Distributed and Hardware Genetic Algorithms Applied to the DNA Code Word Library Generation Problem

Daniel J. Burns
Air Force Research Laboratory/IFTC
525 Brooks Rd.
Rome, NY 13441
315-330-2335
burnsd@rl.af.mil

Morgan Bishop
JEANSEE Corp.
525 Brooks Rd.
Rome, NY 13441
315-330-1556
bishopm@rl.af.mil

ABSTRACT
Two high speed implementations of the genetic algorithm (GA) are described and their performances are evaluated on a highly constrained DNA Code Word Library Generation test case problem. The first is a distributed, or multi-deme, Island Model GA coded in C that uses the Message Passing Interface (MPI) protocol and runs on multiple processors in a cluster. The second is a single population GA coded in VHDL that implements both the GA and the fitness function evaluator in hardware on a single Field Programmable Logic Array (FPGA) chip. While the distributed GA is generally applicable to many problem types, the hardware GA is especially applicable to problems characterized by a fitness function requiring the calculation of a matrix of relatively simple integer-only or Boolean logic functions that can be efficiently implemented in a hardware systolic array.

Categories and Subject Descriptors
I.2.8 [**Artificial Intelligence**]: Problem Solving - Control Methods and Search - *heuristic methods*; B.2.4 [**Hardware**]: Arithmetic and logic structures – High speed Arithmetic – *algorithms;*B.7.1. [**Hardware**]: Integrated Circuits - Types and Design Styles – *Gate Arrays*

General Terms
Algorithms, Performance, Design, Experimentation.

Keywords
Genetic Algorithm, distributed, parallel, hardware, systolic array, speedup, DNA Codes.

GECCO'06, July 8–12, 2004, Seattle, WA, USA.

1. INTRODUCTION
The Genetic Algorithm (GA) is one of many algorithms available attacking hard optimization problems. The simple operators used for selection, mating, and mutation suggest that the GA may ultimately hold a speed advantage over algorithms with more complex arithmetic content, especially when implemented in hardware to achieve high speed solutions. This may be important in applications where real-time decisions are critical, or where best solution times are now hours or months. Fitness function evaluation may consume a large portion of the total solution time for many problems. In such cases it may be useful to parallelize the application, or to implement it in hardware. At this point it becomes an open question whether the GA or any other algorithm can ultimately yield the fastest possible solutions. The relative advantage of one optimization algorithm over another depends in part on the set of arithmetic operations required by the algorithm, and on how efficiently the operations can be executed by a typical CPU or when implemented in special purpose hardware. For example, an algorithm requiring floating point multiplications or gradient calculations involving division may be slower than one with only integer arithmetic and Boolean operations. Similarly, the nature of the application problem fitness function also influences whether a problem is a good candidate for hardware acceleration. In this paper we describe a DNA Code Word Library generation problem that has an integer-only, array type fitness function. Then we describe two GA solvers for this problem that pursue extreme speed-ups. The first is a distributed, Island Model GA that runs on a cluster and achieves ~30x speedup. The second is a hardware implementation of both the GA and fitness function evaluator that achieves ~700X speedup. This work represents preliminary steps toward a third version that targets a hybrid cluster architecture incorporating FPGAs at each processing node. This architecture should be able to achieve speedups of over 10,000, and reduce computation times from months to minutes.

The remainder of this paper is organized as follows. Section 2 describes the test case DNA Code Word Library Generation Problem, its mapping to a GA solution, and results using a baseline software GA run on one processor. Section 3 discusses the parallel GA implementation used in the present work. Section 4 discusses the Hardware GA used in the present work. Section 5 discusses the systolic array fitness function evaluator used in the Hardware GA. Section 6 presents results on the test case problem for the two GA versions. Finally, we offer suggestions for future work in Section 7, conclusions in Section 8, and in Section 9 acknowledge others who made contributions to this work.

