LDPC CODES FOR IRIG-106 WAVEFORMS: PART II–RECEIVER DESIGN

Erik Perrins
Department of Electrical Engineering and Computer Science
University of Kansas
esp@ieee.org

ABSTRACT

Low density parity check (LDPC) codes allow a communications link to operate reliably at signal to noise ratios that are very close to the Shannon limit. Because of this, in the early 2000s they were studied in connection with SOQPSK-TG and were eventually adopted into the IRIG-106. The deployment for SOQPSK-TG has proved to be very successful, which has motivated interest in finding an LDPC solution for PCM/FM and ARTM CPM. Such a solution, however, has proved to be elusive for reasons that were not entirely clear in the past. In our companion paper, we reveal these reasons, which also allows us to develop an LDPC design procedure for *all* IRIG-106 modulations and we apply this procedure to PCM/FM and ARTM CPM. In this paper, our focus is on developing high-speed, parallelizable decoders/demodulators that are suitable for high-throughput applications. We present the performance characteristics of our fixed-point software prototype system. We demonstrate that the coded LDPC system **performs around one dB from the respective channel capacities of these modulations**. As such, these codes can be considered to fill in the LDPC options that are currently absent in the IRIG-106 standard for PCM/FM and ARTM CPM.

INTRODUCTION

The desired LDPC coding scheme is shown in Figure 1 and consists of an LDPC encoder and a CPM modulator that are separated by an interleaver (denoted by the symbol Π). In our companion paper [1], we presented a procedure for designing LDPC codes that are "matched" to a desired continuous phase modulation (CPM) scheme. The CPMs we consider herein are pulse code modulation/frequency modulation (PCM/FM), the Telemetry Group version of shaped-offset quadrature phase shift keying (SOQPSK-TG), and the multi-h CPM developed by the Advanced Range TeleMetry program (ARTM CPM), which are all defined in full detail in the IRIG-106 standard [2]. For convenience, going forward we refer to these, respectively, as ARTM0, ARTM1, and ARTM2.

In [1], we applied our LDPC design procedure to these three modulations to obtain a family of six LDPC codes for each one (18 codes in total). In this paper, our focus is on developing parallelization strategies for the LDPC decoder and the CPM demod/decoder (known as the soft-input soft-output, or SISO, module). We then apply these strategies in a fixed-point software prototype and characterize its performance and execution speed. The results demonstrate that the LDPC–CPM scheme is a viable technology to fill in the missing LDPC options in the IRIG-106 standard.

DISTRIBUTION STATEMENT A. Approved for public release; Distribution is unlimited 412TW-PA-23165.

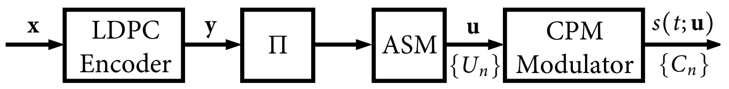

Figure 1: Transmitter model.

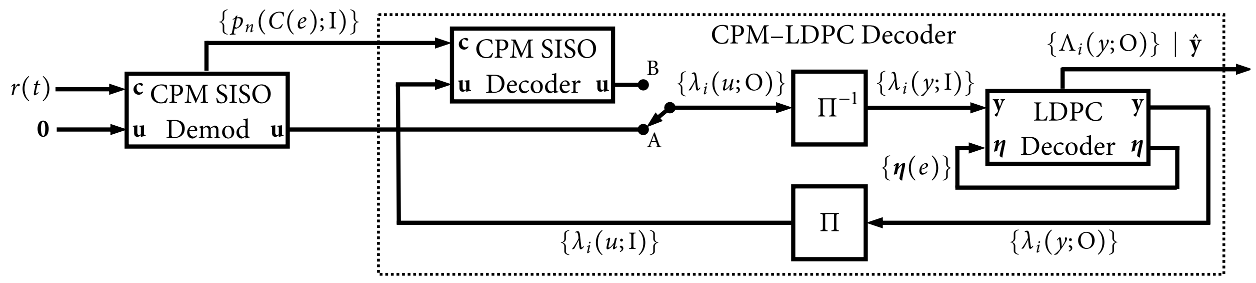
$\{p_n(C(e);\mathrm{I})\}$
$r(t)$
0
c CPM SISO u Demod u
c CPM SISO u Decoder u
B
A
CPM–LDPC Decoder
$\{\lambda_i(u;\mathrm{O})\}$
Π^{-1}
$\{\lambda_i(y;\mathrm{I})\}$
y LDPC y
η Decoder η
$\{\boldsymbol{\eta}(e)\}$
$\{\Lambda_i(y;\mathrm{O})\} \mid \hat{\mathbf{y}}$
Π
$\{\lambda_i(u;\mathrm{I})\}$
$\{\lambda_i(y;\mathrm{O})\}$

Figure 2: Receiver model.

SYSTEM MODEL

The transmitter model is shown in Figure 1 and the corresponding receiver model is shown in Figure 2. The transmitter allows for the insertion of a known bit sequence called the *attached sync marker* (ASM) to help identify the beginning of each code word. Because the ASM is available, the CPM SISO uses a small portion of the ASM ($N_{\mathrm{W}U}$ bits) as a "warm up" in its forward/backward trellis operations.

The receiver has the dual task of CPM demodulation and LDPC decoding. However, the demodulation task can be further subdivided into a traditional CPM demodulator that is positioned *outside* the iterative decoding loop, and a CPM decoder that is located *inside* the iterative decoding loop. The CPM demod operates directly on $r(t)$, the signal received from the AWGN channel and has no (zero) *a priori* information on its "**u**" input. Existing telemetry demodulators can be used with only two minor modifications: (a) the matched filter (MF) outputs that are generated within, $\{p_n(C(e);\mathrm{I})\}$, are preserved and passed to the CPM SISO decoder so they can be "recycled" in future decoding iterations; and (b) the CPM demodulator generates a soft output, and thus it is described as a CPM SISO demod. Once the CPM SISO demod has demodulated to the end of the received code word, its outputs are passed to the CPM–LDPC decoder, as shown in Figure 2. This composite decoder conducts "global" iterations between the CPM and LDPC decoders, which are separated by an interleaver/deinterleaver [3]. For the first global iteration, the CPM decoding task was completed externally by the demod and that result is passed forward via the switch in position "A." For successive global iterations, the switch is placed in position "B" so that the CPM SISO decoder's updates are used. Within the global loop, when it is the LDPC decoder's turn to execute, it may perform one or more iterations in a "local" loop. Due to space limitations, it is not possible to provide a comprehensive treatment of all the details pertaining to Figure 2.

