Pruning algorithm to reduce the search space of the Smith-Waterman algorithm & Kernel extensions to the μC/OS-II Real-Time Operating System

Farhan Ahmed
Department of Electrical and Computer Engineering
Lafayette College, Easton, PA

Honors Thesis

Advisors
Prof. John Nestor
Prof. Jeffrey Gum

May 12, 2005

Acknowledgements

Thesis submitted to the Faculty of Lafayette College
in fulfillment of the requirements for an
Honors Degree
in
Electrical and Computer Engineering

I would like to thank Prof. John Nestor, Prof. Jeffrey Gum, and Prof. Chun-Wai Liew for their guidance and support throughout the year. This thesis could not have been completed without their help.

I would also like to thank my third reader, Prof. Jeffrey Pfaffmann, for taking the time to go over this document and for providing useful insights and advice.

Lastly, I would like to acknowledge the constant support and help from my friends and family, without which I would not be where I am today.

Abstract

This paper is divided into two main parts. Each part summarizes the work done towards obtaining an Honors from the Electrical and Computer Engineering department at Lafayette College.

The first part of this paper presents results of a pruning algorithm used to reduce the search space of the Smith-Waterman algorithm for local sequence alignment. The algorithm *prunes*[1] parts of a given pair of genome sequence (DNA or Protein) such that the results of the Smith-Waterman algorithm on the original sequence match the results of the algorithm over the pruned data set. This reduces the computation time as well the the amount of memory used to process the given sequences without compromising the quality or correctness of results.

The second part of this paper presents kernel extensions made to the μC/OS-II Real-Time Operating System [11] to support features required for a design project undertaken by seniors in the Electrical and Computer Engineering department. Specifically, the features implemented were:

- Support for Coordinated Universal Time
- Cyclical Timer, and,
- Serial Communications Interface

To allow for these features, support was added to the kernel of the μC/OS-II Real-Time Operating System and easy-to-use Application Program Interfaces (APIs) were provided for the programmers to utilize these features. This paper describes the requirements, approach taken to implement the features, the actual implementation with the relevant code blocks, and the APIs provided to the application programmers.

[1]pruning refers to the process of cutting down the original sequence in size without losing any inherent information required for local sequence alignment

Contents

1 Introduction **5**

2 Background **6**

2.1 Genome Data 6

2.1.1 FASTA Format 6

2.2 Sequence Alignment 7

2.3 Scoring Matrix 8

2.4 Global Sequence Alignment 9

2.5 Local Sequence Alignment 10

3 The Smith-Waterman Algorithm **11**

4 Pruning Algorithm **12**

4.1 Past research 12

4.2 The algorithm 13

4.3 Pruning Algorithm - Pseudo-code 13

4.4 Sensitivity Analysis 13

5 Approach **15**

5.1 Parameters 15

5.1.1 Threshold Value 15

5.1.2 Subsequence Length 16

5.2 Automation 16

6 Results **18**

6.1 Constant subsequence length 18

6.2 Constant threshold value 18

7 Conclusion **23**

8 Introduction **25**

8.1 Background 25

8.2 Need for an Operating System 25

8.3 What is a real time operating system? 27

8.4 μC/OS-II Real-Time Operating System 28

9 Kernel Extensions **29**

10 Coordinated Universal Time (UTC) 30
10.1 Representation of UTC 30
10.2 Implementation 30
10.3 User Functions 31

11 Cyclical Timer 32
11.1 Support for Cyclical Tasks 34
11.2 Implementation 34
11.2.1 Task Control Block 34
11.2.2 Changes to OSTimeTick() 34
11.3 Features 35
11.4 User Functions 37

12 Serial Communications Interface 39
12.1 Interrupt System 39
12.2 Intermediate Buffer System 40
12.3 User Functions 40

13 Conclusion 41

List of Figures

2.1 Example of DNA Sequence 7
2.2 Section of a FASTA file 7
2.3 Example of Sequence Alignment 9
2.4 Globally Aligned Sequences 10

5.1 Data analysis approach 17

6.1 Constant subsequence lengths (5 - 75) 19
6.2 Constant subsequence length (100) 20
6.3 Constant threshold values (-50 to -2000) 21
6.4 Constant threshold values (-4000 to -8000) 22

8.1 Task in Infinite Loop Method 26
8.2 3 Tasks in Infinite Loop Method 27
8.3 Comparing infinite loop method and a RTOS 27

11.1 Typical Acyclical Task 32
11.2 Variable Time Delay 33
11.3 Implementation of a cyclical user task 33
11.4 OSTimeTick() Flowchart 36
11.5 Cyclical Task Failure 38

Part 1

Pruning algorithm to reduce the search space of the Smith-Waterman algorithm for local sequence alignment

Chapter 1

Introduction

Computational genomics has been gaining popularity in the last decade and more and more advances are being made every day in this field. The social implications of progress in this area are enormous as the promise of finding cures to deadly diseases, prolonging life and understanding the origin and end of life are becoming more and more probable. With increases in computing power and the decrease in the cost of memory, analyzing the enormous datasets generated by genome sequencing is becoming possible in ways never thought possible. Scientists today, can study the interactions of proteins and DNA down to the level of interaction of individual constituent bases. Even though such advances are encouraging and much needed, we are still years away from the immense amount of computing power that would be required to analyze these datasets completely.

This part of the paper attempts to present the results of an algorithm that reduces the size of such datasets by *pruning* away parts of the data that do not influence the results of the Smith-Waterman [18] algorithm for local sequence alignment. This technique saves precious computing power, but more importantly, allows for the analysis of large datasets that otherwise would not fit in the maximum memory supported by modern 32-bit computers.

The Smith-Waterman algorithm employs a dynamic programming approach which is both CPU and memory intensive. The space complexity of the algorithm is in the order of $O(n^2)$. Thus, to analyze two sequences, each 10,000 characters long, using the Smith-Waterman algorithm and using 16-bit floats to represent the similarity scores, the algorithm would take over 190 MB of memory. This value increases fast as the sequence lengths increase. Moreover, a great deal of overhead is involved in the real implementation of the algorithm. Lastly, most DNA and Protein sequences that are of interest are more than 100,000 characters long and to anaylze them using a strict and original implementation of the Smith-Waterman algorithm is beyond doubt, out of bounds for the maximum memory capacity of the modern 32-bit computer systems.

Chapter 2

Background

This section will provide the background information required to understand the context, functioning and implications of the algorithm presented in this paper. The key points and terminology used in this paper can also be found in Appendix A. The key concepts presented here are:

1. Genome Data
2. Sequence Alignment
3. Scoring Matrix
4. Global Sequence Alignment
5. Local Sequence Alignment

2.1 Genome Data

The genome data used to test the algorithm presented in this paper was obtained from the University of California - Santa Cruz Genome Browser homepage [1]. The available data was encoded in the FASTA format [10]. Genome data is represented as strings of characters (Figure 2.1) each of which represents a unique chemical. These chemicals, also called bases, are the building blocks of protein and DNA.

Specifically, DNA is composed of four key chemicals: adenine, thymine, guanine, and cytosine [13]. Referred to as A, T, G and C respectively, they are attached to a molecule of deoxyribose sugar plus another molecule of phosphate to form a unit called a nucleotide. These nucleotides are the backbone of the DNA. They connect up into dual chains that wrap around each other to form a double helix. These bases bind to each other - adenine to thymine and cytosine to guanine. These pairs of bases are called base-pairs. In almost all human cells, most DNA is found as two versions of 23 continguos segments, called chromosomes. An example of a genome sequence is shown in figure 2.1.

2.1.1 FASTA Format

The FASTA format is a commonly used format to represent genome sequences. The FASTA format descriptions are well defined and are supported by a large number of genome processing software available in the industry. The description of the FASTA format given below has been taken from the *Nationalen Genomforschungsnetz*, Germany [8].

```
C A T T A A G A T C G G T G C C C T T T
```

Figure 2.1: Example of DNA Sequence

A sequence in FASTA format begins with a single-line description, followed by lines of sequence data. The following items describe the structure of a FASTA format file [10].

- The description line starts with a greater than symbol (">")
- The word following the greater than symbol (">") immediately is the "ID" (name) of the sequence, the rest of the line is the description
- The "ID" and the description are optional
- All lines of text should be shorter than 80 characters
- The sequences ends if there is another greater than symbol (">") at the beginning of a line and another sequence begins

An example of a section of a FASTA file is shown in figure 2.2. Even though the DNA sequences are made up of only four key chemicals, the genome sequences in the FASTA format use other characters to represent commonly occuring groups of these 4 chemicals. This reduces the size of the file while at the same time does not lose any inherent information present in the sequence. Table 2.1 contains the characters used in a FASTA format file and their corresponding bases or sequence of bases.

```
>Example1 envelope protein
ELRLRYCAPAGFALLKCNDADYDGFKTNCSNVSVVHCTNLMNTTVTTGLLLNGSYSENRT
QIWQKHRTSNDSALILLNKHYNLTVTCKRPGNKTVLPVTIMAGLVFHSQKYNLRLRQAWC
HFPSNWKGAWKEVKEEIVNLPKERYRGTNDPKRIFFQRQWGDPETANLWFNCHGEFFYCK
MDWFLNYLNNLTVDADHNECKNTSGTKSGNKRAPGPCVQRTYVACHIRSVIIWLETISKK
TYAPPREGHLECTSTVTGMTVELNYIPKNRTNVTLSPQIESIWAAELDRYKLVEITPIGF
APTEVRRYTGGHERQKRVPFVXXXXXXXXXXXXXXXXXXXXXXXVQSQHLLAGILQQQKNL
LAAVEAQQQMLKLTIWGVK
>Example2 synthetic peptide
HITREPLKHIPKERYRGTNDTLSPQIESIWAAELDRYKLVKTNCSNVS
```

Figure 2.2: Section of a FASTA file

2.2 Sequence Alignment

An alignment is a way of stacking one sequence over the other and comparing characters from the two sequences that lie in the same position. If a match between the two characters is given a positive weight and a mismatch a negative weight, then the two sequences could be aligned to give the maximum sum of these weights. This maximum sum producing alignment, which may or may not be unique, is considered the best possible alignment. An example of two sequences aligned differently is shown in figure 2.3. Here, a match is given a weight of +2 whereas a mismatch, either between a character and a gap, or two different characters, is given a weight of -1. It is apparent from this example that the score varies dramatically as the alignment varies.