2. DNA CODE WORD LIBRARY GENERATION PROBLEM

DNA code word libraries contain multiple pairs of Watson-Crick complementary DNA sequences that are free from undesired cross-hybridization between any two non-complementary pairs. They play vital roles in the development of biological and hybrid information systems that operate at the nanoscale [2], e.g. in biological microarrays, nano-circuits, memory devices, robust DNA tags, in breadth-first parallel filtering schemes for solving optimization problems with bio-molecules, and in nano-fabrication schemes that would use self-assembled DNA templates to organize the layout of nano-devices. Various methods have been proposed for building such codes, including the GA [6], Markov generated [2], and Stochastic [13] methods. Recent work [9] has shown that a hybrid GA blended with Conways lexicode algorithm [4] achieves better performance than either alone in terms of generating useful codes quickly. Exhaustive checking is impractical for finding large libraries of code words of lengths greater than about 12 base pairs.

The core of difficulty for this problem is searching the very large number of candidate strands that might be added to the library, and the computational cost of calculating strand binding free energies from thermodynamics in all of the n^2 possible secondary structures which may form from any two DNA strands of equal length. The Levenshtein distance, or edit distance metric is a reasonable but computationally more efficient tool for screening candidate strings during code design. The edit distance defines the minimum number of insertions, deletions, or substitutions needed to transform one string into another. Edit distance can be considered a generalization of the Hamming distance (HD), and a minimum edit distance constraint is much more difficult to meet than HD. HD only considers substitution edits with the strands aligned fully side by side, and it can be calculated in O(2n) time. The Levenshtein distance covers all 'slidings' of two strands past each other, and it utilizes the dynamic procedure shown in Figure 1, which completes in $O(n^2)$ time (in a sequential program).

	A	G	G	C	G	C
A	1	1	1	1	1	1
G	1	2	2	2	2	2
T	1	2	2	2	2	2
G	1	2	3	3	3	3
C	1	2	3	4	4	4
G	1	2	3	4	5	5

cell entry $M_{i,j} = \max(M_{i-1,j},\ M_{i,j-1}, k + M_{i-1,j-1})$

where $k = 1$ if $s1_i = s2_j$ else $k = 0$, and

s1 = sequence along top row, s2 = sequence along left column

Figure 1. Calculation of the Levenshtein edit distance metric.

The DNA code word problem can be mapping to a GA by representing a strand as a string of bits using a substitution such as A=00, C=01, G=10, T=11. Thus a strand with 16 bases can be represented by a 32 bit integer. Well known GA operators such as single point crossover and bit flip mutations are applicable, with modifications. We allow single point crossover to cut only at base pair boundaries, and we use a mutation operator that selects the best of all possible single base mutations. The code design requirement specifies the desired length of the strands or words, e.g. n = 16 bases, and the desired edit distance, e.g. d = 10. In order for a pair of complimentary code words to enter the library, the new word and its reverse compliment (RC) word must meet the edit distance requirement when tested against each other and against each word and each RC word already in the library.

The GA fitness function consists of two numbers that are tabulated for each candidate word in the population. The first is the maximum of the absolute differences between the desired edit distance and the actual edit distances measured between the candidate word and its RC and every word and every RC word already in the library, which we call the max_match. The second is the number of words in already in the library that reject the new word, which we call the number_of_rejecters. A new word and its RC word can enter the library when the number of rejecters = 0. We pack the two numbers into one 32 bit word, with the number_of_rejecters in the upper bits and max_match in the lower bits to obtain a fitness metric that goes to 0 as a word gets 'better'.

Figure 2 shows the results of a typical run (on one processor node, in compiled C) in terms of # Words Found vs. Time, for good parameter tunings of both the baseline software Stochastic (top left curve), Markov (middle), and GA (lower) algorithms. Here the GA uses a population size of 100, no mating, and 1% mutations. The GA is slower at the beginning, about 10-100 times faster in the middle, and about the same near the end of the run. Since optimal (the largest possible) code word sets are not needed in practical applications, one could say that the GA is superior at finding practical code sets. A version of the stochastic method was coded and compared, but was not competitive with these times. Stochastic is similar to GA with only mutation, except that it starts with a full library and seeks to improve the fitness of all words, which requires more checks from the start.

