# The Effectiveness of the Programmed Backoff Method in the Presence of Background Traffic

Norman Matloff

Department of Computer Science
University of California at Davis
Davis, CA 95616
USA

**Abstract** ***In many parallel processing applications, several nodes will complete a task at approximately the same time. If the application is run on an Ethernet, the near-simultaneity of the task completion times implies that when the tasks attempt to communicate with some central task manager, they will bump into each other. This in turn can cause a major slowdown in communication, as the Ethernet hardware generates unnecessarily long backoff times. A solution to this problem, called programmed backoff, has been proposed earlier and shown to work well on Ethernets which are dedicated to the particular application. In the present work, the effectiveness of programmed backoff is investigated in settings in which many unrelated applications are sharing the Ethernet.***

*Keywords:* Ethernet, programmed backoff

## 1 Introduction

In many applications task times (including communication delays) have small degrees of variability [1]. For example, the mean run time for a Heapsort of r items is O(r log r), while the standard deviation is only O($\sqrt{r}$) [4]; the larger the problem, the smaller the standard deviation is relative to the mean. This can cause a slowdown on an Ethernet-based network of workstations (NOW) used for the parallel processing of such applications, for the following reasons.

Suppose for example, we are using a message-passing paradigm in a parallel root-finding program [2]. Here, a function is known to have a single root somewhere in a given interval, which the program finds (to the desired level of accuracy) in a parallel iterative procedure.[1] In any given iteration, the current interval to be searched is divided into m subintervals, where m is the total number of machines. Each "worker" node inspects its assigned subinterval, and then reports to a "manager" node whether the given function has a sign change in that subinterval. Only one of these subintervals will experience such a change, and it will then become the new interval. The manager will broadcast the values of the endpoints of the new interval to the workers, so that they can divide it into new subintervals, and so on.

Function evaluation times will probably have very little variability. Thus the workers will likely finish an iteration at approximately the same time. One of the nodes here will finish the iteration first, and most of the others may finish during the time the first node is sending its report to the manager. Ethernet uses a *1-persistent* CSMA protocol, meaning that if during a transmission by one node one or more other nodes produce frames to send, the latter attempt to send as soon as the current transmission ends. If more than one node is involved, the nodes collide, their Ethernet cards generate a random backoff time, and then try sending again. These hardware-generated random back-

[1] We assume here that the evaluation of the function is lengthy enough to make a parallel search worthwhile. For instance, the function may itself be evaluated through a time-consuming numerical solution of a differential equation.

offs will slow down the application, because in this context the random backoffs produced by the Ethernet hardware are typically much longer than need be.

In [3], an approach to solving this problem was proposed, called *programmed backoff*. Suppose n nodes are currently processing tasks, with the task at node k completing at time $T_k$. At that time, the software running at node k will produce its own backoff, delaying $k\delta$ time before sending its rendezvous frame to the manager node.

Thus for the root-finding application, for instance, pseudocode at worker node i would be as follows:

```
do {
   receive subinterval from
      manager node
   check for sign change in
      this subinterval
   wait for i * delta time
   report whether sign change
      was found
} until root is determined to
     desired accuracy level
```

The goal is that by having the software produce a small, deterministic backoff we can avoid unnecessarily long backoff times produced by the Ethernet hardware. Under the programmed backoff procedure, collisions are still possible. At that point, the Ethernet hardware will take over anyway. But hopefully this will be a relatively rare event.

The method of programmed backoff has been studied via theoretical analysis and simulation [6], and in a real but highly controlled environment [3]. The results of those studies have been quite encouraging. However, all of these investigations involved settings in which the Ethernet is dedicated to the particular parallel processing application. One of the major attractions of performing parallel processing on a NOW is cost: Instead of purchasing a costly machine designed specifically for parallel processing, such as a shared-memory multiprocessor system, one can take advantage of general-purpose machines that already are in use for daily needs such as word processing, e-mail and so on. A division in a company, or a department in a university, will already have a NOW used for these mundane purposes, and we may wish to run a parallel processing application on whichever machines happen to be currently idle.

This has the potential for changing the picture entirely. The idea of programmed backoff arose from the need to deal with hardware backoffs stemming from nearly-simultaneous transmissions from the parallel processing application. Programmed backoff enables these transmissions to miss each other. But now if we take into account other traffic on the Ethernet, will this dilute the effectiveness of programmed backoff?

The purpose of the present work is to investigate this question. Happily, the answer turns out to be that programmed backoff is just as effective in the presence of background traffic as it is on a dedicated network.

# 2 Investigation

## 2.1 Methods

Let $U_k$ denote the actual time at which attempted transmission begins for node k, i.e.

$$U_k = T_k + k\delta$$

To model the background traffic, let $B_1, ..., B_b$ represent the times at which attempted transmissions begin from background nodes. We take as our criterion the expected time $\eta$ until all n nodes of our parallel application have successfully transmitted a message. The values of b in the simulation were taken to be 0, n/4, n/2 and 3n/4.

As in [6], in the simulations the task time distribution was taken to be the uniform density U(1-c,1+c), for c = 0.1 and 0.8. Thus mean task time was 1.0. (In [6] an exponential distribution was also investigated. Some simulations with this distribution were performed in the present work as well, with similar results, but are not presented here.)

