AFRL-RI-RS-TR-2009-131
Final Technical Report
May 2009

MULTI-THREADED DNA TAG/ANTI-TAG LIBRARY GENERATOR FOR MULTI-CORE PLATFORMS

University of North Carolina at Charlotte

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

STINFO COPY

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE
ROME RESEARCH SITE
ROME, NEW YORK

NOTICE AND SIGNATURE PAGE

Using Government drawings, specifications, or other data included in this document for any purpose other than Government procurement does not in any way obligate the U.S. Government. The fact that the Government formulated or supplied the drawings, specifications, or other data does not license the holder or any other person or corporation; or convey any rights or permission to manufacture, use, or sell any patented invention that may relate to them.

This report was cleared for public release by the 88th ABW, Wright-Patterson AFB Public Affairs Office and is available to the general public, including foreign nationals. Copies may be obtained from the Defense Technical Information Center (DTIC) (http://www.dtic.mil).

AFRL-RI-RS-TR-2009-131 HAS BEEN REVIEWED AND IS APPROVED FOR PUBLICATION IN ACCORDANCE WITH ASSIGNED DISTRIBUTION STATEMENT.

FOR THE DIRECTOR:

/s/ /s/

MICHAEL J. HAYDUK, Chief
Emerging Computing Technical Branch
Advanced Computing Division

GERARD J. GENELLO, Chief
Advanced Computing Division
Information Directorate

This report is published in the interest of scientific and technical information exchange, and its publication does not constitute the Government's approval or disapproval of its ideas or findings.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)*	2. REPORT TYPE	3. DATES COVERED *(From - To)*
MAY 09	Final	Jun 08 – Feb 09

4. TITLE AND SUBTITLE

MULTI-THREADED DNA TAG/ANTI-TAG LIBRARY GENERATOR FOR MULTI-CORE PLATFORMS

5a. CONTRACT NUMBER
08-RI-CRADA-06

5b. GRANT NUMBER
N/A

5c. PROGRAM ELEMENT NUMBER
62702F

6. AUTHOR(S)

Arun Ravinfran

5d. PROJECT NUMBER
459T

5e. TASK NUMBER
HA

5f. WORK UNIT NUMBER
EC

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

University of North Carolina at Charlotte
Electrical and Computer Engineering
9201 University Blvd.
Charlotte, NC 28223

8. PERFORMING ORGANIZATION REPORT NUMBER
N/A

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

AFRL/RITC
525 Brooks Rd.
Rome NY 13441-4505

10. SPONSOR/MONITOR'S ACRONYM(S)
N/A

11. SPONSORING/MONITORING AGENCY REPORT NUMBER
AFRL-RI-RS-TR-2009-131

12. DISTRIBUTION AVAILABILITY STATEMENT
APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED. PA# 88ABW-2009-1725 04/27/09

13. SUPPLEMENTARY NOTES

14. ABSTRACT

Report Developed under CRADA 08-RI-CRADA-06.
The report describes a new approach to the problem of generating DNA tag/anti-tag libraries used in experimental computing methods involving bio-molecules, and in biological assay methods. This approach couples multi-threaded coding methods and a highly parallel multi-population genetic algorithm to leverage performance gains made possible by the multi-core CPUs increasingly prevalent in today's commodity workstation computers. We explored and exploited algorithm and architecture trade-offs while developing a multi-threaded code that uses shared memory communication, and minimal synchronization between threads. We also describe experiments that evaluated performance and demonstrated ~ 5X-8X speedups on workstations with dual quad-core CPUs. We observe that coding effort using the C language and Pthreads parallel programming model is greatly reduced compared to two previous approaches that used the VHDL language run on reconfigurable hardware (FPGAs), and the C language with MPI API run on a cluster of computers.

15. SUBJECT TERMS
Threads, genetic algorithm, DNA, tag/anti-tag, Gibbs energy, binding metrics, acceleration

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			Daniel J. Burns
U	U	U	UU	37	19b. TELEPHONE NUMBER (*Include area code*) N/A

TABLE OF CONTENTS

LIST OF FIGURES

1.0 SUMMARY

This report describes a new approach to the problem of generating DNA tag/anti-tag libraries used in experimental computing methods involving bio-molecules, and in biological assay methods. The approach couples multi-threaded coding methods and a highly parallel multi-population genetic algorithm to leverage performance gains made possible by the multi-core CPUs increasingly prevalent in today's commodity workstation computers. We explored and exploited algorithm and architecture trade-offs while developing a multi-threaded code that uses shared memory communication, and minimal synchronization between threads. We also describe experiments that evaluated performance and demonstrated ~5X-8X speedups on workstations with dual quad-core CPUs. We observe that coding effort using the C language and Pthreads parallel programming model is greatly reduced compared to two previous approaches that used the VHDL language run on reconfigurable hardware (FPGAs), and the C language with MPI API run on a cluster of computers.

2.0 INTRODUCTION

The purpose of this CRADA was to leverage recent work at AFRL/RI on the hardware acceleration of DNA Code discovery, and expertise at UNCC on implementing multi-core platform distributed bio-informatics applications to further explore and improve the state of the art for accelerating the solution of problems that require lengthy, expensive computations involving DNA binding affinity models. Both CRADA partners benefited by blending their independent work on algorithms on advanced computing platforms, and by comparing the results to that obtained in previous work. This project explored the efficiency and best performance of various parallelization strategies run on a variety of current leading edge platforms.

3.0 BACKGROUND

The past decade has witnessed a growing interest in exploiting the molecular recognition properties of nucleic acids to perform logical operations, act as covert physical taggants, as labels and probes in bio-assays, and potentially as a means for self-assembly of nano-structures, all of which involve subjecting a combination mixture of DNA molecules to hybridization and enzymatic reactions. The utility of these applications depends on expensive (meaning long run time) computations that identify sets of relatively short segments of non-hybridizing DNA drawn from a large universe of possible strands, or from organism genomes. Unfortunately, these DNA strand searching problems are NP-hard, and this has limited research on both DNA codeword design for synthetic applications (especially for codes of length greater than ten bases), and on probe/target set design and optimization for bio-assay microarrays that use longer DNA strands (16-64 bases). Meanwhile, due to the recent saturation of microprocessor clock frequencies and limitations of superscalar uniprocessor architectures, microprocessor vendors have resorted to integrating multiple processor cores on a single chip. It is expected that the number of processor cores would double with each new silicon generation. The free lunch of software performance increases driven by higher processor clock frequencies appears to be over, and further increases in software performance now critically depend on programmers exploiting higher levels of parallelism inherent in applications. This must be accompanied by programmers developing codes with application programming interfaces (API's) that enable access to the multiple cores available in today's workstations, as wholesale automated parallelization of codes is beyond the capabilities of today's compilers.

This project leveraged recent work at AFRL/RI on the acceleration of DNA Code discovery by using evolutionary algorithms, and UNCC work on grid and multi-core platform distributed bio-computing applications, to further explore and improve the state-of-the-art in computing about DNA. Intellectual property in the form of software program listings was exchanged by the partners. AFRL provided UNC Charlotte a set of three software programs (described in APPENDIX A) written in C, C/MPI, and VHDL that implement DNA code discovery using a genetic algorithm (GA), the Length of the Longest Common Sequence (LLCS), and Pairwise Nearest Neighbor Model (PNNM) Gibbs Energy approximation.

In a series of three technical tasks, workers at UNCC developed and provided AFRL with codes that implemented the three software applications (GA, LLCS, PNNM) using multi-threaded, shared memory methods. Work at both UNCC and AFRL evaluated the performance of the resulting codes on a variety of multi-core architectures. AFRL also developed and tested a version that ran on a Cell Broadband Engine (CBE) cluster platform.

The preliminary results of work under Tasks I (DNA tag/anti-tag building application using LLCS metric) and II (distributed genetic algorithm) were described in an interim technical report delivered to AFRL, and are described in [14]. The remainder of this report is an expanded version of that paper that also describes work under Task III on a version that used thermodynamics based binding energy estimate metric, and additional performance evaluations on other multi-core platforms.

4.0 INTRODUCTION

DNA tag/anti-tag systems are a common essential component in a number of biological assay and genotyping methods [1]. For example, single-nucleotide polymorphism (SNP) analysis by polymerase-mediated single-base primer extension (mini-sequencing) [2] couples a set of probe primer strands (that target SNP sites) with a set of tag strands (that control binding to readout substrates functionalized with anti-tags) to realize massively parallelized analysis of genomic content using fluorescence imaging of DNA microchips, or analysis of microspheres with flow cytometry methods.

The design of high quality tagged probe sets involves two problems, first generating a tag/anti-tag library, and second pairing individual probes and tags, all in a manner that is subject to a number of constraints that control unintended binding among other tags and anti-tags, and other probes and target strands. While paring of individual probes and tags could also be done using a multi-pop GA, that problem is outside the scope of this project, and we focus here on DNA Tag/Anti-tag set design.

The size and quality of the initial tag/anti-tag library affects both the degree of parallelization possible (i.e. how many independent chips or tests must be done to cover all of the target SNPs), as well as the degree of background interference caused by unintended binding that affects SNP call accuracies. Both the initial tag/anti-tag library design process, and the tag selection and pairing process are compute intensive, and motivate work to optimize solutions and utilize advanced computing methods and platforms to obtain solution speedups.

In the Section 5 we briefly review some past work on DNA tag/anti-tag library generation and speedup methods. Section 6 provides an overview of opportunities and challenges offered by multi-core computing. In Section 7 we describe our new work that takes advantage of state-of-the-art multi-core CPUs to implement a multi-threaded version of a multi-population genetic algorithm (GA) to achieve speedup of tag/anti-tag set design. In Section 8 we describe experimental results for the LLCS and Thermo metric versions, and we compare the results with that of previous work using MPI and VHDL FPGA acceleration. Section 9 discusses advantages and limitations of the multi-threaded approach, and finally, Section 10 provides conclusions and recommendations future work.

