In *Handbook on Parallel Computing and Statistics* (2006)
E. Kontoghiorghes, ed., Marcel Dekker, pp. 347–373.

Parallel Programs for Adaptive Designs

Quentin F. Stout* and Janis Hardwick*
*University of Michigan, Ann Arbor, Michigan 48109 USA

Abstract

We discuss the role of parallel computing in the design and analysis of adaptive sampling procedures, and show how some efficient parallel programs were developed to allow one to analyze useful sample sizes. Response adaptive designs are an important class of learning algorithms for a stochastic environment and apply in a large number of situations. As an illustrative example, we focus on the problem of optimally assigning patients to treatments in clinical trials. While response adaptive designs have significant ethical and cost advantages, they are rarely utilized because of the complexity of optimizing and analyzing them. Computational challenges include massive memory requirements, few calculations per memory access, and multiply-nested loops with dynamic indices. We analyze the effects of various parallelization options, showing that, while standard approaches do not necessarily work well, with effort an efficient, highly scalable program can be developed. This allows us to solve problems thousands of times more complex than those solved previously, which helps make adaptive designs practical.

1 Introduction

In situations where data are collected over time, adaptive sampling methods often lead to more efficient results than do fixed sampling techniques. When sampling or “allocating” adaptively, sampling decisions are based on accruing data. In contrast, when using fixed sampling procedures, the sample sizes taken from different populations are specified in advance and are not subject to change. Using adaptive techniques can reduce costs and time, or improve the precision of the results for a given sample size. For example, in many financial situations one tries to optimize rewards by constantly adjusting decisions as information is collected. In a pharmaceutical setting one may initially sample several different compounds to estimate their efficacy, and then concentrate a second round of testing on those compounds deemed most promising. When buying a present one may look at many options and stop when one has found something that seems sufficiently nice. In this latter case, which is an optional stopping problem, we assume some cost for looking time and a reward for appropriateness of the gift.

Fully sequential adaptive procedures, in which one adjusts after each observation, are the most efficient. For example, OECD TG 425 [21] contains guidelines for determining acute toxicity of potentially hazardous compounds. Since experiments are conducted on rats, there has been a strong motivation to develop experimental designs that expose as few animals as possible. Whereas in the past test guidelines had recommended predetermined sampling rules calling for approximately

30 rats per compound, the standard has changed, and TG 425 recommends adaptive procedures requiring far fewer, typically 8 to 12. Unfortunately, fully sequential procedures are rarely used due to difficulties related to generating and implementing good procedures as well as to complications associated with analyzing the resulting data. For example, they assume immediate responses and the ability to rapidly switch between alternatives, and they can involve designs which greatly reduce average sample size but increase the maximal sample size.

A long-term goal of this research program has been to increase access to adaptive designs by creating a collection of algorithms that optimize and analyze a variety of sequential procedures. In particular we have focused on developing serial and parallel algorithms that allow researchers greater flexibility to incorporate diverse statistical objectives and operational considerations. Some of these techniques and applications are detailed in [12, 13, 14, 23].

We differentiate between the design and the analysis of sampling procedures. The design phase might generate optimal or nearly optimal sampling procedures, while the analysis phase may be applied to an arbitrary sampling procedure and may involve a wide range of operational characteristics. The analysis itself may be either exact or approximate. In some situations, optimal procedures may not be used because they are complex or difficult to employ and explain. Still, they provide a basis of comparison to establish the efficiency of suboptimal designs. If one can show that the relative efficiency of a sampling procedure is high compared with the optimal one, then investigators may be justified in implementing a simpler and, typically, more intuitive suboptimal option. Since this collection of algorithms also allows for the optimization of strategies that are constrained to have desirable operational characteristics, the likelihood that investigators can incorporate such goals and still achieve statistical efficiency is increased.

Historically, it has been not only analytically, but also computationally, infeasible to attain exact solutions to most adaptive allocation problems. As an example, in [5] the authors argue that if, for a specific problem, the optimal sequential procedure were "practically obtainable, the interest in any other design criteria which have some justification although not optimal is reduced to pure curiosity." They immediately add, however, that obtaining optimal procedures is not practicable. Then, as an illustration of the "intrinsically complicated structure" of optimal procedures, the authors detail the first step of the optimal solution to a simple sequential design problem involving only three Bernoulli observations. During this same year, 1956, it was pointed out in [4] that problems of this nature could, in principle, be solved via dynamic programming. However, such solutions are still typically viewed to be infeasible. Thirty five years later, in [32], when addressing a variation of the problem in [5], the author reiterates that "In theory the optimal strategies can always be found by dynamic programming but the computation required is prohibitive".

This situation motivated us to work on greatly extending the range of problems that could be analyzed and optimized computationally. While some of the gains can be attributed to the ever increasing power of computers, much is due to algorithms and implementation, as will be shown. To state that a problem can be "solved via dynamic programming" is as vague as saying that one need only "do the math". Careful implementations of complex dynamic programming variations, along with new algorithmic techniques such as path induction, have been necessary to achieve the results reviewed here. While the models for which optimal solutions can be computed are often, albeit arguably, deemed to be "too simplistic", it is nevertheless the case that the insight one garners from evaluating these models is likely to lead to better heuristics that apply as well to more complex scenarios.

The remainder of this chapter is organized as follows. In Section 2 there is a discussion of the basic types of parallel computers available and some useful computing paradigms for working with adaptive designs. In Section 3 we introduce the multi-arm bandit model, which will be used to illustrate the steps used to obtain an efficient parallel algorithm. Section 4 gives a naive serial algorithm for the 3-arm bandit, and then an improved version. Sections 5 and 6 show increasingly more efficient (and complex) parallel algorithms for the same problem, and Section 7 transfers this to a different type of parallel computer. Section 8 gives some illustrative results showing how the optimal 3-arm bandit design is superior to simpler alternatives. Section 9 explains the parallelization of a related but more complex problem, namely 2-arm bandits with delayed responses. Section 10 has some concluding remarks.

2 Parallel Computing Models and Paradigms

In this section we discuss parallel computing platforms and their basic characteristics, along with some computing paradigms that are used in conjunction with adaptive designs.

2.1 Parallel Computing Platforms

Most of our parallel algorithms are developed for *distributed memory*, or *message-passing*, computers, in which data is stored with processors and all communication and access to data is via the exchange of messages between processors. Conceptually these machines are similar to a standard network of computers, where all communication is via basic read and write operations. Distributed memory machines are the most common form of parallel computer, especially since the wide-spread introduction of *clusters*. These relatively inexpensive systems consist of commodity processor boards interconnected by commodity communication systems, typically utilizing open source software.

In Section 7, the distributed memory algorithm is modified into a form written explicitly for a shared memory parallel computer. Such a computer has its memory organized so that any processor can directly access any data, without messages. In general, shared memory machines are preferable since they simplify the process of converting a serial program into a parallel one. While small shared memory machines are increasingly becoming commodities for departmental computing, larger machines require specialized, more expensive, components, and hence are far less common than distributed memory clusters.

By far the most common way to create a parallel program for distributed memory systems is to use MPI, Message Passing Interface. MPI has an extensive collection of operations for exchanging messages and collecting information. Furthermore, because it is a well-developed standard available for most platforms, it helps one develop programs that can be run on a variety of systems. MPI is available for shared memory machines as well, but for them there is an additional standard, OpenMP. This provides compiler directives for automatically parallelizing many loops, which simplifies the parallelization process.

Various timing results are presented throughout to illustrate the performance improvements achieved through various means. Note that the absolute values of the numbers are of little interest since processor performance rapidly improves over time. Relative performance, however, is a useful measure. Further, the basic techniques remain applicable no matter how fast the system is. The distributed memory results presented were obtained using MPI on an IBM SP2, where each processor was an 160 MHz POWER2 Super Chip (P2SC) processor with 1 GB of RAM and 1 GB additional virtual memory. The shared memory results were obtained on a 16 processor SGI Origin with 12 GB RAM, where each processor was a 250 MHz MIPS R10000. Throughout, all times are elapsed wall-clock time measured in seconds. Rerunning the same problem showed very little timing variation, so we merely report average time.

2.2 Farming

There is a simple form of parallelization for adaptive designs that is widely used. It is sometimes called *farming*, and the resultant algorithms are often referred to as being *embarrassingly parallel*. As an example, suppose we have an adaptive design and wish to determine certain of its operating characteristics. It may be infeasible to do this exactly, even with the aid of parallel computers, and hence it is done via simulation. One way to accomplish such simulations, especially if each is itself rather lengthy, is to have several different processors do their own collection of simulations, and then to combine them at the end. As long as one takes care to insure that the random number generators on the processors are independent, this method is quite simple and extremely efficient. There is no communication among the processors except at start-up and in the final collection of data. Thus, even a low-cost distributed memory system with slow communication channels imposes very little overhead and can achieve high efficiency.

A variation on this theme is to do parameter sweeps to tune performance. Many adaptive designs have a variety of adjustable components, such as start-up, stopping and decision rules. For example, with staged sampling, one collects k observations deterministically before any adaptation is employed. However, the most suitable value of k may be unclear, and hence a suite of evaluations, using different values of k, may be required to determine the optimal one. Here too, the parallelization is trivial, since different processors can work on different values of k. Note that this approach is applicable whether the evaluations for a single k are exact or are obtained via simulation.