It is important to emphasize that the first part of the CPM demodulation process takes place only *once* and is positioned *outside* the global iterative loop. This traditional CPM demodulation process is well understood and is responsible for the familiar tasks of timing, phase, frequency, and frame (ASM) synchronization [4–7]. From there the CPM SISO decoder and the LDPC decoder take turns exchanging and updating soft values in the form of log-likelihood ratios (LLRs), which are a specific kind of soft value that is scaled such that its “units” are meaningful in terms of probability (see, for example, [8]). Because our LDPC codes are quasi-cyclic, we have implemented the log-domain LDPC decoder in [8], which is straightforward to parallelize by a factor equal to the circulant size of the quasi-cyclic code. Because there is less literature available on parallelizing the CPM SISO decoder, we devote most of our development to that topic.

BASIC SISO OPERATIONS

The original development of the SISO module goes back to the BCJR algorithm in [9], although our formulation more closely resembles the notation of [10]. We apply several adaptations and customizations that are relevant to our application, including: (1) only one soft output is required, that belonging to the bit stream $\{u_i\}$; (2) the soft input for the “code” actually belongs to a CPM, thus the metric increments must be formulated as they are for the CPM Viterbi algorithm; and (3) the entire algorithm is formulated using values that reside in the *log domain*. There are no special steps taken to arrive in the log domain. But just as is the case with the Viterbi algorithm, the raw MF output are inherently in the log domain from a system analysis perspective, and so remaining in the log domain greatly simplifies the SISO arithmetic (multiplication becomes addition, etc.). The basic operations of the SISO module are given in Algorithms 1 and 2, where the former is specialized to the case of a binary ($M = 2$ symbol alphabet that carries $n_0 = 1$ bit per symbol) CPM (ARTM0 and ARTM1) and the latter is specialized to the case of a $M = 4$ symbol alphabet ($n_0 = 2$ bits per symbol) CPM with a time-varying trellis (ARTM2). We will describe both formulations simultaneously, which allows us to compare and contrast their similarities and differences.

Each edge (branch) in the CPM trellis is labeled with a unique value of the CPM “code” variable, which has an integer or vector representation $C_n \leftrightarrow \mathbf{c}_n$. The various sub-components of $C_n \leftrightarrow \mathbf{c}_n$ can be stored in separate LUTs that are indexed by a given edge, e. The trellis diagrams for all three modulations are shown in Figure 3, where each trellis has N_E edges and N_S states. The first two LUTs are $s^\mathrm{S}(e)$ and $s^\mathrm{E}(e)$, which are the starting state and the ending state, respectively. When there are multiple modulation indexes ($N_h > 1$), as in Algorithm 2, these two LUTs become time varying and thus we add a time subscript, $s^\mathrm{S}_n(e)$ and $s^\mathrm{E}_n(e)$, which is understood to be taken modulo-N_h. The remaining LUTs are: $C(e)$, the code symbol for a given edge; $\mathbf{u}(e)$, which is the length-n_0 bit vector for a given edge, which corresponds to the current edge (branch) symbol $U_0(e)$; $u(e)$, $u_0(e)$, and $u_1(e)$, which are the elements (bits) of $\mathbf{u}(e)$ that are needed for the $n_0 = 1$ and $n_0 = 2$ cases.

The SISO module accepts two *a priori* soft inputs: one for the “code” sequence $\{C_n\}$ and one for the “information” sequence $\{u_i\}$; it returns an extrinsic *a posteriori* soft output for the information sequence. The $\{C_n\}$ input is based on CPM *symbols* and uses a symbol index, n. The $\{u_i\}$ input and output are based on LDPC encoded *bits* and use a bit index, i. These indexes cover the following ranges

$$\begin{aligned} -N'_{\mathrm{W}U} &\le n \le N' + N'_{\mathrm{W}U} - 1 \\ -N_{\mathrm{W}U} &\le i \le N + N_{\mathrm{W}U} - 1 \end{aligned} \tag{1}$$

where N is the length of the LDPC code word and $N_{\mathrm{W}U} \le N_{\mathrm{A}SM}$ is the “warm-up” period for the forward and backward recursions, with $N' = N/n_0$, and $N'_{\mathrm{W}U} = N_{\mathrm{W}U}/n_0$. At the beginning of the

As was mentioned earlier, $C(e) = e$, and thus this LUT is mentioned only for its conceptual value.

In our system, the true information sequence is $\{x_i\}$; however, from the local perspective of the CPM modulator and the SISO module, $\{u_i\}$ fills the role of the information sequence.

algorithm, the sole interface between the bit and symbol indexes takes place at Algorithm 1 Line 6 (which is trivial for $n_0 = 1$) and Algorithm 2 Lines 6–8 (which is more involved due to $n_0 = 2$).

Once the received bit-based LLRs are grouped into symbol-based n_0-tuples, $\boldsymbol{\lambda}_n(\mathbf{u}; \mathrm{I})$, the middle portions of Algorithms 1 and 2 are entirely *symbol based*, i.e. the metric increment, $\gamma_n(e)$, and the forward and backward recursions that generate $A_n(s)$ and $B_n(s)$ involve variables that are based exclusively on n. The forward recursion touches each ending state and performs an update over the edges that merge into that ending state. For this, we define the "ending set" as

$$\mathbf{e}(s^{\mathrm{E}}) \triangleq \{e : s^{\mathrm{E}}(e) = s^{\mathrm{E}}\} \tag{2}$$

which contains all edges such that the ending state of the edge is s^{E} (M edge indexes in total). Likewise, the backward recursion touches each starting state and performs an update over the M edges that merge into that starting state, using the "starting set"

$$\mathbf{e}(s^{\mathrm{S}}) \triangleq \{e : s^{\mathrm{S}}(e) = s^{\mathrm{S}}\} \tag{3}$$

(Extending these definitions to a time-varying trellis is straightforward.) The results of a single forward or backward time step, $\{A_n(s)\}_{s=0}^{N_{\mathrm{S}}-1}$ or $\{B_n(s)\}_{s=0}^{N_{\mathrm{S}}-1}$, respectively, represent an *unnormalized* log-based probability mass function (PMF). The absolute normalization of these PMFs (i.e. that they are made to sum to unity in the linear domain) is not necessary because such normalization would cancel out in the subtraction step at the end of the SISO module (e.g. Algorithm 1 Line 20). As such, the normalization called for in Algorithms 1 and 2 is for the narrow purpose of *preventing numerical overflow* in a finite-precision implementation. This normalization can be as simple as determining the maximum value over the set, and then subtracting this value from all values in the set. In a floating-point implementation, normalization is unnecessary because the finite block length (N) is sufficiently short that overflow is unlikely.