5.0 DNA TAG/ANTI-TAG LIBRARY DESIGN

The basic problem of DNA tag/anti-tag library design is to compose a large set of relatively short (8-32 base pair) Watson-Crick strand pairs that bind perfectly within pairs, but poorly across pairs. A variety of DNA strand hybridization metrics, interaction constraints, search algorithms, coding approaches, and computing platforms have been used by a number of researchers to work toward efficient solutions of this and other related problems [3]. *DNA Code design* has been shown to be NP-complete, thus practically excluding the possibility of finding any procedure to find maximal sets efficiently [4]. Still, there is much room for further work, as improved non-optimized solutions will reduce the cost of large scale genomic analysis, a critical factor in enabling the widespread commercial success of laboratory and point-of-care individualized genomic content analysis technologies. For example, streaming chip multiprocessor micro- architectures that take advantage of the high inter-processor communication bandwidth and new models of programming have been proposed for customized bioinformatics applications [7].

A number of possible strand interactions are usually checked to ensure high quality libraries with low unintended binding across non-interacting pairs. Each word and its reverse complement (RC) word can be checked against every other word and RC word in the library. Also, each word is sometimes checked against its own reverse compliment word. All strands can also be checked for hairpinning, GC content (or melting temperature), and against the genome to be analyzed.

All of these checks require a large number of calculations of some metric that estimates the binding affinity between strand pairs, and this calculation dominates the solution time as word pairs are added to the library. Some simple metrics such as Hamming distance can be calculated in O(n) time for strand length n. Others, such as Edit Distance or Length of the Longest Common Substring (LLCS), require times of order $O(n^2)$, as the dynamic programming methods used to calculate them cannot be parallelized at the lowest level in software. Still more costly estimates are Smith-Waterman similarity, thermodynamic methods based on pairwise nearest neighbors [8], and full thermodynamic estimates (e.g. M-fold).

Because of the large number of possible candidate strands (4^n), exhaustive search using software is impractical for large n due to computational cost, so a variety of greedy heuristic, stochastic, and evolutionary methods have been used to mine the search space for candidate word pairs. Interestingly, for the case of building 16-mer RC codes of Edit Distance 6, our own previous experiments have shown that using a hybrid search approach consisting of 10 minutes of initial library building with a hardware genetic algorithm (HGA), followed by 1.5 hours of hardware exhaustive search of all possible 16-mers, has shown that the HGA phase alone finds ~99% of the words that can be found [6]. The disadvantage of HGA is that it requires porting the application software to a hardware descriptive language (e.g. VHDL), and a special add-on FPGA board. While the experiments we report here are for the case of length 16 RC Edit distance codes, this approach will also work for longer codes, other metrics, and fuller cases of the overall problem.

Our basic algorithm was previously implemented in C using the message passing interface (MPI) protocol [5]. It begins with an empty code library, and uses a distributed, multi-population GA to evolve candidate words for checking and possible inclusion in the library. During the initial generation, good words are harvested from the initial population and are replaced in the population by new random individuals. During subsequent generations, the GA operators are used to improve the fitness of candidate words. We use an adjustable probability of mating, a rank based probability method for

selection for mating, and single point crossover to generate children for the next generation. For mutation we select a specified percent of individuals, and for each we check each of the 48 possible single base mutations (for 16-mers). The fitness of candidate words is measured by checking the desired constraints among strands in the library and forming a weighted sum involving the number of library strands that reject a candidate and a quantitative measure of the worst rejection. The constraints we check are Edit Distance of (word vs. RC word), (word vs. all library words), and (RC word vs. all library words). Finally, we clean the population at the end of generations, replacing clones and library word duplicates in the population with new random words.

6.0 MULTI-CORE COMPUTING

Uni-processor performance has saturated in recent years due to power dissipation issues limiting increases in clock frequencies, growing performance gap between the processor and memory hierarchy, and saturation of the parallelism possible at the instruction level [10]. Processor vendors have responded to this challenge by introducing multi-core processor architectures, where multiple cores and the shared memory hierarchy and interconnect are integrated on the same package and/or die. For example, the Intel Xeon Quad-core processor integrates two dual core Xeon processors in a package with the dual core processors sharing the level two (L2) cache integrated on a die. Similar multi-core architectures are available from vendors such as AMD, IBM and Sun Microsystems. Future multi-core processors are expected to integrate hundreds of processor cores on a single chip. Intel recently announced a prototype multi-core processor integrating 80 cores on a single chip [11]. Another example is the IBM Cell Broadband Engine (CBE) chip which incorporates a general purpose processor and 6-8 special purpose processors optimized for vector calculations. This chip is available in a number of platforms, including the inexpensive Sony Playstation 3 (PS3).

The multi-core approach requires application software to exploit the parallelism made possible by the large number of computing cores and multiple computing threads per core. Inter-processor communication through the shared on-chip memory hierarchy facilitates low communication latency compared to distributed memory clusters making possible successful implementation of parallel algorithms with a relatively high communication-to-computation ratio. Given the dominance of shared memory architectures in today's multi-core processors, thread libraries and OpenMP are widely used parallel programming models.

The next section describes the design of a C/multi-threaded code capable of running on multi-core processors for the DNA tag/anti-tag library design problem.

7.0 METHODS, ASSUMPTIONS PROCEDURES: MULTI-THREADED CODE GENERATOR

Our starting point was the C/MPI code described in [5]. We implemented the multi-threaded version using POSIX Threads (Pthreads) since threading allows for fine grained control and Pthreads is supported by a wide variety of computing platforms. The initial pre-processing is done sequentially in the master thread (thread 0). A user specified number of worker threads are spawned by the master thread with the population distributed equally between the worker threads. Communication between the threads is done through shared global arrays. Similar to the C/MPI version, each worker thread starts with an empty library, and a different population of random individuals. At the end of every generation, all the populations of each worker thread are checked by one of the worker threads (thread 1) to determine the winner thread with the largest library, and the corresponding thread ID is recorded in a shared variable. The remaining threads then copy the winner's library when they reach the end of a generation.

We do not place a blocking mutex on all threads at the end of each generation in order to simplify coding and avoid possible large time penalties due to thread stalls. Due to the lack of synchronization between the worker threads, at any point in time different threads may be at different points of execution in a generation, depending on the scheduling mechanics of the operating system. As a result, before a non-winner thread copies the winning library, either itself or the winner thread may have found more words than the winner reported at the winner's last generation boundary. We use two checks to detect these conditions just before a winner library is copied by a loser thread. First we read update the number of words found by the winner library again. Second, we only copy the winner library if it is larger than the loser's library. Even with these additional checks, words can be lost due to memory cache effects causing the winner and losers' library sizes to be different across the threads. However, this is not a serious limitation because there are a very large number of possible libraries, and the GA simply continues to build one of them without the lost words.

At the end of epochs of a few generations, a small number of the best individuals with highest fitness from each population can optionally be migrated around the threads, in a ring configuration. Again, to maximize execution speed, very little synchronization exists between the worker threads. Each worker thread executes GA generations and library updates as described above until one of the worker threads satisfies one of three user specified termination criteria: a maximum time limit; a maximum number of generations; or a target library size. The terminating worker thread then updates a shared mutex protected global variable thus sending the stop condition to the remaining worker threads. Note that the only synchronization that exists between the multiple threads is reading and possibly updating the shared mutex protected stop variable. When a stop is issued, all threads then join the master thread, and the winning library of words is read and reported by the master thread.

8.0 RESULTS AND DISCUSSION

A. LLCS metric version

This section describes a set of experiments done to characterize the performance of the multi-threaded code using the LLCS metric. We used a Supermicro Superserver 6025B-URB 2U server equipped with a dual socket quad-core (8 cores in all) Xeon 5450 3.00 GHz CPU, 12MB L2 cache and 32GB RAM running Red Hat Linux 4.1.2-14. The code was compiled using gcc 4.1.2 with no compiler optimization options enabled. We did experiments that measured performance in terms (time vs. number of words found), the basic speedup scaling vs. the number of threads (cores), and the effects of certain GA parameters and operator variations. Since the GA is stochastic, the plots in this section are point by point averages over 10 runs. The parameters noted in the heading of the figures in this section are initial population size (i), running population size (r), number of keepers (k, the number of individuals retained from generation to generation and used to for mating), a random number seed (s, -1 meaning keyed to thread and time), the number of generations per epoch (e, 0 meaning no passing at epochs). Other command line options are used to control the number of threads (t), mutation intensity (z), number of words to generate (w), etc. We chose a particular set of parameter settings as a baseline (i=r=k=1024, s=-1, w=230, e=0, max time=6 minutes), and compared results obtained using various other conditions to those obtained with the baseline conditions.

1. Speedup scaling vs. # threads

The first experiment built 16-mer codes of Edit Distance 10 using 1, 2, 4 and 8 threads, as shown in Figure 1. As expected, more words are discovered sooner (lower curves) using more threads.

We note that the total size of the population processed for each thread is the same, e.g. using 1 thread the thread population was 1024 individuals; using 2 threads the each thread population was 512 words, etc. This means that generation durations were shorter using more threads, so this measure of speedup scaling does not truly compare the speed of exactly the same calculation steps at the same point on all curves, but rather is a qualitative comparative measure of solution progress. A linear speedup scaling of 8x, 4x and 2x speedups is seen in the left portion of Figure 2. This portion corresponds to generation 0, when words are simply harvested from the initial population with replacement of picked up words with new random individuals (words 1-50), rather than being produced by the GA operators as in later generations. The linear speed-up of generation 0 is due the absence of interaction costs since the threads operate independently of each other here, and the scaling down of population size per thread with increase in the number of threads. As discussed later, smaller population sizes find words faster early. After generation 0, words are found using the GA operations and the resulting interactions between the threads in the form of passing newly found words at the end of generations causes the sub-linear speed-up seen in the right portion of Figure 2. This effect is also discussed further in Section 8.

Figure 2 shows the speedup scaling, calculated as the ratio of execution times for the same number of words generated by 2, 4 and 8 threads to that of 1 thread.

