Algorithm Design and Optimization for Genome Sequencing*

Louis S. Revor†
Hardin-Simmons University
Abilene, Texas

supervisor :
Dr. Michael Minkoff
Computing and Telecommunications Division
Argonne National Laboratory

13 December 1991

*Work performed at Argonne National Laboratory, a contract Laboratory of the United States Department of Energy.

†Participant in the Fall 1991 Science and Engineering Research Semester program. This program is coordinated by the Division of Educational Programs.

1 Introduction

The basis of this project was the study and design of algorithms for the partial sequencing of the human genome on a variety of sequential and parallel architectures. This work was done under the supervision of Dr. Michael Minkoff of the Computing and Telecommunications Division (CTD). The research was done in support of a genome sequencing project headed by Dr. Radoje Drmanac and Dr. Radomir Crkvenjakov of the Biological and Medical Research Division (BIM). Larry Rudsinski (CTD) also aided in the optimization and design of the implemented code.

The algorithm consists of two main sections. The first section is for the generation of a sample, randomly generated, data table which will simulate the data that will be generated in the lab by the biologists. It will be used for the development of the second section of the algorithm. The second section is the actual reconstruction of the genome from the sample data.

1.1 Table Generation

The generation of the table must be done so as to emulate the experimental results. The table is created to represent all of the data that will be known when the attempt to reconstruct the genome is made. For now, the table will be created using data that would result if no errors occurred in the experiments. However, the experimental data will have errors; therefore, the table generation will eventually have to be modified to simulate false positive (true results that should be false) and false negative (results that are incorrectly false) results.

The first step in the construction of the table is to generate a random sequence that shall serve as the genome. This sequence consists of random combination of the letters 'A', 'T', 'C', and 'G'. These letters represent the four nucleotides that are found in DNA sequences. These nucleotides are known as : adenosine (A), thymidine (T), cytidine (C), and guanosine (G). This sequence must be constructed in a truly random way for all later statistical studies to be considered true. The length of the sequence shall be given in base pairs (bp), where base pairs is the number of the characters in the sequence. All sections of the sequence shall also have their lengths given in this manner. This sequence shall be referred to as the 'genome' or plainly as the 'sequence'.

The next step in the generation process is to break the sequence up into smaller sections called 'fragments'. These fragments are to have a ran-

dom amount of overlap with the previous fragment, however there will be set values for the maximum and minimum amount of overlap two adjacent fragments can have.

Third, a table of 'probes' should be generated. The probes are combinations of nucleotides of varying length. At present all combinations of length six are being considered. The resulting table of probes looks like :

AAAAAA
AAAAAT
AAAAAC
AAAAAG
AAAATA
⋮
GGGGGG

Once the probes and fragments have been generated, the table can be constructed. The table is to show if a given probe is in a given fragment. The process used to determine if a probe is contained in a fragment, or how this data is stored is of little consequence at this juncture, but it will play an increasingly greater role as the study continues.

1.2 Reconstruction

The reconstruction portion of the algorithm is to actually rebuild the sequence (genome) from the table that is given. In the test cases, it would be an attempt to reconstruct the generated sequence in the first step using the table that was created later in the same step. In the actual problem, complete reconstruction is not possible due to areas in the genome that are repetitious. The reason that repetition cannot be treated by the algorithm shall be explained later.

The general idea behind the reconstruction is that given a specific probe from a given fragment, one can look in the table to find a probe that is also in the fragment that overlaps the current probe with a maximum amount of overlap. This new probe shall be called the 'extension' probe. In the test case, all probes, except for the final probe in a fragment, will have at least one probe (with a maximum of four) that have this amount of overlap. In the real case, it is possible for a probe, other than the final probe in a fragment, to not have an extension due to a false negative result in the experiment.