The final step in the SISO module is the completion step, which forms the extrinsic output LLRs. This step makes use of $A_n(s)$ and $B_n(s)$ from the forward and backward recursions, and an *extrinsic* version of the metric increment, which excludes the LLR input. For $n_0 = 1$ (Algorithm 1) there is only one input value to exclude at each time step (compare Lines 7 and 16). For $n_0 = 2$ (Algorithm 2), two separate extrinsic increments are formed, where each takes a turn excluding one of the input values from the current time step (compare Line 9 and Lines 18–19). Each extrinsic output LLR value is formed by marginalizing the PDF over the proper sets of edges for $u = 0$ and $u = 1$. This is done once ($n_0 = 1$) each time step for Algorithm 1 and twice ($n_0 = 2$) each time step for Algorithm 2. Again, as was just pointed out, this subtraction in the log domain is "self normalizing" and yields accurate LLRs regardless of the scale of the input arguments.

The basic operator within the SISO module is "log-based addition," or "max star"

$$\max\star(a, b) \triangleq \ln(e^a + e^b) = \max(a, b) + \mathrm{f}(a - b) \tag{4}$$

where $\max(a, b)$ is the simple maximum value between a and b and the "correction term" is

$$\mathrm{f}(x) = \ln(1 + e^{-|x|}) \tag{5}$$

A low-complexity version of the SISO module is obtained when we drop the correction term and replace all instances of $\max\star(a, b)$ with $\max(a, b)$, which is known as the "max–log" SISO and has a minor performance loss of less than a half dB relative to the optimal version.

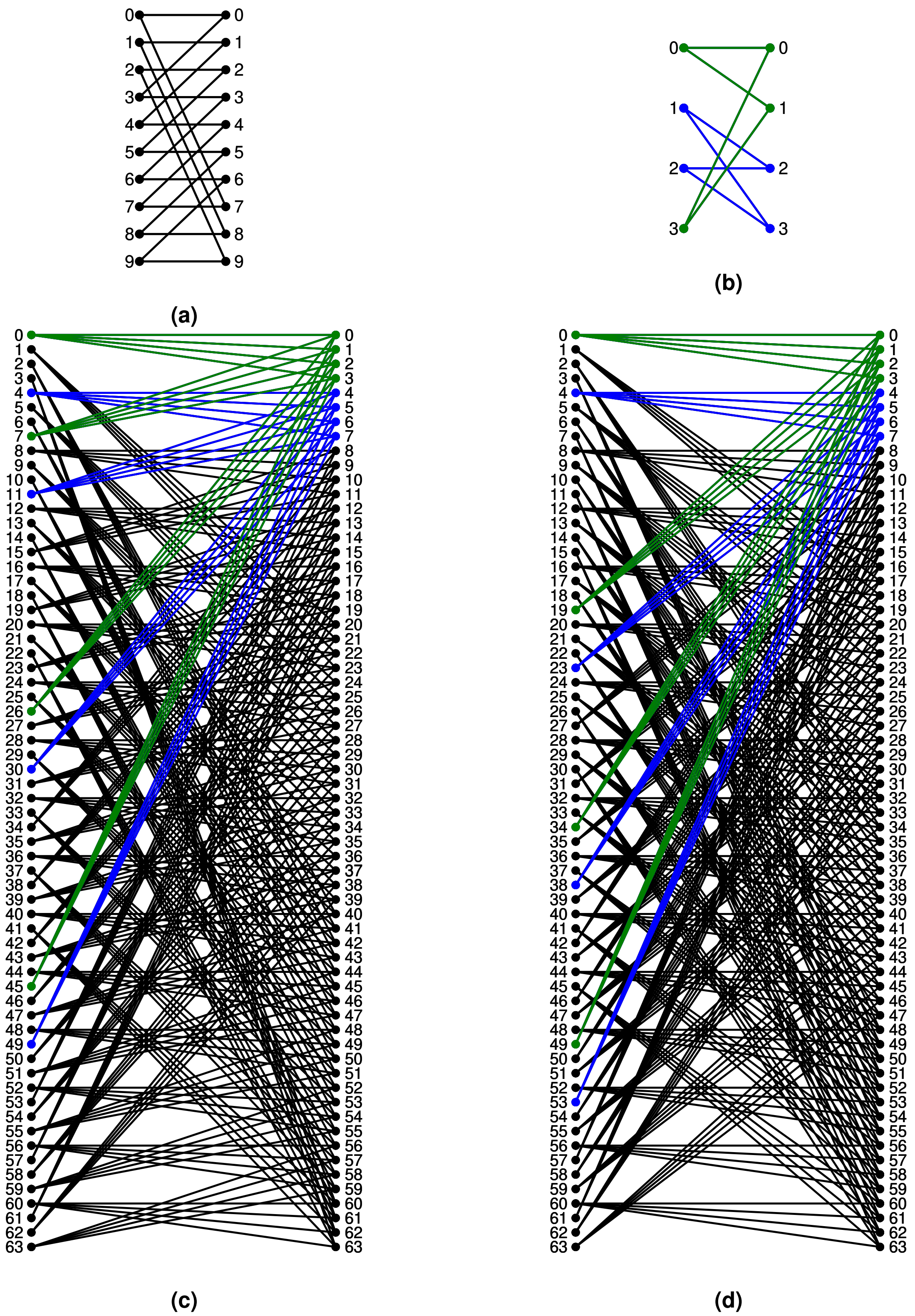

Figure 3: Trellis diagrams for (a) ARTM0 and (b) ARTM1. For ARTM2, which has a time-varying trellis, we have (c) for even symbol indexes (n-even) and (d) for odd symbol indexes (n-odd). The values of the starting state, s^{S}, are shown on the left-hand side of each trellis, and the values of the ending state, s^{E}, are shown on the right-hand side. The ARTM1 trellis can be decomposed into two separate 2-state "butterflies," and for ARTM2 they can be decomposed into 16 separate 4-state "butterflies," the first two of which are shown in green and blue.

