

Parallel BBC Decoding With Little Interprocess Communication

Leemon C. Baird III

William L. Bahn

Academy Center for Cyberspace Research
United States Air Force Academy

USAFA-TR-2009-ACCR-01

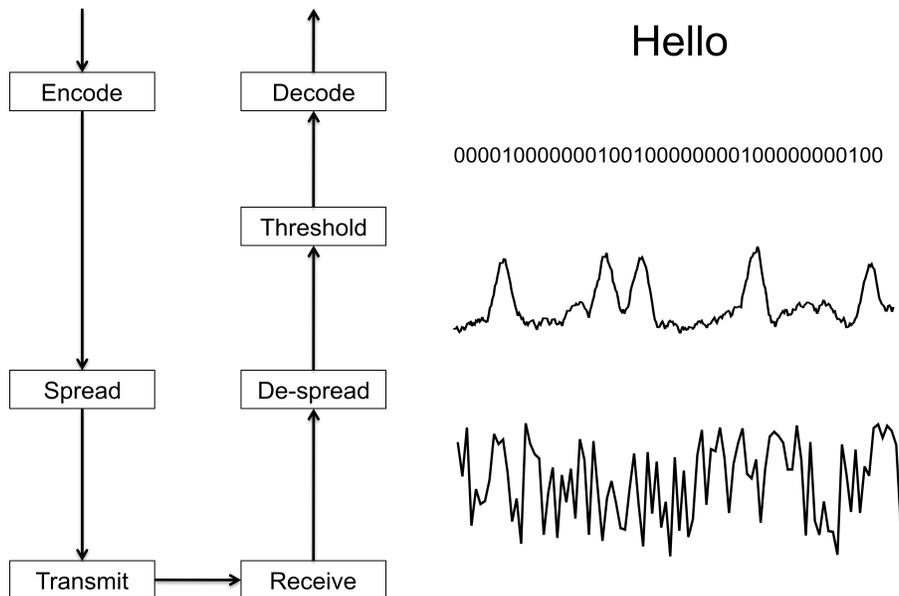
November, 2009

Abstract

Algorithms are proposed for decoding BBC codewords in parallel, as would be used in a BBC-based radio that is continuously decoding. Interprocess communication is minimized, with a simple star topology of interconnections. Randomness is injected at three different places to both improve performance and to resist various attacks.¹

Background

A wireless, jam-resistant, communication system based on BBC (Baird, Bahn, Collins) coding [1] could be organized in this way:



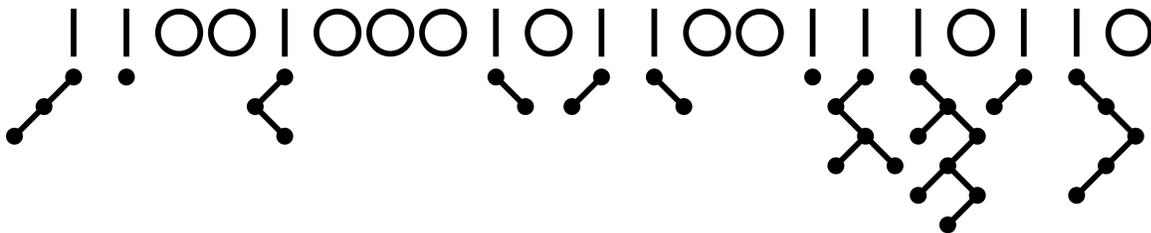
An original message (“Hello”) is BBC encoded to give a codeword that is mostly 0 bits with a few 1 bits. A spreading operation converts this codeword into a waveform that spreads each 1 bit

¹ This work was sponsored in part by the Air Force Information Operations Center (AFIOC), Lackland AFB, TX, and was performed at the Academy Center for Cyberspace Research (ACCR) at the United States Air Force Academy.

in both time and frequency. The waveform is transmitted through a wireless channel, and received at the other end. The receiver can then de-spread the signal, apply a threshold to the result, and recover the codeword. Finally, the codeword is decoded using the BBC algorithm.