When farming is possible, it is almost always the most efficient form of parallelization. That is, one may be able to parallelize the evaluation of a single simulation or exact evaluation, but it is usually more efficient to run different ones in parallel and then combine results rather than run each in parallel. This is because the parallelization of a single task typically adds communication and other overhead, and thus while a single task will be completed most quickly if it is run in parallel, the total set of tasks will be completed quickest if the tasks are run serially.

In a few cases of, say, parameter sweeps, the optimal performance is obtained by a mixture of the embarrassingly parallel and standard parallelization. This occurs when a single task, such as exact evaluation for a specific value of k, runs most efficiently on a small number of processors, rather than on a single one. Examples of this, shown in subsequent sections, can easily occur if the memory requirements exceed the memory available on a single processor. In this case, it is best to find the number of processors that runs a single evaluation at the highest efficiency. If this value is, say p_e, and there are p total processors, then the total time to complete all evaluations is optimized by running p/p_e evaluations simultaneously.

2.3 Exact Optimization

At the opposite end of the spectrum, in terms of the programming effort required, lies the problem of determining optimal adaptive sampling designs. While there are a variety of techniques needed for different problem types, we concentrate here on the problem of optimizing an objective function. Suppose the objective function $\mathcal{O}$ is defined on the terminal states of the experiment, and the goal is to maximize the expected value of $\mathcal{O}$. We assume that the sampling options available, and responses obtained, are discrete.

During the experiment, suppose we are at some state σ and can sample from populations $P_1, \ldots, P_k$. For population P_i, suppose that at state σ there are $r(i)$ possible outcomes, $o^i_0, \ldots, o^i_{r(i)-1}$, and that these occur with probability $\pi^i_0(\sigma), \ldots, \pi^i_{r(i)-1}(\sigma)$, respectively. Let $\sigma + o^i_j$ denote the state where o^i_j has been observed by sampling P_i. Let $\mathcal{E}_{opt}(\sigma)$ denote the expected value of the objective function attained by starting at state σ and sampling optimally, and let $\mathcal{E}^i_{opt}(\sigma)$ denote the expected value of the objective function attained by starting at state σ, observing P_i, and then proceeding optimally.

The important recursive relationship, sometimes called the principle of optimality, is that

$$\mathcal{E}^i_{opt}(\sigma) = \sum_{j=0}^{r(i)-1} \pi^i_j(\sigma)\, \mathcal{E}_{opt}(\sigma + o^i_j) \tag{1}$$

Since the only actions available are either to stop with value $\mathcal{O}(\sigma)$ or sample one of the populations, we thus have

$$\mathcal{E}_{opt}(\sigma) = \max \Big\{ \mathcal{O}(\sigma), \max\{\mathcal{E}^i_{opt}(\sigma) : i = 1, \ldots, k\} \Big\}$$

where the maximum is restricted to those options that are permissible at σ.

Note that not all adaptive designs are for problems with objective functions satisfying such recursive equations. For example, many mini-max objectives cannot be presented this way because they are not defined in terms of expectations with respect to a distribution on the populations, but rather a maximum or minimum over the populations. Thus they do not have the additivity property used above. As an example, in Section 8, we compute a criterion known as $\min$ P(CS), which is the minimum probability of correctly identifying the best arm at the end of the experiment. To do this, we use path induction [13], which is described in the next section.

When recurrences such as (1) do hold, then there is a very powerful technique, *dynamic programming*, for obtaining the optimal design. One starts at the terminal states, and then for each of their predecessor states, determines the population to sample that will optimize the expected value. The optimal action, and the resulting optimal expected value, are recorded for each of these states. Then the optimal actions at predecessors of these states are determined and so on until the initial state is reached.

One important limiting factor of dynamic programming is the need to determine the value and optimal action of every state that can be reached. As will be shown in Section 3, the state space can be exceeding large. This fact is one of the reasons for utilizing parallel computing, since the computational demands of dynamic programming can be more than are feasible with a single processor.

Note that in order to be able to employ dynamic programming, not only does one need for the recursive equations to hold, but one also needs the transition probabilities, $\pi_j^i(\sigma)$, at each state σ. Thus, dynamic programming requires a Bayesian statistical framework in which the π_j^i are random variables whose distributions are updated as data are observed. Technically, this means that one begins with a joint *prior* distribution, Γ, on the π_j^i and proceeds to calculate a *posterior* distribution, which is simply the prior conditioned on the outcomes observed so far. In the calculations, one then uses the posterior mean $E^\Gamma(\pi_j^i \mid \sigma)$ as the value of $\pi_j^i(\sigma)$.

2.4 Exact Evaluation

Since exact analytical evaluations of design operating characteristics are rarely accessible for complex adaptive designs, they are typically obtained computationally. These characteristics can be estimated via simulation or they can be calculated exactly. Often a serial program can be used to generate simulations, or farming can be used to exploit multiple processors. On the other hand, exact evaluations are typically far more complex and may require more sophisticated parallel algorithms.

One well-known technique is *backward induction*, in which the calculations are performed just as in dynamic programming, moving from the end of the experiment towards the beginning. However, to determine the expected value of a state, one uses the, possibly random, choice the design would make at that state, rather than considering all choices and choosing the best. Thus, backward induction can be as computationally challenging as dynamic programming. In some cases, however, it may considerably simpler, such as when it is known that most of the states can never be reached by the given design. Note that such evaluations can be carried out regardless of how the design was created. They may involve either an evaluation for a specific Bayesian distribution, or a collection of evaluations for robustness studies or a frequentist overview.

If many evaluations are needed, then it may be more efficient to use *path induction*. With path induction, there is a preliminary pass from the beginning of the experiment towards the end, and then repeated evaluations are performed on the terminal states. A detailed explanation of this approach appears in [13]. For the purposes of this work, however, the most important feature is that the calculations for the preliminary pass proceed in the opposite order of those for dynamic programming. Hence the same parallelization techniques can be applied to path induction as for dynamic programming, and similar efficiencies can be obtained. Calculations for the evaluations are typically expected values summed over terminal states, and hence these are similar to farming, in that each processor sums over its terminal states and then a global sum is computed. Thus it is quite easy for a parallel computer to perform each evaluation efficiently.

Since backward induction and path induction are so similar to dynamic programming, only the parallelizations for dynamic programming will be discussed in detail.

3 Example: Bandit Models

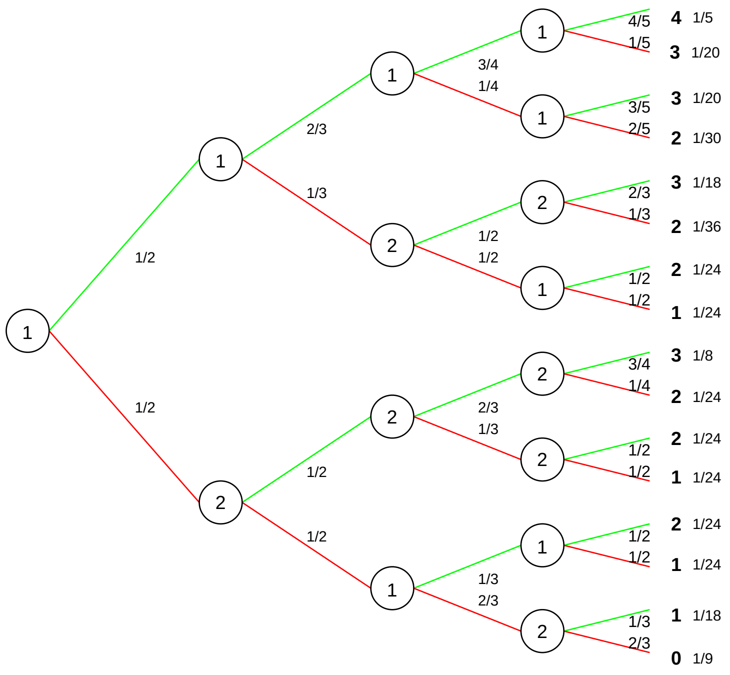
In clinical trials, there are multiple goals that must be considered when designing an experiment. One of these goals is to treat all patients as well as possible, but there are differing viewpoints as to the relevant patient population. For example, if you were a patient, you'd like to be given the treatment currently deemed the best. Physicians sometimes use this viewpoint as well. This is known as *myopic* allocation, since there is no attempt to allocate you with the hope of gaining information from your result to help treat patients in the future. A second viewpoint is that it is all the patients in the clinical trial that are important, in which case one tries to maximize the total number of successful outcomes by the end of the trial. A third viewpoint emphasizes "future" patients who will be treated as a result of the decision made at the trial's termination. Addressing the optimal treatment of this last group has long been considered the classical goal of clinical trials. Still, the need to optimize the well-being of the subjects in the trial itself, be they humans or animals, has drawn increasing attention. Ultimately, the best designs will balance the needs of trial subjects and future patients, although, unfortunately, there is no design that can optimize these goals simultaneously. One way to approach this problem is to attempt to find a design that is optimal from each viewpoint (as is tackled here), and then to develop methods that utilize heuristics from each optimal design. For discussion of the latter approach see [9].

In this section, we describe the design for allocating patients to treatment options such that, on average, the maximal number of positive outcomes is obtained for trial patients. (See Section 8 for discussion of the immediate and future patient criteria.) This objective can be modeled as a *bandit problem* [3]. Such models are important in stochastic optimization as well as in decision and learning theory. In a k-arm bandit problem one can sample from any of k independent arms (populations) at each stage of the experiment. (Here, "arm" = "treatment option".) Statistically speaking, bandit problems are usually presented within a Bayesian decision theoretic framework. Thus, associated with each arm is an initial or prior distribution on the unknown outcome or "reward" function. After sampling from an arm (e.g., allocating a patient to a treatment) one observes the outcome and updates the information to get the posterior for that arm. The goal is to *determine how best to utilize accruing information to optimize the total outcome for the experiment*.