If a probe has more than one extension, then a branching point has occurred. This is called a branching point because two distinct paths are possible. This branch can be resolved in more than one way. Two possibilities are to look in adjacent (overlapping) fragments, or to look in similar sequences. The best way to resolve this has not yet been determined, but the process that will be implemented will be a combination of different methods.

A problem arises when a section of a fragment repeats, or if a probe overlaps itself (such as AAAAAA). This problem arises because there is no way to determine how many times these sections should be repeated. The best way to handle this situation is to note it and move on, because the repetition will eventually end. Also, the biologists are not very interested in these sections of repetition.

Since the proper method for reconstruction has not yet been decided on, most of the research has been on generating data that will help this decision. Also, a lot of attention has been given to deciding which architectures are best suited for this problem.

2 Table Generation

In writing the code for the generation of the table of data, the two main computer science issues had to be taken into consideration. These two issues are speed of the algorithm and storage efficiency. As a result, six versions of the algorithm were written and implemented. Each was written to either improve the speed or storage efficiency. A description of each of the different versions and their advantages is given below.

Table 1 shows the timing for each of the six routines. Timings were done at differing fragment lengths, but the number of fragments was held constant at 100. The probe length that was used was six. Since the number of probes is given by the equation 4^p where p is the length of the probes, there were a total of 4096 probes. All of the timings were done on a Sun4/490. The data is given in seconds.

2.1 Version 1

Since this version is the first version that was implemented, it is very straightforward in its logic and data structures. The sequence, when generated, is stored as a character string, where each element is one of the letters 'A', 'T', 'C', or 'G'. After the sequence is generated, the position in the sequence where each of the fragments are to start is calculated. The fragments

Table 1: Timing (seconds) for Data Table Generation

Fragment Length	Version 1	2	3	4	5	6
50bp	23.5500	4.7833	5.2833	16.7667	.2667	.3500
100bp	46.5667	9.2167	9.8167	37.3000	.4000	.3500
200bp	71.6667	18.7000	19.3000	68.3333	.4833	.4833
300bp	104.5000	28.4000	33.5333	104.2667	.6333	.6500
400bp	135.9667	37.8833	39.2500	131.8167	.8000	.7833
500bp	168.8167	46.9833	47.4667	185.8667	.9500	.8833
1000bp	324.4833	93.4000	95.6167	360.3667	1.6000	1.5333

are then copied out of the sequence and stored into an array of character strings where each element of the array is a distinct fragment. The probes are generated as a character string and stored to a file.

The table, in this version, is a two-dimensional array of logical data type. This is the obvious choice, because all that needs to be stored is a true or false value. The table is generated by reading a probe from the file, then using a string comparison to see if the probe is contained in each of the fragments. If a fragment contains the probe, then a true value is placed in the table at the appropriate position. If the probe is not present in the fragment, then a false value is entered. This process is repeated for all of the probes.

The basic memory requirement (in bytes) for data of this version is given by :

$$s + m * n + m * 4^p$$

where :

$$\begin{aligned} s &= \text{sequence length} \\ m &= \text{number of fragments} \\ n &= \text{fragment length} \\ p &= \text{probe length} \end{aligned}$$

2.2 Version 2

In Version 2, the table is created in a different manner. In this version, each of the fragments are scanned to see which probes are contained in it. Each

probe-length section of the fragment is copied and converted to a number that represents the probe. Using this number, the position in the table that corresponds to the fragment and the probe is marked true. Since the true values will be marked in this fashion, the table must be set to all false values before the routine begins.

The method for converting a probe to a number involves considering the probe to be a base-4 number, because each position in the probe has four distinct values. Letting the letters 'A', 'T', 'C', and 'G' represent the numbers 0, 1, 2, and 3 respectively, a conversion can be easily obtained. Therefore, the probes are now represented by the numbers $0 \ldots 4^p - 1$. Next, the base-4 number must be converted to a base-10 number for use by the program.

Since pattern matching is no longer used, the probes do not need to be generated. As a result of this, disk space is no longer needed to store the file of probes. However, there is no saving in the amount of memory needed by the program for the data.