Let $\tau$ denote the time to transmit a task rendezvous frame. This often will be much smaller than task times. In a message-passing system, for example, in applications in which one uses blocking send() and receive() functions, a large $\tau$ represents too much overhead, and thus one should

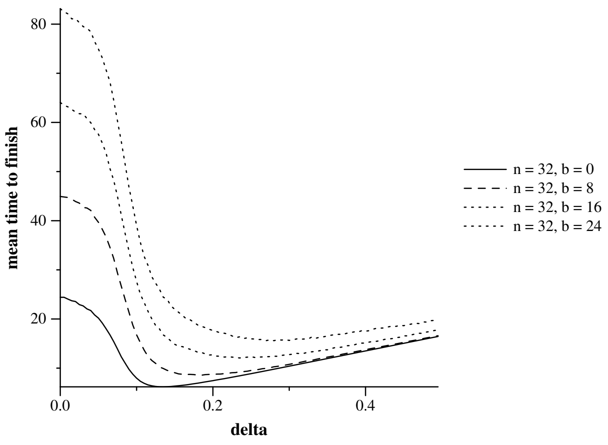


Figure 1: Figure 1, c = 0.1, n = 32

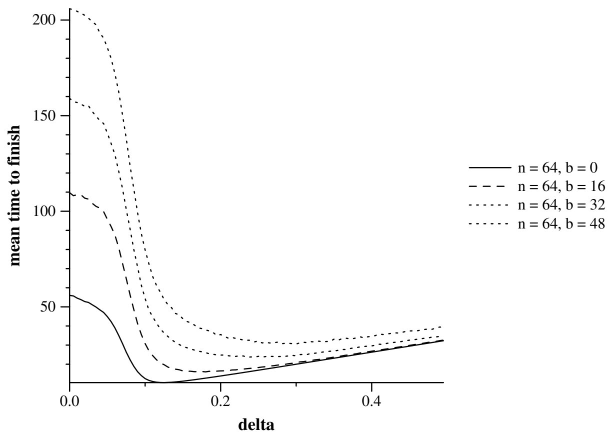


Figure 2: Figure 2, c = 0.1, n = 64

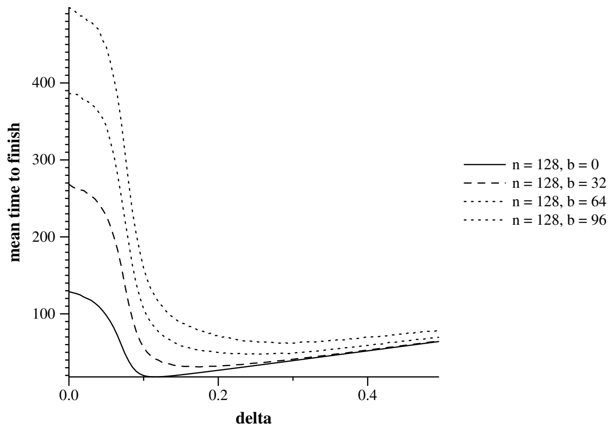


Figure 3: Figure 3, c = 0.1, n = 128

adjust the application granularity to be somewhat coarser. (This may not be the case with nonblocking calls, since one might be able to hide the call latency.) Also, in many applications a frame may contain very little data (such as a 0-1 variable in the root-finding example, indicating whether this node's subinterval produces a sign change for the function). In all the simulations presented here, $\tau$ was taken to be 0.1.

Ethernet hardware backoff was modeled according to the usual binary exponential scheme [5]. It should be noted that we have not modeled the effects of timeshare at each node.

## 2.2 Simulation Results

We begin with the smaller value of c, 0.1, presented in Figures 1 through 3.

Here we have the near-simultaneity in task time completion which formed the fundamental motivation for our work. Several very interesting results arise here:

- Programmed backoff does work in the presence of background traffic, and in fact even better. For example, for n = 32, programmed backoff yielded a 294% improvement in the case of no background traffic, already excellent, but with a background traffic level of b = 8 the improvement was even better, 425%. Of course, absolute performance is worse in the presence of background traffic than in a dedicated network, but if one needs to run on a mixed-use network then we see that programmed backoff yields a dramatic improvement over merely letting the Ethernet hardware generate backoffs.
- The higher the level of background traffic, the larger the value of the optimal $\delta$.
- All of the curves for a fixed n and various values of b appear to have a common straight-line asymptote. This is because large values of $\delta$ spread the node transmissions apart so much that they do not collide with either each other or the background traffic, making the expected time until the last finish time a linear function of $\delta$.

