HIGHLY PARALLEL ARITHMETIC CODING

Robert A. Freking and Keshab K. Parhi

Dept of ECE, University of Minnesota
200 Union St. SE
Minneapolis, MN 55455-0154
{rfreking, parhi}@ece.umn.edu

ABSTRACT

A concurrent technique for arithmetic coding is detailed which permits very high parallelism factors. Remarkably, because serial interprocessor communication has been eliminated, essentially linear speedup is achieved. Full processor utilization is maintained throughout execution by a procedure that promotes automatic load balancing. Because the method does not impose significant restrictions on the style of the arithmetic-coding arrangement, state-of-the-art enhancements such as the promising multiplication-free schemes may be easily incorporated. Although the effect on performance is dramatic, the key processes supporting this solution are straightforwardly realized. The method is suitable for direct implementation in stand-alone hardware or on a general-purpose parallel computer.

1. INTRODUCTION

If the relative merit of compression algorithms was judged solely by attainable compression ratios, arithmetic coding, which boasts essentially entropic optimality, would surely have supplanted its elder rival, Huffman coding, long ago. That this is not so is primarily the outgrowth of performance issues. True to its title, arithmetic coding is processed with the aid of an assortment of full-word arithmetic evaluations, traditionally including multiplication and division. When contrasted with the simplicity of competing compression techniques, these operations are justifiably deemed extravagant. Much effort has been devoted to purging the coding procedure of this computational penalty without unduly impacting its compressive efficacy. It has been suggested in numerous published variants that the more involved multiplication and division operations may be replaced by approximations demanding only a few additions and shifts. Even so, other difficulties attend this method. Most notably, irregularity between the cycle periods of the two constituent processes, compounding localization and scaling, induces a sporadic data-flow pattern that is ill suited to streamed communications. For this reason, arithmetic coding is most often employed in scenarios that admit some form of storage or significant buffering.

In modern design, parallelism is widely recognized as a standard algorithmic transformation for rate enhancement. It is therefore not surprising to find attempts to improve the unremarkable performance characteristics of arithmetic coding with a concurrent approach. Two like-minded encoding developments in the literature rely on expansion and collection of terms of the governing recurrences. The first [1] bisects a sequence of symbols, producing two half-size sequences which are themselves subject to iterative splitting. Continuing until only indivisible symbols remain, a processing dependency in the form of a tree is developed. A derivative of this decomposition process is found in the second offering [2], where the symbol sequence is partitioned only once into multiple indentical-size subsequences. While the latter work omits discussion of the critical mechanism of global normalization, the former work does propose a solution. However, the technique evidences complexities not present in a serial encoder. Of greater concern than the normalization issue is the absence of a decoding counterpart to the parallel encoder. Since decoding is more likely rendered on slower end-user-grade technology, the speed increase delivered by a parallel encoder will prove moot at the decoder. Clearly, a more comprehensive approach is necessary.

In [3] the popular compilation enhancements of loop unrolling and speculative execution are applied to the coding algorithm. As would be expected with such a methodology, the attainable speedup is data dependent. A block-based technique is introduced in [4], where a long sequence of data is separated into approximately equal portions with each assigned to a distinct processing element (PE). Individual coding operations persist over variable durations as a consequence of their

This work was supported by the Army Research Office.

data dependency. As a result, PEs will complete their data allotment on an uncoordinated and unpredictable schedule. Those that consummate early are reassigned to relieve PEs which were apportioned a more computationally intensive set of data. This load balancing effort contributes significant complexity to control tasks. Moreover, the intricacy of PRAM memory is required in support of an almost random data-production order. Processing in blocked increments, data is not feasibly accessible until after all decoding is completed.