2.3 Version 3

In this version the fragments are no longer stored separately from the sequence. A vector is now used to point to the first element in the sequence that is contained in the fragment. Since all of the fragments are of the same length (in the test case), each of the fragments can be exactly determined by looking at the fragment length section of the sequence starting at the character pointed to by the vector.

Since the probes are no longer stored separately, there is a decrease in the amount of storage required, although allocation for the vector is now necessary. The new storage requirement is :

$$s + m * w + m * 4^p$$

where :

$$w \quad = \quad \text{word length}$$

Therefore, there is a savings in the storage requirement when $w < n$.

2.4 Version 4

This version of the algorithm was written primarily for comparison. It is the same as Version 1 with two exceptions. The probes are stored in memory

and not in a disk file. The other difference is that the fragments are not stored separately, but are referenced as in Version 3 above. The table is constructed as it was in Version 1.

With this version I wanted to show the difference in time for the pattern matching routine and the conversion routine. As I predicted, the conversion routine was less time consuming.

The storage requirement for this version is :

$$s + p * 4^p + m * w + m * 4^p$$

2.5 Version 5

Version 5 was written due to the need to reduce storage requirements. It is an improvement to Version 3 above. This version stores the table using one bit for each element. Two routines were written in C to allow for the setting and reading of bits in the table. Because of the time needed to set up a subroutine call, it was decided to set the bits for a fragment at a time. However, setting the bits in this fashion causes the need for another vector to store the information on what bits in the fragment are to be set.

The storage requirement for the table is now decreased by a factor of 8. However, for smaller sized tables, the savings is nullified by the need for the vector. The requirement for this version is :

$$s + m * w + 4^p * w + m * \frac{4^p}{8}$$

Which is less than Version 3 when $w < \frac{7m}{8}$.

As it is noted in the table above, this version of the routine provided the fastest time out of all the six versions. The time is a noticeable decrease when compared to the previous four versions.

2.6 Version 6

This version is a continuation of the improvements made in Version 5. It was written to make the code portable to a wider variety of machines. In the previous version, the sequence prevents compilation on some machines, because these machines do not allow character strings longer than a specific preset length. The sequence in this version is stored in bits. Two bits are allocated for every character that was needed in the previous versions. In the sequence, the bit patterns 00, 01, 10, and 11 represent the letters 'A', 'T', 'C', and 'G' respectively.

Since the sequence is in a different format, the scanning of the fragments is changed to accommodate the differences. Instead of copying p characters out of the fragments when they are scanned, $2*p$ bits will be copied out into an integer variable. When it is copied out of the fragment, the conversion in the previous version is no longer needed. The reason for this is because the conversion is automatically done by the hardware when the value is referenced.

The storage requirement for this version is :

$$\frac{s}{4} + m * w + 4^p * w + m * \frac{4^p}{8}$$

3 Statistical Considerations

The first step in designing an algorithm for the reconstruction of the genome is to find the scale of the problem. The method under consideration for the reconstruction is to extend from probes to a certain length (20-30bp) and then find a method to determine how to combine these extensions to reconstruct the fragments and the sequence. A major concern in this approach is the number of possibilities that might occur when extending from a probe to a specific length. It is obvious that as the extension length increases, the possibilities increase. Therefore, there is a need to balance the length of the extensions and the number of extensions that result.

A code was written that would accept a table of data as input and return the number of possible extensions that occur for various extension lengths. This Code extends on every probe that occurs in each of the fragments. It keeps a total for the number of possible extensions that occur in the fragment for each of the extension lengths. It also notes the number of probes that are in the fragments. Knowing the number of probes that are in the fragment allows for the calculation of the average number of extensions for a probe in the fragment by dividing the totals for the fragment by the number of probes that are present in the fragment.

