A Highly Parallel Finite State Automaton Processor for Biological Pattern Matching

Glen Herrmannsfeldt

Department of Molecular Biotechnology
University of Washington
Box 357730
Seattle, WA 98195-7730
USA

e-mail: gah@mbt.washington.edu

Abstract. Finite State Automata are useful for string searching problems mostly because they are fast. For very large problems, a software implementation will not be fast enough. I describe here a parallel implementation of a hardware Deterministic Finite State Automaton processor. It can rapidly search a large database for approximately matching strings, as a filter for more detailed processing later. As the most important parts, large Random Access Memory chips, are continually getting cheaper, it should be possible and affordable to make large arrays of such processors.

Key words: finite automata, approximate string matching, high-speed searching, deterministic finite automata, massive parallelism

1 Finite State Automata for Biology

An important problem in contemporary molecular biology is sequence comparison. One would like to compare DNA or protein sequences against other DNA or protein sequences and find ones that are most similar. As the database of known DNA and protein sequences is growing exponentially, this problem is continually getting harder. It is useful to have a machine to rapidly compare a group of sequences against a large disk file of sequences and indicate which ones match most closely.

All the examples will be done using a DNA alphabet size of four. The DNA database is much larger than the protein database, and so the larger, faster, processors are needed here first. The processors should be designed to also handle the protein 20 character alphabet.

This paper describes the design of hardware implementations of Finite State Automata [HU79][W87] for processing biological sequences. In all cases described here, the Finite State Automata are Deterministic, though sometimes the acronym FSA will be used instead of DFSA or DFA.

2 The Problem with Insertions and Deletions

The preferred method for approximate sequence comparison is dynamic programming [NW70]. Preferred, that is, unless you are in a hurry. Heuristic methods, based on hashing or finite state automata are faster than dynamic programming, but are not as good at separating the biologically related sequences from statistically insignificant, but similar looking sequences.

While finding an exact match fast is relatively easy, biological problems usually don't work that way. Both natural genetic mutations and errors in sequencing can cause inserted, deleted or substituted bases. Dynamic programming is well suited to doing comparisons with a supplied similarity matrix, giving the score for aligning any database character with any query character, and with specified insertion and deletion penalty values.[G82] Applying a dynamic programming algorithm will give a score for each database sequence aligned with the query sequence, and one can select the highest scoring sequences. The limitation is that dynamic programming is slow.

A modern RISC superscalar processor can do over 100,000,000 mathematical operations (additions or comparisons) per second. Using a dynamic programming algorithm with 10 operations per matrix element, where the number of matrix elements needed is the product of the query length and database length, we can calculate over 10,000,000 matrix elements per second. With a 1,000,000,000 base DNA database, and typical 1000 base query, one search requires 100,000 seconds, or about one day. To compare the entire 10^9 base GenBank against itself would require evaluating 10^{18} matrix elements in 10^{11} seconds. This is over 3000 years, clearly not practical.

One way to speed up the calculation is with special purpose hardware. Systolic array processors work very well for dynamic programming algorithms, [LL85][CH91] and are certainly the way to go to for fast dynamic programming. With an array of 10^3 processors, each evaluating 10^7 cells per second, the GenBank comparison can be done in 10^8 seconds, or about three years. But a large fraction of the sequences being compared will have no relation to the query sequence. If one could do an even faster comparison, as a filter before running the dynamic programming algorithm, it might be possible to speed up the whole process. It should at least be possible to find the interesting results sooner.

3 Finite State Automata for Biological Searching

The BLAST program is the most popular Finite State Automata implementation for biology.[AG90][KA90][KA93] It is probably the most popular overall. BLAST can rapidly scan a database to find possible matches, and then spend more time trying to extend those matches. With the default parameters, and some luck with the choice of query, BLAST may be within a factor of two of the speed that the data can be read off the disk. BLAST is very good at finding close matches, though its performance falls of for less exact matches. There are adjustable parameters which can be used to tune the search strategy. With parameters that do a better job at finding less exact matches, BLAST will slow down.

