Using PVM to Hunt Maximal Snakes in Hypercubes

Brian Paul Rickabaugh[1] and Anil M. Shende[1]

Department of Mathematics, Computer Science, and Physics
Roanoke College
Salem, VA 24153, USA
Phone: +1 540 3752341
FAX: +1 540 3894236
{brickaba,shende}@roanoke.edu

Abstract. Currently, exhaustive search is the only known method for finding longest induced paths in hypercubes. Based on empirical data we conjecture that extending suitable maximal, but not necessarily longest, snakes in an n-dimensional hypercube is a tractable alternative for generating longest snakes in an $(n+1)$-dimensional hypercube. In this paper, we present our use of PVM in generating maximal snakes.

1 Introduction

The problem of finding the length of a longest *snake*, that is, an induced path, in a hypercube was posed in 1958 [Kau58] in the context of coding theory, and is still an open problem. Subsequently this problem has been shown to be of importance in connection with several other applied problems, e.g., electronic combination locking schemes (see, for example, [Bla64, CFT64, Kle70, Jab74, PT96] for discussions on some of these applications).

In this paper we introduce the notion of *maximal* snakes and describe our use of PVM in the hunt for longest snakes in hypercubes using maximal snakes. Unlike the approach described in [PRM+94], our approach is *not* heuristic, and is essentially a method to parallelize an inherently sequential exhaustive search. We believe that this method can be used for solving, using PVM, other similar problems where the only known solution involves an inherently sequential exhaustive search.

2 Snakes in Hypercubes

A *snake in an n-dimensional hypercube* is an *induced path* in the n-dimensional hypercubical, undirected graph, denoted H^n. The *length* of a snake is, by definition, the number of edges in the snake. Longest snakes in three and four dimensional hypercubes are shown in Figure 1. (A related notion is that of a *coil in H^n*, which is, by definition, an *induced cycle* in the H^n. [AK88, HHW88])

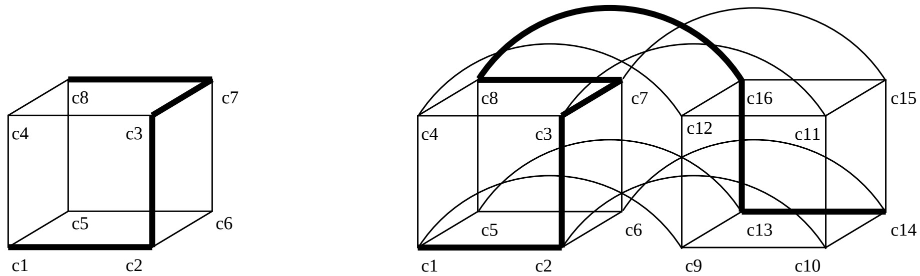

Fig. 1. Longest Snakes in 3 and 4-dimensional Hypercubes

Each of the 2^n vertices of H^n is labelled with a unique sequence of n bits, and for any two vertices with labels v and w, (v, w) is an edge in H^n iff v and w differ in exactly one bit. Without loss of generality, we will assume that the first vertex of every snake is the vertex labelled by n 0's. We will denote an edge in the snake by a number $i, 0 \leq i < n$ so that the labels assigned to the two vertices making up the edge differ in the i^{th} bit (the 0^{th} bit being the rightmost bit in the label). For any path P in H^n, for each $i, 0 \leq i < n$, $P \cdot i$ will denote an *extension* of P by one edge; this edge is between the last vertex v of P and the vertex w in H^n whose label differs from the label of v in the i^{th} bit. ($i \cdot P$ is defined similarly.) A snake S is *maximal* in H^n iff, by definition, for all $i, 0 \leq i < n$, neither $i \cdot S$ nor $S \cdot i$ are snakes in H^n. Note that a maximal snake may not be a longest snake in a hypercube. For example, (see Figure 1) the snake $c1, c2, c3, c7, c8, c16, c12$ is a maximal, but not a longest, snake in H^4. $s(n)$ denotes the length of a *longest snake* in H^n. (Similarly, $c(n)$ denotes the length of a *longest coil* in H^n.)