A-TYPE PARALLELIZATION OF THE SISO MODULE

The SISO module can be envisioned as having a *time* dimension and a *state/edge* dimension. The three main steps of the algorithm (i.e. forward, backward, and completion) each consist of an outer "for" loop that steps through time, and an inner "for" loop that steps through the states/edges. We now outline two distinct parallelization strategies that can be applied to the SISO module. The first offers a way to parallelize the time dimension, and the second offers a way to parallelize the state/edge dimension.

In [11], several possible parallelization schemes are presented for the Viterbi algorithm, which is similar to the SISO module. We select the one described as *algorithmic* parallelization and refer to it as *A-type*. The basic idea is to divide the received code word into P_{A} segments (that overlap to a small degree), and then process these segments in parallel; thus achieving a parallelization factor of approximately P_{A}.

Our received code word has a length of N bits and we allow a SISO warm-up period of $N_{\mathrm{W}U}$ bits on either side of this word, for a total length of $N + 2N_{\mathrm{W}U}$ bits. In terms of CPM *symbols*, we have $N' = N/n_0$, $N'_{\mathrm{W}U} = N_{\mathrm{W}U}/n_0$, and a total length of $N' + 2N'_{\mathrm{W}U}$ symbols. To form the overlapping segments, we divide the code word portion by P_{A}, i.e. $N_{\mathrm{P}} = N/P_{\mathrm{A}}$ and $N'_{\mathrm{P}} = N'/P_{\mathrm{A}}$, and allow each segment to have the full warm-up period of $N_{\mathrm{W}U}$ bits ($N'_{\mathrm{W}U}$ symbols). The symbol and bit indexes that comprise the "base" segment are, respectively,

$$\begin{aligned} -N'_{\mathrm{W}U} \le n_{\mathrm{P}} \ &\le N'_{\mathrm{P}} + N'_{\mathrm{W}U} - 1 \\ -N_{\mathrm{W}U} \le i_{\mathrm{P}} \ &\le N_{\mathrm{P}} + N_{\mathrm{W}U} - 1 \end{aligned} \tag{6}$$

and the indexes for the p-th segment are obtained with the offset

$$\begin{aligned} &n_{\mathrm{P}} + pN'_{\mathrm{P}} \\ &i_{\mathrm{P}} + pN_{\mathrm{P}} \end{aligned} \tag{7}$$

where $p \in \{0, 1, \ldots, P_{\mathrm{A}}-1\}$. A "vectorized" version of the metric increment is formed by stacking values from each segment together as in

$$\boldsymbol{\gamma}_{n_{\mathrm{P}}}(e) = [\gamma_{n_{\mathrm{P}}}(e), \gamma_{n_{\mathrm{P}}+N'_{\mathrm{P}}}(e), \ldots, , \gamma_{n_{\mathrm{P}}+(P_{\mathrm{A}}-1)N'_{\mathrm{P}}}(e)]^T \tag{8}$$

From there, the forward/backward/completion steps take place over the shortened index interval in (6) and generate "vectorized" quantities, $\mathbf{A}_{n_{\mathrm{P}}}(s^{\mathrm{E}})$, $\mathbf{B}_{n_{\mathrm{P}}}(s^{\mathrm{S}})$, and $\boldsymbol{\lambda}_{i_{\mathrm{P}}}(u; \mathrm{O})$. The vector operator $\max\star(\mathbf{a}, \mathbf{b})$ is applied on an element-by-element basis, which is a segment-by-segment basis in this context. After completion, the values in $\{\boldsymbol{\lambda}_{i_{\mathrm{P}}}(u; \mathrm{O})\}_{i_{\mathrm{P}}=0}^{N_{\mathrm{P}}-1}$ can be "unstacked" to yield $\{\lambda_i(u; \mathrm{O})\}_{i=0}^{N-1}$. The number of time steps required by the completion step is reduced by a factor of exactly P_{A}. The forward and backward steps each have a warm-up interval of the original length, $N'_{\mathrm{W}U}$, and so these are reduced by a factor of

$$\frac{N + N_{\mathrm{W}U}}{N/P_{\mathrm{A}} + N_{\mathrm{W}U}} \tag{9}$$

which approaches P_{A} as N becomes very large relative to $N_{\mathrm{W}U}$.

B-TYPE PARALLELIZATION OF THE SISO MODULE

The state/edge processing can be parallelized by organizing the arguments of the forward/backward and completion steps in groups that can be processed together. One trellis structure that can be exploited is a "butterfly," which is defined as a sub-trellis of starting states, ending states, and connecting edges that is disjoint from the rest of the trellis. For example, the ARTM1 trellis in Figure 3 (b) can be decomposed into two separate 2-state butterflies (shown in green and blue), and the ARTM2 trellis in Figures 3 (c) and (d) can be decomposed into 16 separate 4-state butterflies (two of which are shown in green and blue). We refer to this as *B-type* parallelization.

Because of the size of the ARTM2 trellis and the impact that parallelization can have on its throughput, we use this as a detailed case study. In our implementation, we format $A_n(\cdot)$, $B_n(\cdot)$, and $\gamma_n(\cdot)$ large, contiguous arrays. We are able to stride through these arrays in either a *scalar* or *vector* fashion. For example, $A_n(s^{\mathrm{E}})$, $B_n(s^{\mathrm{S}})$, and $\gamma_n(e)$, use n as the time index, and within each time slice, s^{E}, s^{S}, and e, respectively, index the scalar values. If we treat these arrays as a collection of length-P_{B} vectors, then $\mathbf{A}_n(v)$, $\mathbf{B}_n(v)$, and $\boldsymbol{\gamma}_n(v)$ use n as the time index and v as the generic index for each length-P_{B} vector within each time slice.