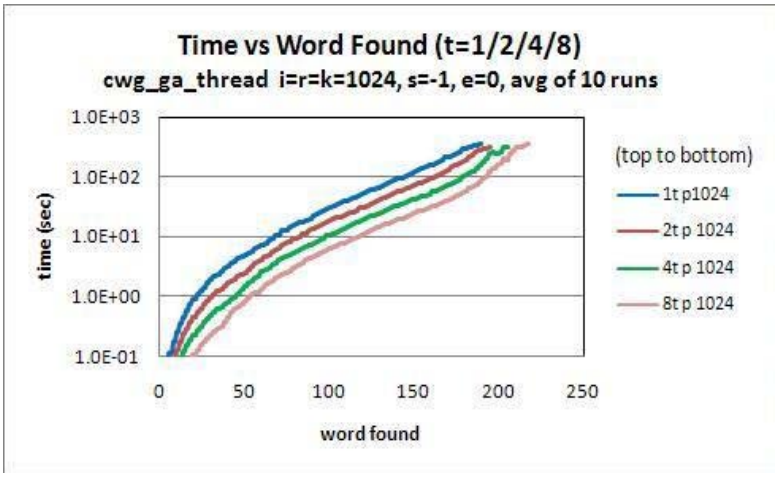

Figure 1. Library growth, # of threads varied.

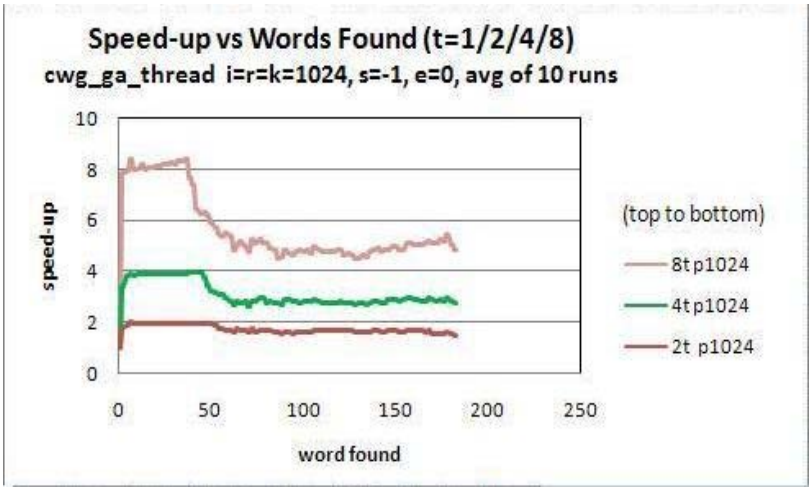

Figure 2. Speedup scaling vs. # threads.

The next experiment checked the effect of population size on performance, by repeating the previous experiment with different total population sizes of 512, 256 and 64 for the case of 8 threads.

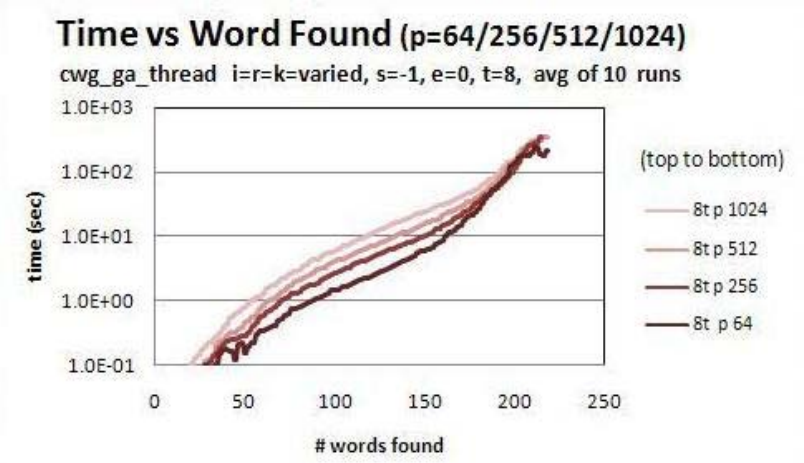

Figure 3. Smaller populations are faster early, but slower as the library size reaches about 150 words.

Figure 3 shows that smaller populations find words somewhat faster early, but the performances begin to converge as the last few words are discovered. The results were similar using 1 thread. Fitness evaluation dominates the computation time (~98%), growing exponentially with library size, and eventually dominating differences due to running the GA operators on different population sizes.

2. Mating:

The next experiment varied the probability of mating by changing the number of individuals kept at the end of generations to produce children (100%, 50%, 10%), again for 8 threads. Figure 4 shows that a high probability of mating is disruptive. Higher mating replaces individuals (that were previously improved by the GA operators) by new ones generated by single point crossover, which can greatly change fitness. Again the results were similar using 1 thread.

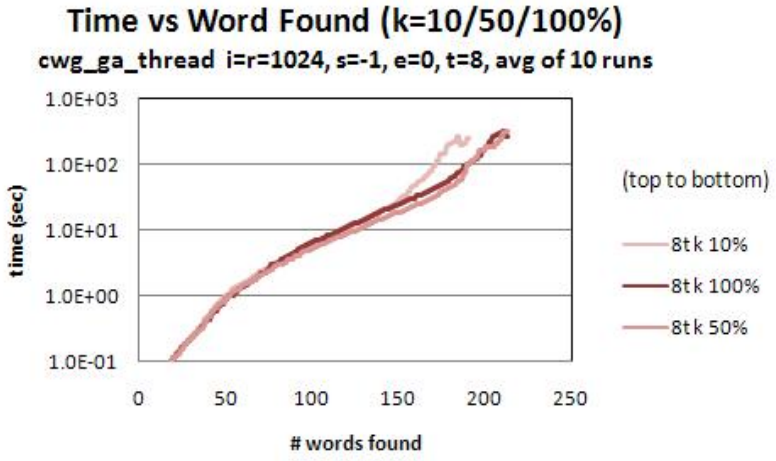

Figure 4. Effect of mating probability, 8 threads.

3. Mutation:

The next experiment tested the effect of mutation intensity, and mutation mode. Mutation intensity is controlled by varying the percent of individuals in the population selected for mutation. We tried two mutation modes which differed if mutation failed to improve fitness: mode 1 mutated the individual at a random base anyway; and mode 2 replaced the individual with a new random individual.

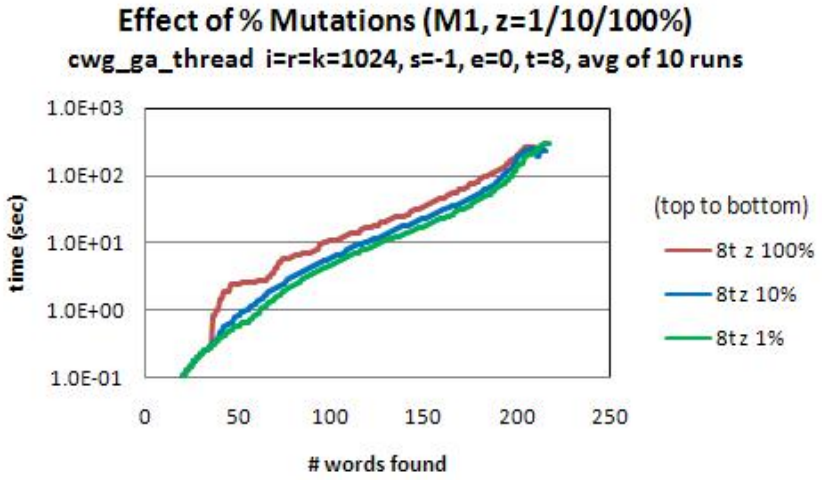

Figure 5. Mutation intensity varied, mutation mode 1, 8 threads. Very high mutation rates are slower.

Figure 5 shows the results for mutation mode 1, for various probabilities of mutation. Heavy mutation is disruptive, and the effect was similar for mutation mode 2, as shown in Figure 6. There was no significant advantage of either mode over the other.

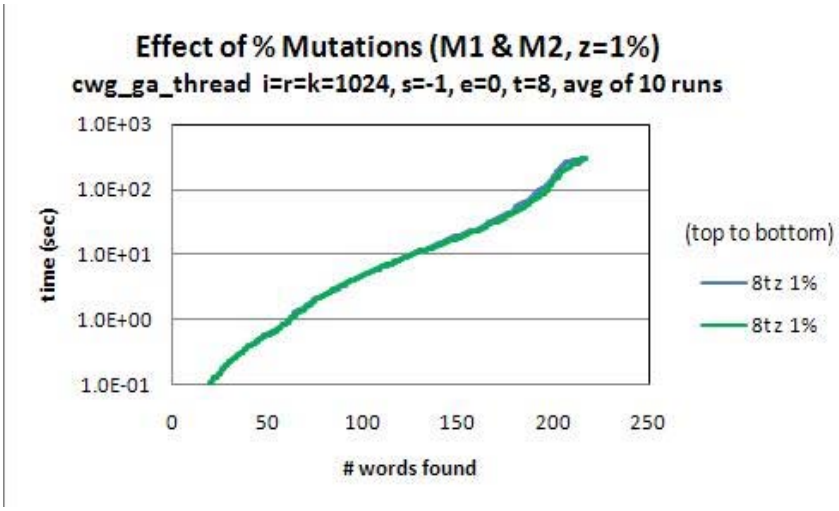

Figure 6. Mutation intensity varied, mutation mode 1, 8 threads. Performance is better with lower percent mutations.

4. GA/Random:

This experiment looked at the effect of using new random words instead of the GA to generate the next generation of the population. Figure 7 shows that this very simple heuristic does build libraries, and that performance scales vs. the number of threads. However, Figure 8 clearly shows that random search stalls, and GA eventually finds far more words.

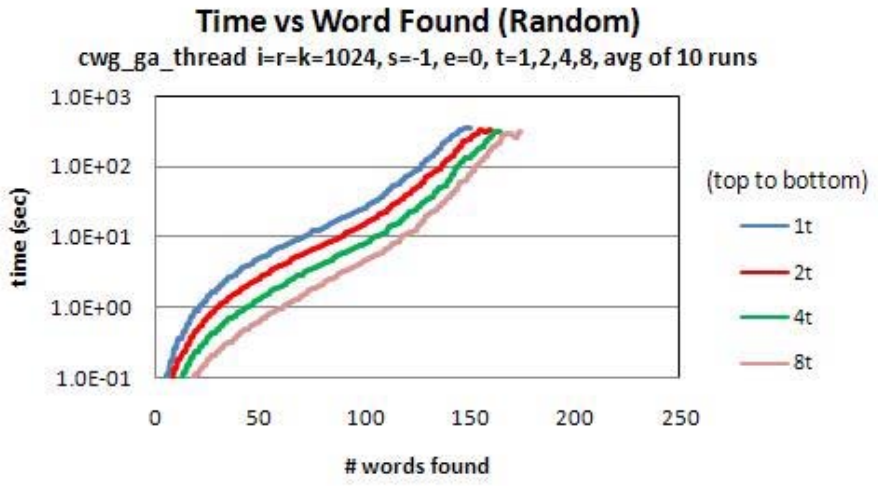

Figure 7. Speedup scaling for random search (no GA).