4 A Hardware Implementation of a Finite State Automaton

Searching with a Deterministic Finite State Automaton is very simple. The new state is obtained through a lookup table from the current state and the incoming database character. In hardware, this is not much more than a large RAM array and a register. One could imagine a single printed circuit board with a large number of processor elements, each consisting of a few RAM chips and a simple programmable logic chip, each processor implementing a DFSA. With the currently popular 64 Megabit RAM chips, and more recently available 256 Megabit chips, a lookup table with 8 million entries, and running with a 10 MHz clock should be very feasible. If one or more query sequence can be compiled into an 8 million state FSA, then we could get hundreds of queries on a system at one time, and search at near the disk streaming transfer rate. That is the motivation for this paper.

If we assume a 10 million character per second streaming rate, that a 1000 character query can be compiled into a single FSA processor, with 100 processors our comparison rate is 10^{12} per second. We can complete the GenBank against itself comparison in 10^6 seconds, about twelve days. This would be fast enough to allow multiple passes, to adjust parameters, and select sequences to feed to a slower, dynamic programming processor.

5 Finite State Automaton Size

A critical parameter is the size of the FSA necessary. Finite State Automata are very good at finding an exact match, as this requires relatively few states. For biological problems, it is necessary to include some substitution data, and usually also some insertions and deletions in the comparison. This is the reason we need such large FSAs. With dynamic programming, the scoring and gap penalties are part of the algorithm, and are used to calculate the score. With an FSA, it is necessary to predetermine all the sequences we will match, including sequences with substitutions, insertions, or deletions. We need a balance between the ability to find less exact matches, and the size of the FSA needed.

If we take a typical query of 1000 characters, and we search for an exact match, we need a 1000 state FSA. (Like BLAST, we trigger on transitions and not states, reducing the required state memory.) If we want to find any 16 character substring, we have 985 substrings of 16 characters each, for 15760 states.

At this point, it will be assumed that the number of states is equal to the total number of characters in the query substrings. Fewer states will be required, because of degeneracy at the beginning of the tree. Each state requires one table entry for each character in the alphabet. For now, we calculate just the total query size, and assume this is close to the number of states needed.

If we want to find any of the 16 character substrings with one of three possible substitutions from the DNA alphabet, in any one of the 16 positions, have to search for the exact match plus 48 possible mismatched strings, for each of the 985 substrings. Thus $49 \times 985 = 48265$ strings of 16 characters each, requiring 772240 states in the FSA. If we add all the single insertion and single deletion strings to the list, we

increase by nearly a factor of three, approaching two million states.

This could be implemented as a two million entry look-up table, with each entry large enough to address the entire table, and in addition, to indicate the required result information. It takes 21 bits to address the two million entries, plus at least one bit to indicate a hit. In matching the final character, we indicate a hit and return to a lower state, the longest suffix of the matching state. In all cases where a match fails, the transition is to the longest suffix of what has been matched.

In the case of DNA, with an alphabet size of four, it is also possible to store the database with four bases per byte, for a 256 character alphabet. This increases the data transfer rate by a factor of four, but it also changes the FSA size. We reduce the length of the query subsequences by a factor of nearly four, but we still increase the number of table entries per database character by a factor of 64. For small FSA on a slow processor, like BLAST uses, this makes sense. If our query substrings were randomly distributed, it would not. However, our query substrings are not statistically independent, and in fact, the substitution/insertion/deletion model gives us many query substrings that differ in only one position. This reduces the number of states required, so it may turn out to be useful. In a hardware implementation, we do better by including more processors. If the processor can run faster than the disk transfer rate, we can unpack the data after it comes off the disk. This would, for example, allow us to use the significantly faster cycle time of static RAM relative to dynamic RAM to our advantage.

6 Finite State Automata as Filters

We consider the Dynamic Programming calculation as the preferred way to score sequence matches. The goal now is to separate likely candidates from unlikely ones. A filter that would reduce the number of comparisons by a factor of ten in the following stage would seem useful, yet practical. If we allow random strings through with a 10% probability, and assume that the real signal is well below this, we should nearly achieve this goal. If we assume DNA with an alphabet size of four, and also assume that the base usage is randomly distributed, we can calculate statistically how many hits we should get with different query substring lengths and numbers of substitutions/insertions/deletions. For the example, each of the 48265 length 16 strings should randomly match one in 4^{16} positions in the database. With a 10^9 database size, or nearly 4^{15}, each of the query strings has about one in four chance of matching. We would like an entire 1000 base sequence to have about a one in ten match rate. One possibility is to increase the length of our query substrings.