Code	Chemical	Remarks
A	adenosine	-
C	cytidine	-
G	guanine	-
T	thymidine	-
U	uridine	-
R	G A	puridine
Y	T C	pyrimidine
K	G T	keto
M	A C	amino
S	G C	strong
W	A T	weak
B	G T C	-
D	G A T	-
H	A C T	-
V	G C A	-
N	A G C T	any
-	gap of indeterminate length	-

Table 2.1: Nucleic Acid Codes

Gaps in sequence alignments are analogous to insertion and removal of genes in nature. These mutations are commonplace in nature and hence are justifiable in the context of sequence alignment algorithms. Usually, there is a cost associated with the insertion or extension of a gap in the sequence and this cost is known as the *affine gap penalty*. This penalty varies with the position of the gap in the sequence. Usually, starting and trailing gaps have low or no penalties whereas the insertion of a gap carries a higher penalty than the extension of one.

There are various applications for sequence alignment problems in computational biology. The biological sequence alignments for sequences representing DNA or protein present an insight into the natural mutations occuring in the strings [6]. Also, similarities between two sequences might suggest evolution from the same genetic tree or mutations over time that occurred in one of the sequences in the given pair of sequences.

Suppose a gene is discovered in a species and it is well understood and its functions are well defined. This gene could be aligned against all sequences in the human genome database and a close match could be found [17]. A close match, called a homology, would perhaps then suggest a similar function of the gene in humans as well. Once this gene has been compared to the sequences of many different species, the evolutionary development of the gene could be charted as it diverged in form from its original ancestor. This analysis holds great value in understanding the origin of certain genes and the functions they perform in the human body or other species.

2.3 Scoring Matrix

In the previous example of sequence alignment, a mismatch between characters was given a score a -1 whereas a match was given a score of +2. While these values are sufficient

```
C A A T G C A G     C - - A A T G C A G
C C C A A T G G     C C C A A T G G - -
Score = -4          Score = 5
```

Figure 2.3: Example of Sequence Alignment

-	a	g	c	t
a	136	-37	-160	-160
g	-37	136	-160	-160
c	-160	-160	136	-37
t	-160	-160	-37	136

Table 2.2: Scoring Matrix

to illustrate the technique of sequence alignment, in real biological applications, a scoring matrix is used to find the score between matches instead. A scoring matrix is a nxn[1] matrix in which each cell gives a score for the corresponding pair of bases. An example of such a matrix is shown in table 2.2. In this table, a match between 'a' and 'a' is given a score of 136, 'a' and 'g' is given a score of -37, 'a' and 'c' is given a score of -160, 'a' and 't' is given a score of -160 and so on and so forth. The scores are determined by taking into account the probability of the two chemical bases occurring together in nature as well as certain probabilistic techniques which are beyond the scope of this paper [2].

While this paper uses the scoring matrix described in table 2.2, it should be noted that the choice of a scoring matrix depends on the context of the sequences. Many such matrices are in use today; two major methodologies used to determine the matrices are PAM or BLOSUM, each of which use a different score for each pair of bases [4]. Also, the penalty of introducing a gap varies from context to context, and also depends on its relative position. For example, there is no penalty to introduce a starting or a trailing gap, while opening a gap carries a higher penalty than extending an already present one.

2.4 Global Sequence Alignment

Global sequence alignment refers to the process of finding the best possible alignment between two given sequences by considering the sequences in their entirity. Gaps can be introduced in between any of the sequences if that produces a better alignment. Gaps are analogous to insertion and deletion of base pairs and are commonly occuring natural phenomenon. The similarity score is the score of the best alignment by adding up all the scores of the matches and the mismatches. Gaps are usually given a negative score along with mismatches, while matches produce a positive score. An example of a globally aligned sequence is shown in figure 2.4.

[1] In case of DNA, n = 4, the number of unique bases that is is composed of

[2] For more information on scoring matrices, please refer to [4]

Sequence 1: AAATGGCAATTTGCCAAATGCTTG

Sequence 2: TTGCAAATGGCATTGTTTAAAAGC

Alignment:
AAATGGCA------ATTTGCCAAATGCTTG
AAATGGCATTGTTTA------AAAGC----

Figure 2.4: Globally Aligned Sequences

2.5 Local Sequence Alignment

Local sequence alignment is the method of finding the best possible alignment between any two *subsequences* of the given sequences. This method uses a similar approach as the global alignment but here subsequences from the two sequences that produce the best alignment are presented as results. Once again, the similarity score is reported by adding up the scores of individual matches, mismatches and gaps. The Smith-Waterman algorithm [18] presented later is a well-known local sequence alignment algorithm that finds the optimal local sequences alignment when given an input of two genome sequences.

Chapter 3

The Smith-Waterman Algorithm

The Smith-Waterman algorithm is a database search algorithm developed by T.F. Smith and M.S. Waterman, and is based on an earlier model appropriately named Needleman and Wunsch after its original creators [19]. The Smith-Waterman algorithm uses dynamic programming to find the best local alignment between any two given sequences. Based on certain criterion, usually a scoring matrix, scores and weights are assigned to each character to character comparison: positive for exact matches/substitutions, and usually negative for insertion/deletions. The exact scores are based on a scoring matrix. The scores are then added together and the highest scoring alignment is reported.

Algorithm 1 Smith-Waterman Algorithm

```
1: Declare a nxn similarity matrix
2: Initialize the top row (i=0) and left column (j=0) with 0
3: for i = 1; i < length(sequence); i++ do
4:    for j = 1; j < length(sequence); j++ do
5:       F(i,j) = max (0, F(i-1,j-1)+ s(xi,yi), F(i-1,j) - d, F(i,j-1)-d)
6:       Save index of term that contributed to the calculated value in F(i,j)
7:    end for
8: end for
9: Find maximum value in nxn matrix
10: Using saved indices in (6), traceback to first 0 encountered
```

(While tracing back in the Smith-Waterman algorithm, moving horizontally or vertically through the matrix represents inserting a gap in the opposing sequence. Moving diagonally represents aligning the two values for that cell.)

The Smith-Waterman algorithm is similar to the Needleman-Wunsch algorithm [15] and differs primarily in the fact that the traceback occurs from the maximum value in the nxn matrix to the first 0 encountered, rather than from the lower right corner to the upper left corner. This difference in the traceback procedure results in the best alignment between subsequences of the original sequences rather than an optimal alignment between the entire sequence.

Obviously, this method is $O(n^2)$ in both time and space. The computations involved are fairly light but the space requirements limit the use of this full matrix algorithm to work on large sequences. As mentioned in the introduction, some sequences that occur in nature are more than a few hundred thousand characters in length and are too big to fit in main memory.

Chapter 4

Pruning Algorithm

The need to analyze large sequences makes it necessary to find ways to reduce the space and time complexity of the Smith-Waterman algorithm. Work has been done in this area and there is ongoing research to tackle this problem.

4.1 Past research

For example, Hirschberg [9] developed a similar algorithm that uses a *divide-and-conquer* approach to reduce the search space for a two-sequence alignment problem from $O(n^2)$ to $O(n)$. This algorithm reduces the search space at the expense of added computations to compensate for the lost values and information during the divide-and-conquer approach. Since the major problem with the sequence alignment algorithms is the space requirements and the computations tend to be fairly light and quick, this algorithm allows for processing of much larger sequences.

Davidson [6] developed an algorithm based on the A^*[1] classic best-first search algorithm well-known in the Aritificial Intelligence community. Davidson's algorithm uses heuristics to ignore parts of the matrix that correspond to the unneeded parts of the sequence. This reduces the search space as well as the time requirements, as parts of the sequence are ignored. Of course, as is with the case with most heuristic-based algorithms, the optimal solution is not always guaranteed in this case.

Evolutionary computation techniques for multiple sequence alignment problems have also been explored by Cai et. al.[3]. Their algorithm searches for near optimal solutions to the local and global alignment problems by using general evolutionary computation techniques. Their results show positive and encouraging results when compared to the results of the straight-forward alignment algorithms.

Parallelization of the Smith-Waterman and other sequence alignment algorithms is another area under research. Combinations of a parallel architecture with the Hirschberg divide-and-conquer algorithm [9] takes advantage of the inherent parallel nature of the divide-and-conquer algorithm, thus resulting in lower space and time requirements for each node. The parallelization technique divides larger sequences into multiple shorter sequences, which are individually evaluated and then pieced together for a final alignment by the master node.

Most work being done in this area focusses on improving the existing algorithms to increase their efficiency and reduce the space-time requirements. The pruning algorithm

[1] A^* : pronounced A-star

presented in this section looks at improving the quality of the sequences before the sequence alignment algorithms use them. This algorithm could be used to process data in the background and as a pre-cursor to full-matrix algorithms such as the Smith-Waterman algorithm to reduce the matrix size needed for computing the optimal alignments.