This paper examines how the last step, decoding, can be done in parallel on a parallel processor, especially an ASIC designed for this specific task.

Note that a BBC jam resistance system will be constantly receiving signals, so the packets to be decoded will overlap in time. Furthermore, we will assume that the threshold on the correlator is set so that every packet has exactly 1/3 of its bits set to 1, and 2/3 set to 0. The following diagram shows how the decoding might look when there are no actual messages being sent, only random noise.



In this case, we assume there are many more bits off the right edge of the figure. When an actual message is sent, the packet will always start with a 1 bit. Therefore, any 1 bit in this sequence is potentially the start of a packet, which must be decoded, but each zero bit cannot be the start of a packet, and therefore does not need to be decoded. Note that every 1 bit in the figure has a decoding tree below it, though the trees are sometimes just a single dot.

It is useful to find the expected size of those trees. Let T be the expected number of nodes in a decode tree, including the root. If the packet is 1/3 filled with 1 bits and the rest 0, then there is a 2/3 probability that any given bit position will lack a tree entirely. In the remaining 1/3 of the cases, there will be 1 node for the root, plus a 1/3 probability that it will have a left subtree with an expected E nodes, and a 1/3 probability that it will have a right subtree with an expected E nodes. Therefore E must satisfy the equation:

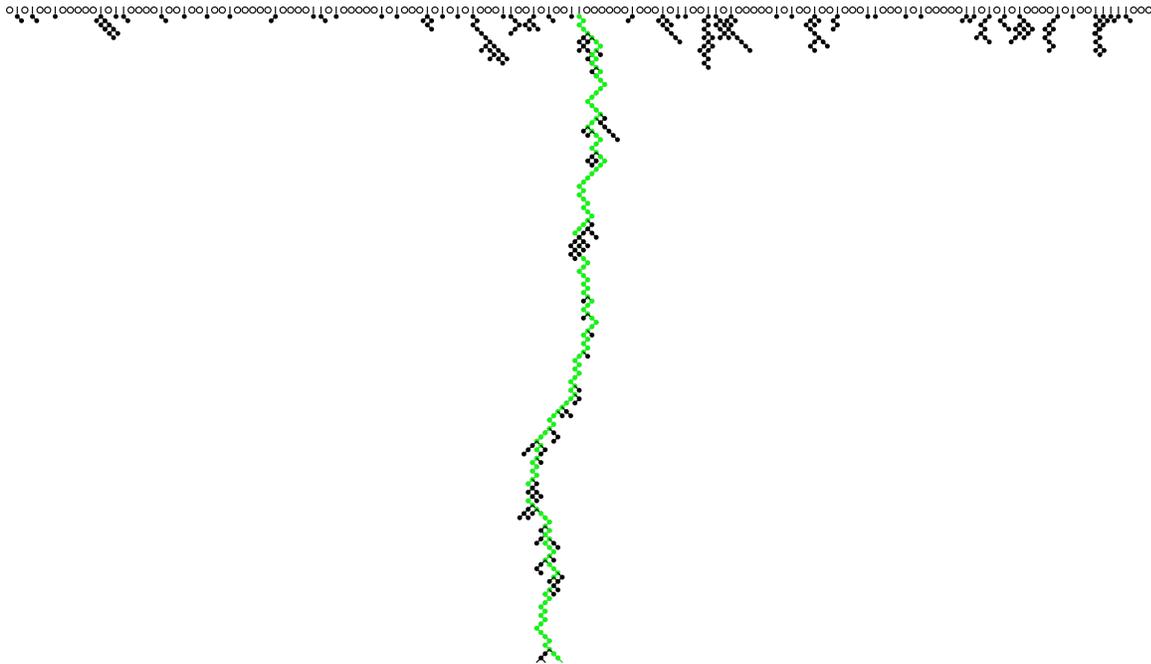
$$E = (2/3)(0) + (1/3)(1 + (2/3)E)$$

Solving for E yields $E=1$. Therefore, in an environment with no real messages, there will on average be only a single tree node to search per received bit, even when the density is 1/3.