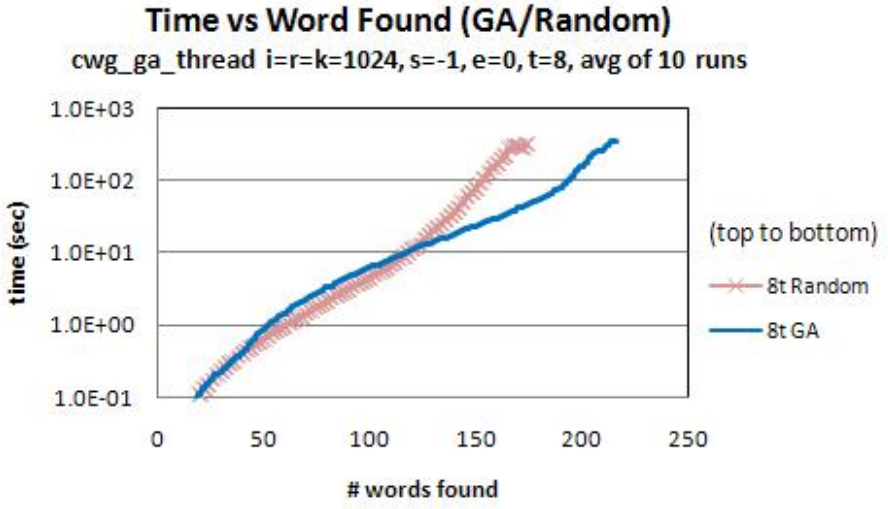

Figure 8. Random candidate generation terminates finding far fewer words that GA.

5. Drifting at Epoch boundaries

This experiment looked at the effect of the number of generations per epoch, and the number of individuals drifted around the ring of threads at epoch boundaries. Variations of either parameter had little effect, as shown in Figures 9 and 10.

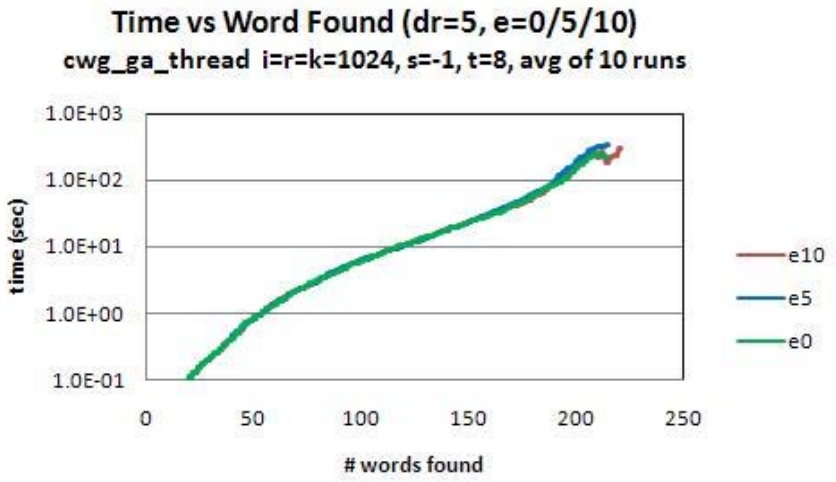

Figure 9. Performance vs. number of generations in epoch.

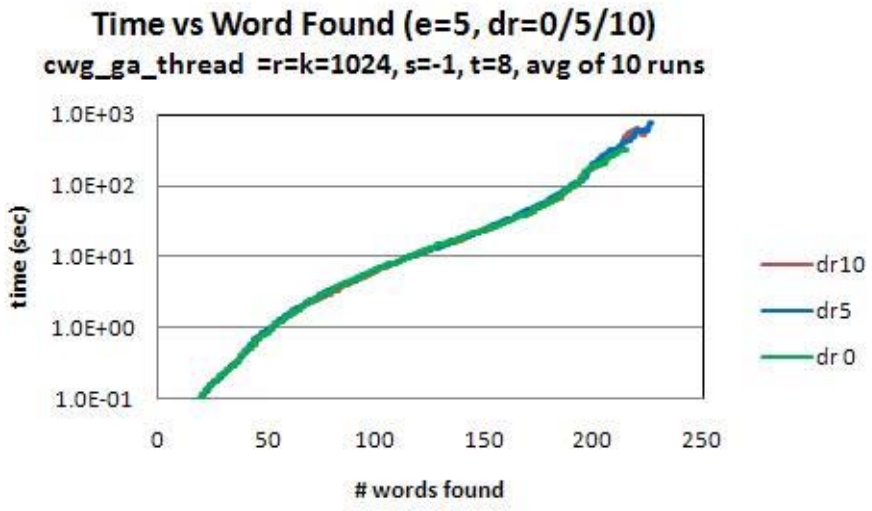

Figure 10. Performance vs. number of individuals drifted at epoch boundaries.

6. *Clean up vs. no clean up*

The cleanup step intended to remove clones and duplicate library words from the GA population had little effect on performance for the baseline conditions because there was no mating, and the development of clones with only mutation is unlikely. However, with mating the population can fill with clones, wasting time operating on identical individuals.

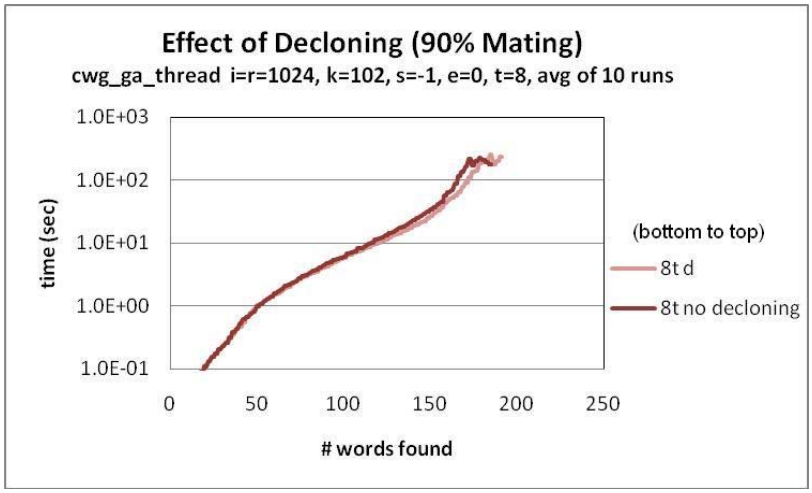

Figure 11. Effect of decloning at end of generations.

7. *Distribution of library lengths*

The final experiment did 100 runs with the baseline conditions, for 15 minutes each, and noted the final size of the library. Figure 12 shows the resulting distribution of the library lengths.

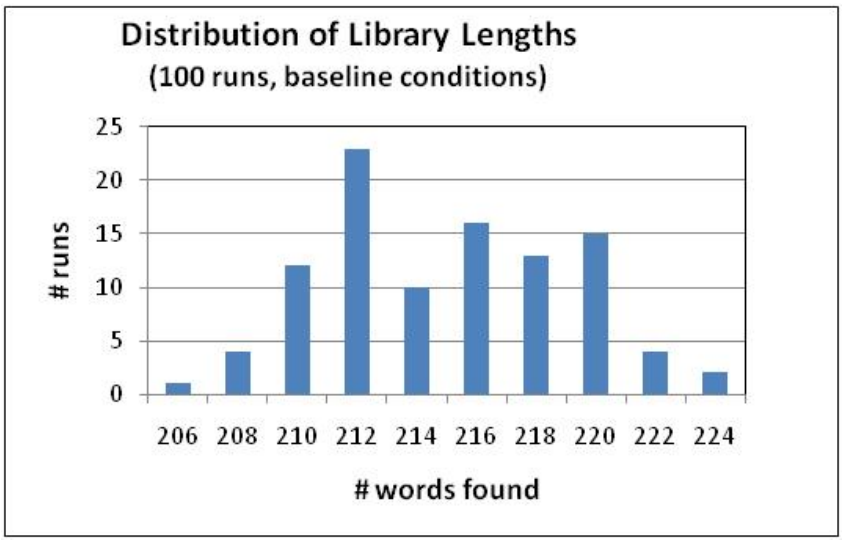

Figure 12. Distribution of library lengths for 100 runs

8. Barrier synchronization

In order to investigate the effects of lack of synchronization between the worker threads during the GA operations, barrier synchronization constructs were incorporated into the threaded code. This ensured that all worker threads were at the same execution point before the winner thread library was copied by the loser threads, thereby preventing possible loss of words found (see Section 6) albeit with time lost due to synchronization. However, as seen in Figure 13, a comparison of execution times for different numbers of threads between the barrier and no-barrier version of the threaded code does not show appreciable difference in the execution times. This suggests that the time lost due to barrier synchronization is compensated by the words found by the threads not being lost.