Using the first butterfly in Figure 3 (c) as an example (shown in green), the forward recursion for $s^{\mathrm{E}} \in \{0,1,2,3\}$, which is vector index $v = 0$ using $P_{\mathrm{B}} = 4$, can be formulated as

$$\begin{aligned}\mathbf{A}_n(0) = \max\star \big\{&\mathrm{rep}_4(A_{n-1}(0)) + \boldsymbol{\gamma}_n(0), \mathrm{rep}_4(A_{n-1}(7)) + \boldsymbol{\gamma}_n(7),\\ &\mathrm{rep}_4(A_{n-1}(26)) + \boldsymbol{\gamma}_n(26), \mathrm{rep}_4(A_{n-1}(45)) + \boldsymbol{\gamma}_n(45)\big\}\end{aligned} \tag{10}$$

where $\mathbf{x} = \mathrm{rep}_{P_{\mathrm{B}}}(x)$ replicates the scalar quantity x a total of P_{B} times, forming a length-P_{B} vector $\mathbf{x}$. Similar formulas can be derived for the remaining butterflies. Likewise, a similar formulation is available for the backward recursion. The end result is that $P_{\mathrm{B}} = 4$ states are updated in parallel and the forward/backward recursions require only 16 vector-based updates at each time step, instead of the original $N_{\mathrm{S}} = 64$ scalar-based updates.

As was implied in (10), our edge metrics are stored in sequential edge order as viewed from the left-hand side of the trellis, e.g. $\boldsymbol{\gamma}_n(0)$ accesses the first P_{B} edge metrics. For the completion step, we assemble the input arguments (one per edge) in this same edge order, with the caveat that we *interleave* length-P_{B} vectors pertaining to Algorithm 2 Lines 23 and 23, like so for the trellis in Figure 3 (c):

$$\mathbf{M}(0) = \mathrm{rep}_4(A_{n-1}(0)) + \boldsymbol{\chi}_n^{(0)}(0) + \mathbf{B}_n(0) \tag{11}$$

$$\mathbf{M}(1) = \mathrm{rep}_4(A_{n-1}(0)) + \boldsymbol{\chi}_n^{(1)}(0) + \mathbf{B}_n(0) \tag{12}$$

$$\mathbf{M}(2) = \mathrm{rep}_4(A_{n-1}(1)) + \boldsymbol{\chi}_n^{(0)}(1) + \mathbf{B}_n(5) \tag{13}$$

$$\mathbf{M}(3) = \mathrm{rep}_4(A_{n-1}(1)) + \boldsymbol{\chi}_n^{(1)}(1) + \mathbf{B}_n(5) \tag{14}$$

$$\vdots$$

$$\mathbf{M}(126) = \mathrm{rep}_4(A_{n-1}(63)) + \boldsymbol{\chi}_n^{(0)}(63) + \mathbf{B}_n(14) \tag{15}$$

$$\mathbf{M}(127) = \mathrm{rep}_4(A_{n-1}(63)) + \boldsymbol{\chi}_n^{(1)}(63) + \mathbf{B}_n(14) \tag{16}$$

where $\mathbf{M}(\cdot)$ is an array with 128 length-4 vector terms, which has a scalar length of $2 \cdot N_{\mathrm{E}} = 512$ terms. This array is divided in half, and the two halves are processed in a massive element-by-element $\max\star\{\cdot,\cdot\}$ vector operation to yield a length-256 scalar result. This halving and processing can be repeated until a length-8 scalar result is obtained (a total of six halving and processing steps). This marginalization yields 8 scalars that constitute two separate 4-ary log-domain extrinsic PMFs for U_n. The first of these extrinsic 4-ary PMFs is marginalized into the LLR $\lambda_{2n}(u;\mathrm{O})$ (the step required by Algorithm 2 Line 23) and the second extrinsic 4-ary PMF is marginalized into the LLR $\lambda_{2n+1}(u;\mathrm{O})$ (the step required by Algorithm 2 Line 26).

This example shows how B-type parallelization by a factor of $P_{\rm B} = 4$ has a natural application to ARTM2, which has $M = 4$. Similar techniques can be applied to ARTM0 and ARTM1 with $P_{\rm B} = 2$ to match their $M = 2$. Even though the ARTM0 trellis in Figure 3 (a) does not have a butterfly structure, it is still possible to group terms in ways that can be processed in parallel.

COMBINED PARALLELIZATION

As we have just outlined, A-type parallelization is applied external to the SISO module with near-negligible impact on the algorithm architecture, while B-type parallelization is applied internal to the SISO module by specializing the processing steps to the particular trellis structure at hand.

Our prototyping platform offers a simple means of single instruction, multiple data (SIMD) parallel processing using 128-bit words. This is via the architecture known as streaming SIMD extensions (SSE), which is available on Intel processors. Our platform is a 2021 MacBook Pro, which has an M1 processor (ARM architecture), which uses a similar 128-bit SIMD scheme called Neon. Our C++ code was written with the "sse2neon.h" header file, which allowed us to write SSE-based code that actually calls the equivalent Neon instructions. By selecting a 16-bit fixed-point integer (i.e. "short int") scalar data type, we are able to achieve 8-way parallelization with a 128-bit wide vector.

For ARTM0 and ARTM1, A-type parallelization is applied externally by a factor of $P_{\rm A} = 4$. Internal to the SISO module, this means that when "scalars" are handled (replicated, copied, etc) they are 64 bits wide, whereas vectors are 128 bits wide with $P_{\rm B} = 2$ "scalars" each. Any mathematical operations ($\max\star$, addition, subtraction) are performed element-by-element on 16-bit elements in a 128-bit vector. When SISO execution is finished, the $P_{\rm A} = 4$ formatting is undone externally, and the results are sent to the LDPC decoder.

For ARTM2, A-type parallelization is applied externally by a factor of $P_{\rm A} = 2$. Internal to the SISO module, this means that when "scalars" are handled (replicated, copied, etc) they are 32 bits wide, whereas vectors are 128 bits wide with $P_{\rm B} = 4$ "scalars" each. However, as before, any mathematical operations are performed element-by-element on 16-bit elements in a 128-bit vector.