4.2 The algorithm

The pruning algorithm removes the redundant parts of the input sequences by eliminating parts of the sequence that do not appear in the optimal alignment. These parts of the sequences are highly dissimilar to the corresponding parts of the other input sequences and hence do not affect the results. By varying the lengths of the subsequences that are checked for a high level of dissimilarity and changing the threshold value that defines high dissimilarity, the algorithm finds the shortest subsequences of the original sequences which still produce the same optimal local alignment through the Smith-Waterman algorithm.

4.3 Pruning Algorithm - Pseudo-code

This section describes the Pruning algorithm in pseudo-code.

Algorithm 2 Pruning Algorithm

```
1:  Allocate temporary buffer space for new str1 and str2
2:  while length of str1 >= sslength[2] do
3:    compare sslength characters of str1 and str2 starting at str1
4:    if score > threshold[3] then
5:      copy the characters to respective buffers
6:      increment str1 and str2 by 1
7:    else
8:      increment str1 and str2 by sslength
9:    end if
10: end while
11: copy buffers back to str1 and str2
```

The pruning algorithm starts at the beginning of both the sequences under consideration and takes groups of subsequences each *sslength* characters long. By comparing the similarity score of the two sequences (using perfect one-to-one alignment) to the threshold value, it determines whether the subsequences are highly dissimilar or not. If the subsequences turn out to be highly dissimilar, they are eliminated from the original sequences, otherwise they are ignored. The process continues by taking the next group of *sslength* characters, until the end of either one of the sequences is reached.

4.4 Sensitivity Analysis

The pruning algorithm is highly sensitive to the subsequence length parameter as well as the threshold value. Theory suggests that since the algorithm considers the entire subsequence of a given length and either keeps it or eliminates it, lower subsequence lengths would invariably produce better results at the cost of being ineffective in reducing the length of the original sequence considerably. On the other hand, larger subsequence length values

would have a lower probability of producing correct results but when they do, the savings in space and time would be enormous. This theoretical intuition does not hold in practice, since through experiments it is observed that smaller dissimilar parts of the sequence are still significant enough to appear in the results. Thus lower subsequence lengths cause dissimilar but significant parts of the original sequence to be eliminated, producing incorrect results in the end.

The threshold value determines whether two subsequences being compared are highly dissimilar or not. If the similarity score of the subsequences is below the threshold value, the subseqences are eliminated from the original sequences. Hence, a lower threshold value (more negative) would result in eliminating only highly dissimilar parts of the original sequence and hence would improve the results. The downside of this argument is that as the threshold value becomes lower, less and less of the original sequence is eliminated, thus affecting the savings in space and time. The results section will delve into some experimental results that show the behavior described above.

Chapter 5

Approach

To compare the effectiveness of the pruning algorithm, the results of the Smith-Waterman algorithm on the pruned data set needed to be compared to the results on the original data set. Moreover, since the pruning algorithm parameters varied, namely, threshold value and subsequence length, various combinations of these values had to be tested for varying effectiveness. The approach to the data analysis followed was as follows:

Algorithm 3 Approach to data analysis

```
1:  From real human genome data, generate files each containing one pair of sequences
2:  Use the generated files as input to the Smith-Waterman algorithm and save the results
3:  Use the generate files in (1) as input to the Pruning algorithm and save the results; the
    parameter values for threshold and subsequence length are varied
4:  Use the files generated in (3) as input to the Smith-Waterman algorithm and save the
    results
5:  Compare the results of (2) and (4)
6:  if Results are equal then
7:      Mark it as a success
8:  else
9:      Mark it as a failure
10: end if
```

Figure 5.1 describes the approach to analysis in graphical form.

5.1 Parameters

The parameters for the Pruning algorithm were varied to test for effectiveness under varying circumstances.

5.1.1 Threshold Value

The threshold value was varied from 0 to 2*(length * -37) in intervals of 10, where *length* signifies the length of the subsequence under consideration. Given the scoring matrix (Table 2.2) used in these experiments, the threshold value of 0 is relatively high. The lower threshold value of 2*(length * -37) is chosen such because -37 happens to be the middle value in the scoring matrix used; the highest and the lowest values were 136 and -160 re-

spectively. Thus a range of 0 to 2*(length * -37) provides a reasonable range of scores for the given pair of subsequences.

5.1.2 Subsequence Length

The subsequences used in the experiments were in the range of 50 - 1000 characters long. The subsequence length parameter for the pruning algorithm was varied from 1 - 100 to provide a good range of values to test. The highest value of the subsequence length was capped at 100 since it is highly improbable that removing larger and larger subsequences from slightly larger sequences would still result in the correct alignment using the Smith-Waterman algorithm. This is because as the subsequence size increases, the probability that a part of the original sequence that occurs in the optimal alignment is within the subsequence under consideration to be removed, increases.

5.2 Automation

To run the experiments over a large number of data sets and to vary the parameters, the entire process was automated using scripting in the *Perl* [16] programming language. The data sets were divided into pairs of sequences, the intermediate results were stored and lastly the results were compared and the successes and failure logged automatically using a group of *Perl* scripts. All scripts are available for download at the project website[1].

[1]To be available by May 9, 2005

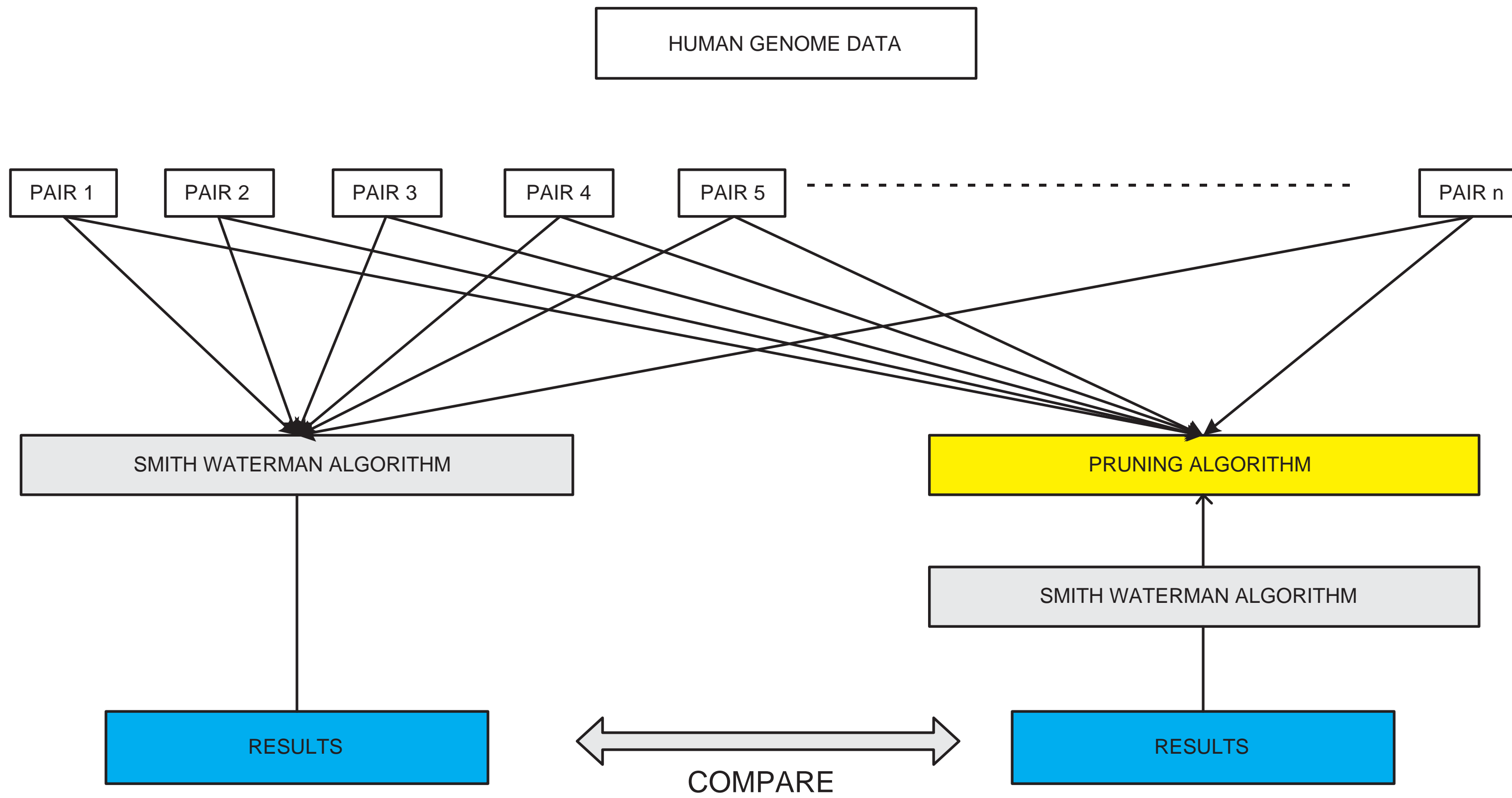


Figure 5.1: Data analysis approach

Chapter 6

Results

Experimental runs of the pruning algorithm were conducted on 10 different pairs of sequences from the real Human Genome database. The results of the experiments were compiled in two ways:

1. The subsequence length was held constant, the threshold value was varied
2. The threshold value was held constant, the subsequence length was varied

6.1 Constant subsequence length

For various values of subsequence length in the range of 5 to 100, the threshold parameter was varied. Figure 6.1 and Figure 6.2 present the results in terms of the number of successes out of 10 for each value of the subsequence length and the corresponding threshold values. The following conclusions can be drawn from the results of the experiments:

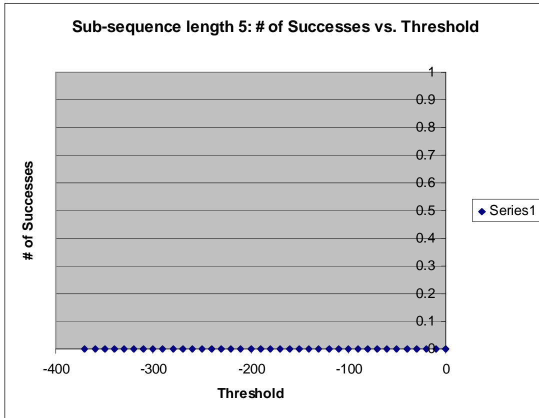
- Elimination of short subsequences is not beneficial
- Success rate increases as the threshold value decreases[1] since fewer parts of the sequences are eliminated
- Usefulness of the subsequence elimination program increases as the subsequence length increases

Further experiments need to be performed to observe the effectiveness of the algorithm on larger sequences. Moreover, the time and space savings of the correct results need to be measured to quantify the benefits of the algorithm.