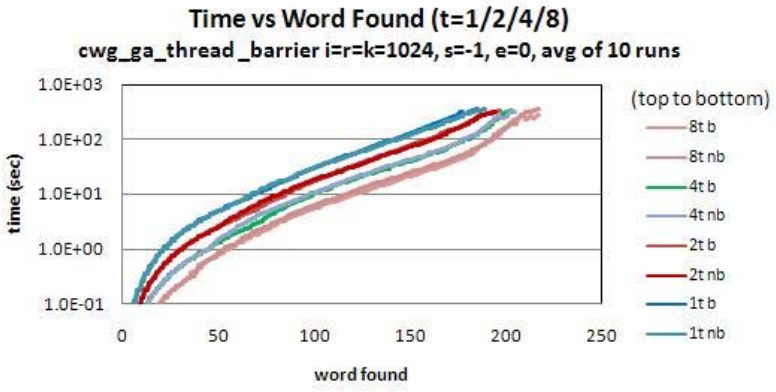

Figure 13. Comparison of speed-up scaling for threaded code with and without barrier synchronization

9. Comparison with Previous MPI and VHDL FPGA Results

The speed-up behavior of a previous C/MPI version of this application that ran on a cluster of compute nodes (using 1 processor core per node) is shown in Figure 14 (heavy lines). Also shown are the speedup curves using the Pthreads version (light lines). The curves for 2 threads is similar, but the speedups for 4 and 8 threads are lower for Pthreads than for MPI. We believe the difference is due to shared memory effects and lost words in the Pthreads version. We also note that the MPI version shows a dip in speedup after generation 0, similar to the Pthreads version, which suggests a similar effect due to communication.

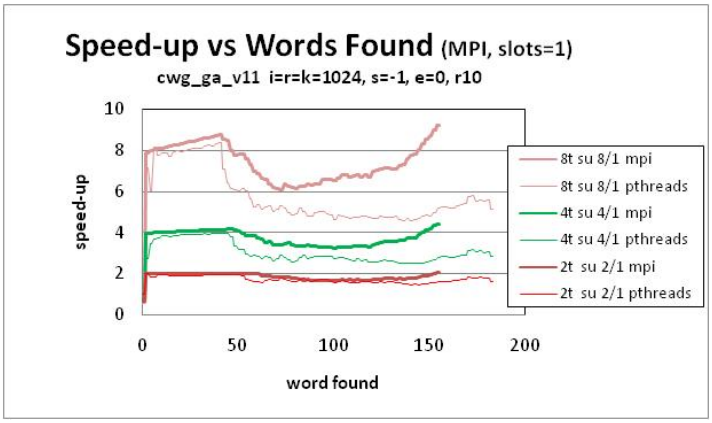

Figure 14. Speed up scaling vs. number of processors in the MPI version.

We also note here that the communication in the MPI version was tuned and quite different, i.e. only new words were passed from winner to loser threads, and the synchronization due to MPI constructs was different from the minimal synchronization in the Pthreads version. It is possible that the Pthreads version speedup could be improved by using more complex Pthreads signaling and synchronization methods.

In [6] we reported results for building similar codes using hardware acceleration methods and an FPGA. Although the mean library sizes obtained in that work were 240 word pairs, those codes included 22 initial words of n/d=16/8 which were determined by construction. Subtracting these initial words leaves gives an average length of about (240-22) = 218 words, which compares reasonably well to the results in Figure 12. The average size was slightly higher in the previous work because the hardware GA was run for 10 minutes, and it achieved a measured speedup of 1000x over software. So, the equivalent software search time in the previous work (10,000 minutes) was significantly higher than the 15 minutes used in the present work.

B. Thermodynamic metric

This section describes a set of experiments done to characterize the performance of the multi-threaded code using the thermodynamic metric [8,13]. Using this metric, we use constraints that allow a word to be added to the library only if the free energy of intended binding between the word and its reverse complement lies within a specified range, and if the free energies of unintended binding between the word and its RC and all other library words fall below a chosen upper limit. Additional parameters were incorporated into the code to specify an upper bound on the unintended binding energy (n), a lower bound on the intended binding energy (o) and an upper bound on the intended binding energy (p). The experimental set up is the same as described above in Section 7, i.e. we mostly used a baseline set of parameters that included i=r=k=1024, e=0, z=1, run time of 3 or 6 minutes, and averaged the results over multiple runs. To get an idea of where the new energy constraint bounds should be set, we calculated the binding energies with the nearest neighbor model for a sample of 1000 randomly selected pairs of 16-mers, as shown in Figure 15. We chose a 'loose' set of constraints to be n=710, o=729, and p=1400, and a tight set of constraints to be n=710, o=925, and p=950. (We note there are methods to translate these energies into melting temperatures).

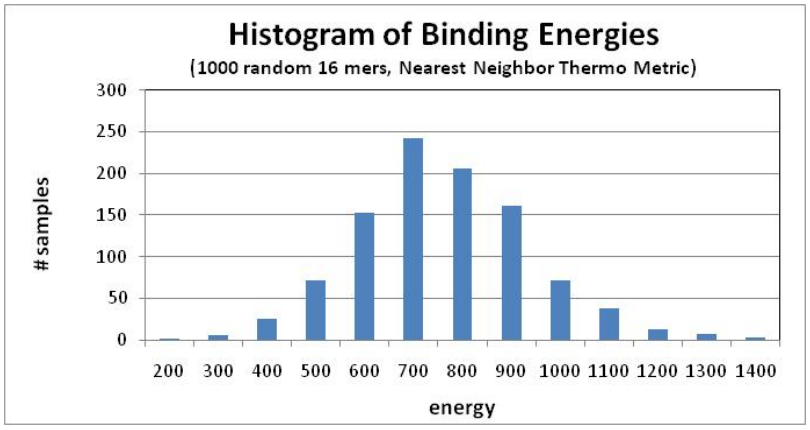

Figure 15. Nearest neighbor binding energy estimates for 1000 randomly selected 16-mer pairs.

1. *Speedup scaling vs. # threads*

The first experiment built 16-mer codes with a tight set of constraints using 1, 2, 4 and 8 threads, as shown in Figure 16. As expected, more words are discovered sooner (lower curves) using more threads.

Again, we note that the *total* size of the population processed for each curve is the same, e.g. using 1 thread the thread population was 1024 individuals; using 2 threads the thread population was 512 words, etc. Comparing Figure 17 with Figure 2 we see that the amount of time to find the first few words is about ~10 times larger using the thermo metric with tight constraints than it was using the LLCS metric. Second, there is slight speed-up apparent early in the curves during the initial random search on generation 0 (again due to a smaller population size being processed faster with no GA operators for larger # threads). But due to the tight constraint, only a small number of words are found on generation 0 before switching to GA search. After switching to GA after generation 0 the speedup clearly depends on the number of threads, as expected. This illustrates that the value of using GA over random search increases for a harder case of our problem. Figure 17 shows the speedup scaling, calculated as before as the ratio of execution times for the same number of words generated by 2, 4 and 8 threads to that of 1 thread.

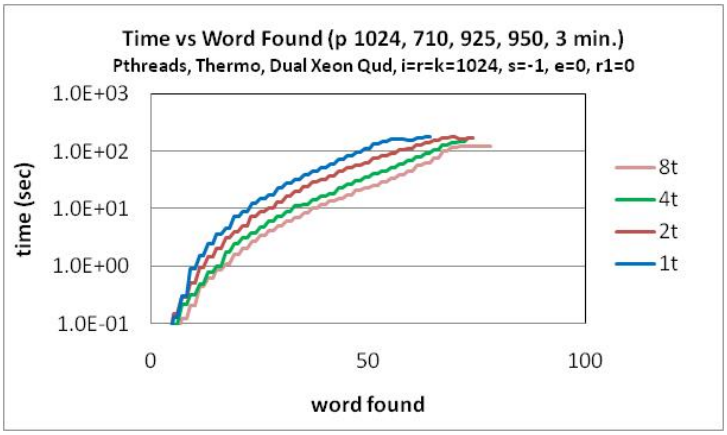

Figure 16. Library growth, # of threads varied.

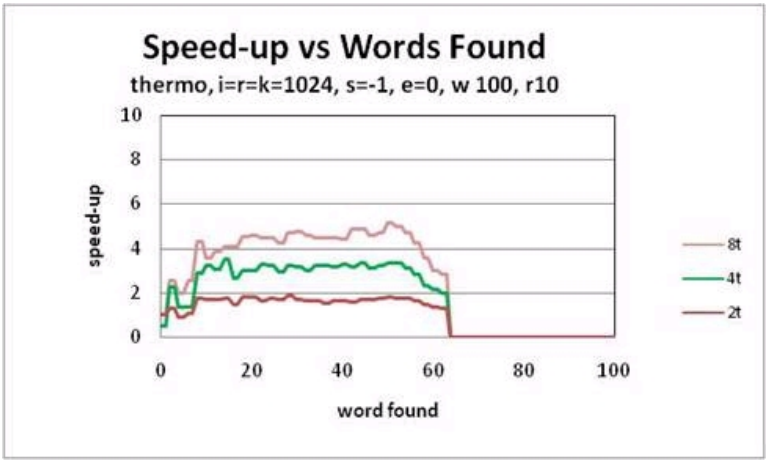

Figure 17. Speedup scaling vs. # threads.

2. *Speedup scaling with no thread interactions*

In order to study the speed-up scaling independent of thread interaction overheads, we disabled all operations involving copying of library words across threads such that the threads effectively ran independent of each other. The thermodynamic constraints are utilized with a looser energy binding constraint. From Figure 18, it is seen that even with no interactions between the threads, for words found beyond generation 0, the speed-up scaling drops with thread scaling. For a given number of words, it is seen that the number of generations required by the single threaded case is less than the multi-threaded case. We believe this effect is due the larger population size for the single threaded case. Note that for each additional generation the computationally expensive mutation operation needs to be done. Although the mutation operation is *O*(population size), the larger number of generations required with thread scaling results in an overall drop in speed-up. For comparison, Figure 19 shows the speed-up when the thread interactions are enabled. With thread interactions enabled, there is clearly a beneficial effect obtained by harvesting words from the larger, distributed, population, and the effect is stronger for more threads (faster processing of smaller populations). Finally, we note that for looser energy binding constraints, the speed-up plots resemble those of Figure 2 (using the LLCS metric).