As has already been suggested, concurrency at the encoder that is not at least matched at the decoder is of limited value. The method of parallelizing explored in this paper achieves such parity. Developed as an extension of a particularly effective parallel Huffman coding technique [5], the arithmetic coding version retains many of the benefits of its predecessor. In particular, very large parallelization factors are possible, speedup is linearly proportional to the number of PEs involved and decoded results are made available at a timely pace. However, intrinsic dissimilarities between the two algorithms force certain restrictions that are not present in the Huffman method. Most significantly, the present application applies to closed systems, where the decoding process is bound to specific characteristics of the encoding process. Migration of encoded data to diverse systems is possible only indirectly under this scheme: as a first step, decoding within the framework of the generating system is necessary. This may not prove as severe a limitation as might be initially suspected since arithmetic coding commonly serves as an adjunct to specialized modes of data handling such as those related to image compression, where short-term, high-rate content delivery may take priority over long-term translational opportunities. Given the many offsetting benefits of this technique, it is reasonable to envision its potential for private networking and dedicated or embedded systems.

The ensuing discussion of this method is arranged as follows. A summary of the essential elements of arithmetic coding is presented in §2, in the process introducing the pertinent notational conventions of this paper. The parallelizing method is introduced and fully detailed in §3, with practical considerations and observations provided in §4. finally, §5 concludes the paper.

2. BACKGROUND

In the following exposition, basic familiarity with the principles of arithmetic coding is assumed. For a thorough guide to the relevant concepts see [6]. A brief recollection of the key points is, nonetheless, in order.

The core of arithmetic encoding, the compounding localization process, is described by the two relations,

$$A_{i+1} = A_i \cdot p_i(s_i)$$

and

$$L_{i+1} = L_i + A_i \cdot q_i(s_i),$$

representing the extent and low codepoint, respectively, of the range undergoing recursive refinement. Here p_i denotes the probability of the symbol s_i at time i and q_i symbolizes the summation of all probabilities associated with symbols assigned ordinals ahead of s_i. Since probabilities are often modeled with the aid of integer counts, it should be understood that p and q may incorporate a division for purposes of normalization. Even if the two expressions are evaluated simultaneously, a delay of one addition and one or more higher-order operations is incurred.

For any practical implementation, the preceding set of relations is inadequate since the necessary level of numerical precision grows with the depth of iteration. A scaling process, which usually entails no more than simple shifts, is required to confine precision. Although trivial and, furthermore, necessary for incremental output, this adjustment imparts timing variations that make the encoding difficult to stream and hence to parallelize as well.

Decoding essentially replicates the encoding process, but with one fundamental augmentation: the coded symbol is first recovered with a simple search. This is typically initiated with a normalization consisting of one or more higher-order operations, e.g., in the worst case a multiplication and a division is involved. Without admitting approximations, two multiplications, two divisions, an addition and a fluctuating number of shifts are endured for each symbol decoded.

Since the method of parallelism to be described is patterned after a method of the same effect in Huffman coding, dissimilarities among these codings must be considered. In contrast with arithmetic coding, a Huffman coding system is capable of transmitting an uninterrupted bitstream from encoder to decoder. This is so because both encoding and decoding may be performed on an incremental basis wherein each progression consumes constant and uniform time. As a result of temporal homogeneity, it is possible to devise procedures referenced solely on bit ordinals, e.g., the mentioned parallelization technique. Arithmetic coding—comprising distinct processes of unequal duration, some of which do not even produce output—affords no such independence from the timing peculiarities of the implementation. In fact, encoding and decoding must be deliberately correlated. For this reason, a solution of the sort to be outlined next must be confined to the

realm of closed systems, where both encoder and decoder are of related design.

Consider a representative arithmetic encoding step. The extent and low codepoint are updated with division, multiplication and addition. Subsequently, b instances of scaling result eventually in b bits of output, though all b bits will not be expelled in the current iteration if an underflow prevention procedure is activated. The corresponding decoding step accepts enough bits to discern the next symbol. Having decoded, the symbol is applied in a process mimicking the encoder. Let the following notation be adopted for the delays associated with the primary processes: compounding localization, τ_l; scaling, τ_s; and decoding search, τ_d. Symbolically, encoding commands $\tau_l + b\tau_s$ time units while the decoding period is $\tau_d + \tau_l + b\tau_s$. The additional τ_d term will prove troublesome for parallelization.

