Optimization of Computations Used in Information Theory Applied to Base Pair Analysis

Team Andrew Coates[1] and Alexey Ilchenko[2]
Faculty Advisors Dr. Matthias K. Gobbert[1] and Dr. Nagaraj K. Neerchal[1]
Clients Patrick O'Neill[3] and Dr. Ivan Erill[3]

[1]Department of Mathematics and Statistics, University of Maryland, Baltimore County
[2]Department of Mathematics, Case Western Reserve University
[3]Department of Biological Sciences, University of Maryland, Baltimore County

Technical Report HPCF–2011–13, `www.umbc.edu/hpcf` > Publications

Abstract

Information theory is used in biology as an approach to analyzing and discovering enzyme binding sites in DNA. By using entropy, a measure of uncertainty, to analyze samples, information can be gained to make predictions more certain, or less uncertain. Information is gained by measuring how uncertain we expect to be at a site and deduct the uncertainty from observing the site.

To get the desired accuracy needed for these predictions, one has to do costly computations. When there are hundreds of samples, values can be approximated sufficiently accurately. When there are very few samples, values can be calculated by exhaustive enumeration. In the intermediate range, however, it's not clear that the existing approximations are optimal and the original method of exhaustive enumeration is very costly with regards to time and memory.

The better these predictions are the better biologists can predict where enzymes will bind when performing expensive wet lab experiments. By predicting where the enzymes bind, they are predicting what function the protein will take.

This work develops an algorithm for the intermediate range of sample sizes that is much faster and uses less memory in the process than the original method. The original naive computation is of exponential time complexity with respect to the number of samples. The new algorithm uses number theory to reduce the time complexity to cubic and further simplify the computation by using assumptions from biology. This is a substantial speedup in all ranges of sample sizes, born out by comparing Python, MATLAB, and serial C code.

Computational experiments that parallelize the studies using the Intel Concurrent Collections (CnC) programming paradigm moreover show that tools from parallel computing can be useful to make larger parameter studies feasible in reasonable time.

1 Introduction

Biologists use information theory as a method of predicting where proteins will bind to DNA. They specifically use entropy and information content as expanded upon in Section 2. Expected entropy is calculated in order to determine information content as expanded upon in Section 3. For the applications under consideration, the biologists are interested in entropy calculations for sample sizes ranging from 1 to about 450. For small sample sizes, the exhaustive calculation of the estimated entropy using its mathematical definition is possible. But already for medium sample sizes, its calculation is very time consuming when done exhaustively and also runs into memory problems. For large sample sizes, a good approximation is available to aid in the calculation of the entropy. Therefore, the most important range for an improved algorithm to compute expected entropy is around 20 to 70 samples.

n	Python	MATLAB	serial C
1	< 0.0001	0.0007	0.0001
2	0.0003	0.0015	0.0002
4	0.0046	0.0053	0.0002
8	0.7603	0.0231	0.0008
13	863.8485	0.0576	0.0012
16	O.M	0.0963	0.0018
32	O.M	0.6016	0.0103
64	O.M	5.0046	0.0877
128	O.M	46.6464	0.5354
256	O.M	498.3383	3.6238

Table 1: List of run times using the exhaustive Python program, more efficient MATLAB program, and translated serial C program. Times are given in seconds. O.M represents Out of Memory.

The number of unique DNA sequences is tremendous and grows as $O(4^n)$ with sample size n. However, some of that DNA sequence information is repetitive with respect to binding site entropy calculations, thus, presents itself as problem of extracting only useful information out of $O(4^n)$ and then effectively optimizing the calculation.

Our team implemented a computational model for calculating entropic values for an enzyme binding using an exhaustive string based approach in Python. This would generate exhaustively all distinct sequences. This leads to memory problems and excessive run times even for relatively small sample sizes, and so a better algorithm is necessary, as expanded upon in Section 4.