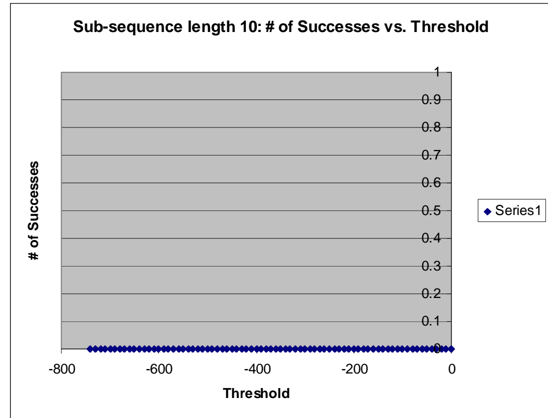
6.2 Constant threshold value

For various values of the threshold value, ranging from -50 to -8000, the subsequence length was varied. Figure 6.3 and Figure 6.4 present the results in terms of the number of successes out of 10 for each threshold value and the corresponding subsequence lengths. The following conclusions can be drawn from the results of the experiments:

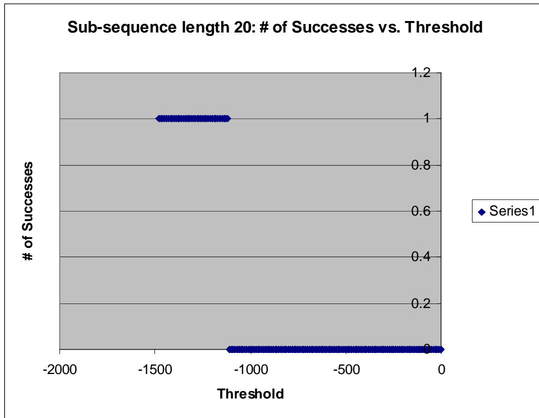
[1] becomes more negative



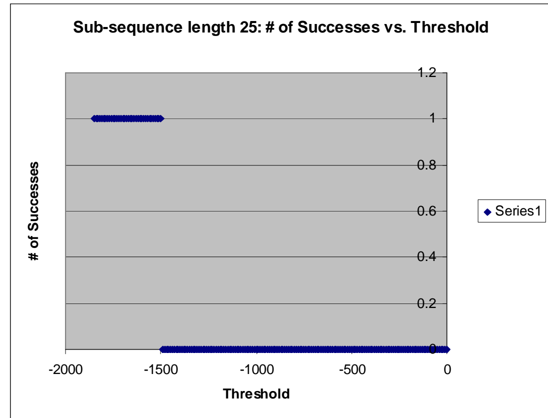
(a) Subsequence Length 5



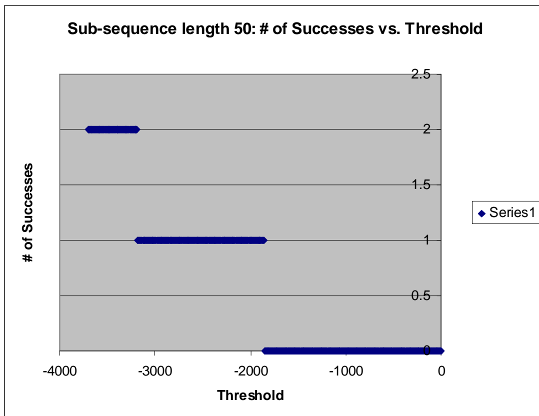
(b) Subsequence Length 10



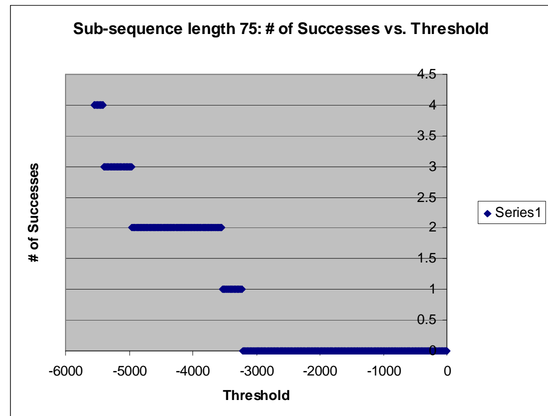
(c) Subsequence Length 20



(d) Subsequence Length 25

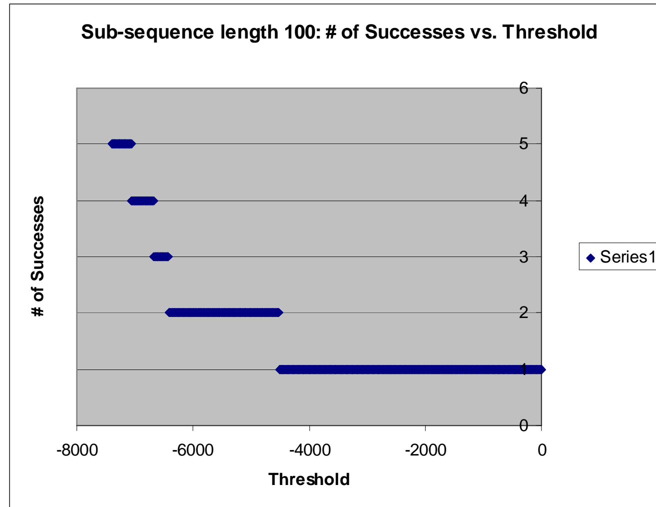


(e) Subsequence Length 50



(f) Subsequence Length 75

Figure 6.1: Constant subsequence lengths (5 - 75)



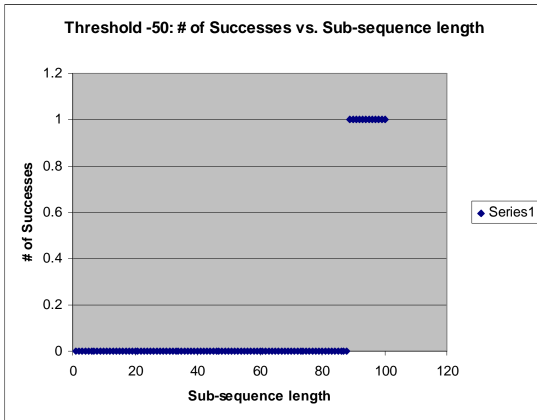
(a) Subsequence Length 100

Figure 6.2: Constant subsequence length (100)

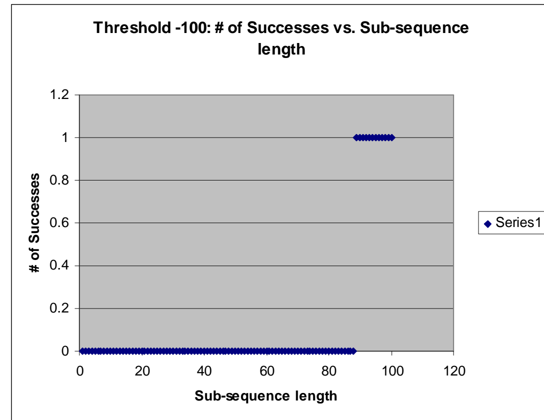
- Smaller threshold values produce better results, since fewer parts of the original sequence are eliminated
- For the 10 sequences tested, reasonable results are obtained for threshold values between -2000 and -8000

The theoretical intuition as described earlier in the paper does not agree with the experimental results. Nevertheless, an alternate explanation could be presented to justify the results. Firstly, the results indicate that smaller threshold values produce better results. This could be attributed to the fact that a lower threshold value indicates that only highly dissimilar parts of the sequence are removed, thus keeping most of the original sequence intact. This obviously would produce correct results as there is rarely any pruning of the original sequence and all the inherent data is preserved.

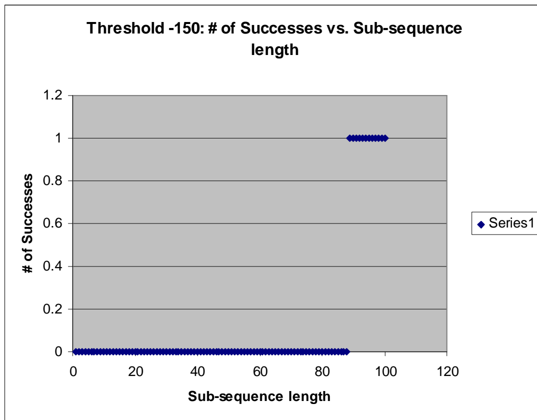
Secondly, the results show that considering larger subsequences produces better results. This is because the similarity score of larger subsequences are rarely below the threshold value because of a large number of matches and a relatively low gap penalty (-5). Thus, larger subsequences are usually not eliminated, thus preserving the original sequence as is. This, again, produces correct results.