The combined parallelization factor, $F_{\rm P}$, is $P_{\rm B}$ times the term in (9), which is a value that is close to (but less than) 8 due to the warm-up overhead. Our observation is that a trellis of many states requires a longer warm-up interval to reach the "steady state." The values of $N_{{\rm W}U}$ we use in our system are 24, 8, and 64, respectively, for ARTM0, ARTM1, and ARTM2.

BER PERFORMANCE, EXECUTION SPEED, AND CONCLUSION

Figure 4 shows bit error rate (BER) simulations for the 18 LDPC codes (six codes times three modulations) that were produced in this study and Table 1 lists their coding gains, which range from 7.0 to 10.8 dB! The coding gains are denoted as Δ_0, Δ_1, and Δ_2 [in dB] for ARTM0, ARTM1, and ARTM2, respectively, and use the *uncoded* BER = 10^{-8} crossing points of $E_b/N_0 \in \{10.8, 12.9, 13.3\}$ dB as a reference. The gains are comparable across the code and modulation types, which validates the consistency of the design approach.

Table 1 also lists the number of global iterations per second (GIPS) achieved by our software prototype: GIPS$_0$, GIPS$_1$, and GIPS$_2$, respectively, for ARTM0, ARTM1, and ARTM2. The prototype was set for a single local (LDPC) iteration per global iteration, although these parameters are an important subject of future study. The prototype was compiled and executed in two configurations: (a) single-precision floating-point *scalar*, and (b) 16-bit fixed-point integer with 8-way

Table 1: Global Iterations per second (GIPS) and Coding Gains of the LDPC–CPM Codes.

K	R	N	GIPS_0	GIPS_1	GIPS_2	Δ_0	Δ_1	Δ_2
1024	4/5	1280	2880, 9250 (3.2x)	5250, 13.1k (2.5x)	550, 3170 (5.8x)	7.6	7.9	7.0
1024	2/3	1536	2320, 9250 (4.0x)	4130, 13.1k (3.2x)	420, 2890 (6.9x)	8.6	9.1	8.6
1024	1/2	2048	1560, 6880 (4.4x)	3190, 11.4k (3.6x)	280, 2270 (8.1x)	9.1	9.9	9.3
4096	4/5	5120	725, 2630 (3.6x)	1340, 4190 (3.2x)	140, 910 (6.5x)	8.7	9.1	8.1
4096	2/3	6144	550, 2320 (4.2x)	1080, 3750 (3.5x)	110, 810 (7.4x)	9.5	10.1	9.5
4096	1/2	8192	380, 1840 (4.8x)	790, 3060 (3.8x)	70, 600 (8.6x)	9.9	10.8	10.2

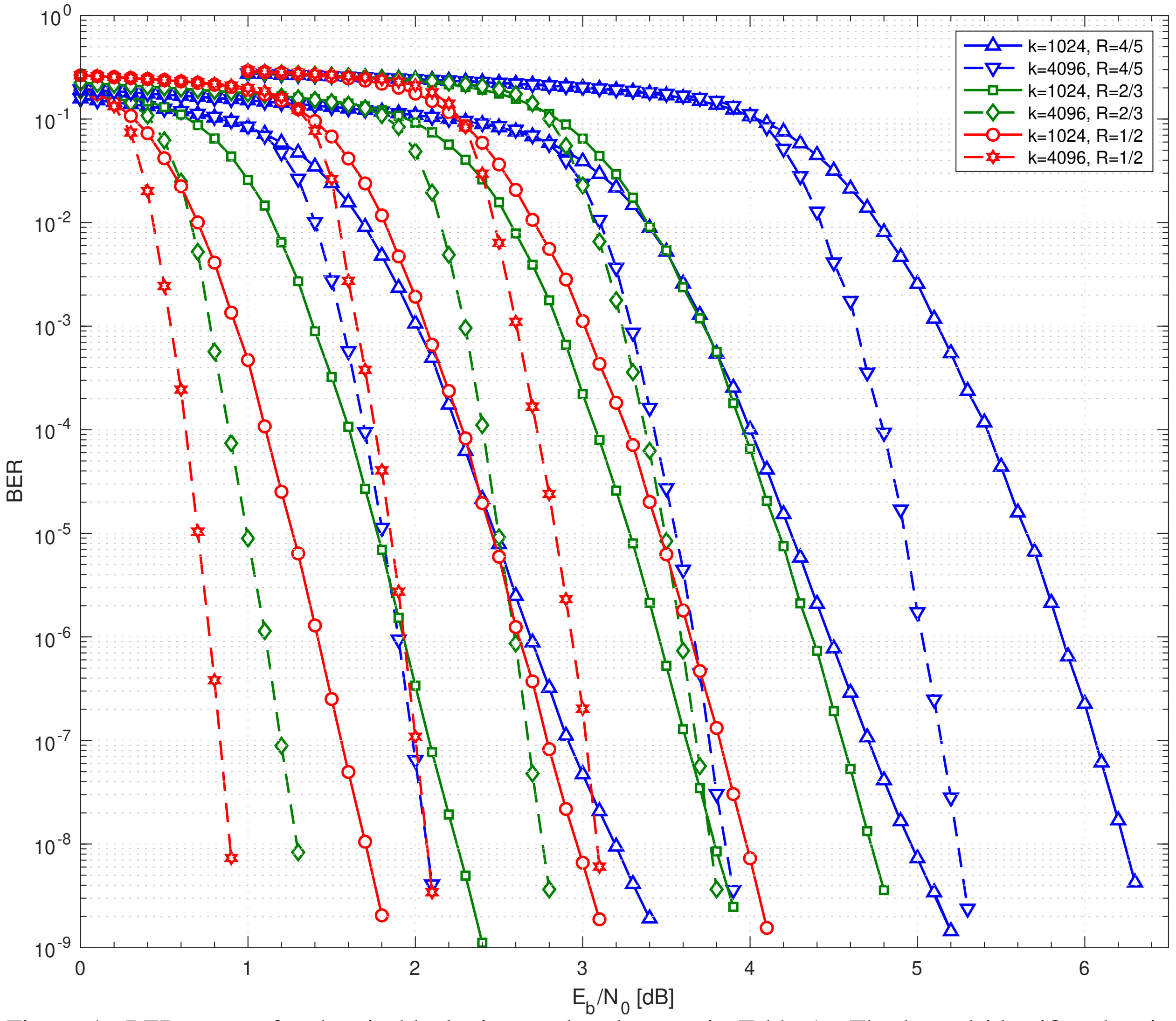

Figure 4: BER curves for the six block sizes and code rates in Table 1. The legend identifies the six cases, however, each modulation type has its own family of six curves: the ARTM0 set is the left-most, the ARTM2 set is the right-most, and the ARTM1 set is in the middle. The coding gains (relative to the *uncoded* cases at BER = 10^{-8}) are listed in Table 1 as Δ_0, Δ_1, and Δ_2 [in dB] for ARTM0, ARTM1, and ARTM2, respectively.