In order to reduce the amount of calculations with respect to the multinomial function, our team noticed that we can group certain DNA sequences together into a class of integer compositions of length 4 where each summand corresponds to the number of base pairs in that particular class of DNA sequences. For example, for a sample of length 4, some sequences are CATT, ATTG, and TATC. Following our approach, the sequences CATT and TATC would be grouped into the same composition, namely, 1 A, 2 T, 1 C, and 0 G. As the entropy calculation is the same for any member in the same composition class, we effectively reduce the calculation to $O(n^3)$ with respect to the multinomial function. We wrote this algorithm in MATLAB as expanded upon in Section 5. Our team then ported the MATLAB algorithm to C, which shows further significant time improvement as expanded upon in Section 6. We also gave our C code to another team in the Summer 2011 REU Site: Interdisciplinary Program in High Performance Computing at UMBC, to use as a test code in their project. This is discussed in Section 7.

Table 1 lists the run time in seconds for our code in Python, MATLAB, and C for various sample sizes in the range from 1 to 256. The data bears out that the Python code is slower than the alternatives for sample sizes 8 and 13 and runs out of memory (indicated by the notation "O.M" in the table) already for medium sample sizes. For the most important range for our work from 20 to 70 samples, both the MATLAB and the C code take less than 1 second in all cases, thus addressing the problem of making those calculations feasible. Moreover, while the MATLAB code begins to take increasingly longer for large sample sizes, the C code continues to perform well.

All run times in Table 1 were gathered from runs on one node of the 86-node distributed-memory cluster tara in the UMBC High Performance Computing Facility (HPCF). The cluster tara consists of 82 compute nodes, 2 development nodes, 1 user node, and 1 management node. Each node has two quad-core Intel Nehalem X5550 processors (2.66 GHz, 8192 kB cache) and 24

Sample	Position			
	1	2	3	4
1	C	T	C	G
2	A	G	G	G
3	T	C	G	G
4	G	T	C	G
5	A	G	G	T
6	G	A	G	C

Table 2: This is a sequence motif for a hypothetical enzyme with $n = 6$ binding sites (across rows), each of length 4 base pairs. Each vertical column lists the observed base pairs at a specific position.

GB of memory. All components are connected by a state-of-the-art InfiniBand (QDR) interconnect. All nodes run the standard UMBC distribution of the Linux operating system, Redhat Enterprise Linux 5. For more information on tara and the UMBC High Performance Computing Facility, visit `www.umbc.edu/hpcf`.

Files for all programs run in this paper may be found along with the PDF version of this report at `www.umbc.edu/hpcf/` under Publications. For specific file names, refer to the end of the relevant section discussing the code of interest. Details on how to run each code can be found in the documentation at the beginning of the major file for each language (Python, MATLAB, and C, respectively).

2 The Biological Problem

To perform a certain biological function, enzymes can bind to a predesignated set of binding sites. For example, when an organism's cell replicates, the genetic information is carried by DNA where the process involves the unzipping of double helix structure and the complimenting of each strand by the family of enzymes known as DNA polymerase. Replication is initiated at a specific genomic sequence of DNA base pairs, Adenine, Thymine, Guanine, and Cytosine, known as replication origin. The enzyme DNA Polymerase can bind to approximately 400 binding sites in yeast (thus effectively making the DNA replication process faster) [4]. Of these numbers of binding sites, suppose n are unique; together with the length of the binding sites, m, specific binding sites of an enzyme are organized in a sequence motif. Table 2 shows a binding motif for a hypothetical enzyme with 6 binding sites, $n = 6$, each of length 4 base pairs, $m = 4$.

With greater number of samples of binding sites, it becomes easier to predict whether another sequence of DNA base pairs is a binding site for an enzyme at hand. However, wet lab experiments are expensive and, therefore, information theoretic methods are used to increase the robustness of the process.