If a processor can search one tree node per time step, then a single processor will be able to handle the decoding of the random trees when there is no actual message. A more realistic

estimate for a depth-first search of the trees might be 2 time steps per node (including backtracking), so two parallel processors would be needed to keep up with the no-message state.

When an actual message is sent, the situation looks like this:

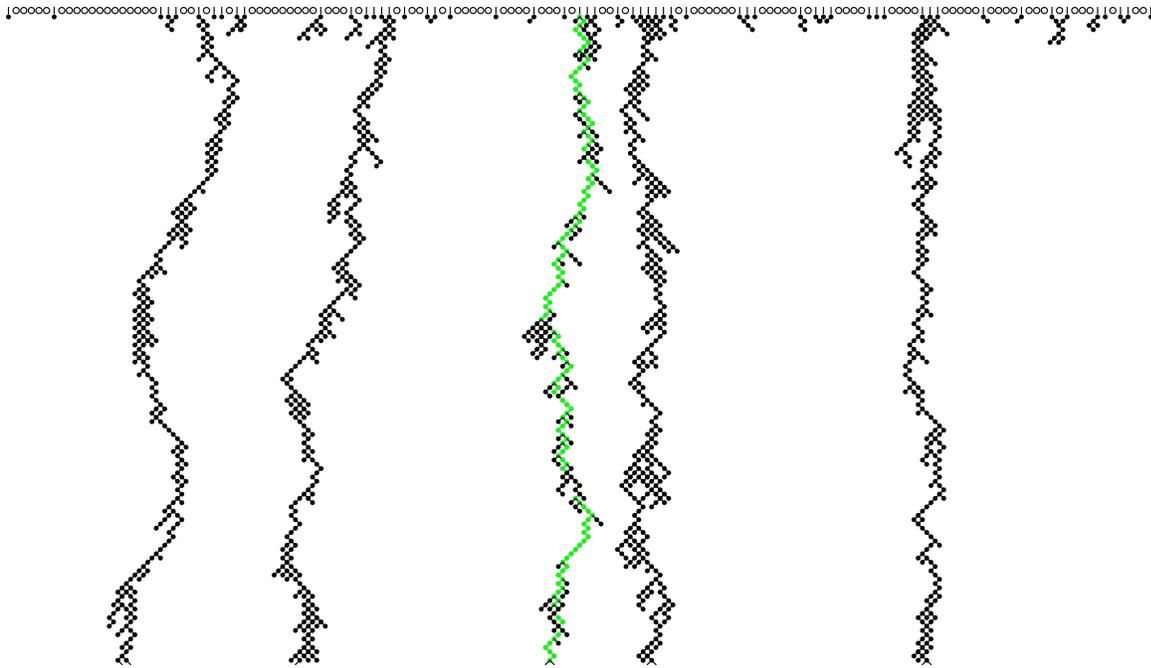


This diagram shows 150 received bits, and a single message of 300 bits. The expected number of nodes to search is 150 nodes for the trees with random noise, plus 300 nodes for the message, plus 300 nodes for the random trees that spawn off of the actual message tree due to random noise.

Assume a large set of simple processors designed to do depth-first searches on these trees. Any time a processor reaches a depth equal to the message length (e.g. 1000 for 1000-bit messages), then it signals a central coordinator that it has a message, and sends that message to the coordinator. As new bits are received, that coordinator will assign idle processors to decode the incoming trees, with one processor assigned to each tree.

A buffer will store the received bits. This buffer will be longer than a single packet. Assume that the buffer is large enough that when an actual message arrives, a single processor assigned to that tree will be able to accomplish a depth-first search of the tree before it slides over and falls out of the buffer. If packets are at least 4 times longer than the message length, then a buffer length of twice the packet length should suffice.

Suppose that the legitimate sender transmits one message, and the attacker transmits 4 messages. The decode trees might look like this:



In this case, 5 processors will end up being assigned to the 5 messages, and will decode each of them before they disappear. Another 1 or 2 processors will be assigned to all of the tiny trees at the top. So 7 processors are enough to decode this. A chip might be designed to hold 100 or 1000 processors, so all but 7 of them would be idle. To avoid this, the coordinator can assign idle processors to trees that are already being searched, in case there are multiple messages in one tree.