For our example, the outcome functions are Bernoulli random variables such that, from state σ and using the notation of Section 2.3, $o_0 = 0$ represents a treatment failure that occurs with probability $1 - p_i(\sigma) = \pi_0^i(\sigma)$, and $o_1 = 1$ represents a success that occurs with probability $p_i(\sigma) = \pi_1^i(\sigma)$, $i = 1, \ldots, k$. Our goal is to maximize the number of successes in n observations.

At each stage, $m = 0 \ldots n - 1$, of an experiment of length n, an arm is selected and the response is observed. At stage m, let (s_i, f_i) represent the number of successes and failures from arm i. Then the state $(s_1, f_1, \ldots, s_k, f_k)$, is a vector of sufficient statistics.

Figure 1 illustrates a simple 2-arm bandit design with $n = 4$. The rate of success of each arm has a uniform or "flat", uninformative, prior distribution. Using a non-adaptive design, one would expect to achieve 2 successes. With the optimal adaptive design one expects 2.27 successes. The advantages of adaptation become more pronounced the longer the trial is and the more arms there are. For example, with $n = 100$ and uniform priors on each arm, non-adaptive allocation will average 50 successes no matter how many arms there are. However, the optimal 2-arm bandit will average 65 successes, and the 3-arm bandit averages 72.



Node: arm sampled; Right cols: successes achieved and prob. reaching that outcome.
Upward line: success; Downward line: failure; Line label: prob. of outcome.

Figure 1: A 2-arm bandit, with $n = 4$ and uniform priors on each arm.

Optimal designs for the bandit problem can be obtained via dynamic programming, but the number of states, and hence the time and space required to evaluate them all, have the formidable growth rate of $\Theta\left(n^{2k}/(2k)!\right)$. We concentrate on the 3-arm version, which has $\Theta(n^6)$ complexity.

Note that the state space grows exponentially in the number of arms. This "curse of dimensionality" often makes exact solutions infeasible. As a result, approximations may be used and the quality of the solution reduced.

Even when parallel computing is employed, major difficulties include:

- Time and space grow rapidly with the input size, so intensive efforts are needed to obtain a useful increase in problem size.
- The time/space ratio is low, making RAM the limiting factor.
- There are few calculations per memory access.
- The nested loops have dynamic index dependencies (see Algorithm 1).

Performance is further exacerbated by the interaction of these aspects. Table 10 illustrates, for example, the dramatic limitations imposed by space constraints and imperfect load balance caused by the loop structure.

3.1 Previous Work

The 3-arm problem had never previously been solved exactly because it was considered infeasible. As noted earlier, researchers have long indicated frustration with the far simpler *2-arm* bandit problem. In particular, in [17], the authors remark that "the space and time requirements for this computation grow at a rate proportional to n^4 making it impractical to compute the decision even for moderate values of say $n \geq 50$". Previously, the largest exact 2-arm bandit solution utilized a IBM 3090 supercomputer with 6 processors to solve n=320 [8]. Here, we solve a problem more than 300 times harder, namely the 3-arm bandit with $n = 200$. Further, because to obtain operating characteristics we must evaluate this design many times, the total work is at least 10,000 times harder than that done earlier; since, had it been feasible, researchers would have used methods available at the time rather than the path induction exploited here.

While there has been scant previous work on the parallel solution of bandit problems, in the computer science community there has been more work on the parallel solution of similar recurrences. Most of this concentrates on theoretical algorithms where the number of processors scales far faster than the input size [15, 20, 25, 26, 27], or where special purpose systems are created [18, 28]. Others [19, 30] look at dynamic programming difficulties when the subproblems are not as well understood.

4 Serial Implementation

The goal of a bandit problem with dichotomous responses is to determine, at each state, which arm should be selected so as to maximize the expected number of successes over the course of the experiment. To solve this via standard dynamic programming (Algorithm 1), first the values of each terminal state (those with n observations) are computed. 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 $m = n-1$ down to 0.

The recurrence at the heart of this dynamic programming algorithm is in the center of the loops. At state $\sigma = (s_1, f_1, s_2, f_2, s_3, f_3)$, one may sample any of the three arms. If arm i is sampled, then the resulting state will be either $\sigma + \widehat{s_i}$ or $\sigma + \widehat{f_i}$, where $\widehat{s_i}$ and $\widehat{f_i}$ denote a single additional success or failure, respectively, on arm i. Given the prior distribution on the success rate for arm i, along with the observations in σ on arm i, one can then compute the probability of these two outcomes. Let the probability of observing a success be denoted by $p_i(s_i, f_i)$. In the previous stage of the dynamic programming, the expected values of these states, assuming one proceeds optimally to the end of the experiment, have been determined. Thus the expected value of sampling from arm i and proceeding optimally, i.e., $V^i(\sigma) = \mathcal{E}^i_{opt}(\sigma)$, is given by

$$V^i(\sigma) = p_i(s_i, f_i) \cdot V(\sigma + \widehat{s_i}) + [1 - p_i(s_i, f_i)] \cdot V(\sigma + \widehat{f_i})$$

Choosing the arm that yields the highest expected value is the optimal decision at σ.

For the purposes of efficient implementation and parallelization, the specific recurrence used to combine values is less important than the indices of the values being referenced, since they determine the memory accesses and communication required. We have a stencil of dependencies, whereby the value at state σ depends only on the neighbor values at $\sigma + \widehat{s_1}$, $\sigma + \widehat{f_1}$, $\sigma + \widehat{s_2}$,

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

```
{$\widehat{s_i}$, $\widehat{f_i}$: 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 }
{$|\sigma|$: number of observations at state $\sigma$ }
{V: the value of the optimal design starting at state $\sigma$, i.e., $\mathcal{E}_{opt}(\sigma)$.
    V(**0**) is the value of the optimal design starting at the beginning.}
{pi(si,fi): prob of success on arm i, if si successes and fi failures have been observed }

**for all** states $\sigma$ with $|\sigma|$=n **do** {i.e. for all terminal states}
  V($\sigma$)=number of successes in $\sigma$

**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
            $\sigma$ = $\langle$s1,f1,s2,f2,s3,f3$\rangle$
            V($\sigma$) = max{(p1(s1,f1) $\cdot$ V($\sigma + \widehat{s_1}$) + (1-p1(s1,f1)) $\cdot$ V($\sigma + \widehat{f_1}$)) ,
                         (p2(s2,f2) $\cdot$ V($\sigma + \widehat{s_2}$) + (1-p2(s2,f2)) $\cdot$ V($\sigma + \widehat{f_2}$)) ,
                         (p3(s3,f3) $\cdot$ V($\sigma + \widehat{s_3}$) + (1-p3(s3,f3)) $\cdot$ V($\sigma + \widehat{f_3}$)) }
```

$\sigma + \widehat{f_2}$, $\sigma + \widehat{s_3}$, and $\sigma + \widehat{f_3}$, along with the priors and σ itself. With minor changes to this equation (and no change in the dependencies), the same program can also perform backward induction (see Section 2.4) to evaluate the expected number of successes for an arbitrary 3-arm design. This allows one to evaluate suboptimal designs which may be desirable for reasons such as simplicity and intuitiveness. The same stencil of dependencies can also be used to optimize and evaluate designs for 2 Bernoulli arms with randomly censored observations [23].

Note that the recurrences involve extensive memory accesses, with little computation per access. There are $\binom{n+6}{6} = \Theta(n^6)$ states, and the time and space complexities are also $\Theta(n^6)$.

4.1 Space Optimizations

Given the vast space requirements needed to solve these problems, good algorithms must incorporate a number of space reduction techniques. The first of these results from the observation that values of V for a given m depend only on the values for $m+1$, so only the states corresponding to these two stages need to be kept simultaneously. This reduces the working space to $\Theta(n^5)$, and by properly arranging the order of the calculations, the space can be further reduced to only that required for one stage's worth of states, i.e., we gain another factor of 2. This corresponds to the

n	first	collapsed	naive comp	best
10	.009	.004	.082	.004
20	.18	.1	3.2	.092
30		1.4	35	.71
40		4.1	186	3.9
50		15	689	13
60			2024	35
70			5047	86
80			11242	185
90			22756	362
100			42900	659
110				1225
120				1922
130				34961
max n	27	54	100	135
limitation	memory	memory	time	time
prog len	193	193	282	419

max n: Maximum problem solvable with 1 GB and time $\leq$ 64,400 sec. (18 hr.)

Table 1: Serial versions, time (sec.) to solve problem of size n.

collapsed column in Table 1. In this table, *max* n shows the maximum problem solvable by a 1 GB RAM machine with a time limit of 18 hours, *limitation* shows which limit was reached, and *prog len* is the size of the version in lines of source code. Note that the collapsed version allows us to solve problems substantially larger, and also results in a slight speedup.