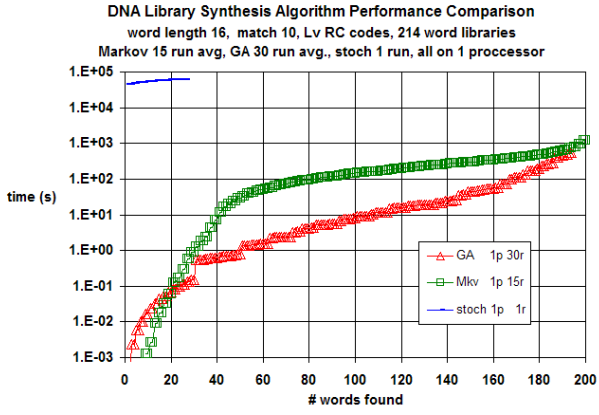

Figure 2. Relative performance of GA, Stochastic, and Markov methods.

To get a sense of problem difficulty, we can calculate the number of edit distance calculations that must be made between a word and its RC and all pairs in the library during a generation. Assuming a population size of 100, a mating probability of 80%, a

mutation probability 1%, and an existing library of 100 pairs, mating requires (80 children x 2 checks) x (100 words x 2 checks) or 32,000 checks. Mutation requires another (0.01 x 100) x (47 possible mutations x 2 checks) x (100 words x 2 checks) = 18800 checks. At 10 us per check this takes 0.5 seconds, but the constraints are so severe that many words must be checked to find one that can be added to the library, leading to run times of hours to assemble large libraries. Since the edit distance calculation consumes a large portion of the computation time involved in building such codes, regardless of the search algorithm used, this problem is a good candidate for speedup by parallel and hardware implementations, which are discussed next.

3. THE PARALLEL GA

Both the DNA Code Word Library generation problem and the GA are embarrassingly parallel. Much previous work has been done on parallel and distributed GAs [5,7].

We use an Island Model GA that passes the top 5 individuals to adjacent nodes in one direction in a ring configuration after epochs of 40 generations. We choose the population size so that the minimum number on individuals at each processor node was about 100. We typically use from 1 to 30 nodes in the cluster, with a total population size of 3000 individuals, split as evenly as possible among the processors. In our version of the Island Model, communication occurs between a master node and all other nodes at startup, when one of the termination criteria is met (maximum time, maximum # of generations, or desired # words found), and also in two other cases. First, when any processor finds a word, it is shared immediately with all other processors in order to keep them all working on extending identical libraries. Second, the top 5 individuals are passed around the ring in one direction at epoch boundaries. We did limited experiments with epoch lengths between 3 and 100 and found that 40 worked well. We observed that single point crossover mating was very disruptive to average population fitness, so we typically set crossover to a low value, or used only mutation. We replace clones in the population with new random individuals at the end of each generation. We used either rank or fitness based selection, with neither clearly better. Our experiments typically did 30 runs using 1 processor, 30 runs using 2 processors, etc., and we averaged the results over all 30 runs for each # of processors.

Figure 3 shows typical results in terms of the average # words found vs time (left), and a speed-up curve (right). The different plots at the left correspond to different #'s of processors, and each point in each curve is the average over 15 runs. The speedup curve shows the average time to find 200 words vs # processors used, normalized to the value for 15 processors. The 3 speedup curves show ideal linear speedup (red), uncensored speedup (blue), and censored speedup (green). Censored speedup was intended to exclude atypical runs. The 2 rows of boxes (lower right) show the number of runs at each processor value that found all 200 words (all did), and the lower row of boxes show the number of runs that finished in a time within one standard deviation of the average time of all successful runs. In this experiment the # processors was scanned from 15 to 30, the code words were 16 bases long with a desired max_match edit distance of 10, the GA used a population of 3000, there was no crossover, the mutation probability was 1% , and the 5 best individuals were passed at epochs of 40 generations. These results show an approximately linear speedup with # processors.