Table 1 shows the results if we allow up to one substitution, insertion, or deletion in different length (l) substrings of a 1000 character query. The final column shows the expected number of matches of a random string in a four character alphabet matching against a 10^9 character database.

We can see that 10% is somewhere between 25 and 26 character query substrings, and nearly six million states needed in the FSA. With strings this long, though, one error in 26 may not be enough. If we increase the allowed number of errors, then we need even longer query substrings to maintain the 10% hit rate.

As an additional complication, we should remember that our query strings are not necessarily statistically independent. With the one error allowance, many strings

l	n	e	t	f	x
10	991	91	90181	901810	86003303.53
11	990	100	99000	1089000	23603439.33
12	989	109	107801	1293612	6425440.31
13	988	118	116584	1515592	1737236.98
14	987	127	125349	1754886	466961.41
15	986	136	134096	2011440	124886.63
16	985	145	142825	2285200	33254.04
17	984	154	151536	2576112	8820.56
18	983	163	160229	2884122	2331.64
19	982	172	168904	3209176	614.47
20	981	181	177561	3551220	161.49
21	980	190	186200	3910200	42.34
22	979	199	194821	4286062	11.07
23	978	208	203424	4678752	2.89
24	977	217	212009	5088216	0.75
25	976	226	220576	5514400	0.20
26	975	235	229125	5957250	0.05
27	974	244	237656	6416712	0.01
28	973	253	246169	6892732	0.00
29	972	262	254664	7385256	0.00

The required FSA size is approximately proportional to the total number of characters in the query substrings. We tabulate for different lengths (l) the number of substrings in a 1000 character query (n), the number of cases of each with no errors, or a single substitution, insertion, or deletion using an alphabet size of four (e). Also, the total number of query substrings (t), and total number of characters in those substrings (f). The final column (x) is the expected number of times one of the query subsequences should match in a 10^9 character database, assuming they are statisically independent.

Table 1

are very similar. In any case, to make a useful filter, we must do better than match a single substring with up to one error.

7 A Two Level Finite State Automaton

In order to implement approximate string matching in a Finite State Automaton, we need a vary large number of states. However, many of the states end in a similar result. One possible way to get around this, and allow for a more reasonable memory size, is to drive a second FSA from the output of the first. Suppose we want to match all substrings of length 24 with up to three mismatches. We could generate a FSA to match all substrings of length six with up to three mismatches, and to output a value indicating how many mismatches it found. Then a second FSA matches the patterns in the output of the first. If we have an exact match, the first FSA will generate a series of exact match states. We now have to match four exact match states each six positions apart. There are still a large number of states to match, but the large number of strings with a given number of mismatches will all map into the same state in the output of the first stage.

To see how the number come out, we calculate the case just described. For a DNA alphabet size of four, strings of length six, and up to three substitutions, insertions, or deletions, including the first and last characters when appropriate, the results are shown in table 2. A total of 21065 strings of length six are needed for the first FSA.

For the second FSA we have an alphabet size of 21, the states tallied, plus the "none of the above" state, and strings of length up to 22. We need up to 19 for the exact match, and up to 22 to find the three insertions case. We need to distinguish substitutions, insertions, and deletions to match up the different sub-matches. The result, though, will be that the second FSA is even larger than a single FSA would be.

If we really need to detect all such matches, and only such matches, that is what would be required. But for our filter application, we can use a more statistical method. An FSA that finds six or eight character substrings will find many more of them in a reasonably similar sequence. In a region of a long exact match, it will continuously find matching substrings. In a long approximate match, we will have many consecutive single error matches. We need, then, a way to statistically recognize a good match from the output of a FSA matching smaller substrings. We implement the first FSA to output only the number of errors, zero, one, two, three, or "more", where "more" is too many to be useful.