Explicitly *finding* a longest snake in H^n seems to be intractable with state of the art technology (see [PRM$^+$94]). Furthermore, even simply *determining* the value for $s(n)$ has proven to be a notoriously difficult problem; the exact values for $s(n)$ are known only for dimensions less than 8. Much research has gone into determining upper and lower bounds for $c(n)$, the length of a longest *coil* in H^n, (see, for example, [AK91] and [Sne94]). These bounds are not tight and are not applicable to the lengths of longest snakes in hypercubes [PRM$^+$94] except for the obvious relationship: $s(n) \geq c(n) - 1$.

Currently, exhaustive search is the only known method, albeit an intractable one, for finding longest snakes. [PRM$^+$94] reports success using a *genetic algorithm* [Hol75] in finding a longest snake (of length 50) in dimension 7. Unfortunately, this approach did not prove successful in dimension 8 [PRM$^+$94], even in a parallel computing environment using PVM [JPP94].

2.1 Why Hunt Maximal Snakes?

We started with the naive, intuitive idea that since H^n is constructed from two copies of $H^{(n-1)}$, a longest snake in H^n might be an extension of a longest snake in $H^{(n-1)}$. Notice that the longest snake, shown in Figure 1, in H^4 *is* an extension of a longest snake in H^3. Analysis of known longest snakes in dimensions up through 6 proved this intuition faulty. For example, no longest snake in H^5 has, as a sub-snake, a longest snake in H^4. An interesting fact revealed by this analysis was that *every* longest snake in H^5 has, as an initial segment, a snake of length 6 in H^4. Furthermore, these snakes of length 6 are *maximal* in H^4. Similarly, it turns out that every longest snake in H^6 is an extension of some *maximal* snake of length 9 in H^5, and every longest snake in H^7 is an extension of some *maximal* snake of length 26, or length 25, in H^6. Based on these observations and an exhaustive list of all maximal snakes in dimensions up through 6 we conjectured that:

Conjecture 1.

1. In H^n, $n \geq 2$, for each l, $2(n-1) \leq l \leq s(n)$, there is a maximal snake of length l.
2. Every longest snake in H^n is an extension of some *maximal* snake in $H^{(n-1)}$.

Both parts of the above conjecture are true for $n \leq 6$, and remain open for $n > 6$.

Assuming Conjecture 1, starting from a maximal snake of length 46 in H^7, we generated a snake of length 97 in H^8. Prior to this, the longest known snake in H^8 was of length 89 [PRM+94]. This empirical success suggests a promising approach to generate longest snakes in H^n: generate snakes starting with a "suitable" maximal snake in $H^{(n-1)}$.

Our work presented in this paper is motivated by our desire to gather adequate data *in reasonable time* to:

1. resolve the above conjecture for all n, and
2. to try and characterize the "suitable" maximal snakes.

3 Generating Maximal Snakes

Given the high symmetry of a hypercube, each snake in a hypercube has several other snakes obtained by permuting the dimensions of the hypercube. The following notion of *progressive* snakes and Proposition 1 suggest a method for generating non-isomorphic snakes up to reversals.

Suppose S is a snake in an H^n. For each $d, 0 \leq d < n$, let $p_S(d)$ be the first position of d in the snake S. Then, S is a *progressive* snake, by definition, iff for each $d, d', 0 \leq d < d' < n, p_S(d) < p_S(d')$. Proposition 1 then follows from the definition of progressive snakes.

Proposition 1. *In H^n, for every snake S', there exists a unique, up to reversal, progressive snake S that is isomorphic to S'.* □

Thus it suffices to generate all and only progressive snakes.

The following useful Corollary is also immediate from the definition of progressive snakes.

Corollary 1. *Every progressive snake has* 012 *as its initial segment.* □

3.1 Generating Maximal Snakes in Parallel

We use a master-slave model for this PVM application. Our master process coordinates slave processes and the slave processes perform the actual generation of snakes. The master maintains two FIFO queues, a snake registry (`sr`) which stores snakes that need to be sent to the slaves, and a request queue (`rq`) which stores the id numbers of idle slaves waiting for snakes. The master receives 3 types of messages: `SNAKE_RECEIVE` indicating a slave has sent a snake to the master, `SNAKE_REQUEST` indicating that a slave is requesting a new base snake, and `SNAKE_MAXIMAL` indicating that a slave has found a maximal snake.