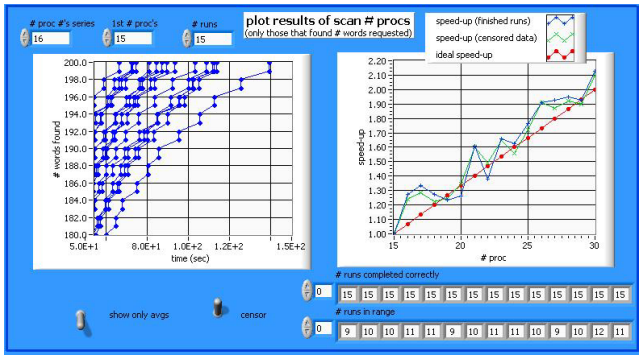

Figure 3. Average # words found vs # processors and speedup curve for an experiment scanning the # processors from 15 to 30 (average of 15 runs at each # processor value).

This code was instrumented with MPI extensions that allow logging the time of the beginning and end of events at each processor during execution. This analysis is useful for optimizing the type and placement of MPI communication events. Figure 4 shows typical results after tuning the MPI communication calls. This tuning decreased the generation time from ~65 ms to ~8 ms.

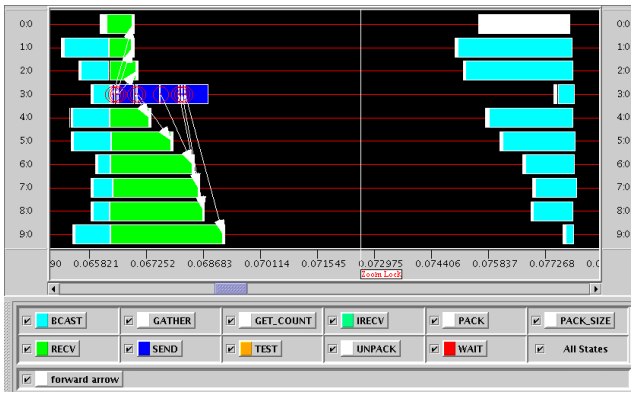

Figure 4. Timing of communication (blue and green) and calculation (red line) during a GA generation for nodes 0-9. Generation boundary is at the beginning of the green area.

The communication overhead is about 25% for the cycle shown in Figure 4. There is skew and jitter in the total generation times among the processors due to startup effects, MPI response latencies, and due to the stochastic nature of the GA. This analysis is also useful for determining a population size large enough to guarantee that communication delays do not dominate the total generation time. Finally, the code was instrumented with GNU Gprof [8] to observe the duration of various tasks in the generation cycle (black in Fig. 4). This analysis showed that the subroutine that calculates the edit distance consumed 98.13% of the generation time. Therefore, the fitness function evaluator could be sped up (e.g. by hardware acceleration) by a factor of about 98.13/(100-98.13) = 112 before the GA algorithm time would be equal the fitness function evaluation time. Significant speedup beyond that would require both hardware fitness function evaluator and a hardware GA, which we discuss next.

4. THE HARDWARE GA

There has been some interest in speeding up GAs by implementing them in hardware. Reviews and examples of past work can be found in [12,11,1,14]. These studies are usually coupled with a particular type of GA and problem, and the results are often highly problem dependant. Overall speedups of 3-1000X have been reported.