We implement the second FSA similar to an accumulator, where its state would rise in high match regions, and fall in low match regions, accumulating match scores. If it reached a sufficiently high state, either due to an exact match, or a longer, but less exact match, it would signal the hit. This is a little similar to the way dynamic programming algorithms accumulate scores, though more statistical. It is different than dynamic programming in a special way: at each position it does not destinguish which query substring matched, only that one did. While this would be a disadvantage to dynamic programming, it may be an advantage to us. There are many sequences of marginal similarity that we would otherwise miss. If the query substrings are long enough, they should represent biological features, even if we wouldn't otherwise recognize them. Doing this well implies understanding the details of the sequence

Match condition	Cases	Strings
Exact match	1	1
One substitution	6	18
One deletion	6	6
One insertion	5	20
Two substitutions	30	90
Two deletions	30	30
Two insertions	20	320
Three substitutions	120	3240
Three deletions	120	120
Substitution and Insertion	30	360
Substitution and Deletion	30	90
Substitution and Insertion	30	360
Substitution and two deletions	120	360
Two substititions and deletion	120	1080
Substitution and two insertions	120	5760
Two substitution and insertion	120	4320
Insertion and deletion	30	120
Insertion and two deletions	100	400
Two insertions and deletion	80	1280
Insertion substutution deletion	125	3000

For a six character string, we tabulate the number of cases of substitutions, insertions, deletions, and combinations. The numbers get large very fast, and FSA for these cases must be considered carefully. These numbers are approximate, as they don't include degeneracy in the original sequence, but give an idea about how the query space increases with increasing allowable errors.

Table 2

better than I have described here. The important point, though, is that for longer queries we can use a more statistical approach to the scoring, and still filter what we need to filter.

The biology of approximate matches is a little difficult to describe, but maybe another example will make it more obvious. Imagine an FSA to recognize english words. For an exact answer, one should include an entire dictionary, but realizing the non-uniform distribution of letters and letter groups, one could score based on these groups. Using digraph (two letter) or trigraph (three letter) frequencies in a FSA would recognize possible english words with a high probability. Protein sequences can be similarly recognized by groups of amino acids, even if the groups are in a different order. It is this feature that FSA can find though dynamic programming cannot.

8 Counting FSA States

Until this point, only the total query size was used as a measure of FSA size. Here, the calculation gets more detailed. With a large number of query substrings, the lower FSA states will be well populated. With an alphabet size of four, and the FSA expanding like a tree, the first level will have four states, the second level 16, and the third will have 64. In the terminal branches, the number of states will equal the number of query substrings, each one indicating a hit. In between, it transitions from the saturated lower states, to the sparse terminal states. It is these transition states where the FSA spends most of the time during a search.

For different levels of the tree, a four character alphabet and 10000 query substrings, the numbers are shown in table 3.

The table shows, for each level i of the tree, l the number of states at that level, r the probability of that string matching a random string of that length, and p the probability of matching a string of that length and *not* matching a longer string. This last column gives the probability that the FSA will be at this level of the tree, at an average point during the search.

For an alphabet size of a, the number of possible states at level i is a^i, and there won't be more than the number of query subsequences. If statistically independent, the fraction not used is $(1-a^{-1})^j$ Where j is number of query subsequences per state at the previous level. That is, at each branch of the tree at level i we have j query subsequences to divide up among a new branches. Then l is this fraction multiplied be a multiplied by the number of states at the previous level. Next, the probability p of being at or past level i is la^{-i}, the number of states divided by the number of possible states. The incremental probability, Δp is $p_i - p_{i+1}$, the probability of being at least at level i minus the probablility of being higher than level i.

9 Scaling Laws for FSA Processors

With the ability to put multiple processors on one board, the balance between the size of each processor and the number of processors becomes important. For a fixed board size or fixed cost, we would like to know how many of what sized processors to use. High overall processing speed is achieved by having many processors all running

i	l	p	Δp
1	4	1	0
2	16	1	0
3	64	1	0
4	256	1	1.32e-05
5	1024	0.999987	0.060238
6	3849	0.939748	0.445070
7	8105	0.494678	0.346872
8	9687	0.147806	0.109828
9	9956	0.037978	0.028448
10	9994	0.009531	0.007147
11	9999	0.002384	0.001788
12	10000	0.000596	0.000447
13	10000	0.000149	0.000112
14	10000	3.73e-05	2.79e-05
15	10000	9.31e-06	6.98e-06
16	10000	2.33e-06	1.75e-06
17	10000	5.82e-07	4.37e-07
18	10000	1.46e-07	1.09e-07
19	10000	3.64e-08	2.73e-08
20	10000	9.09e-09	6.82e-09
21	10000	2.27e-09	1.71e-09
22	10000	5.68e-10	4.26e-10
23	10000	1.42e-10	1.07e-10
24	10000	3.55e-11	2.66e-11
25	10000	8.88e-12	6.66e-12
26	10000	2.22e-12	1.67e-12

For each level (i) of the FSA tree for alphabet size four, and 10000 query substrings, the number (l) of states there are likely to be, and the probability of being at least at this level. The final column, Δp, indicates the probability of being at this level and not a higher level.