Some slave receives from the master the initial empty snake as its first base snake. We will refer to this slave as the *first slave*. Subsequently, other slaves receive from the master other snakes as base snakes. The slave processes depend on a predetermined constant $\delta > 0$. Each slave process begins with its base snake S_0 with length l. It extends S_0 by δ edges to obtain a snake of length $l + \delta$, then extends the snake by 1 more edge, and then registers with the master process this snake, say S_1, of length $l + \delta + 1$. This process is repeated for S_1 to register a snake S_2 and so on. Clearly for each i the length of S_{i+1} = length of $S_i + \delta + 1$.

Below are sketches of the algorithms for our master and slave processes, respectively. This version shall be referred to as "Version 1".

Algorithm for the Master

```
Initialize the Snake Registry with an empty snake;
Create Slaves; done = false;
while (!done){
  while (sr is not empty and rq is not empty){
    i = topq (rq); dequeue (rq);
    S = topq (sq); dequeue (sq);
    send S to slave with ID = i;
  }
  wait for a message from a slave;
  switch (message type)
  {
    SNAKE_RECEIVE: S = received snake;
                   enqueue (sr, S); break;
    SNAKE_REQUEST: i = ID of slave making the request;
                   enqueue (rq, i); break;
    SNAKE_MAXIMAL:  report Maximal Snake; break;
  }
  done = ((sr == empty) and (rq == empty));
}
```

Algorithm for Each Slave

```
while (true){
  wait for a base snake;
  S = received snake;
  if (S == empty) then Extend (``012'', 0, 0);
  else{
    let S be S'.d;
    Extend (S', 0, (d+1));
  }
}
```

Extend (S, dist, startdim)

```
if (dist == delta){
  let d be the least non-negative number < n such that S.d is a
          progressive snake;
  register S.d with the master;
  Extend (S.d, 0, 0);
}
else
  for (d = startdim; d < n; d++)
    if (S.d is a progressive snake)
      Extend (S.d, (dist++), 0);
```

It is important to note here that

Proposition 2.

1. *All and only progressive maximal snakes are generated by the above algorithm, and*
2. *no snake is generated more than once.*

Proof.

1. By Corollary 1, it suffices to start the generation process with the snake 012, as is done by the first slave. Past that point, by construction, only progressive snakes are considered by the slaves.
2. Suppose a slave process gets a base snake S with length l. This slave process considers every possible extension to S except for extensions that have a length of $l + k \cdot (\delta + 1)$, $k > 0$. Each of these extensions is registered with the master process and is eventually sent to some slave process as a base snake. Thus, eventually, *every* possible extension of S is considered.
3. By construction, no two slave processes start with the same base snake. Furthermore, if one base snake S is an initial segment of another base snake S', then the slave process that gets S will differ from S' in its first extension (see the `else` clause in the main algorithm for the slave process above).

The above three items collectively prove both the statements in the proposition. $\square$

3.2 Revision of the Algorithm

After analysis of our data from the trials using Version 1, we wanted to reduce processing time and increase the processing ratio. A possible means to achieve this goal was a reduction in the number of messages passed and the idle time. We altered the slave process to include a local snake registry (lsr). When registering a snake S, the slave process first tries to register S in its lsr. If its lsr is full, the slave process registers S with the master. When a slave process needs a new base snake, it first uses a snake, if any, from its lsr before requesting a snake from the master. This revision shall be referred to as "Version 2."