Our design is different than previous work in that we have targeted a single chip FPGA implementation with the population stored and manipulated in fast, on-chip SRAMs that avoid delays associated with using off chip memory. Also, we use a systolic array on the same chip for the fitness function evaluator. We used the relatively inexpensive, commercially available Annapolis Microsystems Wildcard (PCMCIA) FPGA board that contains a Xilinx Virtex-II X2CV3000 FPGA chip. Basically, in this approach a PC executes a C 'Host' program that passes run parameters to an FPGA processing element 'PE' that implements the GA and DNA Code Word application. The Host then receives reports from the PE when words are found, and when the hardware application terminates. The PE function is described in VHDL, which we composed and simulated using the Mentor Graphics ModelSim tool. We used the Xilinx ISE Webpack or Synplicity Synplify tools for synthesis.

The FPGA design effort was focused mainly on implementing a fast function evaluator because the total execution time is dominated by fitness function evaluation. Details of the design will be described elsewhere, but here we give an outline of the main processes. They are initialization, checking fitness, picking up good words from the population into the library, mutation, decloning, and reporting results to the host. The FPGA chip contains static block RAMs (BRAMs) that can be configured as 96 separate 512 word x 32 bit RAMs. We use one BRAM to hold the population (up to 512 individuals), a second BRAM to hold the fitnesses, and third BRAM to hold the code word library. We use an overall architecture that was simply a pipeline of processes connected between sets of these 3 BRAM types. The BRAMs are dual ported, which facilitates connection between separate input and output processes. We determined that with a population size of 100 and with 100 words in the library, the time overhead for passing the entire population, fitness, and library BRAMs around the pipeline was less than 2% of generation time.

A 32 bit pseudo-random number generator (PRNG) was implemented in this design by an array of 32 32 bit linear feedback shift registers. The output word is formed by concatenating one bit from each of the 32 registers. The PRNG can be seeded by the Host to repeat a run with the same sequence of random numbers, or with a different set. A new random number is available at each PE clock edge, and all possible 32 bit numbers are represented in the sequence before it repeats. The population can be initialized with random individuals, and the library can be initialized as empty, or it can be seeded by the Host with an existing partial library.

The main GA is a tight loop of three processes, the first picking up good new words from the population into the library, the second for mutation, and the third for decloning. The pickup process looks for new good words in the population, and when it find one it moves it to the library and replaces it with a new random individual. When a new word enters the library, all of the fitnesses must be recalculated. In this step operand pairs are read from the population and library BRAMs and presented to the fitness evaluator at the PE clock rate, as described in the next Section. Each population word is checked against its own RC, and against each library word and each RC library word. The results are also analyzed and the fitness metrics are determined at the PE clock rate and stored in the fitness BRAM. During the mutation process words are selected from the population, and for each word all 47 possible single base mutations are assembled into a BRAM and their finesses are checked in a manner similar to population checking. The mutation that results in the best improved fitness is used to replace the original word, or if no mutation improves fitness, one of the 47 mutations is chosen randomly to replace the original word. When the required number of words are mutated, or if a mutation is found with 0 fitness that can enter the library, the mutation process ends and the decloning process starts. This process actually does several things. First it finds and records the best fitness in the population (without sorting). Then it replaces any words in the population that are already in the library with new random individuals. Such words will have quite good fitness, since only two words in the library will reject them, but their fitness will never be good enough. Finally, any clones in the population are replaced with new random individuals because there is no point in keeping clones given the type of mutation we use. We do not use mating, so the next generation then starts with the pickup up process.

Although we don't typically use mating for this problem, it was implemented in VHDL. In the interest of speed, we used a table look up method that implements rank based selection from the top k individuals of a population of n individuals, where k and n are BRAM addresses between 0 and 511. For example, for the case of selection from the top 10 individuals in a population of 100, a table of 10 9 bit numbers is calculated from the appropriate cumulative selection probabilities. The index of a parent is chosen by sampling a 9 bit random number from the lower bits of the 32 bit PRNG, and running an index pointing into the table up from 0 until it points to tabled value larger than the random number. One less than this index is used as the parent's index. Using this approach the Host can calculate the tabled values for any k and n <=511 and pass them in to the PE at the start of a run. This avoids resynthesis of the FPGA, which takes time.