Another significant space reduction results from the fact that, due to the constraint

$$s_3 + f_3 + s_2 + f_2 + s_1 + f_1 \leq n,$$

only a corner of the 5-dimensional V array is used (approximately 1/5! = 1/120 of the total). To take advantage of this, the 5-dimensional V array is mapped 1-1 onto a linear array $\mathtt{V}_\ell$. Unfortunately, this mapping also requires that all array references be translated from the original five indices into their position in the linear array. From a software engineering viewpoint, the best way to implement this translation is to use a function which takes as input the five indices and yields their position in the array, i.e., a mapping of the form

$$\mathtt{V(s1,s2,f2,s3,f3)} \mapsto \mathtt{V}_\ell(\mathtt{T(s1,s2,f2,s3,f3)})$$

Unfortunately, this is extremely costly as the translation function T is a complicated 5^{th} degree polynomial which must be evaluated for every array access. This version, the *naive comp* in Table 1, can solve larger problems, but is significantly slower than the *collapsed* version. For the *best* version, we broke the translation function into a series of offset functions, where each offset function corresponds to a given nested loop level., i.e.,

$$\begin{aligned} T(s_1, s_2, f_2, s_3, f_3) &= T_{s3}(m) + T_{f3}(m-s_3) + T_{s2}(m-s_3-f_3) + T_{f2}(m-s_3-f_3-s_2) \\ &\quad + T_{s1}(m-s_3-f_3-s_2-f_2) + s_1 \end{aligned}$$

An offset function only needs to be recalculated before its corresponding loop is entered, and the more expensive offset functions correspond to the outermost loops.

This method dramatically reduces the translation cost down to a usable level, but greatly increases program complexity, as is shown by the increase in *prog len*.

The simplified Algorithm 1 ignores the fact that in order to utilize the design, one needs to record the arm selected at each state. Unfortunately these values cannot be overwritten and the storage required is $\Theta(n^6)$. Fortunately, this too involves only values in one corner, allowing a reduction by a factor of 1/6! = 1/720. These values are stored on disk and do not reduce the amount of memory available for calculation. Using run-length encoding or other compression techniques would likely reduce this to $\Theta(n^5)$, but so far this has not been necessary. Note that if one only needs the value of the optimal design, but not the design itself, then this storage is not needed. Such a situation arises, for example, when the optimal design is only used to gauge the performance of simpler designs.

5 Initial Parallel Algorithm

To parallelize the recurrence, we first address load balancing. In the initial parallelization the natural step of dividing the work among the processors was taken. The outermost `m` loop behaves very much like "time" in many simulations and cannot be parallelized, so instead one parallelizes the second outermost loop, `s3`. At stage `m`, processor $\mathcal{P}_j$ is assigned the task of computing all values where `s3` is in the range `start_s3(j,m)` ... `end_s3(j,m)`.

Because the number of states corresponding to a given value of `s3` grows as `(m-s3)`^{4}, determining the range of `s3` values assigned to each processor is nontrivial. Thus, simply assigning all processors an equal number of `s3` values would result in massive load imbalance and poor scaling. We evaluated two solutions to this problem. Optimal `s3` partitioning is itself a small dynamic programming problem which takes time and space $\Theta(mp)$. However, it was easy to develop a fast $\Theta(m)$ greedy heuristic which was nearly optimal, and it is this heuristic which was used in the initial program.

5.1 Communication

The communication needed can be divided into *array redistribution* and *external neighbor acquisition*. Array redistribution occurs because, as the calculation proceeds, the number of states shrinks. To maintain load-balance, the `s3` range owned by a processor changes over time. At stage `m`, processor $\mathcal{P}_j$ needs the states with `s3` values in the range `start_s3(j,m)` ... `start_s3(j,m+1)-1` from $\mathcal{P}_{j-1}$. Redistribution includes the cost of moving the states currently on the processor to create space for these new states.

External neighbor acquisition occurs because the calculations for a state may depend on its neighbors in other processors. To calculate states with `s3=end_s3(j,m)` during stage `m`, $\mathcal{P}_j$ needs to obtain a copy of the states with `s3=end_s3(j,m)+1` from $\mathcal{P}_{j+1}$. Note that external neighbor acquisition negates round-robin or self-scheduling approaches to load-balancing the `s3` loops, as this would result in a dramatic increase in the communication requirements. This does not necessarily hold for shared memory systems, however, as can be seen from the OpenMP version

Algorithm 2 Scalable Parallel Algorithm

```
{$\mathcal{P}_j$: processor j}
{start_$\sigma$(j,m), end_$\sigma$(j,m): range of $\sigma$ values assigned to $\mathcal{P}_j$ for this m value,
 with start_$\sigma$(j+1,m)=end_$\sigma$(j,m)+1 }

{For all processors $\mathcal{P}_j$ simultaneously, do}

for $\sigma$=start_$\sigma$(j,n) to end_$\sigma$(j,n) do {i.e. for all terminal states}
  V($\sigma$)=number of failures in $\sigma$

for m=n-1 downto 0 do {compute for all states of size m}
  for $\sigma$=start_$\sigma$(j,m) to end_$\sigma$(j,m) do
    determine s1, f1, s2, f2, s3, f3 from $\sigma$
    compute V as before

  {Array redistribution}
  Send needed V values to other processors
  Receive V values from other processors

  {External data acquisition}
  Send needed V values to other processors
  Receive V values from other processors
```

in Section 7. Shared memory computers are able to utilize these approaches because their much faster communication systems reduce the latency to a tolerable level.

6 Scalable Parallel Algorithm

The initial load-balancing approach is simple to implement and debug because it makes minimal changes to the serial version. Unfortunately, it has imperfect load and working space balancing and this severely limits scalability (see Table 2) and solvable problem size (see Table 10).

For a more scalable version (Algorithm 2), instead of partitioning the states using the coarse granularity of the `s3` values, we partition them as finely as possible, i.e., by individual states. The assigned states are specified by `start_`σ and `end_`σ. However, this leads to numerous difficulties. The first is that a processor's `V` array can now start or end at arbitrary values of `s3, f3, s2, f2, s1,` and `f1`, so one can no longer use a simple set of nested loops to iterate between the start and end value. Our first attempt to solve this problem had nested if-statements within the innermost loop, where the execution rarely went deep within the nest. While logically efficient, this turned out to be quite slow because it was too complex for the compiler to optimize. A solution that the compiler was able to cope with was to use a set of nested loops with if-statements in front of each loop so that it starts and stops appropriately. This solution was almost as fast as the original serial nested loops.

Another difficulty was that the offset calculations are not uniformly distributed along the range of the V array, and this leads to a noticeable load imbalance. Storing the results of the offset equations in arrays significantly decreases the cost of each offset calculation and reduces the load imbalance to a more acceptable level. However, there is still some slight load imbalance that could be addressed by including the cost of these array lookups in the load balancing.

6.1 Communication

The move to perfect division of the V array also caused complications in the communication portion of the program. The main complication was that data needed for either external or redistribution aspects was no longer necessarily located on adjacent processors. This resulted in a considerable increase in the complexity of the communication portions of the program.

Our initial version of the communication functions used a natural strategy when space is a concern: each processor sent the data it needed to send, shifted its remaining internal data, and then received the data sent to it. Blocking sends were used to insure that there was space to receive the messages. Unfortunately, this serialized the communication, because the only processor initially ready to receive was the one holding the end of the array, i.e., the only processor which does not redistribute to any other processor. The next processor able to receive was the second from the end, because it sent only to the end processor, and so on. The performance of this version was unacceptable. The next version removed the interaction and performed adequately but synchronization costs became more of a problem. To remove these, we switched to non-blocking communication wherever possible. This made communication fairly efficient, although there may still be room for some slight additional improvement.

Unfortunately, non-blocking communication requires additional buffers to accommodate incomplete sends and receives. In general there is a serious conflict between extensive user space requirements and minimizing communication delays. The communication buffers needed to overlap communication and calculation, and to accommodate non-blocking operations, can be large.

6.2 Scalable Timing Results

Table 2 shows the efficiency, $e(p)$, of the initial and scalable parallel versions as the number of processors p increases. Table 3 shows the effect on timing and scaling of each of the major changes detailed in Section 6, contrasting 1 processor and 8 processor versions, where $t(p)$ is the time. Note that the improvements reduced the serial time, and increased the parallel efficiency relative to the reduced serial time.

Table 4 contains the percentage of the total running time taken by different parts of the scalable program as the number of processors increases. *Calc* is the percentage of time taken by the dynamic programming calculations, *file* is the cost of writing the decisions to disk, and *misc* is the part of the time not attributed elsewhere. Under *array redist*, we show the cost of shifting data among the processors to maintain load-balance, where *comm* is the cost of calculating the redistribution and communicating the data between the processors, and *shift* is the cost of moving the data on the processor. Below *external comm* is the cost of getting neighbor states from other processors, including the cost of determining which processor has the data, where to put it on the

p	efficiency $e(p)$	
	initial	scalable
1	1.00	1.00
2	.96	.96
4	.93	.94
8	.81	.91
16	.64	.86
32	.48	.81

Table 2: Scaling results, $n = 100$.

version	$t(1)$	$t(8)$	$e(8)$
first scalable	1044	178	.734
improved loops	775	143	.678
offsets in array	766	134	.715
scalable comm	762	106	.903
non-blocking comm	760	104	.913

Table 3: Stepwise improvements in scalable version, $n = 100$, 1 and 8 processors.

				array redist		external
p	calc	file	misc	comm	shift	comm
1	98	1.9	0.1	0.0	0.0	0.0
2	94	1.6	0.9	1.9	1.2	0.4
4	88	1.6	0.1	4.5	2.0	3.8
8	84	1.4	0.2	6.5	2.0	5.9
16	73	1.2	0.7	11.0	2.1	12.0
32	57	1.1	0.0	16.1	1.7	24.1

Table 4: Percentage distribution of time within scalable version, $n = 100$.

current processor, and the cost of communicating the data.