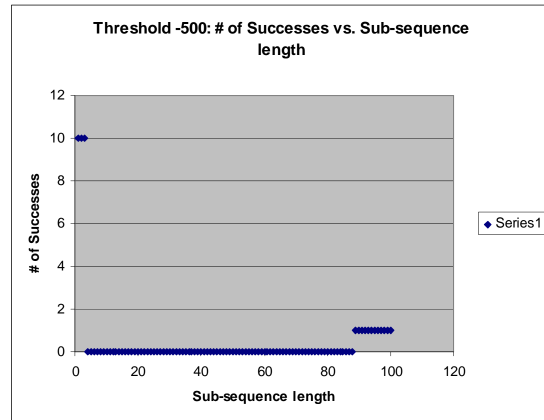
(a) Threshold Value -50



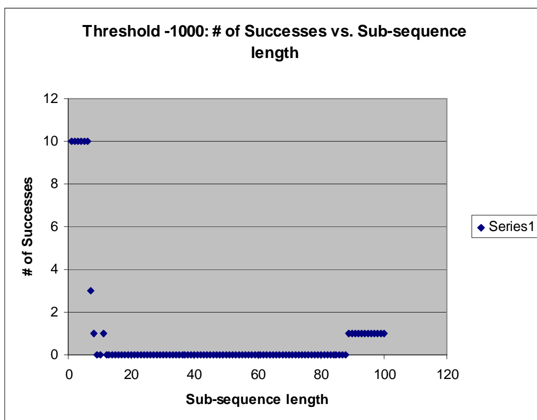
(b) Threshold Value -100



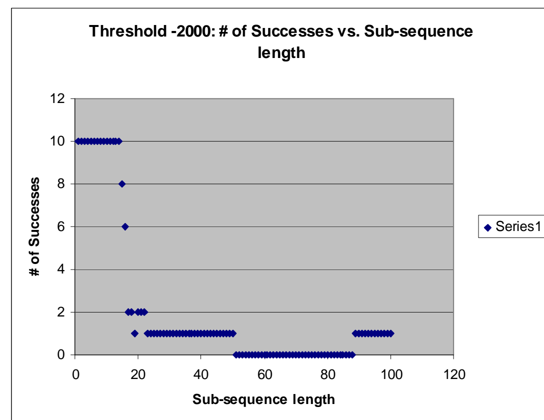
(c) Threshold Value -150



(d) Threshold Value -500

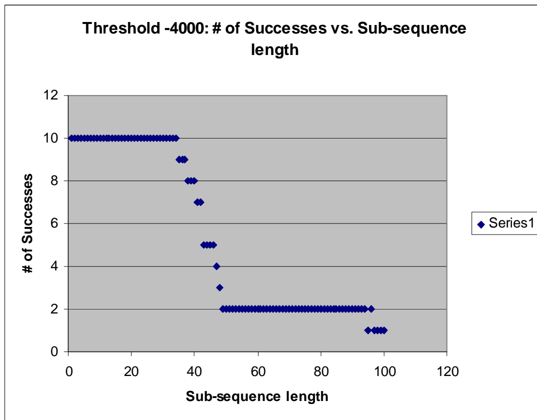


(e) Threshold Value -1000

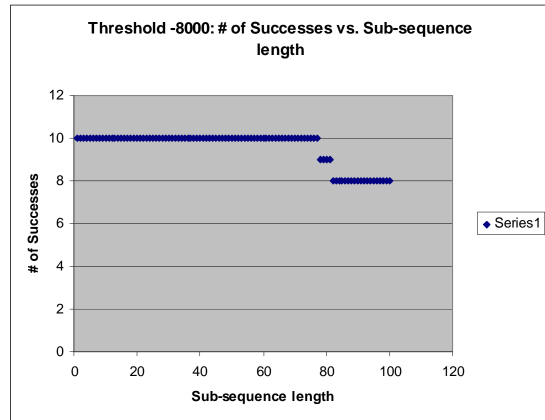


(f) Threshold Value -2000

Figure 6.3: Constant threshold values (-50 to -2000)



(a) Threshold Value -4000



(b) Threshold Value -8000

Figure 6.4: Constant threshold values (-4000 to -8000)

Chapter 7

Conclusion

The pruning algorithm reduces the search space of the Smith-Waterman algorithm by eliminating unneccessary parts of the input sequences. Even though the algorithm produces reasonable results in certain cases, the number of failures are still high. Work needs to be done to make the algorithm more *intelligent* by using probabalistic models to determine the dissimilarity of subsequences rather than simply checking the similarity score against a set threshold value. This would result in better results since groups of characters that frequently occur together in nature will then not be eliminated just based on the similarity score.

Also, the scoring matrix used in this paper (Table 2.2) was very simple. Different results might be obtained using other scoring matrices since the similarity score entirely depends on this matrix. Further work needs to be done to compare the results using different scoring matrices. Also, a model needs to be developed to determine the best scoring matrix to use based on the context of the given sequences.

Lastly, the savings in space and time need to be measured to determine the effectiveness of the pruning algorithm. By comparing these results against other heuristics based alignment algorithms like BLAST [14], the applicability of this algorithm in the industry could be determined.

Part II

Kernel Extensions to the μC/OS-II Real-Time Operating System

Chapter 8

Introduction

8.1 Background

Every senior in the Electrical and Computer Engineering department needs to complete a one semester long design project, termed the Senior Design Project, before he or she can graduate with a diploma from the department. This year, the seniors are involved in developing a prototype low power wireless sensor environment on the Lafayette College campus. This environment is required to support both fixed and mobile sensors allowing for environmental monitoring, alarm monitoring and vehicle/material position tracking. The actual statement of work and the final revised requirements document are attached to this paper as Appendix B.

8.2 Need for an Operating System

As mentioned in the requirements document, each node is required to perform several tasks at once. Some of the tasks need to be scheduled to run cyclically after a specified interval of time while others are asynchronous and should be triggered only after an interrupt[1] occurs. Moreover, the priorities of the task can change depending on events, network traffic and other external factors. All this together makes it virtually impossible to program the microprocessors in a traditional infinite loop fashion since it does not allow for the features mentioned above. An operating system provides the core functionality needed to implement the complicated system. The main features that require the use of an operating system are:

- Multitasking
  - Multitasking enables several tasks to run at the same time. Scheduling the tasks to run is a complicated process and is handled entirely by the operating system. This makes the design of the tasks easy, as the programmer does not have to worry about the intricacies of time-sharing and can concentrate on the problem at hand.
- Sufficient number of interrupt levels

[1] *interrupts* are alerts from the hardware which signify the occurence of an event. When a hardware interrupt occurs, the operating system takes control of the processor and hands it to an *interrupt service routine (ISR)* that acknowledges the interrupt and runs the necessary code

- The interrupt levels define the source of interrupts. There are basically two types of interrupt levels: system and bus. The system and bus interrupts are generated from the micro-controller bus and system I/O. Examples of system interrupts are the timer and serial link interrupts [12].

• Task Priority

  - Task priorities are essential when programming multiple tasks. Critical tasks can be given a higher priority so that they are not pre-empted by a low priority task during the execution of time critical code. Moreover, by efficiently assigning task priorities, relatively more CPU time can be allocated to more CPU intensive tasks while pushing the other tasks to background.

A good example that demonstrates the need for an operating system involves running multiple tasks on the board with different priority and scheduling requirements [2]. For example, a single task that needs to run forever can be simply coded in assembly language using an infinite loop. This is shown diagrammatically in Figure 8.1.

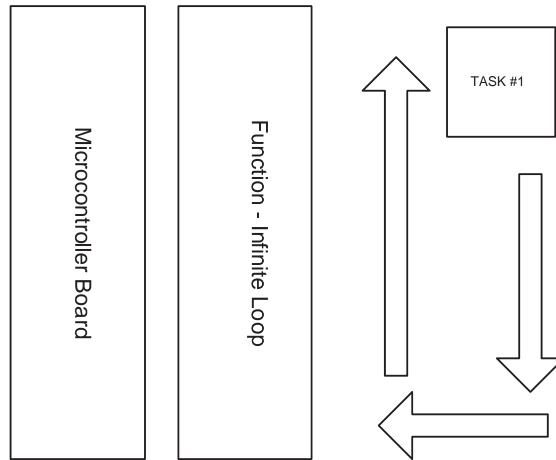


Figure 8.1: Task in Infinite Loop Method

Now, let's imagine the number of tasks needs to be increased to 3. This, again, can be done simply by calling the 3 functions in sequence, or adding code for the 3 tasks one after the other and then looping infinitely. This procedure is shown diagrammatically in Figure 8.2.