All communication between the Host and PE is handled by an interface that is supplied as an example with the Wildcard software. It allows both the Host and PE to read and write to a common 32b register and a common BRAM that reside in the PE. Communication occurs at the beginning of a run to set GA and code design parameters, again when the PE finds each word, and finally at the end of a run. The PE records the generation each new word is found on, and also the best fitness in the population vs generation, and it passes this information back to the Host.

At this writing the entire design has been composed and simulated. The PRNG and fitness evaluator described in the next section have been synthesized. Together they use less than 20% of the FPGA chip resources, and the maximum clock frequency is higher than our 100MHz goal.

5. THE HARDWARE SYSTOLIC ARRAY FITNESS FUNCTION EVALUATOR

Systolic arrays [10] basically can perform calculations in a 3 dimensional array of cells simultaneously (2 physical dimensions and the time dimension). They are driven with fast streams of operands flowing into two edges of an array. In the case of the Levenshtein calculation, the results corresponding to any two operands flow as a diagonal front away from those edges toward the opposite corner of the array where a stream of results is read out from the lower right cell output. After a latency period of 18 clocks an answer for each set of input operands appears in the output stream at every clock period.

Figure 5 shows a block diagram of the systolic array fitness function evaluator and its feeder registers. Register arrays are needed along the top and bottom edges to sequence portions of the operands into the inputs of the edge cells at the right times. Bases in the input words at the right along the top edge and toward the bottom along the left edge must be delayed in a staggered manner before being presented to the edge cells. Each cell in the array contains the circuitry for calculating the max() function shown in Figure 1, as well as registers for passing bases in the operands down along columns and to the right along rows through the array. The array operates in a checkerboard fashion, with the cells on the even rows on even columns and odd rows on odd columns in one group, and the others in a second group. The first group loads inputs on one clock edge, and latches outputs on the next clock edge. The cells in the second group do the opposite.

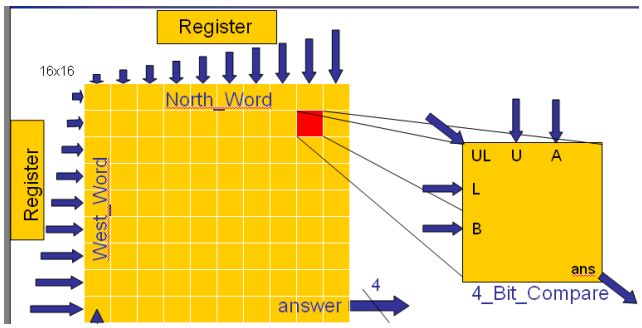

Figure 5. Block diagram of hardware systolic array for fitness function evaluation.

In the present design we actually use two instances of the fitness function evaluator, one for pickup process, and one for the mutation process. This is a side effect of writing the source code with 'hierarchical' structure (with multiple processes that can be added or debugged and changed easily), rather than in a 'flat' structure (with all functionality in one big process). Since only one hardware process can drive a signal, we need to duplicate or multiplex the inputs and outputs of any component that is used by more than one process. Multiplexing adds delay and complexity and can make routing interconnects more difficult, so we duplicated the fitness evaluator to avoid that potential problem.

6. RESULTS AND DISCUSSION

Results for the baseline software GA DNA Code Word application run on one processor were shown in Figure 2 and compared favorably with results using the best known algorithm found in the literature, the Markov method. Figure 3 also showed that the performance of the distributed version of the GA scales approximately linearly with the number of processors used. These results are shown again in Figure 6 along with a performance curve for 1 run of a simulation of the hardware version (blue), and one run of the baseline software GA run on 1 processor using the same conditions as the hardware for comparison (lower red). These two new curves were for a population size 16, vs 100 for the previous (upper) GA curve in Figure 6.