3 Mathematical Formulation

The solution consists of deriving, optimizing, and parallelizing an algorithm to compute an entropy value (or, analogously, information content) associated with a given sequence of DNA base pairs. When an organism's cell replicates, the genetic information is carried by DNA where the process involves the unzipping of the double helix structure and the complimenting of each strand by the family of enzymes known as DNA polymerase. Replication is initiated at a specific genomic sequence of DNA base pairs, Adenine, Thymine, Guanine, and Cytosine, known as replication origin. These replication origin sites occur with some frequency in the genome; by calculating the entropy value of the sites, one can investigate how information is distributed across the site and compare one site to another. One way to categorize information is to use entropy (as suggested by Shannon [9]),

$$H(x) = -\sum_{x \in \Delta} p(x) \log_2 p(x), \tag{1}$$

where Δ is the set $\{A, T, C, G\}$ as further used by Scheiner to calculate entropic values for sequences of DNA in [8]. For the remainder of this paper, let log be understood to be $\log_2$. The probability $p(x)$ of $x \in \Delta$ can be taken from the genome or from observation of some string of length n. Entropy in information theory represents the amount of uncertainty. The number value of entropy can be understood to be the number of yes/no questions required to determine what value we have. For example, if we have $p(A) = p(T) = p(C) = p(G) = .25$, then we will compute an entropy of 2. This tells us that on average it will take us two questions to determine what value in Δ we have. For a more in depth introduction to entropy see [3].

How does this contribute to DNA replication? Certain proteins are found to bind at a similar set of sites on DNA. That is, they tend to attach to similar base pair sequences. By knowing the frequency that these base pairs appear in the genome, and then by observing how frequently they show up at particular binding sites we can determine learn which type of base pairs appear most often. By doing this we should be able to predict where certain proteins will bind.

3.1 Information Content

Schneider uses Information Content, which we will denote as IC, and he calls $R_{sequence}$ in his paper [8]. IC is computed to be literally mean the deduction of uncertainty. Equivalently stated, IC is the amount of information gained. Schneider uses the formula

$$IC = H_x - H_{x|y} \tag{2}$$

to define IC. H_x here is the entropy of this site based on genomic frequencies and $H_{x|y}$ is the entropy of this site given that the protein binds there based on the observed frequencies at the site. Schneider makes an adjustment to this equation because the computed $H_{x|y}$ is subject to undersampling bias. He adds an error term, ϵ_n, making (2) into

$$IC = H_x - (H_{x|y} + \epsilon_n) \tag{3}$$

with ϵ_n defined to be

$$\epsilon_n = H_x - E(H_n), \tag{4}$$

where $E(H_n)$ is the expected entropy of a string of length n. By substituting back into (3), we can simplify IC down to

$$IC = E(H_n) - H_{x|y}. \tag{5}$$

3.2 Expected Entropy

This is where the problem comes in. $E(H_n)$ is defined as

$$E(H_n) = \sum_{s \in \Delta^n} H(s)p(s) \tag{6}$$

where Δ^n denotes all sequences of length n made up by values in Δ. You can also think of Δ^n as the cross between Δ with itself n times. We should note that Δ^n contains 4^n elements. For small n, say less than 10, we can compute $E(H_n)$ easily because of the small scale of the problem. Additionally, for very large n we are able to compute $E(H_n)$ accurately using Basharin's approximation [2] because it becomes more accurate as n gets larger. What we can't solve easily is when n is of an intermediate value, or somewhere in the 20 to 70 range. It becomes very time costly to compute by exhaustive enumeration, but also the error in Basharin's approximation is too large to be useful. Our goal is to find a more efficient method to calculating $E(H_n)$ in this 20 to 70 range.

4 Exhaustive Python

We have created a program in Python which computes the expected entropy of a sample of size n with given genomic probabilities. The reason for using Python to do this is that it is easy to handle strings with it. This is beneficial since the program creates the exhaustive list of all possible combinations of a sample size n before computing the expected entropy, $E(H_n)$.