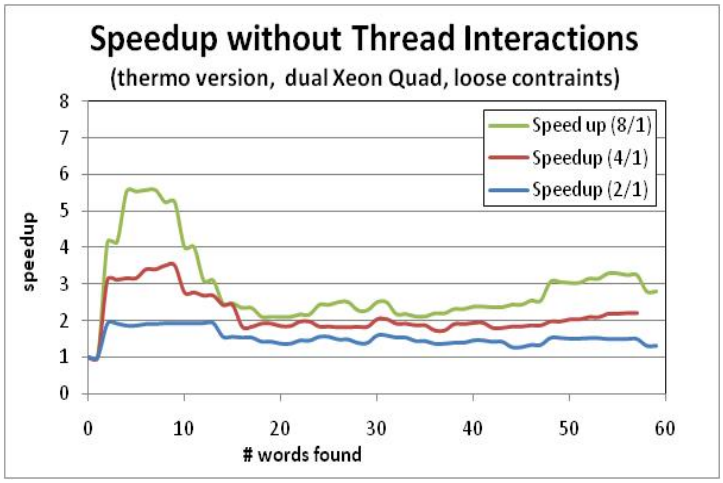

Figure 18. Speed-up scaling with # threads with no interaction between threads.

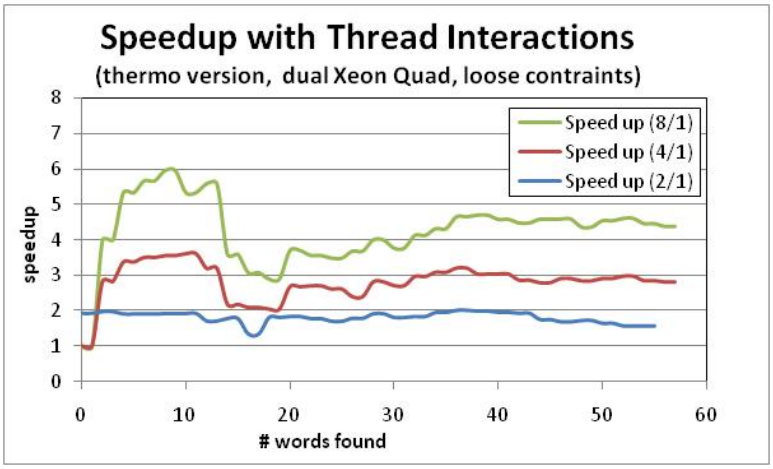

Figure 19. Speed-up scaling with # threads with interaction between threads.

3. Speedup scaling on the AMD Opteron quad-core

We investigated the scaling behavior of the multi-threaded DNA tag/anti-tag library generator employing loose energy binding constraints executing on a work station with dual socket quad-core AMD Opteron processor (8 cores in all). Figure 20 compares the speed-up scaling for 8 threads for the dual socket quad-core AMD Opteron, dual socket quad-core Intel Xeon, and a single socket quad-core Intel Xeon. From Figure 20 we note that overall the quad-core AMD Opteron shows a better speed-up scaling compared to the qaudcore Intel Xeon.

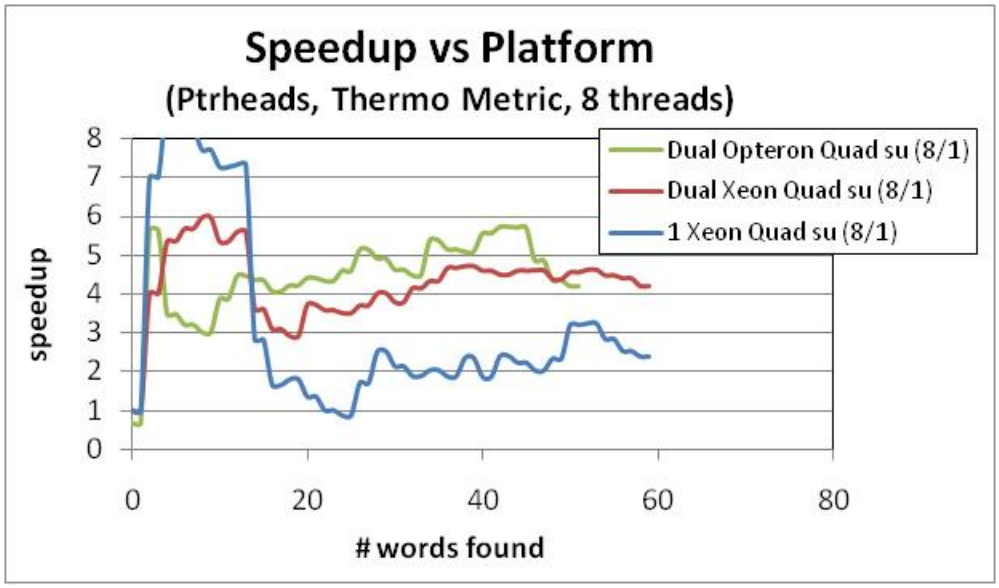

Figure 20. Speed-up scaling with # threads on various platforms.

C. Speedup scaling on the Cell Broad Band Engine(CBE)

A preliminary version of the Pthreads, LLCS metric application was written for the CBE, and evaluated on a cluster of CBEs at AFRL containing a number of Sony Playstation 3's. Each CBE contains a PPE which is a 64-bit, Dual Threaded Power PC (32KB L1 cache, 512KB L2 cache), and 6 SPEs (RISC Core with 256KB Local Store), with DMA communication between PPEs and SPEs. This version used MPI between CBE PPEs, and the IBM SDK 3 API to access the 6 SPEs on each CBE. The method of parallelizing the application was significantly different in this version compared to the previous Pthreads versions. First, similar to the MPI version used in previous work [5], it distributed the GA population across CBE PPEs, here using openmpi, (instead of running separate populations on different threads on the same node as in the Pthreads version). Second, it parallelized the evaluation of the 48 mutation candidates in chunks divided across the SPEs (instead of evaluating mutation candidates sequentially on the same node as in the Pthreads version). We thought this approach would be a better match with the strengths of the CBE platform by simplifying the code run on the SPEs. As in the Pthreads version, the SPE threads were set up and started in the main thread running in the PPE, and were left running throughout the solution, to avoid overhead due to thread creation and joining for each SPE call. Due to development time constraints, the code in the SPE was not vectorized, and the branches in the SPE code were not hinted, and the two SPE pipelines were not tuned; all of which would be expected to improve performance. Two experiments were done with this version, the first to evaluate speedup vs. the number of SPEs used on one CBE node compared to PPE speed, and the second to evaluate MPI speedup between large numbers of CBE nodes using only the PPE.

Figures 21 and 22 show the results of the first experiment. First, we see that there is no speedup during generation 0 (up to about 60 words), because the GA population is managed on the PPE and is a constant size regardless of the # SPEs used. Second, we see that there is a speed penalty when moving from the PPE to one SPE, which is expected. Third, there is speedup when more SPEs are used, and performance is higher using 6 SPEs than when using the PPE, at least for the part of the Figure 21 between 65 and 120 words. Figure 22 also shows an estimate of the performance for 6 SPEs that we would expect by adding 4 way vectorization in the SPE code, since it should be possible to process four 16-mer checks simultaneously using 64 bit vectorized instructions. The projected ~7X peak speedup is in line with expectations for this platform, i.e. about a 10x speedup for 'pointer chasing' applications (i.e. not regular array processing like matrix multiplication).

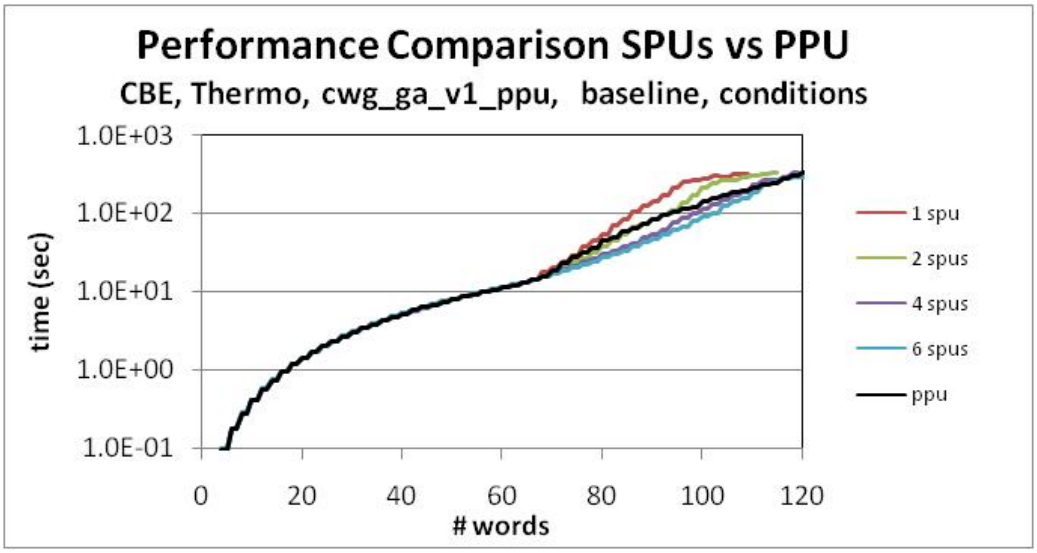

Figure 21. Performance of CBE version on 1 PS3 node, with # SPEs varied.

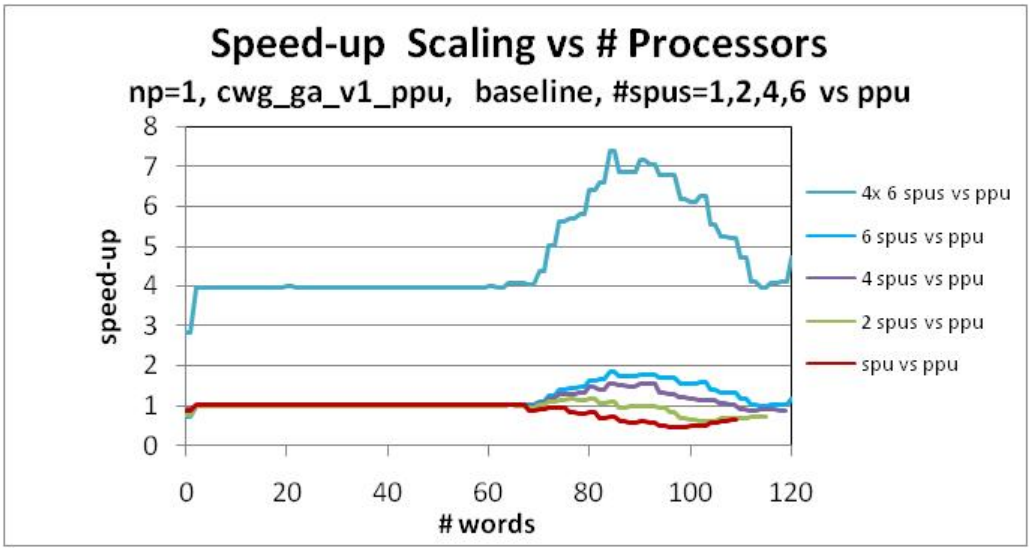

Figure 22. Speed-up scaling (SPUs vs. PPU) with vs. threads on the PS3 CBE.
(Top curve is estimated performance with 4 way vectorized code).