The results show that the hardware version is about 100 times faster in the early stage of the problem with few words found. The hardware version (lowest curve) was not extended because the simulation is too slow to run out to more than a few words.

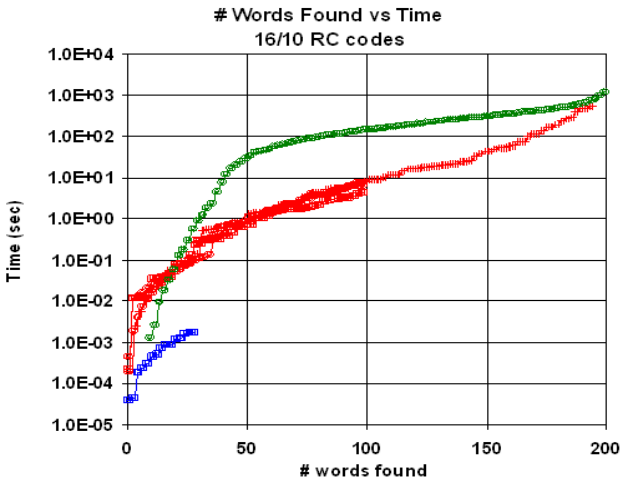

Figure 6. Relative performance of GA, Markov, and hardware GA.

To get a better idea of how the hardware version should perform in the later stages of the problem, we analyzed the simulated waveforms of the hardware version and constructed a clock cycle accurate spreadsheet model that calculates total generation time as a function of population size and the number of words in the library. We then used the model to construct a curve of generation time vs # words in the library, for the case of population size 100. The clock frequency of the FPGA was assumed to be 100MHz. We also measured the corresponding actual generation times for the baseline software GA run on one processor, also for population size 100. The results are shown in Figure 7, and they indicate that the hardware version (lower curve) should be about 700X faster than the software version (upper curve).

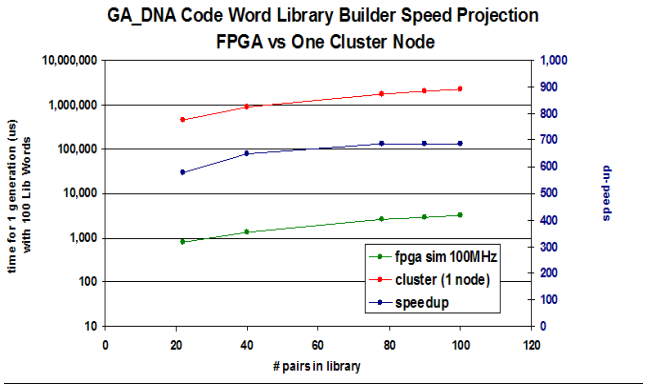

Figure 7. Comparison of generation time # words in library for software and hardware GA DNA Code Word application. (Population size = 100, composing 16/10 RC codes, no mating, 1% mutations).

7. FUTURE WORK

We plan to synthesize the hardware version and evaluate its performance. It would be desirable to add thermodynamic binding free energy calculations and other metrics used in the Markov method, such as tabulating stacked pairs, which are adjacent bases that bind between two words. This would enable a search for an improved (faster) mutation heuristic that would seek to eliminate stacked pairs. An exhaustive search option would also be useful. Presently there is no way to know whether another word exists that can be added to a library without searching. We estimate that with about 250 words in the library the present hardware systolic fitness function evaluator could check all 2^(32-1) candidate words in about 3 hours. This would actually be faster than the using the present algorithms, which can run for days before finding words. This would be useful to those interested in improving the known bounds on the size of optimal code word libraries. Finally, it would be of interest to implement a distributed hardware GA version. It might be possible to process more than one population on the same chip. Another approach would be to use fast FPGA to FPGA communication mechanisms to implement a mutli-chip distributed hardware GA. It would also be of interest to explore hardware versions of the Markov method, or other evolutionary algorithms.