Table 3

at the same time. Making one large processor, is inconvenient. Some parts, including addressing logic, get bigger as the logarithm of processor size. For result collection reasons, it is more convenient to have the processor size close to the size for a single query sequence, of approximately 1000 bases.

If we make the processor too small, we are limited by the number we can get on a board, and the overhead in board area and cost per processor. The control logic should be done using programmable logic[X96]. Though a simple PAL may be enough, though something a little more complex would probably be useful. In any case, the control logic should be a single chip, and the board area that this takes up will be the most important parameter determining the overall logic density.

Toshiba makes a 64k by 32 bit synchronous static RAM with a 15ns cycle time [TOS98]. This could be clocked at 66 MHz, and hold a 65,536 state FSA. The package itself is 22mm by 16mm. If the control logic was a similar size, and the necessary space between chips added, we need about 10cm^2 of board area.

Samsung makes 128k by 36 bit and 256k by 18 bit synchronous static RAMs with a 6ns cycle time.[SAM98] The access time, the time between the address being clocked in and the data being clocked out is about one half the cycle time, so some time is available for the control logic. Though it will be very difficult to keep up with a 166 MHz clock.

Samsung also makes a 16M by 16 bit synchronous dynamic RAM, with a 72ns read cycle time. (A read cycle requires 9 cycles of a 125 MHz clock.) While the cycle time is much slower than the static RAM, it is closer to the rate we are likely to get data into the processor array. The much larger FSA size will allow overall a greater throughput. As this RAM is only 16 bits wide, two will be required to be able to address all the states. The package size of 10mm by 22mm, a little more than half the SRAM sizes, allows again for about a 10cm^2 processor unit. The processor runs 12 times slower than the SRAM processor, but allows 64 or 128 times the number of states. Except for the extra difficulty of designing for DRAM, this is certainly worth using.

The physical size of a RAM package depends very little on the number of bits stored. Packaging technology is keeping up with current RAM densities, and the silicon size is growing very slowly, approximately logarithmically with the number of bits. With a given size for the control logic and RAM package, what is the optimal number of RAM units to use per control unit?

If the control chip area and RAM chip area are about equal, and we use this size as our unit area, we want to maximize the number of query sequences multiplied by the number of states (input characters) per second we can process. The required number of states is about equal to the number of query sequences per processor. The area for the control logic increases with the required number of address bits, again logarithmically with RAM size.

With the size and speed of the RAM factored out, the number of states per unit area is proportional to $n/(1+n)$, increases asymptotically with increasing n. The number of address lines, and things proportional to the number of address lines increase as $\log n$, so that $n/((1+n)\log n)$ is a better measure, which still increases with increasing n. However, generating the FSA also gets more difficult with increasing n. It is because of this, and simplifying result collection, that we size the RAM to the largest RAM we can get with the smallest number of chips with sufficient data

output lines.

10 Result Collection

One of the easiest details to overlook is the collection and storage of result data. If this becomes a bottleneck, it will limit the speed of the entire system. In this case, we need to at minimum know which query sequence matched which database sequence. Optionally, we would know where in the query or database the match was found. As a filter, this additional information is less important.

To find query and database sequence information, we must have an indication of sequence boundaries stored in the database. If we allow only one query per FSA processor, we only need to know which processor detected the hit. We should then latch this state until the next database sequence, avoiding multiple records of the same match.

While the goal is that 10% of the sequences will have hits, in some cases it could approach 100%, and hit collection should be designed to tolerate this case. We could, then, require millions of hits per query per search. Most economical, is to have a small number of hit processors for the entire array. We then have to either buffer hits, or store the entire hit record on each cycle. Buffering requires FIFO (First In First Out) memory in the result path. Storing the entire record, with only one bit per processor, means storing a bit vector at each hit signal. If we allow only one hit per sequence per processor, then we can store the bit vector, one bit per processor, at the end of each database sequence for which we have at least one hit, along with the database sequence number. This is a reasonable compromise in memory required for storing hits.