The results of the second experiment are shown in Figure 23 and 24. Here we have run the distributed GA population on a large number of openmpi nodes, using 2 slots per CBE PPE (since it is hyper-threaded). Again we see significant speedup in the first part of the curves due to processing smaller populations with more nodes during generation 0, similar to the Pthreads version. Speedup improves during the later part of the curves, probably because of less communication since few words are found on later generations, again similar to what was seen with the Pthreads version. We did not investigate cache misses with this version, but we would expect that it might be an issue, even though each node is running 2 threads, compared to the Pthreads version running up to 8 threads.

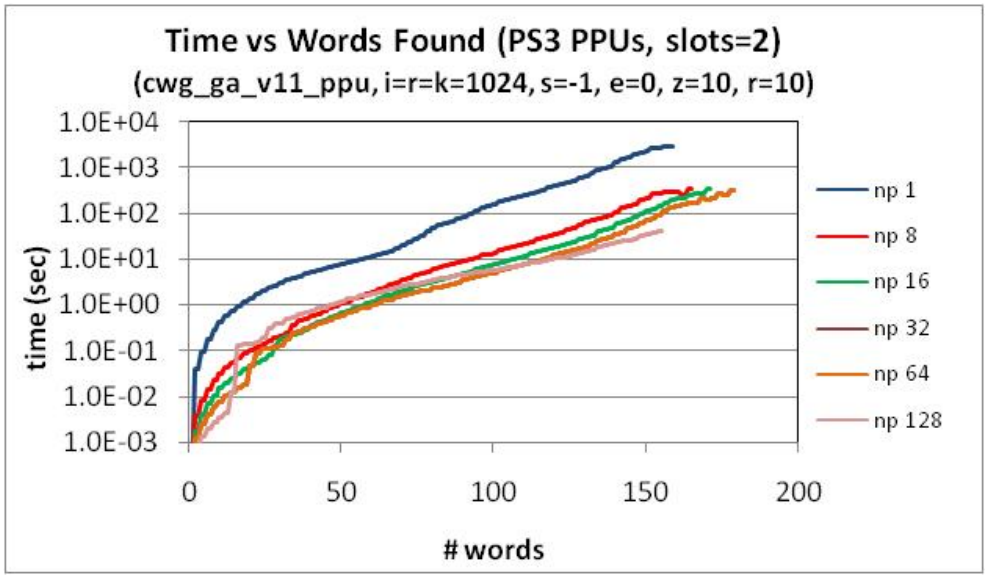

Figure 23. Performance vs. # MPI nodes using 2 processes running on each PPE, on multiple CBE nodes.

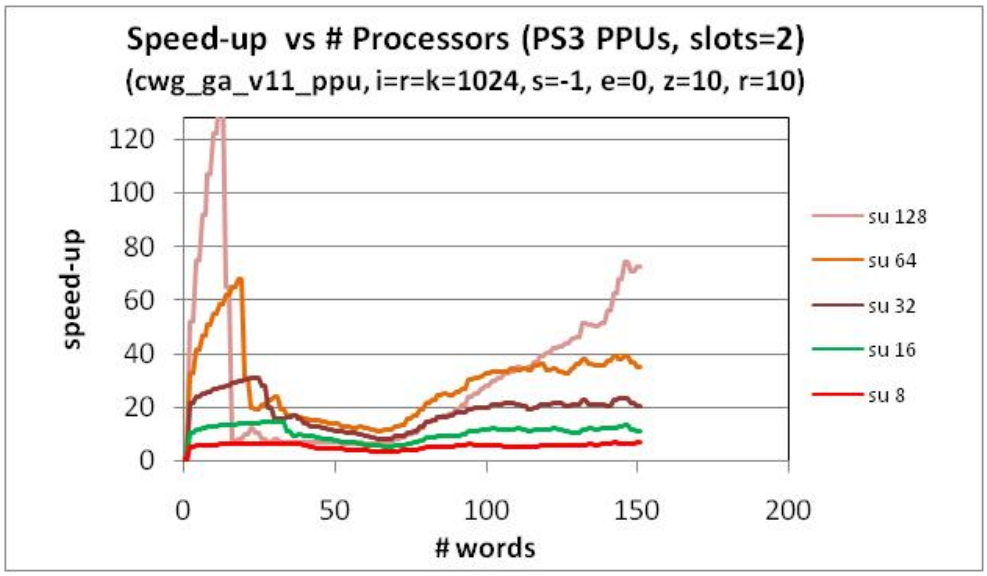

Figure 24. Speedup for the data of Figure 23.

Finally, Figure 25 shows a plot of speedup for this experiment, compared to perfect linear speedup. The speedup falls off with large numbers of nodes, although it is nearly linear for small numbers of nodes.

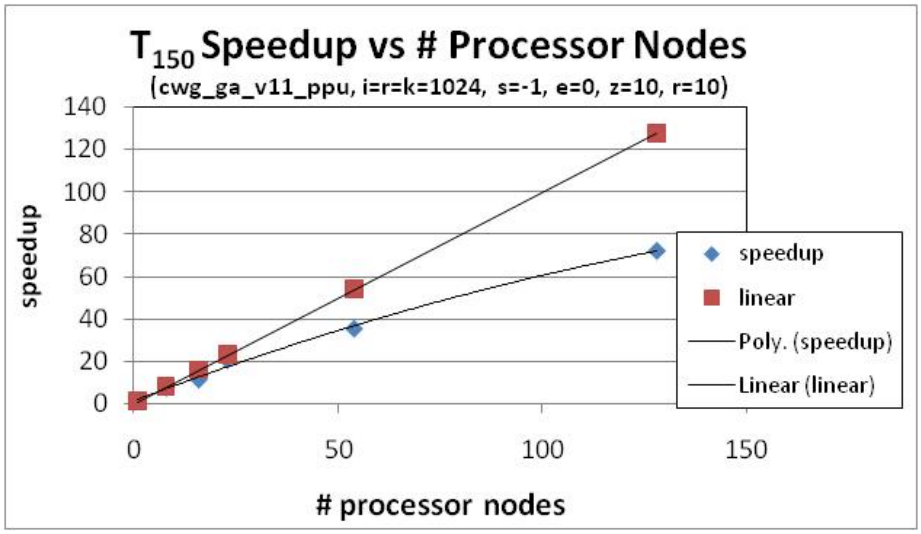

Figure 25. Speedup of MPI CBE experiment, with termination criteria of 150 words found.

9.0 DISCUSSION

The multi-core computing paradigm allows exploitation of coarse grained parallelism available in many computationally intensive problems. Since the programming can be done using conventional high level languages such as C/C++ or Java, it is more accessible to a broader group of users than the programming required for hardware accelerators based on FPGAs. For example, an FPGA version of the same algorithm implemented using VHDL achieved a 1000x speedup [6] but was expensive both in terms of design time and resources - it required a $16K add-on hardware FPGA board. Moreover, the FPGA platform is far less flexible and scalable compared to a multi-core implementation. Our multithreaded code was relatively easily extended to incorporate the nearest neighbor pair-wise thermodynamic metric [8], and could be modified with minimal effort to incorporate other binding energy estimate metrics such as Smith-Waterman [7], or full thermodynamic estimates (e.g. M-fold) compared to casting those in VHDL and FPGA.

One can also argue that the C/Pthreads programming paradigm is more accessible than the C/MPI programming paradigm. For example, the previous C/MPI version of this application required careful placement, profiling, and tuning of a number of communication events, and selection of GA parameters to achieve a high compute to communication time ratio. However, the present C/Pthreads version was successfully parallelized with minimal synchronization between threads.

With hundreds of cores expected to be available on commercial multi-core processors, orders of magnitude speed up of algorithms is possible without expensive hardware. However, a suitable parallelization scheme must be chosen that allows load balance and minimization of stalls due to data dependences.

We have demonstrated that by carefully exploiting the algorithm it is possible to minimize the critical sections in the code, allowing all threads to operate in parallel and obtain good speed up. In our case, the heuristic nature of the GA algorithm enabled us to sacrifice some of the words generated in exchange for minimum synchronization between the threads. Even so, care must be taken to ensure that poor data locality and false sharing does not cause the shared cache hierarchy to become a performance bottle neck. In the quad-core Xeon processor used in this work, two dual-core Xeon chips are paired onto a single multi-chip module (MCM). Each core includes a 32KB L1 cache and each chip (two cores) share a 4 MB L2 cache. As a result, for more than two threads, the interaction occurs through the slower main memory resulting in the increased speed-up degradation for the 4 and 8 thread cases of Figure 2. To verify this further, we measured the L2 cache miss events using the Intel VTune performance analyzer [9]. For a baseline run of the multi-threaded code, the number of L2 data cache misses increases by 10.5% when we increased the number of threads from 2 to 4. However, no further change in the number of L2 data cache misses is observed by increasing the number of threads from 4 to 8. A similar 10.5 % increase in the number of L2 data cache misses was observed in the two threaded cases when the two threads were bound to cores that did not share the L2 cache. These misses partially undo the scaling benefits of adding additional cores to the system.

Unlike the Intel Xeon quad-core, AMD Opteron quad-core integrates all four cores on a single die with private L1 and L2 caches and a shared L3 cache. Consequently, we see a better speed-up scaling with the Opteron quad-core as compared to the Xeon quad-core. As multi-core processors with an increasing number of cores sharing the on-chip cache hierarchy are introduced by processor vendors, we can expect better thread scaling from a multithreaded implementation of the DNA tag/anti-tag computing problem. For example, the recently announced Intel 8-core Nehalem processor integrates 8 processor cores on a die with a shared L3 cache [12].