3. PARALLELIZATION

An idealized physical model for concurrency in this situation may be conceptualized as a composition of p independent coding systems, each consisting of an encoder-decoder pair linked by a private means of communication. This extreme degree of autonomy engenders the most compelling attributes of parallelism, including full processor utilization and unlimited scalability with PE count. Achieving such separation in actuality is not trivial and, in fact, not necessarily ensured. Nevertheless, to attain high-performance parallelism it is crucial that PE tasks be as independent as possible. As affirmed by Amdahl's Law, it is interprocessor communication that is principally responsible for the rapid degeneration into saturation observed in most parallel implementations.

The model just described can be brought nearer to practicality by including some means of distributing a single data stream among the p independent channels as well as a complimentary means of reassembling the data stream from p sources at the output, as depicted in fig. 1. As will be elucidated shortly, these introductions entail unexpected complexity. Also in the same figure, a data transport layer has been indicated. Although a necessary concession to the realities of communication, this addition leaves the effective connectivity intact. This refined model constitutes the essence of our approach to parallelism. Solidifying the functional definition of the three unspecified components—stream segmentation, channel sharing and stream reassembly—in a unified and efficient manner will occupy the remainder of this paper.

As presented, the nature of data transport layer is distinct from the stream related manipulations. In fact,

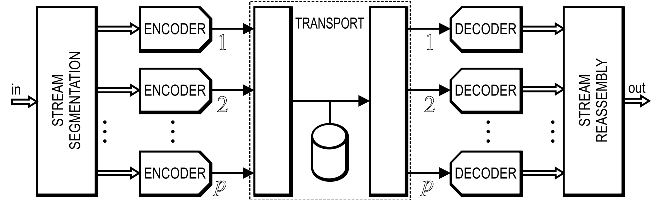

Figure 1: Parallel system showing key components

it is closely bound to the underlying architectural arrangement. Dedicated hardware most likely suggests a more formal communication environment, where a single serial channel must relay all p substreams. To maintain correspondence between encoders and decoders, time-division multiplexing (TDM) with one-bit blocks may be applied. By contrast, if the parallel coding system is implemented on a general-purpose parallel-computing platform, intermediate storage can be leveraged to fulfill the transport layer. Specifically, a dataword may be formed by juxtaposing one bit of data from each encoder. When recalled from storage, the dataword is dispersed among the decoders in a process mirroring the withdrawal from the encoders. Associations are, hence, preserved in somewhat automatic fashion since most parallel computers natively provide the bit-broadcast operations necessary to collect and disperse such datawords. While not mandatory for arithmetic coding, temporal correspondence may also be achieved with either TDM or parallel storage. In what follows, a general-purpose-computing setting will be assumed since arithmetic coding is poorly suited to the punctual rigors of the serial channel environment.

For the present discussion, it must be presumed that the unparallelized input data stream possesses no inherent separability. Therefore, any segmentation scheme which yields a series of substreams for independent processing cannot rely on permanent independence. Rather, the original stream must be reconstructed at the output from those substreams. Unfortunately, if encoding and decoding are permitted to consume periods differing nonlinearly in time, no method for reunifying the distinct substreams of data without interprocessor communication would exist. In fact, while it need not be rigidly maintained throughout, temporal correlation must be revived at the decoders if simple collective and dispersive operations are to be sufficient vehicles for data transfer. As it turns out, with proper encoding and adequate intermediate buffering the decoders themselves can recreate this correlation.

While options for stream segmentation schemes are many, only the one that achieves optimal utilization of

all PEs is of interest. Assigning the next symbol to the next available PE accomplishes the desired effect. However, to enable later reassembly it is necessary to subject this process to a further constraint: when more than one PE is available assignment proceeds in accordance with PE precedence. Whether lower ordinals take priority over higher ordinals or vice versa is irrelevant so long as the same convention is followed at the decoder. Precedence-based assignment appears to threaten parallelism by establishing a serial availability-status query, but this need not be the case. Instead, the bit-broadcast facility can be utilized with every PE contributing an availability status bit. Each PE can interpret the received dataword by bit-count/prefix-sum operations synthesized with lookup tables and, accordingly, identify the next symbol for encoding from among all symbols pre-loaded into each PEs local memory.