11 An Implementation Detail

While many implementation details should be left up to the system designer, there is one very interesting one that I describe here. In any technology, dynamic RAMs are much larger in bits stored, than static RAMs, for a similar silicon area and price. In the RAM array, it takes six transistors for a traditional static RAM cell, but only one for a dynamic RAM cell, so it is interesting to consider a design with dynamic RAM. To keep a dynamic RAM refreshed, it is necessary that every value of some of the low order bits be accessed every few milliseconds. RAM is normally implemented as a square array of data cells. In a DRAM read cycle, a row of bits are destructively read out, then stored back into the array. The column address then selects the appropriate bit from the row. It is the read and write back that refreshes the stored charge in the entire row, which happens on any read or write cycle. Normally, processors cannot be depended on to generate addresses sufficiently randomly to rely upon this to refresh the data. Dynamic RAM controllers will add special refresh cycles to guarantee that the data is refreshed in time. But in a processor that naturally cycles through the array, video displays being a common example, this is not necessary.

In designing a Finite State Automaton, it is possible to cycle the low order bits, assuming a little randomness to the input stream. To use real numbers, a certain 256 Megabit DRAM requires each of 8192 rows to be refreshed every 64ms.[SAM98B] If

the cycle time is close to 64ns, that means that each row must be accessed every one million cycles. A FSA search processor will normally have some states that it spends much of its time in. Table 3 shows this for a specific combination of parameters. If we take this small number of states, and arrange a succession of equivalent states for them, which cycle the appropriate address bits, it should work. For the case in Table 3, there are 3849 states that together represent 47% of the cycles. If we replace each of these states by ten or more equivalent states, and distribute the addresses among these states, it should be possible to cover the 8192 states needed for the refresh condition.

It may take a fair number of states to do this, but with an 8 million state machine, there should be states to spare.

If this can't be depended upon, it would also be possible to add extra data to the database to insure randomness at the appropriate point. Of course, one could always implement standard refresh logic in the controller.

12 Generating the Finite State Automaton

The ability to process large FSA fast requires the ability to generate them fast. Once we have a list of query sequence fragments, including ones with substitutions, insertions or deletions, a single FSA is generated from them.

With the four character DNA alphabet, we generate a quaternary (base 4) tree for the FSA. To do this, we read in the query sequences and follow through the states of the current FSA. When we try to take a branch with no successor state, we generate a new state and fill all entries with zero, add the branch from the previous state to the new state, and continue on through the sequence.

After we have generated the tree, there will be many states left still with no successor.[W87] It is necessary to back fill these states, to point to the state that the automaton would be in if that state didn't exist. Consider the FSA to match only the query `ACGTACGC`: The states, in succession, will have matched `A`, `AT`, `ACG`, `ACGT`, `ACGTA`, `ACGTAC`, `ACGTACG`, before reaching the final tt ACGTACGC. Now, consider the input `ACGTACGT`. Before the final `T` the FSA will be in the `ACGTACG` state, attempting to match the final `C`. When the `T` is read, it must return to the `ACGT` state, the state matching the longest suffix of the current input.

To do this, an algorithm devised by Gish[Gish] for use in BLAST[AG90] is used. This algorithm starts from the root state and considers every branch that leaves the root state. Each adds onto a circular queue of *from* and *to* states, the branch from the root state to its successor state. Then the queue of *from* and *to* states is then processed. For each branch out of the *to* state that is still zero, we replace it with the corresponding branch out of the *from* state. This does exactly what is needed: it branches to the same place we would have gone if the current state didn't exist. The FIFO (first in first out) queue is important here. The next state must be the state representing the longest match to the input stream. As the tree is traversed, deeper states, representing longer matches, get filled first. The scan and fill process is very fast and executes in time linear in FSA size.

If we are generating a two million state FSA from an input stream, we should consider how fast this process is. With current size machines, we should be able to fit the whole table in processor real memory. In the tree building phase, for each input

character, we need to check the current table entry, add a new state if it doesn't exist, and then move to the next state. For the backfill process, we need to progress through the tree as states are added to and removed from the FIFO queue. The queue depth could approach the number of states in the FSA, and each queue entry needs two state pointers.