The heterogeneous architecture of the Cell Broadband Engine also provides interesting opportunities for accelerating the DNA tag/anti-tag library generation problem. That work is ongoing and is outside the scope of this CRADA.

10.0 CONCLUSIONS

In this report we have described a new approach that couples multi-threaded coding methods and a distributed multi-population genetic algorithm and illustrated their use in solving a simple example case of the DNA tag/anti-tag code design problem. This approach obtains speedups over single threaded codes by utilizing the multi-core CPUs present in modern workstations. We believe that the programming difficulty of implementing this approach is reduced compared to a previous C/MPI version run on a cluster, and the platform is certainly less expensive both in terms of cost and power. While the performance levels reached by this application run on today's workstations (up to 8x-64x depending on the platform product) do not reach those obtainable using reconfigurable hardware acceleration (1,000x), we believe the programming methods again are a better fit to those widely taught to problem domain experts. They will also become increasingly important as more cores appear in commodity workstations

We observed that there are complex interactions between the algorithm and the underlying architecture when parallelizing the algorithm. While the computing power is generally enhanced by using additional cores, there may be incremental penalties such as cache misses and lost code words due to thread interaction overheads (architecture effects) the need to process more generations due to smaller population size per thread, and the possibility of lost code words due to communication style (algorithm effects) that present interesting trade-offs. The results so far suggest that unless the overall problem size is scaled with the number of threads, bringing a large number of cores to bear on the problem will not yield a strictly proportional speed-up. We also note that the probabilistic nature of the algorithm and the hardware cache coherence mechanism of shared address space machines make the theoretical analysis of the problem difficult.

11.0 RECOMMENDATIONS

Improvements to this work could be made by tuning the inter-thread communication style to incorporate more sophisticated Pthreads interactions, and perhaps by trying other parallelization schemes. For example, the present work used separate threads to host separate GA populations, but a more fine grained parallelization strategy might use separate threads to calculate strand binding affinity for one candidate word against all words in the library, or for one strand-strand calculation. Also, SIMD instruction processing is provided on most CPUs, and this could be leveraged to achieve additional speedup by performing calculations parallelized at the instruction level, e.g. four (or possibly more) calculations on different strand pairs can be done in using one 64 bit SIMD instruction.

Another possible area for future work to build on the results of this project would be to develop analytical models to quantify the cost/benefit relationships associated with certain architectural and algorithm tradeoffs. For example, communication style between Pthreads , cache coherency effects, and smaller population sizes with more Pthreads are some factors that can lead to lost words and longer computing times. Although we observed that their negative effects are tolerated by the robust genetic algorithm analytical models would allow optimization studies that might identify further performance improvements. Also, dynamic adjustment of GA parameters during run-time might prove interesting.

Another area for future work would be the development of hybrid MPI and Pthreads versions, i.e. using MPI between nodes, and Pthreads within nodes in a cluster of (possibly heterogeneous) multi-core nodes. In fact, we did preliminary work on such a hybrid version that used openmpi between CBE PPEs, and the normal IBM SDK 3 mechanism for using multiple SPEs in the CBE chips, but a similar version could be developed for use on clusters of multi-core commodity Xeon, Opteron, and SPARC CPUs.

Finally, the methods described in this report (i.e. the use of evolutionary computing methods and the Pthreads programming model) could be applied to other similar or larger problems (e.g. probe/target design for biological assay methods), could incorporate different binding metrics, and could be evaluated on other computing platforms.

12.0 REFERENCES

[1] J.P. Nolan, F. Mandy, “Multiplexed and Microparticle-based Analyses: Quantitative Tools for the Large-Scale Analysis of Biological Systems”, *Cytometry A*. 2006 May; 69(5): 318–325.

[2] L. Kaderali, A. Deshpande, J.P. Nolan, P.S. White, “Primer-design for multiplexed genotyping”, *Nucleic Acids Res.* 2003 March 15; 31(6):1796–1802.

[3] S.-Y. Shin, I.-H. Lee, D. Kim, and B.-T. Zhang, “Multiobjective Evolutionary Optimization of DNA Sequences for Reliable DNA Computing”, *IEEE Transactions on Evolutionary Computation*, vol. 9(20), pp.143-158, 2005.

[4] Vinhthuy Phan and Max H. Garzon , “On Codeword Design in Metric DNA Spaces”, *Natural Computing*, DOI 10.1007/s11047-008-9088-6, Springer Netherlands, June, 2008.

[5] D. Burns, K. May, T. Renz, and V. Ross, “Spiraling in on Speed-Ups of Genetic Algorithm Solvers for Coupled Non-Linear ODE System Parameterization and DNA Code Word Library Synthesis,” *MAPLD International Conference*, 2005.

[6] Qinru Qiu, D. Burns, Q. Wu, and Prakash Mukre, “Hybrid Architecture for Accelerating DNA Codeword Library Searching*,” Proc. IEEE Symposium on Computational Intelligence in Bioinformatics and Computational Biology*, April 2007.

[7] R.K. Karanam, A. Ravindran, and A. Mukherjee, “A Stream Chip-Multiprocessor for Bioinformatics”, *ACM SIGARCH Computer Architecture News*, Vol. 36, No. 2, May 2008, pg. 2-9.

[8] A.G. D’yachkov, A.J. Macula, W.K. Pogozelski, T.E. Renz, V.V. Rykov, and D.C. Torney, “A Weighted Insertion-Deletion Stacked Pair Thermodynamic Metric for DNA Codes,” *Lecture Notes in Computer Science*, Vol. 3384/2005, pp. 90-103, Springer Berlin/Heidelberg.

[9] J. Reinders, “VTune Performance Analyzer Essentials”, *Intel Press*, 2005.

[10] Eric Bangeman, “Intel pulls the plug on 4GHz Pentium 4”, *Ars Technica*, October, 2004.

[11] Sriram Vangal, et al. "An 80-Tile Sub-100-W TeraFLOPS Processor in 65-nm CMOS," 1. *IEEE Journal of Solid-State Circuits*, Vol. 43, No. 1, Jan 2008.

[12] “Next Generation Intel Micro Architecture” (Nehalem), *Intel White Paper,* 2008.

[13] Qinru Qiu, Prakash Mukre, Morgan Bishop, Daniel J. Burns, Qing Wu: Hardware Acceleration for Thermodynamic Constrained DNA Code Generation. DNA 2007: 201-210.

[14] A. Ravindran and D. Burns, “Multi-Threaded DNA Tag/Anti-tag Library Generator for Multi-Core Platforms”, 2009 IEEE Symposium on Computational Intelligence in Bioinformatics.and Computational Biology, Nashville, TN, Apr. 2009.

13.0 LIST OF SYMBOLS, ABBREVIATIONS, AND ACRONYMS

AFRL	Air Force Research Laboratory
AMD	Advanced Micro Devices, Inc.
API	Application Programming Interface
CBE	Cell Broad Band Engine
CPU	Central Processing Unit
CRADA	Cooperative Research and Development Agreement
DMA	Direct Memory Access
DNA	DeoxyriboNucleic Acid
FPGA	Field-Programmable Gate Array
GA	Genetic Algorithm
GB	Giga-Byte
GC	Guanine-Cytosine
HGA	Hardware Genetic Algorithm
IBM	International Busness Machines Corporation
ID	IDentification
LLCS	Length of the Longest Common Subsequence
MB	Mega-Byte
MCM	Multi-Chip Module
mer	oligomer
M-fold	Multiple-fold
MPI	Message Passing Interface
mutex	mutual exclusion
NP	Nondeterministic Polynomial-time
PNNM	Pairwise Nearest Neighbor Model
POSIX	Portable Operating System Interface for Unix
PPE	Power Processor Element
PPU	Power Processor Unit
PS3	Playstation 3
Pthreads	POSIX Threads
RAM	Random Access Memory
RC	Reverse Complement
RI	Information Directorate
RISC	Reduced Instruction Set Computing
SDK	System Develoment Kit
SIMD	Singe Instruction Multiple Data
SNP	Single-Nucleotide Polymorphism
SPARC	Scalable Processor Architecture
SPE	Synergistic Processing Element
SPU	Synergistic Processing Unit
UNCC	University of North Carolina at Charlotte
VHDL	VHSIC Hardware Description Language
VHSIC	Very High Speed Integrated Circuits

APPENDIX A

Software Code Descriptions

A1. The VHDL code that contain the GA and LLCS metric systolic array functionalities that AFRL/RI will provide to UNCC under this CRAA reside in directory "C:\annapolis\wildcard4\examples\HGA_DNA" on a Gateway workstation in the office of the AFRL/RI POC. It consists of a number of .c, .h, and .vhdl files that can be used to simulate and synthesize an FPGA personalization that implement an application that can design non-hybridizing DNA libraries of 16 base pair duplexes that satisfy certain user selectable constraints. The C programs were supplied in a source, include, and build directories that contain C and other supporting files, executables and a synthesized FPGA image file, and .h files, respectively. The source files are in C, and the target FPGA is that of the Annapolis Micro Wildcard4 PCMCIA card. The VHDL codes will be supplied in the format of a ModelSim project that includes synthesizable source VHDL files for implementing the GA/DNA Library design application. The codes include references to supporting software specific to the target FPGA device and card, which cannot be distributed by AFRL, so to completely simulate the codes would require that UNCC purchase of those products (~$2K) in order to simulate the application. VHDL codes were not further developed under this CRADA. This code and performance results were described in [6].

A2. The codes that implement a C/MPI version of a distributed Island Model GA/DNA code design application for workstation cluster reside in file "C:\Danb\BioComputing\BAA_01-26_BioComputing\biospice\DNA code word generation\zips\200603_03_7_runs_on_latte.zip" on a Gateway workstation in the office of the AFRL/RI POC. This code and performance results were described in [5].

A3. The codes that implement a sequential C version of a GA/DNA library design application using the thermodynamic pairwise nearest neighbor metric reside in directory "C:\annapolis\wsiipro_pci\thermo version" on a Gateway workstation in the office of the AFRL/RI POC. This code and performance results were described in [13].