As a consequence of the additional search step, the decoders, working as rapidly as possible, would be unable to duplicate the schedule of the encoders. The best that could be hoped for would be operation at a rate which is a slower multiple of the encoding rate. This can be accomplished by considering the search and compounding localization steps as a single operation. Note that unless the period of the search is compulsorily constant irrespective of the complexity of the particular search, temporal correspondence would be unattainable. The delay associated with the maximum number of search iterations is therefore allotted. Having fixed the period of a composite decoding operation, the scaling steps must be slowed proportionately. Thus, a scaling step consumes $\left[\left(\max(\tau_d) + \tau_l\right)/\tau_l\right]\tau_s$ times units. Because τ_s is relatively small the penalty associated with this approach is modest. An alternative permits full-rate decoding by modifying the encoder. Since encoding and decoding differ in delay by $\max(\tau_d)$, if the encoding process is idled for this duration every iteration, the processes will operate on identical schedules, namely, the fastest decoding schedule. This is arguably the preferable method since it is unusual that encoding would assume a significance superior to that of decoding. fig. 2 illustrates the two timing options.

The imposed scheduling conformity induces the necessary temporal correlation enabling stream reassembly without the exchange of coordinating information among PEs. The instant at which a decoded symbol is released is critical. In particular, it must be ejected after a constant delay from the initiation of decoding. Since the initiation of a decoding cycle accurately reflects the moment of assignment at the corresponding encoder, the assignment ordering is implicitly extant

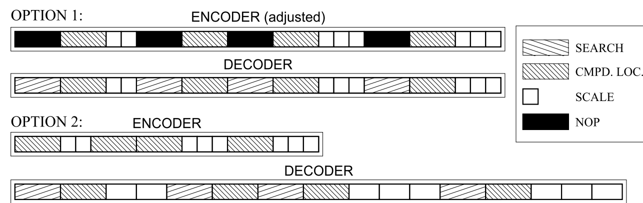

Figure 2: Scheduling options for temporal correspondence.

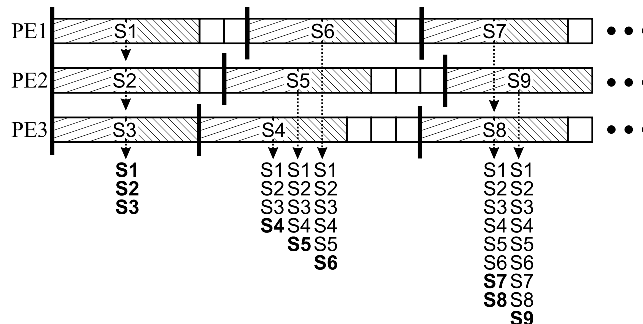

Figure 3: Because the decoder process replicates the timing of the encoder, symbols—which are discovered and output after a fixed search period (as shown)—are expelled in the same order they were assigned at the encoder.

in the decoded symbol record. If symbols are removed consistent with the PE precedence specified at the encoders, the original symbol stream is recovered. A pictorial overview summarizing the process is presented in fig. 3.

4. PRACTICAL CONSIDERATIONS

If encoding is conformed to decoding, as is recommended, then all decoding PEs are 100% utilized. Conversely, encoding PEs benefit from 100% utilization if decoding is tuned to a multiple of the timing of encoding. In either case, buffering is required to ensure an ample supply of codebits at the decoders. Since codebits are produced sporadically, storage routines must wait until every PE has contributed a bit before archiving the resultant dataword. In the meantime some PEs will generate a potentially large number of bits awaiting output. Even in standard serial implementations, intermittent data flow has always plagued arithmetic coding. Because the question of buffering here is equivalent to the oft-addressed problem of buffering for continuous flow in serial coder, it will not be considered in this paper. However, it should be noted that