Table 5 presents the running times of the scalable program for $n = 200$ for 16 and 32 processors. Note that the speedup is more than a factor of two. This occurred because on 16 processors the program must make extensive use of disk-based virtual memory. A similar effect can be seen in Table 1 as n increases from 120 to 130. This illustrates an often overlooked advantage of parallel computers, a bonus increase in speed simply because dividing a problem among more processors allows it to run in RAM instead of in virtual memory. However, this can be successful only if the parallelization load-balances the memory and computation requirements.

7 Shared Memory Implementations

To measure the performance of the 3-arm bandit code on a shared memory machine we implemented four separate versions.

The first version, which we call MPI, uses the shared memory implementation of the MPI libraries. Aside from a few changes due to differences in the versions of Fortran on the two machines, this version is identical to the scalable version of the code previously described.

The next version, OpenMP, uses OpenMP directives to implement a shared memory version of the code. This version is very similar to that in Algorithm 1, except for the addition of a

p	$t(p)$
16	10463
32	1965

Table 5: Timing results, $n = 200$, scalable version.

second copy of the `V` array. This second copy is necessary because, while using a shared memory implementation the same `V` array is shared among all the processors, which may be acting on different sections of it at arbitrary times. This means there is no longer a guarantee that every calculation that uses a state will have read the state's value before it is overwritten. Thus, we need to have a second array to hold the current stage's inputs while the current stage's outputs are being stored. After a stage is completed its output array is copied into the input array for the next stage.

To convert the code, OpenMP parallel-do directives were used around the outermost loop, `s3`, of the dynamic programming setup, and the `s3` loop in the dynamic programming. Both of these loops use OpenMP dynamic scheduling, where each processor grabs a user defined chunk size number of iterations, performs them, and then, when completed, grabs another set. This process continues until all the iterations of the loop have been completed.

To compute the chunk size for each stage, we first determine the average amount of work per processor at that stage. The chunk size is then the maximum number of initial iterations whose combined work is no greater than the average work. Note that this will not be the number of iterations divided by the number of processors since the work per iteration varies dramatically. This will create many chunks with diminishing amounts of work which will be taken by under-loaded processors as they complete their tasks, helping to insure approximately even load balance. Such a dynamic scheduling approach is not useful for distributed memory systems because of the increase in complexity that would result from tracking the location of the states and synchronizing access to them, and the cost of moving so much state information among processors.

The third version of shared memory code, Auto, was generated by using the SGI Fortran auto-parallelizer on the serial version of our code. Unfortunately, due to the dependencies inside the `V` array described above, the auto-parallelizer was only able to parallelize the innermost, `s1`, loop of the dynamic programming setup.

The final version of shared memory code, Auto+Copy, again used the auto-parallelizer, but this time on the double `V` array code described above for OpenMP. The reduction in dependencies allowed it to do slightly better. It parallelized the innermost, `s1`, loops of both the setup and the main body of the dynamic programming.

Table 6 shows the results of our measurements on these four versions. As can be seen, the hand parallelized versions perform far better than those done automatically. In fact, Auto has almost no discernible increase in speed as the number of processors increases. Auto+Copy does slightly better, but is still far inferior to the others. The winner clearly is OpenMP, which was to be expected as it has far less overhead than MPI. Note, however, that OpenMP's scalability will degrade as the number of processors increases because it cannot allocate less than one `s3` loop per processor. (Because we had only 16 nodes on our SGI Origin, we can not provide numbers for more processors). Implementing a fully scalable code using OpenMP would be difficult, and in the

	MPI		OpenMP		Auto		Auto+Copy	
p	$t(p)$	$e(p)$	$t(p)$	$e(p)$	$t(p)$	$e(p)$	$t(p)$	$e(p)$
1	439	1.00	406	1.00	471	1.00	454	1.00
2	290	.76	209	.97	473	.49	419	.54
4	155	.70	113	.90	465	.25	404	.28
8	90	.61	72	.70	473	.13	403	.14
16	73	.38	59	.43	470	.06	397	.07

Table 6: Efficiency of shared memory implementations, n=100

end would probably result in something quite similar to the MPI version.

8 Illustrative Results for 3-Arm Bandit

To illustrate the use of the parallel algorithm in Algorithm 2, it was applied to the problem of comparing three sequential allocation procedures involving 3 arms. We continue with the example of designing a clinical trial to address the ethical obligation to optimize patient treatment. In Section 3, three interpretations of treating patients optimally were offered. We examine one design for each interpretation and then look to see which of these designs seems to address all three interpretations the best. Computationally, the intent is to show that the parallel program provides heretofore unattainable exact evaluations of these procedures and their operating characteristics for practical sample sizes. The procedures are:

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

Myopic A fully sequential design that chooses, at each state, the arm that has the highest probability of producing a success. For the current patient, this is the desirable "personal physician" approach.

Equal Allocation (EA) A commonly used fixed sampling approach, in which each arm receives $n/3$ pulls. This is the classical allocation procedure that is expected to perform well with respect to choosing the best treatment to apply to future patients once the trial has terminated.

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 is the Bayesian criterion, *expected number of failures*, "E^{Γ}(F)". For this example, the prior distribution, Γ, on the treatment means was the product of independent uniform, Beta(1,1), distributions. Since we use the same Γ throughout the example, we drop the superscript to simplify notation. Recall that E(F) is the criterion minimized by the bandit procedure. Myopic allocation, on the other hand, assumes that the next observation is the last one, and as such it calls for allocation to the treatment that presently looks the best. For clinical trials with small sample sizes, this goal is virtually the same as trying to minimize E(F) among the n trial patients. Note that while dynamic programming is needed to determine E(F) for bandit allocation, backward induction is used to determine the E(F) for myopic allocation. For equal allocation, E(F) is simply the sum of $n/3$ times the prior probabilities of failure for each arm. In the uniform case, this sum is simply $n/2$.

To address the third interpretation, which is to optimize future patient well-being, we focus on correctly identifying the best treatment arm at the end of the trial. The decision rule is to select the arm with the highest observed rate of successes, with the intent to treat all future patients with the selected therapy. In case of ties, the winner is selected randomly, as is standard. We wish to do this with high probability, so we examine the *probability of correct selection,* P(CS)=P(CS $\mid p_1, p_2, p_3$), where $(p_1, p_2, p_3) \in \Omega = [0, 1]^3$.

There exists no allocation procedure that maximizes P(CS) for all combinations of $(p_1, p_2, p_3) \in \Omega$. While one can carry out a pointwise comparison of two designs, assessing P(CS) over Ω for each, in general it is more tractable to utilize a summary measure to assess overall performance. As with the E(F) criterion, one could work with a Bayesian version of P(CS), and integrate with respect to a prior distribution on the treatment success rates. Optimizing this measure can be done via dynamic programming. A different approach is needed, however, if a frequentist measure is desired. First, let $(p_{(1)}, p_{(2)}, p_{(3)})$ be the order statistics for (p_1, p_2, p_3). In other words, $p_{(1)}$ is the smallest success rate, $p_{(2)}$ the second smallest, and $p_{(3)}$ the largest. Fix a $\delta > 0$, and say that a selection of arm i is *correct to within* δ, denoted CS$_\delta$, if $p_i > p_{(3)} - \delta$. (In the examples, $\delta = 0.1$.) Let Ψ be the class of all allocation designs of length n (notation for length suppressed). For $\psi \in \Psi$ define

$$\mathrm{P}_\delta(\psi) = \min_{(p_1,p_2,p_3)\in\Omega} \mathrm{P}(\mathrm{CS}_\delta \mid \psi; p_1, p_2, p_3)$$

Then, a popular optimization goal is to locate $\psi^* \in \Psi$ such that

$$\mathrm{P}_\delta(\psi^*) = \max_{\psi\in\Psi} \mathrm{P}_\delta(\psi)$$

Unfortunately, ψ^* is unknown when there are 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 $\mathrm{P}_\delta(\psi)$. However, when $k = 2$, the optimal procedure is to allocate equally to each arm. Thus, while for 3 or more arms there exist adaptive procedures that are better than equal allocation on this measure, EA has the potential to be a very good suboptimal procedure.

For an arbitrary allocation algorithm, it is not known which values of (p_1, p_2, p_3) yield the minimum over Ω, and it is not possible to determine this exactly through backward induction. This indicates that a search throughout the parameter space is needed to determine $\mathrm{P}_\delta(\psi)$. $\mathrm{P}_\delta(\psi)$ is an example of a criterion for which an allocation design needs to be evaluated multiple times. Because of these multiple evaluations, path induction was used to search for P$_\delta$ for the bandit and myopic designs. For 2 arms it can be shown that the minimum always occurs when $p_{(1)} = p_{(2)} - \delta$, reducing the dimension of the relevant search space. Here the search was over arm probabilities such that $p_{(1)} = p_{(2)} = p_{(3)} - \delta$. While for 3 or more arms there are contrived designs where P$_\delta$ is not attained in this region, for the designs considered here it seems to be a reasonable assumption.

In Figure 2, E(F) for each procedure is plotted as a function of the sample size. Similarly, P$_\delta$ versus sample size is presented in Figure 3. As noted, uniform priors have been used throughout.

All arms have uniform priors.