8-way parallelization (as presented earlier). The table lists two GIPS value for each case, (a), (b), respectively. The parallelization throughput gain is most pronounced for ARTM2, ranging from 6x to 8x. It was less pronounced for ARTM 1 (2.5x to 4x) and ARTM0 (3.2x to 5x), which is likely due to areas of "overhead" that were not parallelized, such as the MF bank. Although higher throughputs would be expected on dedicated hardware, these execution speeds on our software prototype are sufficient to support data links in the 5–10 Mbps range. Furthermore, our software decoder has a high enough throughput to simulate down to very low BERs, especially with multiple instances running simultaneously on a multi-core processor (all BER curves in Figure 4 were generated with our high-throughput fixed-point configuration). These high-throughput fixed-point simulations clearly demonstrate that an optimized FPGA design in current receiver hardware is realizable.

In future work, we plan to refine and finalize a specific LDPC code for each of the above cases, i.e. 18 pairs of $\mathbf{H}$ and $\mathbf{G}$, that can be considered for the IRIG-106 standard. Final BER results and other details (parameter optimizations, etc.) will be published at a later date.

Overall, these LDPC codes paired with ARTM0 (PCM/FM) and ARTM 2 (ARTM CPM) can be considered to fill in the "missing" LDPC coding options in the current version of IRIG-106.

REFERENCES

[1] E. Perrins, "LDPC codes for IRIG-106 waveforms: Part I–Code design," in *Proc. Int. Telemetering Conf.*, (Las Vegas, NV), Oct. 2023.

[2] Range Commanders Council Telemetry Group, Range Commanders Council, White Sands Missile Range, New Mexico, *IRIG Standard 106-2022: Telemetry Standards*, 2022. (Available on-line at https://www.irig106.org).

[3] E. Perrins and M. Rice, "A simple figure of merit for evaluating interleaver depth for the land-mobile satellite channel," *IEEE Trans. Commun.*, vol. 49, pp. 1343–1353, Aug. 2001.

[4] E. Perrins and M. Rice, "Reduced complexity detectors for multi-h CPM in aeronautical telemetry," *IEEE Trans. Aerosp. Electron. Syst.*, vol. 43, pp. 286–300, Jan. 2007.

[5] E. Perrins and M. Rice, "Optimal and reduced complexity receivers for M-ary multi-h CPM," in *Proc. IEEE Wireless Commun. Netw. Conf.*, (Atlanta, Georgia), pp. 1165–1170, Mar. 2004.

[6] P. Chandran and E. Perrins, "Symbol timing recovery for CPM with correlated data symbols," *IEEE Trans. Commun.*, vol. 57, pp. 1265–1270, May 2009.

[7] E. Perrins, "FEC systems for aeronautical telemetry," *IEEE Trans. Aerosp. Electron. Syst.*, vol. 49, pp. 2340–2352, Oct. 2013.

[8] D. Divsalar, S. Dolinar, C. R. Jones, and K. Andrews, "Capacity-approaching protograph codes," *IEEE J. Select. Areas Commun.*, vol. 27, pp. 876–888, Aug. 2009.

[9] L. Bahl, J. Cocke, F. Jelinek, and J. Raviv, "Optimal decoding of linear codes for minimizing symbol error rate," *IEEE Trans. Inform. Theory*, vol. 20, pp. 284–287, Mar. 1974.

[10] S. Benedetto, D. Divsalar, G. Montorsi, and F. Pollara, "A soft-input soft-output APP module for iterative decoding of concatenated codes," *IEEE Commun. Lett.*, vol. 1, pp. 22–24, Jan. 1997.

[11] G. Fettweis and H. Meyr, "High-speed parallel Viterbi decoding: Algorithm and VLSI-architecture," *IEEE Commun. Mag.*, pp. 46–55, May 1991.

Algorithm 1 Log-Based SISO APP Algorithm for $M=2$ ($n_0=1$).

```
1: Input: $\{p_n(C(e);\mathrm{I})\}$, for $-N'_{\mathrm{WU}} \le n \le N'+N'_{\mathrm{WU}}-1$, the received a priori PMFs from the CPM MF bank;
2: Input: $\{\lambda_i(u;\mathrm{I})\}$, for $-N_{\mathrm{WU}} \le i \le N+N_{\mathrm{WU}}-1$, the a priori LLRs from the LDPC decoder;
3: Output: $\{\lambda_i(u;\mathrm{O})\}$, for $0 \le i \le N-1$, the extrinsic a posteriori LLRs;
4: Initialization: $A_{-N'_{\mathrm{WU}}-1}(s)=0$ for $0 \le s \le N_{\mathrm{S}}-1$;
5: Initialization: $B_{N'+N'_{\mathrm{WU}}-1}(s)=0$ for $0 \le s \le N_{\mathrm{S}}-1$;
   Metric Increment:
6: $\boldsymbol{\lambda}_n(\mathbf{u};\mathrm{I}) = [\lambda_n(u;\mathrm{I})]^T$,   for all $n$;
7: $\gamma_n(e) = -[\boldsymbol{\lambda}_n(\mathbf{u};\mathrm{I})]^T a_2(\mathbf{u}(e))/2 + p_n(C(e);\mathrm{I})$;
   Forward and Backward Recursions:
8: for $n=-N'_{\mathrm{WU}},\dots,N'-3,N'-2$ do
9:     $A_n(s^{\mathrm{E}}) = \max^{\star}_{e\in\mathbf{e}(s^{\mathrm{E}})} \{A_{n-1}(s^{\mathrm{S}}(e)) + \gamma_n(e)\},\ \forall s^{\mathrm{E}}$;
10:    Normalize $\{A_n(s)\}_{s=0}^{N_{\mathrm{S}}-1}$
11: end for
12: for $n=N'+N'_{\mathrm{WU}}-2,\dots,1,0$ do
13:    $B_n(s^{\mathrm{S}}) = \max^{\star}_{e\in\mathbf{e}(s^{\mathrm{S}})} \{B_{n+1}(s^{\mathrm{E}}(e))+\gamma_{n+1}(e)\},\ \forall s^{\mathrm{S}}$;
14:    Normalize $\{B_n(s)\}_{s=0}^{N_{\mathrm{S}}-1}$
15: end for
    Extrinsic Increment:
16: $\chi_n(e) = p_n(C(e);\mathrm{I})$;
    Completion Step:
17: for $n=0,1,\dots,N'-1$ do
18:    $\lambda_n(u;\mathrm{O}) =$
19:    $\max^{\star}_{e:u(e)=0}\{A_{n-1}(s^{\mathrm{S}}(e)) + \chi_n(e) + B_n(s^{\mathrm{E}}(e))\}$
20:    $-\max^{\star}_{e:u(e)=1}\{A_{n-1}(s^{\mathrm{S}}(e)) + \chi_n(e) + B_n(s^{\mathrm{E}}(e))\}$;
21: end for
```