While the infinite loop method suffices thus far, if more restrictions are put on the tasks, the problem becomes more complicated. For example, let's suppose the following restrictions need to be put on the tasks:

1. Task 3 is more important than Task 1 and it should run three times as often
2. Task 2 is more important than Task 1 and it should run twice as often

Even though a solution still exists using the infinite loop method, it is undoubtedly more complicated than a comparable solution using an operating system. Using an operating system, the priorities of the tasks can be specified at task creation time. Moreover, using a cyclical timer, the run interval of the tasks can be specified and can be dynamically

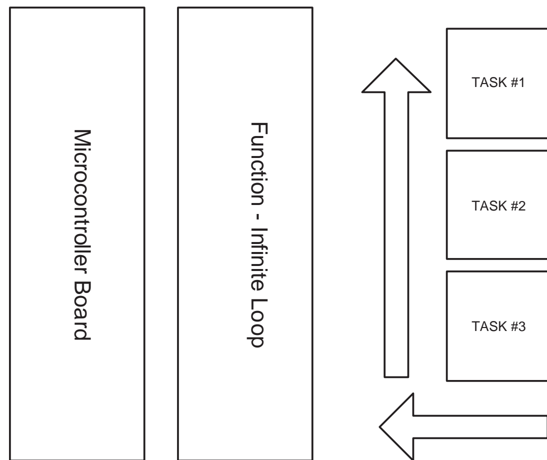
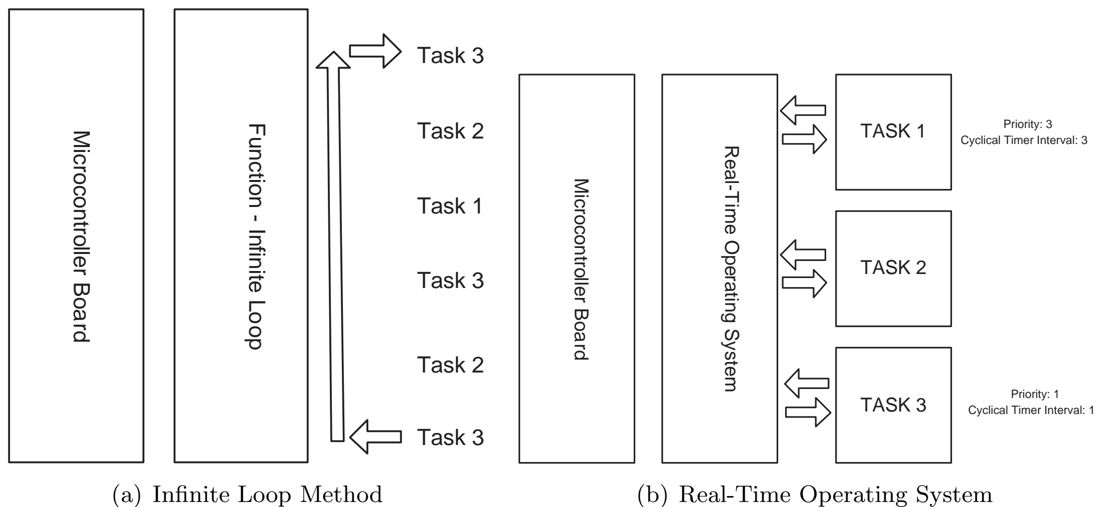


Figure 8.2: 3 Tasks in Infinite Loop Method

changed. The difference in the complexity of the two approaches, one that uses the infinite loop method and one that uses an operating system can be clearly seen in Figure 8.3. Thus, through this example, it is clear that use of an operating system makes programming multiple tasks with complicated run schedules, much easier than the traditional non-operating system infinite loop method.



(a) Infinite Loop Method (b) Real-Time Operating System

Figure 8.3: Comparing infinite loop method and a RTOS

8.3 What is a real time operating system?

A real time operating system (RTOS) differs from a traditional operating system in the fact that a RTOS guarantees a certain capability within specified time constraints [20]. Real time operating systems are used wherever the responsiveness of a system is highly critical owing to the changing external variables and the need to process these changes immediately. In a real time operating system, it is critical that tasks run at specified times and within a

specified time frame after an event occurs. Since the nodes being designed by the seniors need to process real time data and react to an ever-changing environment, a real time operating system was the best choice for the task at hand.

8.4 μC/OS-II Real-Time Operating System

μC/OS-II is a real-time operating system developed by Jean A. Labrosse [11] and is widely used in education as well as the industry. This operating system has been ported to a wide variety of architectures including the Motorola 68HC11 and provides all of the necessary features mentioned above. Using a port of this OS on the 68HC11 microprocessor, individual tasks could be written and configured to fulfill the time, priority and scheduling requirements. Moreover, an excellent resource for this operating system is available in the form of the book called 'MicroC/OS-II: The Real Time Kernel' written by the operating system's developer. Due to its rather complete functionality, availability of a port free of charge and good documentation, this operating system was chosen as the operating system of choice for the Senior Design project. While it provides most of the basic functionality, the requirements of the Senior Design project demanded extensions to the kernel to provide a complete set of functionality.

Chapter 9

Kernel Extensions

Three major features were added to the kernel:

- Support for Coordinated Universal Time (UTC)
- Cyclical Timer
- Serial Communications Interface

All features provide an easy to use API[1] for application programmers to use which has been documented in a users' manual as well as described in this paper. The rest of the paper will describe the need, implementation and the programmer's API for each of the features added and mentioned above.

[1] Application Program Interface

Chapter 10

Coordinated Universal Time (UTC)

The project required use of a coordinated timing system for the nodes to ensure data integrity and reporting in a timely fashion. Packets sent from the nodes were encoded with a time stamp to ensure that all nodes communicated within an acceptable time frame. Moreover, the timing feature enabled performance testing on the boards by measuring the response time of the packets sent to and fro from the controller node. The coordinated timing system also allows for geographically widely distributed nodes to have the same time reference for communications, data gathering and reporting purposes.

Coordinated Universal Time is an internationally accepted time standard that is based on seconds (SI) and is defined and recommended by the International Radio Consultative Committee (CCIR) and maintained by the *Bureau International des Poids et Mesures (BIPM)* [7]. This standard was chosen because of its popularity in the field and ease of implementation.

10.1 Representation of UTC

[5] UTC is represented in two parts by a 29-bit integer. The least significant 19 bits are referred to as the **time of week** (TOW) and count from 0 to 403,199, incremented by 1 every 1.5 seconds. The 10 most significant bits are called the **week number** (modulo 1024). Since a native 29-bit number is not available on the 68HC11 (or any popular architecture for that matter), a 32-bit unsigned integer was used instead. It was ensured that the most significant 3 bits were always masked off as 0.

10.2 Implementation

The Motorola 68HC11 board used for development and deployment was configured to produce a system tick every $1/100^{th}$ of a second. Two global counters were then added to the kernel - one for the time of week (**utc_tow**) and one for the week number (**utc_wno**). A third counter counted the number of ticks since the board started up. A function was added to the OSTimeTickHook () function, which was called at every system tick, that incremented the value of utc_tow every 1.5 seconds. The board was configured to issue a system tick every 1/100th of a second, and so 1.5 seconds corresponded to 150 system ticks. Code was also added to specify rollover after time of week reached 403,199 and at

that point, the week number was incremented by 1 and time of week was re-initialized to 0. Code to make the week number rollover after 1023 was also added.

10.3 User Functions

- get_utc()

  This function returns the current system UTC time as a 32-bit unsigned integer. The lower 19 bits of the integer specify the time of week; the next 10 bits specify the week number and the upper 3 bits are set to 0.

  Parameters:

  None

  Return value:

  The function returns a 32-bit unsigned integer that contains the time of week and the week number.

- set_utc(INT32U val_to_set)

  This function sets the coordinated universal time of the system.

  Parameters:

  INT32U val_to_set: This is a 32 bit unsigned integer that specifies the value to be set. The lower 19 bits of the value passed must specify the time of week. The next 10 bits must specify the week number and the upper 3 bits must be set 0. No error checking is performed in this function.

  Return value:

  None

Chapter 11

Cyclical Timer

The requirements of the project made it necessary for certain tasks running on the nodes to run at fixed time intervals. This feature was required to gather data from the sensors after a fixed time interval and also for the nodes to send reports back to the controller in a periodic fashion. Since the operating system did not natively support cyclical tasks, extensions to the kernel had to be written to allow support for such a feature.

Two main enhancements were made to the kernel for this purpose to allow:

- Any task to be made cyclical at run-time
- Active reporting in case of a task missing execution after being scheduled

Traditionally, μC/OS-II supports acyclical tasks. If these tasks are to be run after every specified time interval, they need to be delayed for a number of system ticks and then resumed[1].

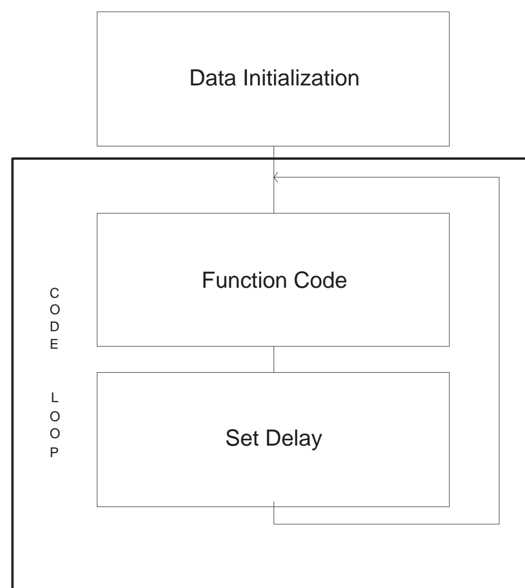


Figure 11.1: Typical Acyclical Task

[1] μC/OS-II requires tasks to run in an infinite loop. Hence acyclical tasks need to be suspended and resumed as and when required.

