimality in clinical trials. Int. Statist. Rev. 53, 15–24.

Atallah, M. (Ed.), 1999. Algorithms and Theory of Computation Handbook. CRC Press, Boca Raton, 1999.

Bather, J.A., Coad, D.S., 1992. Sequential procedures for comparing several medical treatments. Sequential Anal. 11, 339–376.

Berry, D.A., Eick, S.G., 1995. Adaptive assignment versus balanced randomization in clinical trials — a decision-analysis. Statist. Med. 14, 231–246.

Berry, D.A., Fristedt, B., 1985. Bandit Problems: Sequential Allocation of Experiments. Chapman & Hall, London.

Betensky, R.A., 1996. An O'Brien-Fleming sequential trial for comparing three treatments. Ann. Statist. 24, 1765–1791.

Betensky, R.A., 1997. Sequential analysis of censored survival data from three treatment groups. Biometrics 53, 807–822.

Buringer, H., Martin, H., Schriever, K., 1980. Nonparametric Sequential Selection Procedures. Birkhauser, Basal.

Coad, D.S., 1995. Sequential allocation rules for multi-armed clinical trials. J. Statist. Comput. Simul. 52, 239–251.

Gittins, J.C., 1989. Multi-Armed Bandit Allocation Indices. Wiley, New York.

Gropp, W., Lusk, E., Skjellum, A., 1995. Using MPI: Portable Parallel Programming with the Message Passing Interface. MIT Press, Cambridge, MA.

Gupta, S.S., Liang, T., 1989. Selecting the best binomial population: parametric empirical Bayes approach. J. Statist. Plann. Inference 23, 21–31.

Hardwick, J., Oehmke, R., Stout, Q.F., 1997. A parallel program for 3-arm bandits. Comput. Sci. Statist. 29, 390–395.

Hardwick, J., Oehmke, R., Stout, Q.F., 1998. Adaptive allocation in the presence of missing outcomes. Comput. Sci. Statist. 30, 219–223.

Hardwick, J., Stout, Q.F., 1993. Exact computational analyses for adaptive designs. In: Flournoy, N., Rosenberger, W.F. (Eds.), Adaptive Designs. Institute Math. Stat. Lec. Notes, Vol. 25, 223–237.

Hardwick, J., Stout, Q.F., 1999. Using path induction for evaluating sequential allocation procedures. SIAM J. Sci. Comput. 21, 67–87.

Jones, P., 1992. Multiobjective Bayesian Bandits. Bayesian Statistics 4: Proceedings of the fourth Valencia Int'l Meeting, 689–695.

Kulkarni, R., Kulkarni, V., 1987. Optimal Bayes procedures for selecting the better of two Bernoulli populations. J. Statist. Plann. Inference 15, 311–330.

Oehmke, R., Hardwick, J., Stout, Q.F., 1999. Scalable parallel implementation of high-dimensional dynamic programming. In preparation.

Palmer, C., 1993. Selecting the best of k treatments, In: Flournoy, N., Rosenberger, W.F. (Eds.), Adaptive Designs. Institute Math. Stat. Lec. Notes, Vol. 25, 110–123.

Siegmund, D., 1993. A sequential clinical trial for comparing three treatments. Ann. Statist. 21, 464–483.

Wang, Y.-G., 1991. Sequential allocation in clinical trials. Commun. Statist. Theory Methods 20, 791–805.