8. CONCLUSIONS:

We have shown that a GA approach to solving the DNA Code Generation Problem is competitive with the best known methods in the literature. We have described a hardware systolic array implementation of the Levenstrein matrix calculation that achieves about a factor of 1000X speedup of the fitness function evaluator for this problem. We have shown that distributed and hardware GAs offer significant performance improvements of 30X and ~700X, respectively.

9. ACKNOWLEDGEMENTS

Larry Merkle of Rose Hulman Institute of Technology contributed valuable guidance and encouragement to this project during a Summer Research Faculty assignment at AFRL. Kevin May contributed to the coding of the distributed and hardware GA fitness function evaluator, and he tested the distributed GA while an undergraduate at Clarkson University studying Computer Engineering. Tony Macula of the Mathematics Dept. of SUNY Genesseo contributed suggestions for the fitness function and developed the Markov generated algorithm.

10. REFERENCES

[1] Aporntewan, C. and Chongstitvatana, P. , "A Hardware Implementation of the Compact Genetic Algorithm", Proceedings of the 2001 Congress on Evolutionary Computation, pp. 624-629, 2001.

[2] Bishop, M. , Macula, A. , Pogozelski, W. , and Rykov, V. , "DNA Codeword Library Design", Proc. Foundations of Nanoscience – Self Assembled Architectures and Devices, (FNANO), April 2005.

[3] Brenneman, A. and Condon, A.E., "Strand Design for Bio-Molecular Computation" , Theoretical Computer Science, Vol. 287, Issue 1, Sept. 2002, Natural computing, pp. 39-58.

[4] Brualdi, R. and Pless, V. , Greedy Codes, Journal of Combinatorial Theory,(A) 64:10-30, 1993.

[5] Cant-Paz, E. , *Efficient and Accurate Parallel Genetic Algorithms,* Kluwer Academic Publishers,Norwell, MA, 2000.

[6] Deaton, R., Garzon, M, Murphy, R.C., Rose, J. A. Franceschetti, D. R. and Stevens, S. E., Jr., "Genetic search of reliable encodings for DNA-based computation," Koza, John R., Goldberg, David E., Fogel, David B., and Riolo, Rick L. (editors), Proceedings of the First Annual Conference on Genetic Programming 1996.

[7] De Jong, K.A. and Sarma, J. , "On Decentralizing Selection Algorithms", *Genetic Algorithms: Proceedings of the Sixth International Conference (ICGA95)*, pp. 17-23. Morgan Kaufmann, July, 1995

[8] gprof: http://www.cs.utah.edu/dept/old/texinfo/as/gprof.html.

[9] Houghten, S.K. , Ashlock, D. and Lennarz, J. , Bounds on Optimal Edit Metric Codes, Brock University Tech. Rept. # CS-05-07, July, 2005.

[10] Kung, S.Y. , VLSI Array Processors, Prentice-Hall, Inc., Upper Saddle River, NJ, 1987.

[11] Megson, G.M. and Bland, I.M. , "Synthesis of a Systolic Array Genetic Algorithm", 12th International Parallel Processing Symposium / 9th Symposium on Parallel and Distributed Processing (IPPS/SPDP '98), pp. 316-320, Proceedings. IEEE Computer Society 1998.

[12] Scott, S. D. , Samal, A., and Seth, S. , "HGA: A hardware based genetic algorithm", Proc. ACM/SIGDA 3rd Int. Symp. FPGA's, 1995, pp. 53-59.

[13] Tulpan, D.C. , Hoos, H. , Condon, A. ,"Stochastic Local Search Algorithms for DNA Word Design", Eighth International Meeting on DNA Based Computers(DNA8), June 2002.

[14] Wells, E.B., et. al., "Applying a Genetic Algorithm to Reconfigurable Hardware -- a Case Study", paper 179, 2004 MAPLD Conference, Washington, DC, 2004. http://klabs.org/mapld04/papers/e/e169_wells_p.doc