B = Bandit; E = Equal; M = Myopic

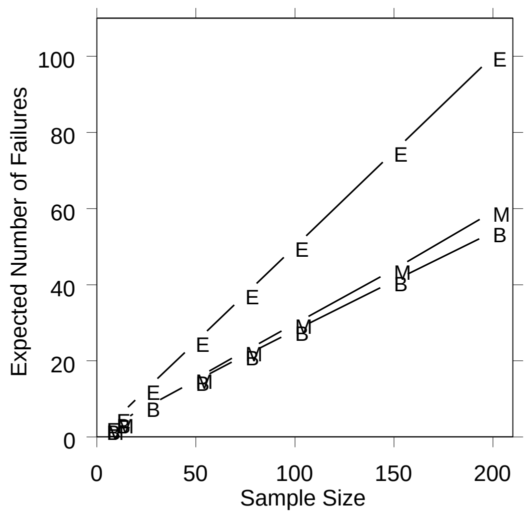


Figure 2: Sample Size vs. E(Failures)

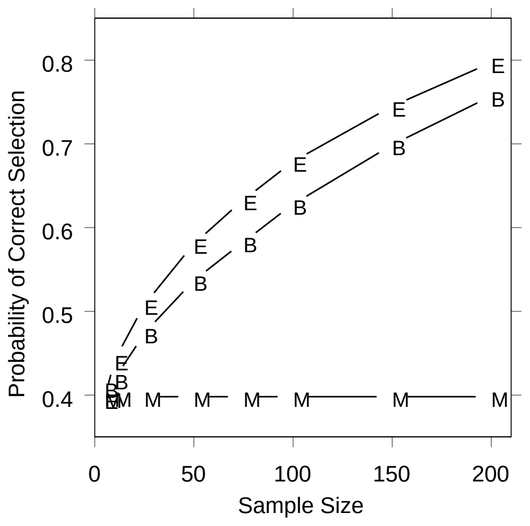


Figure 3: Sample Size vs. $\mathrm{P}_\delta, \delta = 0.1$

These were used mainly because they are a common basis for comparison across the literature. 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 high P_δ 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. Thus, we find that the bandit design seems to be the preferred design overall since it performs optimally on E(F) and also attains high values of P_δ.

For the indifference region of $\delta = 0.1$, the minimum P(CS$_\delta$) 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 trial 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 $1/2$. The myopic rule randomizes in the first stage and if a success is obtained, the parameter estimate for the arm selected is updated to $2/3$, while the other arm estimates remain at $1/2$. This procedure selects the next observation from the arm with the expected success rate of $2/3$. 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 P_δ significantly improves with very little increase in failures; however, a discussion of this is beyond the scope of this work.

9 Delayed Response Problem

An interesting dilemma that arises with adaptive designs is that information may not accrue at the rate of allocation. In this case, new questions that arise are (a) how to optimize an experimental design knowing that responses will be delayed and (b) how to model the response delay. *Delayed responses* are a significant practical problem in clinical trials, and are often cited as a difficulty when trying to apply adaptive designs [1, 29]. We have applied our scalable parallelization approach to a version of this problem in which there are 2 Bernoulli arms. Moreover, like the 3-arm problem, no nontrivial delayed response problem has been fully optimized previously, either analytically nor computationally. Again, determining the optimal design has been seen as being intractable, although some special cases have been analyzed. These include 2-stage designs where the first stage is equal allocation [6], and designs where one arm has a known success rate and the problem is to decide when to stop trying the unknown arm [7, 31].

There are many different models of the delay, appropriate for varying circumstances. Here we assume that the response times for each arm are exponentially distributed, and that patients arrive according to a Poisson process. We call the optimal design for this problem the *delayed 2-armed bandit*, D2AB. In this setting, the natural states are of the form $(s_1, f_1, u_1, s_2, f_2, u_2)$, where u_i is the number of patients assigned to treatment i but whose outcomes are unknown.

As before, we have the condition that $s_1+f_1+u_1+s_2+f_2+u_2 \leq n$, which allows compression, and all nonnegative values of s_1, f_1, s_2, f_2, s_3, f_3 satisfying this constraint are valid, just as they were for the 3-arm bandit. However, a critical difference is that the recurrence for $V(\sigma)$ depends upon $V(\sigma+\widehat{u_1})$, $V(\sigma+\widehat{s_1}-\widehat{u_1})$, $V(\sigma+\widehat{f_1}-\widehat{u_1})$, $V(\sigma+\widehat{u_2})$, $V(\sigma+\widehat{s_2}-\widehat{u_2})$, and $V(\sigma+\widehat{f_2}-\widehat{u_2})$. That is, either a patient is assigned a treatment and the outcome is initially unknown, or we have just observed the outcome of a treatment. See [11] for the detailed form of the recurrence and its derivation.

While the recurrences for the delayed response model again have a stencil of neighbor dependencies, they are much more complicated. To go through the calculations systematically, one needs the appropriate notion of "stage", corresponding to m in the 3-arm program. In general, the stage of a state σ should be the maximum path length to the state from the initial state **0**. In the 3-arm problem, all paths to σ from **0** took the same number of steps, which was the sum of the entries. Here again all paths have the same length, but it is $2(s_1+f_1+s_2+f_2)+u_1+u_2$, i.e., the components do not contribute uniformly.

Because all the paths from σ from **0** are the same length, states at stage k (i.e., at distance k) depend only on states at stage $k+1$, which allows one to store only 2 stages at a time. Further, as in the 3-arm problem, by carefully analyzing the dependencies and going through the loops in the correct order, this can be reduced down to 1 stage. However, there are now $2n$ stages for the outermost loop, as opposed to the n used previously. This has the negative effect of doubling the number of rounds of communication, which significantly reduces the parallel efficiency. It does have a positive effect, however, of slightly reducing the memory requirements since the same number of states are spread over more stages. The nonuniform roles of the indices make the array compression calculations somewhat more complex, and make it harder to determine the indices of the states depended on.

An additional complication comes from the fact that for the 3-arm problem, any combination

				array redist		external
p	$e(p)$	calc	misc	comm	shift	comm
1	1.00	95.8	0.0	0.0	4.2	0.0
2	.93	89.5	0.0	3.7	3.8	3.0
4	.79	75.7	0.0	12.4	3.6	8.3
8	.67	61.9	0.1	18.4	2.8	16.8
16	.41	41.5	0.2	28.2	2.0	28.1
32	.27	25.8	0.2	31.8	1.2	41.0

Table 7: Analysis of delay program on new system, n=100.

of nonnegative entries having a sum of m was a valid state at stage $m \leq n$. Now, however, there can be a valid stage $m \leq 2n$, and a combination of nonnegative entries having that weighted sum, but the combination does not correspond to a state. For example, if $n = 100$, then $(0, 0, 75, 0, 0, 75)$ is not a valid state, even though it is at stage 150. The reason is that it violates the constraint that $s_1 + f_1 + u_1 + s_2 + f_2 + u_2 \leq n$. Previously this constraint was automatically satisfied, but this is no longer true. This situation complicates the compressed indexing and access processes. Details can be found in [22].

Table 7 contains the timing and scaling analysis of the program, which incorporates all of the features of the most scalable 3-arm program. This was run on a newer version of the computer system where policies had been adjusted to improve disk usage but which had the unintended effect of reducing scalability. Hence we would expect the performance to be degraded somewhat, but the drop in efficiency is rather significant, caused by the complex indexing and extra rounds of communication. Perhaps further tuning would have improved this, but it was sufficient for our purposes. This is an important aspect of parallel computing, in that improving parallel performance can be a never-ending process, and hence one needs to assess the tradeoffs between programmer effort and time versus computer time.

9.1 Randomized Play-the-Winner

One popular ad hoc sampling rule is known as the randomized play the winner (RPW) rule, which first appeared in [33]. The RPW is an urn model containing "initial" balls that represent the treatment options. Patients are assigned to arms according to the type of ball drawn at random from the urn. Sampling is with replacement, and balls are added to the urn according to the last patient's response.

An advantage of urn models like RPW is the natural way in which delayed observations can be incorporated into the allocation process. When a delayed response eventually comes in, balls of the appropriate type are added to the urn. Since sampling is with replacement, any delay pattern can be accommodated. We call this design the *delayed RPW rule* (DRPW). A DRPW strategy, in which responses occur with a fixed delay, is mentioned in [16]. In [2] the authors consider a slightly altered version of this rule for a related best selection problem. However, only asymptotic results have been obtained for these cases. These results are consistent with ours when the delay is not large compared to the arrival rate, but they do not correctly predict behavior when the delay is comparable to the sample size times the arrival rate (see Figure 4).

9.2 Sample Results

We carried out exact analyses of the exponential delay model for both the D2AB and DRPW. Here we present results for $n = 100$. For the DRPW the urn is initialized with one ball for each treatment. This particular initial urn may be thought of as having roughly the effect of the uniform priors used in the bandit design. If a success is observed on treatment i then another ball of type i is added to the urn, while if a failure is observed then another ball of type $3 - i$ is added, $i = 1, 2$.

For comparative purposes, we look at base and best case scenarios. The best fixed-in-advance allocation procedure is the base case, i.e., the optimal solution when no responses will be available until after all n patients have been allocated. To maximize successes one should allocate all patients to the treatment with the higher expected success rate. We denote the expected number of successes in the base fixed case by $\mathrm{E_{bf}[S]}$. Here, we consider only uniform priors on the treatment success rates p_1 and p_2, in which case any fixed allocation rule works equally well, yielding an expected return of $\mathrm{E_{bf}[S]} = n/2$.