In this code, the probes are referred to by their position in the data table. The following table shows the correspondence between the probes and the numbers that refer to them :

Probe	Reference #	Bit Representation
AAAAAA	0	000000000000
AAAAAT	1	000000000001
AAAAAC	2	000000000010
AAAAAG	3	000000000011
AAAATA	4	000000000100
⋮	⋮	⋮
GGGGGA	4092	111111111100
GGGGGT	4093	111111111101
GGGGGC	4094	111111111110
GGGGGG	4095	111111111111

Each of these numbers is represented in memory in the same fashion as the bit representations used in version 6 of the data table construction code. Representing the probes this way allows for probes of all lengths to be referenced without any changes to the code.

It is interesting to note that the four possible extensions to a probe are next to each other in the table. Knowing this, if the first extension is calculated, the other three are exactly determined. Considering the bit representations for these numbers, the extension can be obtained by zeroing the left two bits in the number and then left shift the number two bits with a zero fill. For example, 'CGGGGG' has 'GGGGGA', 'GGGGGT', 'GGGGGC' and 'GGGGGG' as its extensions. It can be seen in the table that all four of these can be determined by knowing that 'GGGGGA' is the first of these, and that the other three follow in the table.

The bit representation for 'CGGGGG' is '101111111111'. In order to discover that 'GGGGGA' is the first of the extension probes, the left two bits from the bit representation of 'CGGGGG' must be removed to obtain '1111111111'. Next, the new bit pattern must be shifted left two bits with a zero fill. The result of this operation is the pattern '111111111100'. This new pattern is the pattern for the probe 'GGGGGA', which is what was needed to be obtained. The extension probes for any probe can be determined in this fashion. It is also not dependent on the probe length.

Table 2 is a sample of output from the code. The data was generated from a table that was created using fragment lengths of 500bp and probe length of 6bp.

The routines that generate the data table should be modified so the fragments do not have overlap regions when generating the data for statistical analysis. The reason for this is that regions of overlap will have a greater

Table 2: Example of Output for Number of Possible Extensions

Extension Length (bp)					Probe
10	15	20	25	30	Total
1462	6198	26872	115082	494390	470
1276	4959	18881	71129	267959	462
1474	6439	26657	110536	462400	466
1483	6626	29202	129432	573950	462
1443	6212	27632	122941	545237	481
1359	5252	19926	76044	288892	466
1335	5313	20962	79736	302002	467
1488	6177	26076	108992	453810	466

influence on the results of the analysis. The results will not be statistically true if there are regions of overlap. When doing an analysis to find the average number of extensions from a given probe, the same number of probes should be used from each fragment. If one fragment contributes data from more probes than the others, it will have a greater influence on the outcome and the results will not be true again.

Table 3 was generated from 100 fragments, each contributing data from 3 probes that were chosen randomly from the possible probes in the fragment. The data in the table is the average number of extensions that each probe had when extending to the given lengths. Extension lengths include the length of the initial probe which was 6bp.

Table 3: Average Possible Extensions

Extension	Fragment Length(bp)			
Length	250	500	750	1000
10bp	1.80	2.95	4.45	6.55
15bp	4.11	12.37	31.08	71.93
20bp	9.45	52.35	220.43	794.91
25bp	22.55	226.46	1579.68	8867.17
30bp	56.41	999.89	11490.20	99704.09

In trying to determine an overlap between two fragments using only the

data table, all probes that are in both of the fragments must be considered. However, if a probe is in both of the fragments, but outside the region of overlap, there is no way of determining that it is not suppose to be in the overlap region. It is interesting to compare the possible extensions from a region of overlap to the possible extension from non-overlapping fragments of the same length as the overlap region. It is obvious that the number of possible extensions from an overlap region should be slightly higher because probes could be considered that are in both fragments, but not in the overlap. This increase is of particular interest.

Table 4 shows the difference between the number of possible extensions from an overlap region and a non-overlapping fragment of the same length.