This program is very limited. It takes a long time to run for these small n. Table 1 shows the time used in some Python runs. The amount of memory used in the matrix of all permutations is of the order $O(n4^n)$ characters or about $8n4^n$ bytes. For $n = 13$, we estimate the required memory to be 6.5 GB, and for $n = 14$, we estimate 28 GB. Each node of the cluster tara has 24 GB of memory, so the memory needed to generate all the permutations is too large for $n > 13$. However, it is already apparent from the results for $n \leq 13$ that the run times blow up quickly. For instance, between $n = 8$ and $n = 13$, the run time increases from < 1 second to about 15 minutes. It is clear that even if there were no memory problems, this method would take far too long to run for n between 20 and 70.

The necessary code for this section is contained in the file `EHn.py` which may be found along with the PDF version of this report at `www.umbc.edu/hpcf/` under Publications. Details on the code can be found in the documentation at the beginning of the code.

5 MATLAB Code Optimized by Compositions

Our first step in optimization comes with an observation of the nature of the computation of $E(H_n)$. The same $H(s)$ and the same $p(s)$ is computed for sequences having the same number of A's, T's, C's and G's. Hence the product $H(s)p(s)$ is the same for these sequences. We can skip exhaustive generation if we can take advantage of the equality of this product in sequences with the same counts of each base pair. The following paragraph is an explanation of the theory used in generating these counts.

In number theory, the concept of integer partitions [5] can give us all these counts. This is to say, for a given integer n, the set given by the partitions of n is all the non-negative integers which add up to n. For example, if $n = 4$, the partitions of the set are $\{(4), (3+1), (2+2), (2+1+1), (1+1+1+1)\}$. The set of the compositions of n are all the distinct permutations of the set of partitions of n. We are interested in the compositions of length 4. In this way we are assigning a spot in each combination to the total number of base pairs found in a particular sequence. This means our $n = 4$ example would have to look like this:
$\{(4+0+0+0), (3+1+0+0), (2+2+0+0), (2+1+1), (1+1+1+1)\}$ and all of their permutations.

The total number of compositions of length 4 is akin to the number of ways of arranging $n+3$ objects into 4 positions, namely

$$\binom{n+3}{3} = \frac{(n+1)(n+2)(n+3)}{6}.$$

So for our $n = 4$ example, there will be 35 compositions total.

We used MATLAB to write our next program because it is built for mathematical calculations and more particularly matrix and vector operations. This is beneficial because our program generates all the length four compositions of n into a matrix sized $\binom{n+3}{3} \times 4$ in $O(n^3)$ time complexity. We define a row in the composition matrix to be of form (N_A, N_T, N_C, N_G). Where the count of each base pair is given by N_A, N_T, N_C, N_G.

The nature of the problem calculates the same product $H(s)p(s)$ for all sequences of DNA, which fall into the same integer composition class. When computing $p(s)$, we do the same calculation in each iteration with one exception, the exponents that each genomic probability is raised to. These exponents are the counts. That is, $p(s) = P(A)^{N_A}P(T)^{N_T}P(C)^{N_C}P(G)^{N_G}$. When computing $H(s)$, the only values that matter are the counts. The value $p(x)$, used in computing $H(s)$, is found by dividing the count of each base pair by n and then $H(x)$ is computed by iterating through the probabilities. So by generating the matrix of all possible counts, we can avoid generating all the strings which saves $O(n^4)$ time and replaces it with $O(n^3)$ time.

The problem at this point is that the compositions matrix doesn't account for the different permutations within each row of the matrix. For example, let's return to the $n = 4$ example. Let's evaluate a specific row in the matrix, $(2+2+0+0)$. This row says that there are two A's and two T's. This composition includes all the unique ways to permute two A's and two T's. In this case there are six: AATT, ATAT, ATTA, TTAA, TATA, and TAAT. This is simply given by the multinomial coefficient [7]. That is, with 4 factors,

$$\binom{n}{N_A, N_T, N_C, N_G} = \frac{n!}{N_A!\, N_T!\, N_C!\, N_G!}, \tag{7}$$

where $N_A + N_T + N_C + N_G = n$. We now have the tools to compute $E(H_n)$.