Algorithm 2 Log-Based SISO APP Algorithm for $M=4$ ($n_0=2$) and a time-varying trellis.

```
1: Input: $\{p_n(C(e);\mathrm{I})\}$, for $-N'_{\mathrm{WU}} \le n \le N'+N'_{\mathrm{WU}}-1$, the received a priori PMFs from the CPM MF bank;
2: Input: $\{\lambda_i(u;\mathrm{I})\}$, for $-N_{\mathrm{WU}} \le i \le N+N_{\mathrm{WU}}-1$, the a priori LLRs from the LDPC decoder;
3: Output: $\{\lambda_i(u;\mathrm{O})\}$, for $0 \le i \le N-1$, the extrinsic a posteriori LLRs;
4: Initialization: $A_{-N'_{\mathrm{WU}}-1}(s)=0$ for $0 \le s \le N_{\mathrm{S}}-1$;
5: Initialization: $B_{N'+N'_{\mathrm{WU}}-1}(s)=0$ for $0 \le s \le N_{\mathrm{S}}-1$;
   Metric Increment:
6: $\boldsymbol{\lambda}_n^{(0)}(\mathbf{u};\mathrm{I}) = [\lambda_{2n}(u;\mathrm{I}),\quad 0\quad]^T$,   for all $n$;
7: $\boldsymbol{\lambda}_n^{(1)}(\mathbf{u};\mathrm{I}) = [\quad 0,\quad \lambda_{2n+1}(u;\mathrm{I})]^T$,   for all $n$;
8: $\boldsymbol{\lambda}_n(\mathbf{u};\mathrm{I}) = \boldsymbol{\lambda}_n^{(0)}(\mathbf{u};\mathrm{I}) + \boldsymbol{\lambda}_n^{(1)}(\mathbf{u};\mathrm{I})$;
9: $\gamma_n(e) = -[\boldsymbol{\lambda}_n(\mathbf{u};\mathrm{I})]^T a_2(\mathbf{u}(e))/2 + p_n(C(e);\mathrm{I})$;
   Forward and Backward Recursions:
10: for $n=-N'_{\mathrm{WU}},\dots,N'-3,N'-2$ do
11:    $A_n(s^{\mathrm{E}}) = \max^{\star}_{e\in\mathbf{e}(s^{\mathrm{E}})} \{A_{n-1}(s_n^{\mathrm{S}}(e)) + \gamma_n(e)\},\ \forall s^{\mathrm{E}}$;
12:    Normalize $\{A_n(s)\}_{s=0}^{N_{\mathrm{S}}-1}$
13: end for
14: for $n=N'+N'_{\mathrm{WU}}-2,\dots,1,0$ do
15:    $B_n(s^{\mathrm{S}}) = \max^{\star}_{e\in\mathbf{e}(s^{\mathrm{S}})} \{B_{n+1}(s_n^{\mathrm{E}}(e))+\gamma_{n+1}(e)\},\ \forall s^{\mathrm{S}}$;
16:    Normalize $\{B_n(s)\}_{s=0}^{N_{\mathrm{S}}-1}$
17: end for
    Extrinsic Increments:
18: $\chi_n^{(0)}(e) = -[\boldsymbol{\lambda}_n^{(1)}(\mathbf{u};\mathrm{I})]^T a_2(\mathbf{u}(e))/2 + p_n(C(e);\mathrm{I})$;
19: $\chi_n^{(1)}(e) = -[\boldsymbol{\lambda}_n^{(0)}(\mathbf{u};\mathrm{I})]^T a_2(\mathbf{u}(e))/2 + p_n(C(e);\mathrm{I})$;
    Completion Step:
20: for $n=0,1,\dots,N'-1$ do
21:    $\lambda_{2n}(u;\mathrm{O}) =$
22:    $\max^{\star}_{e:u_0(e)=0}\{A_{n-1}(s_n^{\mathrm{S}}(e)) + \chi_n^{(0)}(e) + B_n(s_n^{\mathrm{E}}(e))\}$
23:    $-\max^{\star}_{e:u_0(e)=1}\{A_{n-1}(s_n^{\mathrm{S}}(e)) + \chi_n^{(0)}(e) + B_n(s_n^{\mathrm{E}}(e))\}$;
24:    $\lambda_{2n+1}(u;\mathrm{O}) =$
25:    $\max^{\star}_{e:u_1(e)=0}\{A_{n-1}(s_n^{\mathrm{S}}(e)) + \chi_n^{(1)}(e) + B_n(s_n^{\mathrm{E}}(e))\}$
26:    $-\max^{\star}_{e:u_1(e)=1}\{A_{n-1}(s_n^{\mathrm{S}}(e)) + \chi_n^{(1)}(e) + B_n(s_n^{\mathrm{E}}(e))\}$;
27: end for
```

ACKNOWLEDGMENT

This work was supported by the Spectrum Relocation Fund (SRF) under the project "Forward Error Correction Codes for IRIG-106 CPM Waveforms." The author would like to thank Kip Temple, Bob Selbrede, and Program Manager Bobbie Wheaton.