Table 4: Difference Between the Number of Possible Extensions from Overlapping Regions and Non-overlapping Fragments

Extension Length	Region Length(bp) 500	750	1000
15bp	1.53	7.57	27.69
20bp	9.39	87.63	537.17
25bp	54.79	903.31	9081.68
30bp	312.03	8761.61	144023.94

4 Optimization and Analysis

During the creation of the code for counting expansion possibilities, an effort was made to optimize the code for use on various machines and architectures. It was decided to analyze the performance on the various machines because the reconstruction code will behave in a fashion similar to that of the expansion code. Because of the size of the genome problem, it will be necessary to know which architectures are best suited for this type of problem. The computers that were used in this analysis are : Sun4c, Sun4/490, HP 720, Intel iPSC/860, CRAY X-MP/14 and the Intel Touchstone Delta. A version of the code was also written implementing P4 [1] to simulate an

[1] P4 is based on "Argonne Macros" which is documented in the book "Portable Programs for Parallel Processors", by Boyle, Butler, Disz, Glickfeld, Lusk, Overbeek, Patterson, and Stevens. Major contributions to P4 were also made by Bob Olson, University

MIMD distributed memory system across a network of workstations.

The first implementations of the code were run on workstations because of the ease in code development and accessibility. The timings involved the expansion out to 30bp from every probe in twenty fragments. Each of the computers expanded on the same table, so the amount of work being done on each of the machines is identical. Table 5 shows the results of the timings on the different machines.

Table 5: Expansion Timing (seconds) for Uniprocessor Workstations

Fragment	Machine		
Length	Sun4/490	Sun4c	HP 720
250bp	3.2500	6.2500	2.8333
500bp	67.8000	103.0333	57.8167
750bp	767.7834	1126.7167	658.1000

The next implementation was on a single processor CRAY X-MP/14. Transporting the code and compilation required little to no effort from the workstation versions. A second version was written in Cray assembly language (CAL) to utilize the Cray's vector registers to minimize the accesses to main storage. Only the routine that actually counts the number of expansion possibilities was rewritten in CAL. The rest of the code was left as is. The timings for the CRAY X-MP is given in Table 6

Table 6: Expansion Timing (seconds) for the CRAY X-MP/14

Fragment	CRAY X-MP	
Length	Normal	CAL
250bp	3.0379	2.1467
500bp	69.0349	47.4215
750bp	778.8751	533.9692

The timings for the Cray using the original routines show that the Cray's extra architectural enhancements do not apply to this type of problem. It is

of Illinois, Remy Evard, University of Oregon, and David Leibfritz, Argonne National Laboratory.

interesting to note that the Sun4/490 and HP 720 both had better timings than the Cray until the CAL routine was written. Due to the expense of using the Cray, it is not a very cost-effective solution to the problem.

Since each of the fragments are totally independent from each other, the expansions for each of the fragments can be computed simultaneously. Because of this, the code was ported to an Intel iPSC/860(Gamma). Since the Gamma is an MIMD distributed memory system, the code had to be split into two sections. The main (driver) section of the code handled the generation of the table data and the control of the message passing to the independent processors. The node section of the code was to count the extension possibilities for a given fragment. Because the system is distributed memory, each of the processors only knew the data for the particular fragment it was working on. This data is passed to the processor by the driver section when the node becomes available.

The revision to the code required a lot of changes to the structure of the code, but the portions of the code that did the actual computations remained the same. The code was set up for the driver section to run on the front-end (Intel 310) of the Gamma, while the node section was put on each of the processors. Table 7 shows the timings for the Gamma using 1, 2, 4, and all 8 of its hypercube processors. Note that the timings for the Gamma are the maximum time needed by one of its processors. Also, since each of the Gamma processors is dedicated, it keeps track of real time from when the node process is started. Unlike the other machines, time spent waiting to receive messages is included.