We compute $E(H_n)$ by iterating through the rows of the composition matrix. For each row, we compute $H(s)$ and $p(s)$. We then multiply these together with (7) to get that rows value and then sum it with the other rows. That is,

$$E(H_n) = \sum_{v \in A} \binom{n}{v_1, v_2, v_3, v_4} H(v)p(v), \tag{8}$$

where v_i represents the first through fourth item in the row for $i = 1, 2, 3, 4$, respectively, and A is the composition matrix. This method of calculating $E(H_n)$ reduces the time complexity from $O(4^n)$ to a multiple of $O(n^3)$ in factorial complexity.

It is necessary to mention that with the use of factorial functions comes the limitations of factorial functions. Our calculation breaks down with the breaking down of the factorials in (7) for n too large. We've found that this occurs for $n > 280$. That is, MATLAB will calculate $n!$ to be `inf` which makes $E(H_n) =$ `inf` as a result.

It is evident from comparing the Python column to the MATLAB column in Table 1 that this algorithm using optimization by compositions implemented in MATLAB is much faster than the exhaustive code. The exhaustive code takes about 15 minutes to run for $n = 13$ where as the algorithm code runs in about 8 minutes for $n = 256$. Unfortunately, as n gets larger from n to $2n$ the factor of increase grows.

The necessary code for this section is contained in the file `EHn.m` which may be found along with the PDF version of this report at `www.umbc.edu/hpcf/` under Publications. Details on the code can be found in the documentation at the beginning of the code.

6 Serial C Code

We translated our MATLAB compositions code into C code for the sake of speed up and for the reasons discussed in Section 7. Table 1 shows us that as n doubles, the time goes up by a factor of a little bit more than 8. This is expected since the time complexity of our algorithm is of order $O(n^3)$ with respect to the iterative multinomial function and $2n$ results in $8n^3$ operations, a slow down from n causing n^3 operations. The times are also much faster in C than in MATLAB despite using the same algorithm. This is just a result of C being a faster language than MATLAB. Our client is concerned with the range of n from 20 to 70 which will take less than .2 seconds. Other researchers in this field concerned with higher n will still be served well as it takes less than 4 seconds for $n = 256$.

The necessary code for this section is contained in the files `EHn.c`, `EHn.h`, and `Makefile` which may be found along with the PDF version of this report at `www.umbc.edu/hpcf/` under Publications. Details on the code and compiling can be found in the documentation at the beginning of the code. To see which compilation flags were used, see the Makefile.

7 Parallel C Code using CnC

This work was conducted as a part of the REU Site: Interdisciplinary Program in High Performance Computing at the University of Maryland, Baltimore County (UMBC) (`www.umbc.edu/hpcreu`). Another team in the program conducted a study on the performance and ease of use of the Intel Concurrent Collections (CnC) parallel programming paradigm [6] as compared to using Message Passing Interface (MPI). Their conclusion is that CnC is very suitable for parameter studies, especially because of the ease to set them up, once a serial function for the simulation of one set of parameters is available.

This situation is exactly applicable in our case, if it is necessary to compute entropies for *all* sample sizes 1 to n, instead of one (or a few) specific n. Then the acceptable run times for each sample size in Table 1 will add up very rapidly, and a serial run for all of them will take unacceptably long. This motivates the idea to distribute all serial runs onto several parallel processors. But since each run will take a different amount of time, it is difficult to distribute the jobs to the available processors a priori in a way that balances the load overall. The reseach by the other REU Site team on CnC demonstrates that this is an ideal case for using CnC for a parameter study, since that parallel paradigm will distribute the jobs at run time automatically, since we can identify them as independent of each other. To allow the other team to set this up, all we need to supply to them is a serial C (or C++) code for each sample size (which becomes the heart of the compute step in CnC terminology), and the list of desired sample sizes (tags in CnC terminology). See [1] for more details on the implementation. Being able to take advantage of CnC in this way was another reason to translate the MATLAB code to C.