4 Experimental Conclusions

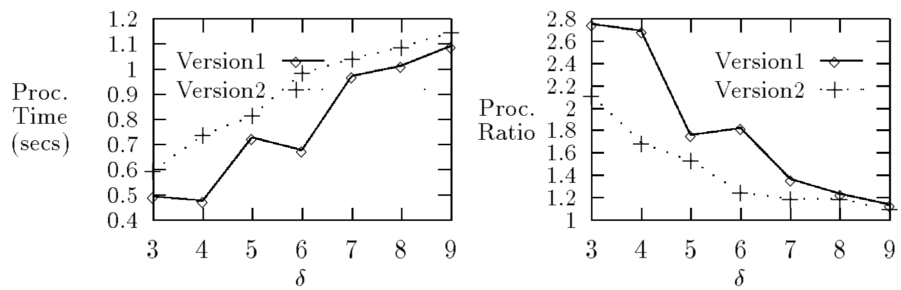

Fig. 2. Running Times and Processing Ratios for Dimension 5

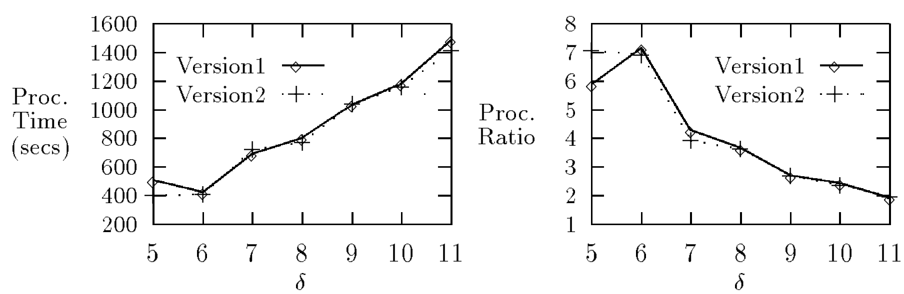

Fig. 3. Running Times and Processing Ratios for Dimension 6

Figures 2 and 3 show statistics obtained by running both versions for $n = 5$ and $n = 6$, respectively. Each run was executed on a PVM with 10 hosts and 10 slave processes. The figures show processing times and processing ratios obtained for each version with varying δ; $3 \leq \delta \leq 9$ for $n = 5$ and $5 \leq \delta \leq 11$ for $n = 6$. Processing Time is defined to be the total amount of time, in seconds, the run took to execute including slave idle time, but not including the time it took to populate the PVM with slave processes. Processing Ratio is defined to be the ratio of the cumulative processing time, not including idle time, of all the slaves and the total processing time.

It is important to note that for each version and n there exists a δ that is favourable to decreasing processing time. In general, as δ increased, processing time increased, and processing ratio decreased. It is also apparent that, contrary to our expectations, Version 2 produced the same or worse results in our trials.

δ	Dimension 5; Version 1; Number of Trials per Slave Process									
3	635	431	269	173	171	167	143	143	127	78
4	600	386	377	371	371	68	68	37	34	25
5	1361	219	200	176	159	116	75	21	5	5
6	1245	181	178	174	136	114	100	88	61	60
7	1669	119	110	87	69	69	66	59	52	37
8	1893	89	63	58	54	53	53	39	23	12
9	2016	59	46	41	38	35	34	28	28	12
δ	Dimension 5; Version 2; Number of Trials per Slave Process									
3	804	707	168	124	118	106	84	83	77	66
4	1395	362	344	93	48	34	31	11	10	9
5	1533	222	175	134	99	68	66	21	16	3
6	1893	133	129	64	43	26	19	15	11	4
7	1984	99	59	55	47	33	21	14	13	12
8	1954	74	60	54	51	50	36	28	21	9
9	2129	43	37	30	29	26	26	8	5	4