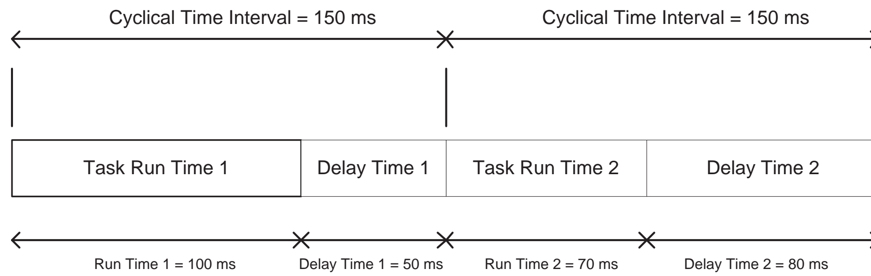


Figure 11.2: Variable Time Delay

The calculation for the number of system ticks, and eventually the time period, that the task needs to be delayed for is done by calculating the run-time of the task and then subtracting it from the cyclical time interval. But the run-time of a task cannot be deterministically assigned at compile time since it depends on factors such as CPU availability, I/O delays, and interrupts. Thus, using traditional delay mechanisms fall short of fulfilling the requirements of a cyclical task.

Figure 11.2 shows the concept of a variable time delay. Here, the required cyclical task interval is $150ms$. The initial run time of the task is $100ms$ and hence the first delay time is set to $150 - 100 = 50ms$. Now, suppose the run time of the task becomes $70ms$ because of increased CPU availability. The delay time must now be changed to $150 - 70 = 80ms$ instead of the original $50ms$. This is not possible at run-time and hence this method of making tasks cyclical is not very convenient or accurate.

A schematic of the implementation of a cyclical task is shown in Figure 11.3.

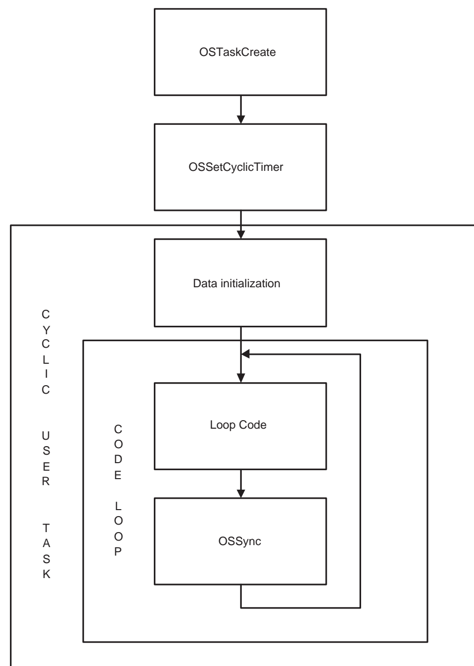


Figure 11.3: Implementation of a cyclical user task

11.1 Support for Cyclical Tasks

It is worthy to note that support for cyclical tasks is missing from most modern day operating systems such as GNU/Linux, Windows or variations of BSD. These modern day operating systems rely on user level applications such as 'cron' to program cyclical tasks. This method, though sufficient for most purposes, lacks high resolution for cyclical time intervals and does not support active reporting mechanisms in case of a task skipping execution after being scheduled. These shortcomings make user level cyclical timers inadequate for system critical tasks. Support for cyclical tasks needs to be implemented at kernel level to ensure that tasks can be programmed to run, theoretically, at resolutions that match the system ticks. Moreover, kernel level support also allows for user defined actions to be taken in case of task run failure or missing a time interval by pre-empting all existing tasks and running recovery code.

11.2 Implementation

11.2.1 Task Control Block

Two new fields were added to the task control block *struct*[2]. OSTCBCyclicalTimer is a 16-bit unsigned short that was set to the value of the time interval that the task is supposed to cyclically run at. OSTCBNScheduled is a 16-bit unsigned short value that stored the number of times a task has been scheduled to run.

The task control block is a data structure that holds critical information about the current state of various tasks running on the operating system. Among the values stored in the task control block are, pointers to the top and bottom of the stack, pointers to the previous and the next task control block, delay counter for the task (OSTCBDly), unique priority of the task (OSTCBPrio), variables that control the messaging and queueing functions of the operating system, variables that specify the current run state of the task in the ready table and variables that hold the values related to the cyclical timer aspect of the task. The operating system holds a singly linked list of pointers to the task control blocks of all the tasks created by the operating system and cycles through them at every system tick.

11.2.2 Changes to OSTimeTick()

The OSTimeTick () call is an internal system call that is called every system tick. This function goes through the task control blocks of all the tasks running and decrements the value of the OSTCBDly variable. The OSTCBDly variable holds the value of the number of ticks that the task must wait before being scheduled to run. For acyclical tasks, the OSDelay () function sets the value of this variable and delays the task. If the OSTCBDly variable becomes 0 and the task is not suspended, then it is made ready to run.

Changes were made to this call by adding support for cyclical task. Code was added to check if a task is cyclical (value of OSTCBCyclicalTimer is not equal to 0), and if it is, then the OSTCBDly variable is set to the value in OSTCBCyclicalTimer and the value in OSTCBNScheduled is incremented by 1. Following this, the task is made ready to run by changing the status of the task in the *task ready table*. A flowchart explaining the flow

[2] *struct* is a *C* language data structure that encapsulates numerous data fields and variables into one entity

of control through this system call is shown in Figure 11.4 and the pseudo-code for the OSTimeTick () function is given in Algorithm 4.

Algorithm 4 OSTimeTick ()

```
1:  Get next task from a linked list of task control blocks
2:  if Current task is the idle task then
3:     Exit the function
4:  else
5:     Decrement OSTCBDly for current task by 1
6:     if OSTCBDly = 0 then
7:        if OSTCBCyclicTimer > 0 then
8:           Set OSTCBDly of the task to OSTCBCyclicTimer
9:           Increment OSTCBNScheduled by 1
10:       end if
11:       Make task ready to run
12:    end if
13:    Go to step (1)
14: end if
```

11.3 Features

The cyclical timer support added to the kernel makes it possible for a task to be made cyclic at any point during the execution of a task. Moreover, a task can be reset to an acyclical status by simply calling the OSSetCyclicTimer () with a time interval of 0.

If a call to the OSDelay () function, which is reserved exclusively for acyclical tasks, is made on a cyclical task, the task re-synchronizes at the next time interval. If a time interval is missed due to any reason, the return value of the OSSync () function gives the user a chance to decide if any recovery code needs to be run. This feature allows for monitoring of system critical tasks and allows for appropriate action to be taken in case of a task failure due to excessive run-time in CPU intensive code. Also, the OSSync () function, resets the variable that holds the number of times a task has been scheduled so far and allows the task to resynchronize to the fixed time interval when the next time interval occurs. This way, the task continues to function properly even after a task run failure occurs.

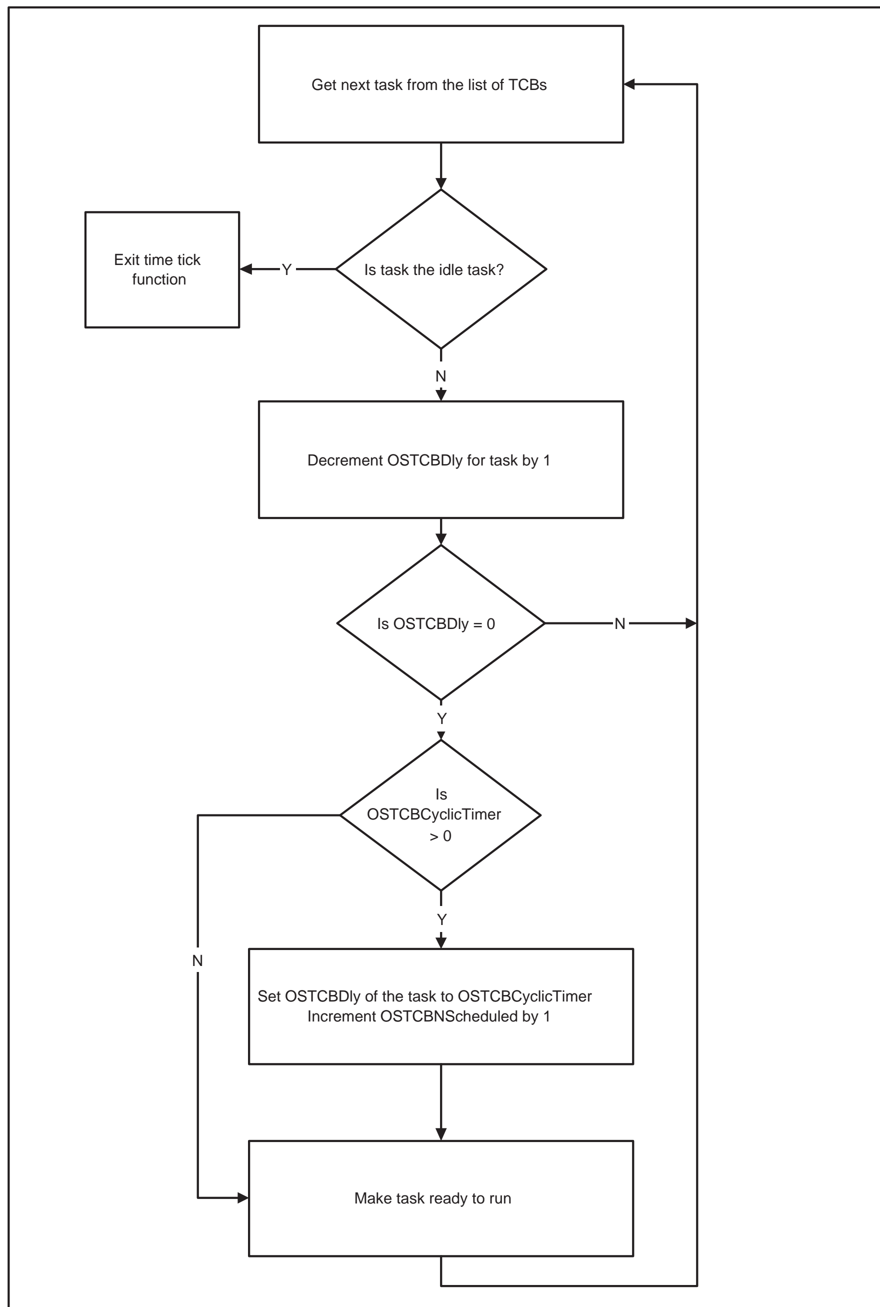


Figure 11.4: OSTimeTick() Flowchart

11.4 User Functions

- OSSetCyclicalTimer (INT8U prio, INT16U timer)

  This system call was added to the kernel to set the timer interval for a specified task and hence make it cyclical. The call to this can be made at any time during the program execution and hence the value of the time interval for a cyclical task can be dynamically altered. This is useful since tasks can be configured remotely and at run time.