Table 3: Wall clock time in seconds for calculating DNA entropy for all sample sizes from 1 to n using 1, 2, 4, and 8 threads.

num threads	$n = 32$	$n = 64$	$n = 128$	$n = 256$	$n = 512$
1	0.075	0.910	13.704	219.137	3531.563
2	0.044	0.506	7.530	118.186	1847.807
4	0.029	0.292	4.201	68.140	1094.142
8	0.026	0.228	3.252	48.577	771.015

Table 3 is the copy of a corresponding table in [1] and shows the wall clock times in seconds for studies with sample sizes from 1 to n. The serial runs using 1 thread in the first row demonstrate that the total for larger n can become significant. For instance, for $n = 256$, which takes 3.6 seconds for only that case in Table 1, doing all runs from 1 to n takes 219 seconds. Using 8 threads on a node with two quad-core processors reduces this to under 49 seconds again. For larger n, both effects become more pronounced.

This demonstrates that tools from parallel computing can be extremely useful to make simulations feasible within acceptable times. And it shows that CnC in particular holds great potential due to its ease of use.

Acknowledgments

This research will be conducted during Summer 2011 in the REU Site: Interdisciplinary Program in High Performance Computing (`www.umbc.edu/hpcreu`) in the UMBC Department of Mathematics and Statistics, funded by the National Science Foundation (grant no. DMS–0851749). This program is also supported by UMBC, the Department of Mathematics and Statistics, the Center for Interdisciplinary Research and Consulting (CIRC), and the UMBC High Performance Computing Facility (HPCF). The computational hardware in HPCF (`www.umbc.edu/hpcf`) is partially funded by the National Science Foundation through the MRI program (grant no. CNS–0821258) and the SCREMS program (grant no. DMS–0821311), with additional substantial support from UMBC.

References

[1] R. Adjogah, R. Mckissack, E. Sibeudu, A. M. Raim, M. K. Gobbert, and L. Craymer, *Intel Concurrent Collections as a method for parallel programming*, Tech. Rep. HPCF–2011–14, UMBC High Performance Computing Facility, University of Maryland, Baltimore County, 2011.

[2] G. P. Basharin, *On a statistical estimate for the entropy of a sequence of independent random variables*, Theory of Probability and its Applications, 4 (1959), pp. 333–336.

[3] I. Erill, *A gentle introduction to information content in transcription factor binding sites*, 2010. 30 pages, June 01, 2011, accessed August 14, 2011, compbio.umbc.edu > Notes.

[4] A. J. F. Griffiths, S. R. Wessler, and R. C. Lewontin, *Introduction to Genetic Analysis*, vol. 10 of Introduction to Genetic Analysis, W. H. Freeman and Co., 2008.

[5] S. Heubach and T. Mansour, *Combinatorics of Compositions and Words*, Discrete Mathematics and its Applications, CRC Press, 2009.

[6] K. Knobe, M. Blower, C.-P. Chen, L. Treggiari, S. Rose, and R. Newton, *Intel Concurrent Collections for C++ 0.6 for Windows and Linux.* 1997, accessed July 11, 2011, http://software.intel.com/en-us/articles/intel-concurrent-collections-for-cc.

[7] S. M. Ross, *A First Course in Probability*, Pearson Prentice Hall, 2010.

[8] T. D. Schneider, G. D. Stormo, and L. Gold, *Information content of binding sites on nucleotide sequences*, Journal of Molecular Biology, 188 (1998), pp. 415–431.

[9] C. E. Shannon and W. Weaver, *The Mathematical Theory of Communication*, Illini books, University of Illinois Press, 1964.