The best possible case arises when all responses are observed immediately (full information). In this situation, DRPW is simply the regular RPW and the D2AB is the regular 2-armed bandit. Recall that the regular 2-armed bandit optimizes the problem of allocating to maximize total successes. Letting $\mathrm{E_{opt}[S]}$ represent expected successes in the best case, $\mathrm{E_{opt}[S]}$ = 64.9 for this example. Using the difference $\mathrm{E_{opt}[S]} - \mathrm{E_{bf}[S]}$ as a scale for improvement, one can think of the values on this scale, (0, 14.9), as representing the "extra" successes over the best fixed allocation of 100 observations. For an allocation rule ψ define

$$\mathrm{R}_\psi = \frac{\mathrm{E}_\psi[\mathrm{S}] - \mathrm{E_{bf}[S]}}{\mathrm{E_{opt}[S]} - \mathrm{E_{bf}[S]}}$$

to be the *relative improvement* over the base case. While R_ψ also depends on n, the prior parameters, and the response and arrival rates, these are omitted from the notation.

Note that, for fixed arrival and delay rates, $\mathrm{R_{D2AB}} \to 1$ as $n \to \infty$. However, this is not true for $\mathrm{R_{DRPW}}$, since asymptotically the urn contains a nonzero fraction of balls corresponding to the inferior arm. If the arm probabilities are p_1, p_2, let $q_{(i)} = 1 - p_{(i)}$, $i = 1, 2$ (using the order statistic notation introduced in Section 8). Then $\mathrm{R_{DRPW}}(p_1, p_2) \to (q_{(1)} - q_{(2)})/(q_{(1)} + q_{(2)})$. For uniform priors, $\mathrm{R_{DRPW}} \to 0.545$. However, this asymptotic behavior gives little information about the values for practical sample sizes, and exact solutions for fixed values of n are not known. Hence their performance must be determined computationally.

Tables 8 and 9 contain the expected successes for the D2AB and the DRPW rules, respectively. Patient response rates, λ_1 and λ_2, vary over a grid of values between 10^{-5} and 10^1, and the patient arrival rate is fixed at 1. Note that, for both rules, when $\lambda_1 = \lambda_2 = 10^{-5}$, E[S] $\approx$ 50. When $\lambda_1 = \lambda_2 = 10$, the delayed bandit rule gives E[S]=64.9 as one would expect. Note that in the best case scenario for the DRPW, E[S] = 57.9, which gives an R of 0.53. With the RPW, we can expect to gain only 7.9 successes as compared to the 14.9 for the optimal bandit.

Moving away from the extreme points, consider the case when λ_1, λ_2 and λ_s are all the same order of magnitude. The D2AB rule is virtually unaffected, with an R value of 0.99. This is

λ_1	λ_2						
$\downarrow$	10^{-5}	10^{-4}	10^{-3}	10^{-2}	10^{-1}	10^{0}	10^{1}
10^{-5}	50.1						
10^{-4}	51.2	51.2					
10^{-3}	55.4	55.4	55.8				
10^{-2}	59.3	59.4	59.9	61.5			
10^{-1}	60.9	61.0	61.6	63.1	64.1		
10^{0}	61.3	61.3	61.9	63.5	64.5	64.8	
10^{1}	61.3	61.3	62.0	63.5	64.6	64.8	64.9

Table 8: Bandit: E[S] as (λ_1, λ_2) vary, $n = 100$, $\lambda_s = 1$, uniform priors

λ_1	λ_2						
$\downarrow$	10^{-5}	10^{-4}	10^{-3}	10^{-2}	10^{-1}	10^{0}	10^{1}
10^{-5}	50.0						
10^{-4}	50.2	50.4					
10^{-3}	51.6	51.7	52.6				
10^{-2}	54.8	54.8	54.9	55.7			
10^{-1}	56.5	56.5	56.5	56.7	57.3		
10^{0}	56.9	56.9	56.9	57.1	57.6	57.8	
10^{1}	57.0	57.0	57.0	57.2	57.6	57.8	57.9

Table 9: RPW: E[S] as (λ_1, λ_2) vary, $n = 100$, $\lambda_s = 1$, uniform priors

---

true because, on average, there is only one allocated patient whose outcome hasn't been observed throughout the trial. For the DRPW, $\mathrm{R}_{\mathrm{DRPW}}$ = 0.52, which is only slightly smaller than the non-delayed case. Both rules seem quite robust to mild to moderate delays in adaptation. It is only when *both* response rates are at least three orders of magnitude below the arrival rate that results begin to degrade seriously. When $\lambda_1 = \lambda_2 = 10^{-3}$, for example, $\mathrm{R}_{\mathrm{D2AB}}$ is only 0.40, and $\mathrm{R}_{\mathrm{DRPW}}$ is a dismal 0.17. It is also interesting to note that even when the response rate is only 1/100th the arrival rate, the D2AB does better than the RPW with immediate responses. Figure 4 illustrates the expected successes for DRPW and D2AB when the response rates are both one but the arrival rate varies between 10^{-5} and 10^{5}.

When we consider scenarios in which only one treatment arm supplies information to the system, we see an interesting result. For example, using uniform priors, when $\lambda_1 = \lambda_s = 1$ but $\lambda_2 = 10^{-5}$, the relative improvement is 0.76 for the D2AB and 0.47 for the DRPW. This is an intriguing result for the DRPW since its R-value is 89% of the best possible RPW value. Still, one clearly prefers the D2AB since there is only a 24% loss over the optimal solution while excluding half the information.

One way to view this problem independently from the allocation rules is to examine the expected number of allocated but unobserved patients when a new patient allocation decision must

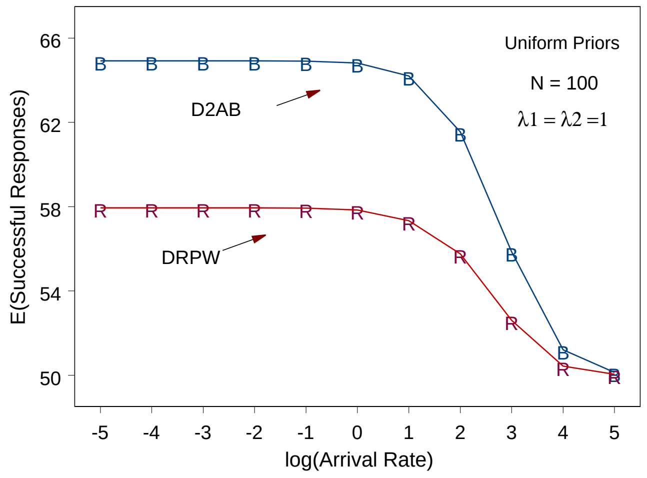


Figure 4: Expected successes for D2AB and DRPW, $\lambda_1 = \lambda_2 = 1$

be made. As noted, when the response delay rate is 1, at any point in time one expects only a single observation to be delayed, and the impact on performance is minimal. When $\lambda_1 = \lambda_2 = 0.1$, once approximately 20 patients have been allocated there is a consistent lag of about 10 patients. Connecting this value to the results in Tables 8 and 9, one finds that a loss of roughly 10% of the total information at the time of allocation of the last patient (and a significantly higher loss rate for earlier decisions), corresponds to a loss of only about 5% in terms of the improvement available from D2AB, and about 8% from DRPW.

When the response rate is about 100 times slower that the arrival rate, asymptotically there will be approximately 100 unobserved patients at any point in time. Fortunately, for a sample size of 100, one is quite far from this asymptotic behavior, and approximately 37% of the responses have been observed by the time the last allocation decision must be made. This allows the D2AB to achieve 77% of the relative improvement possible, while the DRPW rule attains only 38%. Note that this is an example where asymptotic results would be quite misleading, and thus a computational approach is required to determine the true behavior.

While for space reasons this work has only analyzed problems in which both treatments have uniform priors, similar results hold for more general priors.

10 Conclusions

There is considerable interest in using adaptive designs in various experiments because they can save lives (human or animal), time, cost, or other resources. For example, for a representative delayed response problem with $n = 100$, uniform priors, and response delay rates 10 times patient arrival rates, simple equal allocation averages 50 successes. The most commonly suggested adaptive technique, randomized play the winner (RPW), achieves only a 14.7% improvement, while the newly obtained optimal solution (D2AB) achieves a 28.4% improvement (see Figure 4). In fact, the optimal solution is nearly as good as the optimal solution for the case where there are no

n	uncompressed	initial	scalable
100	100	1	1
200	∞	21	16
300	∞	∞	173

Max problem solvable: uncompressed: 105; initial: 231; scalable: ∞.

Table 10: Min. processors (p) needed to solve problem of size n, using 1 GB per processor.

delays. Note that this is also the first exact evaluation of RPW in this setting, accomplished via backward induction. Its improvement over equal allocation, as well as its degradation relative to the results obtained by the optimal design, were not known. The former could have been estimated via simulation, while the latter could not have been.

Note that a Bayesian approach was needed for dynamic programming to create the optimal designs. However, analysis phases, such as the evaluation of RPW or the myopic rule in Section 8, can be done on any design, whether it is ad hoc, Bayesian, or frequentist. The analyses may evaluate a mix of Bayesian or frequentist criteria, independent of the design. This point is pursued further in [12].

However, overall the complexity of adaptive designs has proven to be a major hurdle impeding their use. Our goal is to reduce computational concerns to the point where they are not a key issue in the selection of appropriate designs. This chapter has concentrated on the parallel computational aspects of this work, while other papers analyze the statistical and application impact of new serial algorithms [12].