  Parameters:

  INT8U prio: 8-bit *unsigned char* that specifies the priority of the task being addressed. The priority of a task is a unique value that identifies any given task.

  INT16U timer: 16-bit *unsigned short* that specifies the time interval at which the task shall repeatedly run. The value is given in the number of system ticks and the actual time value is board-dependent.

  Return value:

  The return value is an 8-bit *unsigned char*. The value is 1 if the task control block of the specified task was successfully altered, 0 is an error occurred.

- OSSync()

  This system call was added to the kernel to disable a task after execution to make it wait until the next time interval elapsed. This call also resets the value of OSTCBNScheduled after returning the value stored in it minus 1. This is called from within the code loop of a cyclical task after all the task specific code and before the closing parentheses of the infinite loop. The code for OSSync () is written in a critical section such that the function is not pre-empted by an interrupt request.

  Parameters:

  None

  Return value:

  This function returns a value that specifies the difference between the number of times a task has been scheduled to run and the number of times it has actually run. A value of 0 thus signifies that the task is performing as desired whereas a value of 1 or more signifies that the task skipped execution even after being scheduled to run. This could arise due to the run time of the task being longer than the time interval after which it is supposed to run cyclically or due to other higher priority processes taking up too much of the CPU time.

  In mission critical systems, it is important to ensure that tasks run when they are supposed to since most tasks invariably affect the performance of the system and are necessary for the completion of a job. Failure in running tasks at scheduled times can create bottlenecks for other tasks or could result in system crashes. Hence, to avoid this, in addition to a high dependability of the cyclical timer, a reporting mechanism is necessary to report any failures that might occur. This mechanism then allows the programmers to take the appropriate action when these failures occur to maintain system integrity.

  Figure 11.5 shows a scenario where a task may skip a scheduled run. Suppose a task is supposed to run every $70ms$ and the usual run time of the task is less than this

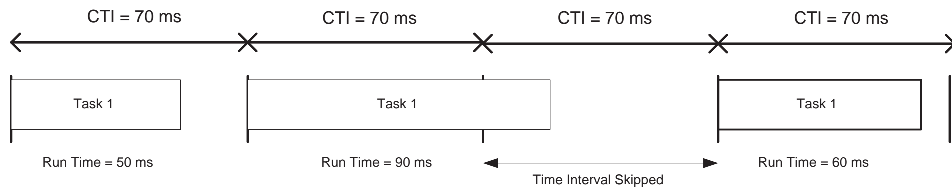


Figure 11.5: Cyclical Task Failure

value. Now, lets suppose, for reasons like CPU unavailability or I/O delays, the task takes longer than $70ms$ to complete a run. In this case, the task will be scheduled to run, but will not be made *ready*, owing to its current status as already *active*. Thus, the next time OSSync () is called, the return value of the function will be 1 instead of 0, thus signifying that the task skipped a run. At this point, the user may instruct the processor to run any recovery code needed.

Chapter 12

Serial Communications Interface

The senior design project required the use of wireless serial communications between nodes and between the controller and different nodes. A few uses of such communication capability are:

- Reporting of sensor data
- Remote diagnostics and configurability
- Requesting data on demand
- Failure reporting and correction

Moreover, the requirements document specified that 4 48-byte packets must be able to be sent from each node per minute.

A communications interface was thus implemented that allowed transmitting and receiving 64 byte packets. The packet length was increased from the original 48 required to 64 because of the need to add header information to the raw data being transmitted over the network.

12.1 Interrupt System

The interrupt system on the Motorola 68HC11 board can be configured to call a function whenever data arrives at the serial port. Moreover by enabling the transmit interrupt, data can be sent over the serial port by writing the appropriate byte to the serial communications data register (SCDR).

A function called SCIHandle () was written to handle all interrupts related to serial communications. When an interrupt related to serial communication peripherals occurs, the SCIHandle () function is called. The function checks the status of the Serial Communications Status Register (SCSR) and determines whether data is waiting to be read at the serial port or if the interrupt happened because of the transmit interrupt system being enabled. If data is waiting at the serial port, the byte is read from the SCDR and added to a buffer. If the interrupt happened due to the transmit interrupt system, then the byte to be sent is written to the SCDR.

12.2 Intermediate Buffer System

Bytes are read as they are received and put into a temporary 64-byte array. The kernel sets a flag to alert the user when 64 bytes of data[1] have been received at the serial port. The user program can continually check the status of this flag and then call the GetByte () function once the flag is found to be set. This call to GetByte () resets this flag and makes data available in another data buffer by copying over the values of the temporary buffer. This way, a packet received at the serial port is available to be processed for some time[2] and is not overwritten by new bytes arriving continuously at the port.

12.3 User Functions

- SendPacket(INT8U *c)

  This function sends 64 bytes of data over the serial port. The function does not return until all 64 bytes have been successfully sent.

  Parameters:

  INT8U *c: This is a pointer to the start of a 64 byte unsigned *char* array. The array must contain the data in order in which it is to be transmitted. The function does not check for validity of the array and hence the user must ensure that the pointer points to a valid array of at least 64 bytes.

  Return value:

  None

- GetByte()

  This function must be called after the user determines that the **data_rdy** flag has been set. A call to this function copies the data from the intermediate buffer to the data buffer and makes it accessible for the users. It also resets the **data_rdy** flag so that the program is not tricked into thinking that two packets have been received. The users can then access the received packet (64 bytes) by accessing the pointer to the start of the data buffer specified by **datas0**.

  Parameters:

  None

  Return value:

  None

[1] 1 packet was designed to be 64 bytes long
[2] till the next packet is received

Chapter 13

Conclusion

The kernel extensions made to the μC/OS-II RTOS made it possible for the senior design group to write tasks that were cyclical in nature, and use a coordinated timing system to ensure data integrity and reporting. The use of an operating system was worth the time and effort since it simplifies the complex requirements to simply writing individual tasks, as the operating system handles the rest of the complexity natively. The cyclical timer support implemented includes features such as synchronization and reporting in case of scheduling and task execution errors. These features are critical for efficient and dependable execution of multiple tasks on the operating system. While this project still remains to be completed, the implementation of an operating system and the extensions to the kernel made make the task at hand much simpler.

Bibliography

[1] UCSC Genome Bioinformatics. http://genome.ucsc.edu/, 2005.

[2] Sherry R. Bishop, J. Implementation of the uc/os-ii real-time operating system on a motorola 68hc11. Independent Study, 2000.

[3] Juedes D. Liakhovitch E. Cai, L. Evolutionary computation techniques for multiple sequence alignment. Technical report, Ohio University, Athens, OH, 2000.

[4] S. Cates. Scoring matrices. http://cnx.rice.edu/content/m11062/latest/, February 2005.

[5] Seismology Research Center. Tn199812b - gps week number rollover. http://www.seis.com.au/TechNotes/TN199812B_GPS_WNRO.html, 2005.

[6] A. Davidson. A fast pruning algorithm for optimal sequence alignment. Technical report, University of Alberta, Edmonton, Alberta, Canada.

[7] Institute for Telecommunication Sciences. Coordinated universal time. http://www.its.bldrdoc.gov/fs-1037/dir-009/_1277.htm, 2005.

[8] National Genomforschungnetz. http://www.ngfn.de/, 2005.

[9] D.S. Hirschberg. A linear space algorithm for computing maximal common subsequences. *Association for Computing Machinery*, pages 341 – 343, 1975.

[10] Aumann K. D. Huehne, R. Fasta format description. http://ngfnblast.gbf.de/docs/fasta.html, 2005.

[11] J. J. Labrosse. *MicroC/OS-II The Real-Time Kernel*. CMP Books, Berkeley, CA, 2002.

[12] International Business Machines. Kernel extensions and device support programming concepts. http://publibn.boulder.ibm.com/, April 2005.

[13] S. K. Moore. Understanding the human genome. *IEEE Spectrum*, 11:34 – 35, November 2000.

[14] NCBI. Blast information. http://www.ncbi.nlm.nih.gov/, 2005.

[15] S. Needleman and C. Wunsch. A general method applicable to the search for similarities in the amino acid sequences of two proteins. *Journal of Molecular Biology*, 48:443 – 453, 1970.

[16] Perl. The perl programming language. http://www.perl.org.

[17] H. H. Jr. Ropelewski A., D. D. II. Strategies for searching sequence databases. *Biotechniques*, 28(6), 2000.

[18] T.F Smith and M. S. Waterman. Identification of common molecular subsequences. *Journal of Molecular Biology*, 147:195 – 197, 1981.

[19] Stanford University. Smith-waterman algorithm. http://www-cse.stanford.edu/classes/sophomore-college/projects-00/computers-and-the-hgp/smithwaterman.html, 2005.

[20] Whatis.com. Real time operating systems. http://whatis.techtarget.com/, 2002.