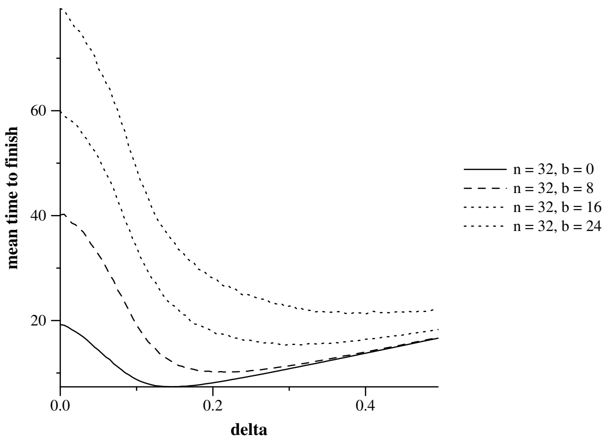


Figure 4: Figure 4, c = 0.8, n = 32

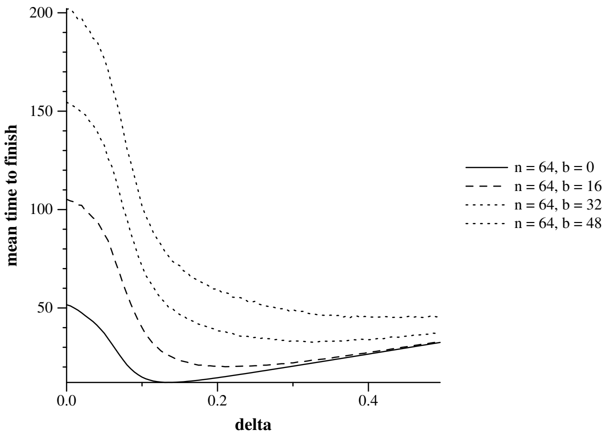


Figure 5: Figure 5, c = 0.8, n = 64

The case c = 0.8 is shown in Figures 4 through 6.

Here there is much more variation in task times, and accordingly the speedups in this case are somewhat more moderate, consistent with what was found in the no-background-traffic setting studied in [6]. For example, for n = 32 programmed backoff yielded a 161% improvement in the case of no background traffic, and with a background traffic level of b = 8 the improvement was 296%, compared to 294% and 425% we saw earlier for c = 0.1 However, note that programmed backoff's relative advantage in the case of background traffic is actually greater in the case of more-variable task times (c = 0.8), 296/161 = 1.84 versus 425/294 = 1.46 for c = 0.1.

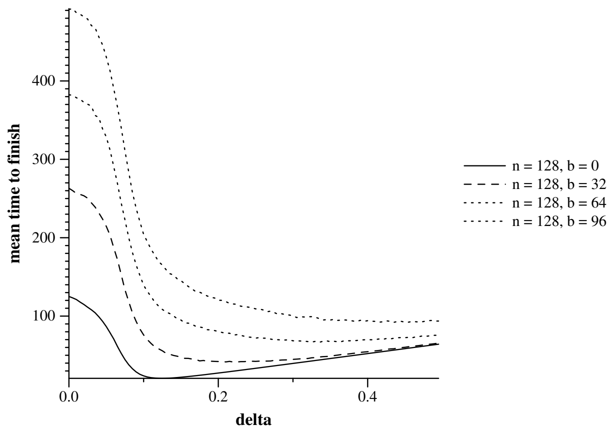


Figure 6: Figure 6, c = 0.8, n = 128

# 3 Discussion and Conclusions

We have found that the performance enhancement possible from programmed backoff for dedicated Ethernets carries over to the case of mixed-use Ethernets, which have background traffic which compete with the parallel application for access to the network. In fact, the potential improvement attainable from programmed backoff appears to be even higher in the case of mixed-use networks.

The author recently learned of certain work in synchronizing locks for buses or multistage interconnection networks for shared-memory multiprocessors which is to some degree analogous to programmed backoff on Ethernets. In order to reduce bus or interconnect contention, processors executing a spin lock were originally proposed to wait exponential backoff times between successive tries on the lock. An alternative, termed *proportional backoff*, involves waiting a programmer-controlled amount of time, proportional to the estimated time before the task currently owning the lock releases it. This is not quite the same as programmed backoff; for example, proportional backoff does not propose having the wait time depend on the processor's identification number. Nevertheless, the results in the work here suggest that proportional backoff techniques might be profitably employed in a "background traffic" setting too. Some multiprocessor systems are dedicated to running a single parallel application at a time, while others simultaneously run a mix of applications. The results we have seen above suggest that proportional backoff may work well in a mixed-use setting too.

## References

[1] Vikram S. Adve and Mary K. Vernon, "The Influence of Random Delays on Parallel Execution Times," *Proceedings of the 1993 ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems*, May 1993, pp. 61-73.

[2] S.G. Akl. *"The Design and Analysis of Parallel Algorithms"*, Prentice Hall, Inc, 1989.

[3] Gregory Davies and Norman Matloff. "Network-Specific Performance Enhancements for PVM," *Proceedings of the Fourth IEEE International Symposium on High-Performance Distributed Computing*, August 1995, pp. 205-210.

[4] G. Gonnet. *Handbook of Algorithms and Data Structures*, Addison-Wesley, 1984.

[5] Gilbert Held. *Ethernet Networks* (second edition), John Wiley, 1996.

[6] Norman Matloff. *Network-Based Parallel Computing*, Dhabaleswar Panda and Craig Stunkel, eds., Lecture Notes in Computer Science 1362, Springer-Verlag, pp. 110-117.