Unfortunately, the recurrences involved have attributes that make it difficult to achieve high performance and scalability. Memory requirements tend to be the limiting factor, and trying to ameliorate this causes overhead and a significant increase in program complexity. As noted in Section 6, increases in program complexity can cause severe performance problems when the compiler is unable to optimize the inner-most loops, and hence one must select alternatives with the compiler's limitations in mind. Space constraints, and low calculation to communication ratios, also complicate the ability to reduce the effects of communication latencies and overhead. However, by working diligently, it is possible to achieve significant speedups and scalable parallelizations, although this comes at a cost of increased program length and more complex program maintenance.

In Table 10, the effects of memory limitations on the 3-arm problem, using 1 GB per processor, are illustrated. *Uncompressed* refers to a parallel program using load-balancing as in the initial parallel version, but without compressing to a 1-dimensional array. Note how the scalable version needs fewer processors to solve large problems, and that it can solve arbitrarily large problems, while the other versions cannot go beyond a fixed problem size no matter how many processors are available. This is due to the imperfect load balancing in the earlier versions which were unable to allocate less than a single `s3` loop per processor.

Besides being able to compare alternative parallelizations, we can also compare to the work of others. Using only 16 processors of an IBM SP2 we solved the 3-arm, n=200 problem. This is approximately 500,000 times harder than the problem called "impractical" in [17], and more than 300 times harder than that solved in [8] on a parallel IBM 3090 of approximately the same computational power. Further, the more than 100 evaluations used for determining P_δ would have taken these authors a 100 fold increase in time, while by using path induction it only roughly tripled the time required to find the design.

Similar parallelization steps can be used to solve problems involving 4 or more arms, arms with more than two outcomes, designs with staged allocation, and so forth. However, since the computational requirements of these problems grow more rapidly than those of the problems considered here, the largest problems solvable with the same resources will be smaller. Note that the parallelization process described applies much more broadly than adaptive designs for clinical and preclinical trials, although this in itself is an important application. The bandit model is widely used in areas such as operations research, artificial intelligence, economics and game theory. Further, our work generally applies to neighbor recurrences using stencils. This common class of recurrences includes many dynamic programming problems such as the generalized string matching used in some data mining and bioinformatics applications.

Despite some successes, it is important to realize the limitations of parallel computing. Parallel computing can only do a little to overcome the curse of dimensionality that plagues many uses of dynamic programming (and relatives such as backward induction and path induction) for adaptive designs. When computational time increases as $\Theta(n^6)$, as is true for determining the optimal design for the 3-arm bandit and 2-arm bandit with delayed response problems, then doubling the sample size results in a 64-fold increase in computational time. Thus to double the size of the largest problem solvable on a serial computer, yet solve it in the same amount of time, would require 64 processors with perfect efficiency, or even more processors with more realistic efficiency. One may have access to such a computer and program, and the doubled problem size may be what is needed to solve a specific problem. However, if another doubling is needed then it is unlikely the researcher will have access to suitable parallel resources and hence other methods will have to be employed.

Finally, merely throwing a parallel computer at a problem is unlikely to be of much help. For farming-like applications this is a relatively simple process and likely to attain the desired improvements, but for many other problems the process is far more complicated. As was shown, extensive work was needed to achieve useful problem sizes and reasonable efficiency. Often it is easiest to utilize shared memory systems, but typically only modest performance will be achieved without significant work. One important aspect of the parallelization process that should be kept in mind is that to make the most of the programming effort, one should emphasize the use of software standards. MPI is the dominant message-passing system, and is widely and freely available. Similarly, OpenMP is the dominant parallelization method for shared-memory machines. By using these, porting code to new, typically more powerful, platforms is greatly simplified. Since significant effort may have been put into the parallelization process, one would like to be able to use the resulting program for an extended period of time.

Acknowledgments

The parallel programming was done by Robert Oehmke, and further details of algorithm improvements and parallelization aspects can be found in his thesis [22]. This research was partially supported by NSF grant DMS-0072910. Parallel computing support was provided by the University of Michigan's Center for Advanced Computing. Most of the material presented here first appeared in [10, 11, 24].

References

[1] P Armitage, The search for optimality in clinical trials. Int Statist Rev 53:1–13, 1985.

[2] U Bandyopadhyay, A Biwas. Delayed response in randomized play-the-winner rule: a decision theoretic outlook. Calcutta Statist Assoc Bul 46:69–88, 1996.

[3] DA Berry, B Fristedt. Bandit Problems: Sequential Allocation of Experiments. Chapman and Hall, 1985.

[4] R Bellman. A problem in the sequential design of experiments. Sankhya A 16:221–229, 1956.

[5] R Bradt, S Karlin. On the design and comparison of certain dichotomous experiments Ann Math Statist 27:390–409, 1956.

[6] H Douke. On sequential design based on Markov chains for selecting one of two treatments in clinical trials with delayed observations. J Japanese Soc Comput Statist 7:89–103, 1994.

[7] S Eick. The two-armed bandit with delayed responses. Ann Statist 16:254–264, 1988.

[8] J Hardwick. Computational problems associated with minimizing the risk in a simple clinical trial. In: Contemporary Mathematics: Statistical Multiple Integration, ed.'s N Flournoy & R Tsutakawa, American Math Assoc 115:239–257, 1989.

[9] J Hardwick. A modified bandit as an approach to ethical allocation in clinical trials. In: Adaptive Designs, ed.'s N Flournoy & W Rosenberger, IMS Lecture Notes – Monograph Series 25:65–87, 1995.

[10] J Hardwick, R Oehmke, QF Stout. A program for sequential allocation of three Bernoulli populations. Comp Stat and Data Analysis 31:397–416, 1999.

[11] J Hardwick, R Oehmke, QF Stout. Optimal adaptive designs for delayed response models: exponential case. In: MODA6: Model Oriented Data Analysis, A Atkinson, P Hackl, W Müller, eds, Physica Verlag, 2001, pp. 127–134.

[12] J Hardwick, QF Stout. Flexible algorithms for creating and analyzing adaptive sampling procedures. In: New Developments and Applications in Experimental Design. IMS Lec Notes–Mono Series 34:91–105, 1998.

[13] J Hardwick, QF Stout. Using path induction to evaluate sequential allocation procedures. SIAM J Scientific Computing 21:67–87, 1999.

[14] J Hardwick, QF Stout. Optimal few-stage designs. J Statist Plan and Inf 104:121–145, 2001.

[15] OH Ibarra, H Wang, T Jiang. On efficient parallel algorithms for solving set recurrence equations. J Algorithms 14:244–257, 1993.

[16] A Ivanova, W Rosenberger. A comparison of urn designs for randomized clinical trials of $k > 2$ treatments. J Biopharm Statist 10:93–107, 2000.

[17] R Kulkarni, V. Kulkarni. Optimal Bayes procedures for selecting the better of two Bernoulli populations. J Stat Plan and Inf 15:311–330, 1987.

[18] A Lew, A Halverson Jr. Dynamic programming, decision tables, and the Hawaii parallel computer. Computers and Mathematics with Applications 27:121–127, 1993.

[19] G. Lewandowski, A Condon, E Bach. Asynchronous analysis of parallel dynamic programming algorithms. IEEE Trans Parallel and Distributed Systems 7:425–438, 1996.

[20] B Lokuta, M Tchuente. Dynamic programming on two dimensional systolic arrays. Inf Proc Letters 29:97–104, 1988.

[21] Test Guideline 425: Acute Oral Toxicity — Up-and-Down Procedure, Organization for Economic Cooperation and Development (OECD), www.epa.gov/oppfead1/harmonization/docs/E425guideline.pdf, 2001.

[22] R Oehmke. High-Performance Dynamic Array Structures on Parallel Computers. PhD dissertation, University of Michigan, Ann Arbor, MI, 2003.

[23] R Oehmke, J Hardwick, QF Stout. Adaptive allocation in the presence of censoring. Computing Science and Statistics 30:219–223, 1998.

[24] R Oehmke, J Hardwick, QF Stout. Scalable algorithms for adaptive statistical designs. Scientific Programming 8:183–193, 2000.

[25] S Ranka, S Sahni. String editing on a SIMD hypercube multicomputer. J Parallel and Distributed Computing 9:411–418, 1990.

[26] W Rytter. On efficient parallel computations for some dynamic programming problems. Theoretical Computer Science 59:297–307, 1988.

[27] D Tang. An efficient parallel dynamic programming algorithm. Computers and Mathematics with Applications 30:65–74, 1995.

[28] R Sastry, N Ranganathan. A systolic array for approximate string matching. Proc IEEE Int'l Conf on Computer Design, pp 402–405

[29] R Simon. Adaptive treatment assignment methods and clinical trials. Biometrics 33:743–744, 1977.

[30] SA Strate, RL Wainwright, E Deaton, KM George, H Bergel, G Hedrick. Load balancing techniques for dynamic programming algorithms on hypercube multicomputers. Applied Computing: States of the Art and Practice, pp 562–569, 1993.

[31] X Wang. A bandit process with delayed responses. Stat and Prob Letters 48:303–307, 2000.

[32] Y-G Wang. Sequential allocation in clinical trials. Comm in Statist: Theory and Meth 20:791–805, 1991.

[33] LJ Wei, S Durham. The randomized play the winner rule in medical trials, J Amer Stat Assoc 73:840–843, 1978.