The generate and backfill algorithm should be fast enough to keep the system running. During a search, the processor should be able to stream data at 16 million characters per second. The time needed to generate and load the FSA should be less than the search time. With upcoming genome projects expected to generate 30 gigabase data sets within three years, there should be enough data to keep the system running.

If our search time is on the order of hours and we need hundreds of FSA's generated in that time, we must generate them in minutes. We should be able to generate two million states in times the order of seconds on 100 MHz processors. Writing the generated FSA out to disk is the slowest part.

For the dynamic RAM version using statistical refresh, we want multiple copies of the more commonly occupied states. If we generate a complete tree for the first levels, with multiple identical copies of the level common states distributed through the low order address bits, and then add new states onto this, we should have a good start.

13 Conclusion

A large array of Finite State Automaton processors can be built for a reasonable price. This array can be used to rapidly search a database for some set of query sequences, and to store information related to the query sequences found. In some cases, this may be enough, otherwise, it can be used as input to a more detailed search. It should increase the value of the more detailed search by concentrating the useful sequences.

C code fragment to backfill states in a FSA

```
q=0;
for(i=0;i<ALPHABET;i++) {
   j=fsa[i];
   j &=  ACCEPT;
   if(j==0) continue;
   fifo[q].from=0;
   fifo[q].to=j;
   q++;
   }
head=q-1;
tail=0;
while(q>0) {
   from=fifo[tail].from;
   to=fifo[tail].to;
   tail=(tail+1)%(state+1);
   q--;
   for(i=0;i<ALPHABET;i++) {
      j=fsa[from+i];
      k=fsa[to+i];
      if(!k) {
         fsa[to+i]=j;
         continue;
         }
      if(k & ACCEPT) {
         fsa[to+i]=j | ACCEPT;
         continue;
         }
      if(!(j & ACCEPT)) {
         head=(head+1)%(state+1);
         q++;
         fifo[head].from=j;
         fifo[head].to=k;
         }
      }
   }
```

Figure 1: After all the query sequences are added to the FSA it is necessary to backfill it. Each link that is not part of a query sequence must point back to the state that the FSA would be in with the same input if the current state did not exist. The significant feature of this algorithm is the FIFO queue of states to be done.

References

[AG90] Altschul, S.F., Gish, W., Miller, W., Myers, E.W., Lipman, D.J., *J. Mol. Biol.* **215**, 403-410 (1990).

[CH91] Chow E.T., Hunkapiller T., Peterson J.C., Zimmerman B.A., Waterman M.S., A systolic array processor for Biological Information Signal Processing. Proc. of International Conference on Supercomputing (ICS-91) June 17-21, 1991.

[Gish] Personal communication.

[G82] Gotoh, O., An improved algorithm for matching biological sequences, J. Theor. Biol., **162**, 705-708 (1982).

[HU79] Hopcroft, J.E., Ullman, J.D.: Introduction to automata theory, languages, and computation. Addison-Wesley, Reading, Massachusetts.

[KA90] Karlin, S., Altschul, S.F., *Proc. Natl. Acad. Sci. USA* **87** 2264-68 (1990).

[KA93] Karlin, S., Altschul, S.F., *Proc. Natl. Acad. Sci. USA* **90** 5873-7 (1993).

[LL85] Lipton, R.S., Lopresti, D., A Systolic Array for Rapid String Comparison, 1985 Chapel Hill Conference on VLSI (1985)

[NW70] Needleman, S.B., Wunch, C.D., A general method applicable to the search for similarities in the amino acid sequence of proteins, *J. Mol. Biol.*, **48**:443-453 (1970).

[SAM98] 256K x 18 bit Synchronous Pipelined Burst SRAM, KM718V889, Samsung Electronics, (1997).

[SAM98B] 4M x 16bit x 4 Banks Synchronous DRAM, KM416S16230A, Samsung Electronics, (1997).

[TOS98] 65,536 word by 32 bit Synchronous Pipelined Burst Static RAM, TC55V2325FF-7, Toshiba Corporation (1998).

[W87] Wood, Derick: Theory of Computation. Harper & Row, New York, New York.

[X96] Xilinx, Inc., *The Programmable Logic Data Book,* Xilinx, Inc., 1996