each PE could simply provide enough local memory to receive the encoded sequence in its entirety. Afterward, in a post-process, datawords could be developed in a straightforward manner. It must be recognized that, as termination nears, variations in substream length prompt the interposition of arbitrary placeholder bits in positions corresponding to PEs that have exhausted their streams. Barring an abnormal condition in which the substreams are appreciably unbalanced, this inefficiency should be confined to a short period. In exchange for this slight degradation, unwieldy PRAM memory as well as the complexities of buffering has been avoided. While the post-process must endure minor difficulties, availability-based assignment has automatically assured load balancing for the main encoding and decoding processes without involved regulation efforts.

With the described parallelism an adaptive model is appropriate for setting probability ranges since each substream is encoded individually. Because an initialization phase elapses before a proper context is assembled, this method would not be suited to coding short data sets. Further reinforcing this restriction is the irregularity that occurs toward the end of a run wherein dataword formation introduces compression overhead via the aforementioned placeholder bits. Moreover, the independence of substreams dictates the encoding of a separate end symbol for each substream. These compression-rate burdens must be amortized over a reasonably large data set before they may be safely neglected.

The maximum parallelism factor is not limited by interprocessor communication as it is in many of the more traditional approaches. Instead architectural issues suggest practical upper bounds. The native bit-broadcast operations that undergird this technique are normally available only in units of the computer's dataword size. Hence, a parallelism factor equal to the dataword size yields the optimum processing efficiency. This optimum notwithstanding, advantages remain at higher parallelism factors—especially multiples of the dataword size. Thus, parallelism up to the number of available PEs is feasible.

5. CONCLUSIONS

A method for concurrent arithmetic coding has been described. Featuring up to 100% PE utilization without explicit load-balancing and high, if not unlimited parallelism factors, this approach delivers benefits not attainable by other schemes. Because encoding and decoding must be matched, the technique should be implemented in a closed system. Minor substream-length disparities may either be solved by buffering or with the admission of a limited amount of compression overhead near the end of a run. The extent of the latter can be mitigated by defraying the inefficiency over a long symbol sequences. Other factors also suggest the appropriateness of the method for lengthier sequences. It is notable that this technique does not impose any restrictions on the structure of the core coding operations. As a result, varied sophisticated arrangements such as multiplication-free constructions may be employed. Perhaps most intriguing, the lack of interprocessor communication enables essentially linear speedup, i.e., p PEs can encode/decode p times faster than a single PE. This sort of superior rate enhancement exemplifies both the obligation and the appeal of parallelism.

REFERENCES

[1] J. Jiang and S. Jones, "Parallel design of arithmetic coding", IEE Proc. E: Comp. & Dig. Tech., pp.327-33, vol.141, no.6, 1994.

[2] H. Y. Lee., et. al., "A parallel architecture for arithmetic coding and its VLSI implementation", in *Proc. 39th Midwest Symp. Circ. Syst*, vol.3, pp.1309-12, Ames, IA, 1996

[3] G. Feygin, P. G. Gulak and P. Chow, "Architectural advances in the VLSI implementation of arithmetic coding for binary image compression", in *Proc. 1994 Data Compression Conf.*, pp.254-63, Snowbird, UT, Mar. 1994.

[4] P. G. Howard and J. S. Vitter, "Parallel lossless image compression using Huffman and arithmetic coding", in *Proc. 1992 Data Compression Conf.*, pp. 299-308, Snowbird, UT, Mar. 1992.

[5] R. A. Freking and K. K. Parhi, "An unrestrictedly parallel scheme for ultra-high-rate reprogrammable Huffman coding", in *Proc. Int. Conf. Acous. Speech Sig. Process., 1999*, vol. 4, pp 1937-40, Phoenix, March,1999.

[6] I. H. Witten, R. M. Neal and J. G. Cleary, "Arithmetic coding for data compression", Commun. ACM, vol. 30, no. 6, pp. 520-40, 1987.