Each new processor assigned to a tree will start over at the top, and decode it with a depth-first search. But the search will be randomized to avoid finding the same messages as the other processors assigned to that tree. In a depth first search, the first time a node is reached, the searcher travels down to one of its children. When it later backtracks to that node, it travels down the other child. It does not matter which child is traversed first. So each processor can randomly decide which child to choose first in each layer. Given this randomized approach, if a tree has many more processors than messages, then it is very likely that every message will be found before the tree falls out of the buffer. If there are many more processors than messages, then the system simply doesn't have the computational power to decode all the messages, but it is likely that each processor will find a different message, and so the decode has graceful degradation. It decodes as many messages as its processing power allows, and misses the others, relying on higher-level protocols like TCP/IP to request the message be resent.

We note that for this algorithm, if every message ends up in a separate tree, then the parallel search algorithm gives essentially perfect, linear speedups. Doubling the number of processors on the chip will double the number of messages per second that can be received. An attacker can only jam the system by expending the energy needed to flood the receiver with more than that many messages. And even then, the system degrades gracefully, receiving as many messages as its hardware allows, where it is essentially random which messages are chosen.

If the attacker manages to send many messages that start at the exact same bit position as a legitimate message, then the system has slightly less than linear speedup. So this would be the ideal attack for an attacker. However, it is an extremely difficult attack. The attacker must start his attack packet at exactly the same bit position as the legitimate message. If packets are thousands or millions of nanoseconds long, the attacker must synchronize to within only a few nanoseconds. This requires the attacker to know the exact distances between himself, the sender, and the receiver, with all 3 distances known to within a few feet. And he must be continually sending the first few bits of his packet so that when he discovers the sender has started sending, he will have already started his packet at the correct time. All of this can be done, but it is difficult, and it still gives only a small help to the attacker.

To further resist the attack, two additional sources of randomness are injected into the system. Any legitimate sender should send each of his messages in a separate packet, and must delay the start of the packet by a random amount. Note that this does not affect the bit rate by much, because the sender's packets can overlap. So the each packet might be delayed by a random duration on the order of only 1% of the packet length. Second, the sender should preface each message (that's message, not packet) with 16 random bits, and the receiver should ignore the first 16 bits of any decoded message. This randomizes the tree corresponding to any given message, which makes it difficult for the attacker to target certain parts of the tree, even if the attacker knows exactly what data the sender is planning to send.

The algorithm described here can be briefly summarized as follows:

- Assign a processor to each incoming tree
- If you have an idle processor:
 - Assign to oldest tree with 1 processor
- If you run out of processors:
 - Take from a tree with more than 1
 - If none, take from the oldest
 - Variant: almost oldest

- Randomness
 - Delay before packet (which has 1 message)
 - First 16 bits of message
 - Order for depth-first search

It appears that this algorithm will be resistant to attacks, and to random noise and traffic from other users. This should be confirmed with simulation and actual experiment. It also appears that this can be easily implemented in hardware. The various processors do not need to communicate with each other. They only need to communicate with a coordinator, in a star topology. Only a few signaling lines are needed for the coordinator to assign a processor to the tree corresponding to a bit position in the buffer, or tell it to stop because its tree has fallen out of the end of the buffer, or to detect whether the processor has found a message, or has completed the entire depth-first search of the tree. It should be possible to use an architecture with a general-purpose CPU for the coordinator and massively-parallel chips for the search processors, and the communication bandwidth should be acceptable, even if these are spread across multiple chips. We note that the search itself can be efficient if the hash function is a linear feedback shift register, and if the received bits are continually broadcasted to all processors, and every processor keeps its own local copy of the buffer. Given all that, this appears to be a working solution for parallel BBC decoding.

References

[1] Baird, Leemon C. III, Bahn, William L. & Collins, Michael D. (2007) Jam-Resistant Communication Without Shared Secrets Through the Use of Concurrent Codes, Technical Report, U. S. Air Force Academy, USAFA-TR-2007-01, Feb 14.