Table 1. Trials per Slave in Dimension 5

δ	Dimension 6; Version 1; Number of Trials per Slave Process									
5	776612	559677	532960	532033	512006	495286	479088	474127	446178	426629
6	771522	531918	527818	518040	494175	489436	485682	472888	471852	471265
7	1254970	485526	475390	461839	459544	452653	418598	416598	416005	393474
8	1476660	447410	443649	441184	429089	421991	418075	412170	389553	354813
9	1994880	397700	391087	381980	378334	355781	348687	347522	323988	314634
10	2201000	365925	363580	351531	348329	335406	320526	320067	317396	310836
11	2624210	309390	295695	295139	294920	289723	286457	281863	279685	277511
δ	Dimension 6; Version 2; Number of Trials per Slave Process									
5	749178	537179	532485	512060	511048	509196	499088	498552	495618	390192
6	769981	552773	545269	527894	513533	505435	490117	462202	440364	427028
7	1297840	526878	478255	457089	427263	422677	421980	409122	407851	385639
8	1487560	451554	447256	444093	417693	410762	409958	397568	396193	371962
9	1998720	408672	396257	379320	370792	367645	352223	337521	317126	306322
10	2204790	366459	362165	335601	334257	330381	328905	327804	323597	320641

Table 2. Trials per Slave in Dimension 6

In order to explain this behaviour we looked at the number of trials performed by each slave process. The number of trials performed is the number of possible extensions investigated by a process. In particular, for a slave process, this is the number of times a check for a snake being progressive is performed in the function `Extend`. Tables 1 and 2 list the number of trials, in descending order, performed by the 10 slaves for $n = 5$ and $n = 6$, respectively. Notice that for some values of δ the trials were fairly evenly distributed. Nonetheless, for each δ, one slave process always dominated. However, for large values of δ the domination was significantly stronger. We determined this dominant slave to be the first slave.

Slave processes receiving relatively short base snakes obviously perform more trials than others. Hence the slave that gets the initial base snake, i.e. the first slave, performs the greatest number of trials. In Version 2, this problem is compounded by the first slave registering the first few short snakes in its own *lsr* and thus working on a comparatively large number of very short snakes.

Based on the above analysis we intend to revise our algorithm in one or more of the following ways:

1. Deactivate *lsr* for the first slave, for the initial base snake.
2. Use a smaller δ for the initial base snake.
3. Vary δ based on the length of the received base snake.

References

[AK88] H. L. Abbott and M. Katchalski. On the snake-in-the-box problem. *Journal of Combinatorial Theory*, 45:13–24, 1988.

[AK91] H. L. Abbott and M. Katchalski. On the construction of snake in the box codes. *Utilitas Mathematica*, 40:97–116, 1991.

[Bla64] W. L. Black. Electronic combination locks. Technical Report 73, Research Laboratory of Electronics, M. I. T., 1964.

[CFT64] R. T. Chien, C. V. Freiman, and D. T. Tang. Error correction and circuits on the n-cube. In *Proceedings of the 2nd Annual Allerton Conference on Circuit and System Theory*, pages 899–912, 1964.

[HHW88] F. Harary, J. P. Hayes, and H. J. Wu. A survey of the theory of hypercube graphs. *Computer Math. Applications*, 15:277–289, 1988.

[Hol75] J. H. Holland. *Adaption in Natural and Artificial Systems.* The University of Michigan Press, Ann Arbor, 1975.

[Jab74] S. V. Jablonskiĭ. Discrete mathematics and mathematical problems of cybernetics. *Nauka*, 1, 1974.

[JPP94] M. Juric, W. D. Potter, and M. Plaksin. Using the parallel virtual machine for hunting snake-in-the-box codes. In *Proceedings of the 7th Conference of the North American Transputer Users Group*, October 1994.

[Kau58] W. H. Kautz. Unit distance error checking codes. *IRE Transactions on Electronic Computers*, 7:179–180, 1958.

[Kle70] V. Klee. The use of circuit codes in analog to digital conversion. In B. Harris, editor, *Graph Theory and its Applications*. Academic Press, New York, 1970.

[PRM+94] W. D. Potter, R. W. Robinson, J. A. Miller, K. J. Kochut, and D. Z. Redys. Using the genetic algorithm to find snake-in-the-box codes. In *Proceedings of the 3rd International Symposium on Artificial Intelligence and Mathematics*, January 1994.

[PT96] Kenneth G. Paterson and Jonathan Tuliani. Some new circuit codes, 1996.

[Sne94] H. S. Snevily. The snake-in-the-box problem: a new upper bound. *Discrete Mathematics*, 133(1-3):307–314, October 1994.