Table 7: Expansion Timing (seconds) for the Intel iPSC/860

Fragment Length	Nodes			
	1	2	4	8
250bp	5.862	3.210	2.513	2.330
500bp	86.278	44.329	22.841	13.713
750bp	958.991	493.225	250.007	146.288

This table shows that the Gamma was not as efficient as some of the workstations when using only one node. When the number of nodes is increased by one, the comparison between the Gamma and the workstations becomes more interesting. As the length of the fragments increase, the Gamma increasingly outperforms the workstations; however, at the smallest

fragment length, the Gamma did not do as well.

In preparation for running the code on the Intel Touchstone Delta(which is possibly the world's fastest supercomputer), the code that was used on the Gamma was modified so the driver section of the code would run on one of the Gamma's processors as well. This version takes full advantage of the Gamma's hypercube message passing system. The decrease in time required for message passing is shown by the timings. The timings for this version is given in Table 8. Because the driver section requires a node in this version, the number of processors that can be used by the node section is decreased by one.

Table 8: Expansion Timing (seconds) for the Intel iPSC/860 With Driver Executing on a Node

Fragment	Nodes		
Length	1	3	7
250bp	3.815	1.471	1.011
500bp	84.015	29.397	15.676
750bp	950.403	325.311	156.969

This new code was then ported to the Intel Touchstone Delta (Delta). Due to the number of processors on the Delta (512), the size of the problem used in all of the prior timings is trivial. Because of this the number of fragments was increase to 512 (from 20) so the Delta's timings would be of interest. Table 9 shows the timings that were done on the Delta.

Table 9: Expansion Timing (seconds) for the Intel Touchstone Delta (512 Fragments)

Fragment	Mesh Size		
Length	(4,8)	(8,8)	(8,16)
500bp	71.016	39.424	25.692
750bp[2]	887.070	489.265	314.623

[2]The timings for the 750bp fragment lengths were done on the Delta after it had been booted with the slow debug kernal. I am not sure how this affects the timings, but they

The Delta uses a Mesh message passing system unlike the Gamma's Hypercube. The Mesh size refers to the size of the grid that is being used by the program. To find the actual number of nodes being used, all one has to do is multiply the two coordinates.

After seeing the these results, we thought that it would be interesting to see how a network of Sun workstations would compare. Therefore, the original Gamma code was converted to P4 so that it could be run parallel on a network of workstations. The conversion from Intel iPSC/860 message passing calls to P4 message passing calls required very little work. However, when using P4, all node processes must be self terminating. This requirement did not exist on the Gamma. This required a little change in the structure of the code that was used on the nodes.

When running the P4 version, a Sun4/490 was used to run the driver section of the code, while the nodes processes were all run on a Sun4c. Table 10 shows the results of these timings. The time shown is the minimum node time. Unlike the Gamma this time is user time and not real time. This means that time waiting for messages is not included in the time.

Table 10: Expansion Timing (seconds) Using P4 and a Network of Workstations

Fragment Length	Slaves			
	1	2	4	8
250bp	4.2500	2.2000	1.2833	1.0833
500bp	99.5000	51.0000	26.4500	15.4667
750bp	1134.7167	583.6000	304.0667	177.4333

After seeing the results of these timings, it is clear that this type of problem is best solved in parallel. Out of all the versions implemented, I believe the most cost effective version is the P4 version running on a network of workstation. It is not as fast as the Gamma, but it is not far behind. However, the Gamma version can be easily scaled up to run on more processors by porting it to the Intel Touchstone Delta, while the P4 version requires the addition of more workstations.

will still provide an interesting benchmark.

5 Reconstruction

Due to various delays, the actual algorithm for the reconstruction of the simulated genome has not yet been developed. However, the research presented in this paper has lead to a better understanding of the problem. Once the algorithm is completed and tested, the code to generate the genome (and table data), as well as the code for the reconstruction, must be altered to allow for experimental errors. Out of all the tasks mentioned thus far, I predict that this last step mentioned shall be the most difficult.