Searching Biological Sequence Databases Using Distributed Adaptive Computing

Nicholas P. Pappas

Thesis submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Master of Science
in
Computer Engineering

January 24, 2003

Bradley Department of Electrical and Computer Engineering
Blacksburg, Virginia

Keywords: configurable computing, adaptive computing, FPGA, sequence comparison, sequence alignment, bioinformatics, Smith-Waterman algorithm

Searching Biological Sequence Databases Using Distributed Adaptive Computing

Nicholas P. Pappas

Abstract

Genetic research projects currently can require enormous computing power to processes the vast quantities of data available. Further, DNA sequencing projects are generating data at an exponential rate greater than that of the development microprocessor technology; thus, new, faster methods and techniques of processing this data are needed. One common type of processing involves searching a sequence database for the most similar sequences. Here we present a distributed database search system that utilizes adaptive computing technologies. The search is performed using the Smith-Waterman algorithm, a common sequence comparison algorithm. To reduce the total search time, an initial search is performed using a version of the algorithm, implemented in adaptive computing hardware, which is designed to efficiently perform the initial search. A final search is performed using a complete version of the algorithm. This two-stage search, employing adaptive and distributed hardware, achieves a performance increase of several orders of magnitude over similar processor based systems.

Dedication

To my parents,
for their infinite support in the past,
and hopefully their continued support in the future.

To my friends,
Adrienne, Andrei, Anne, Annette, Bobby, Doug, Eliza, Forest, Jean, Jennifer, Julianne, Laura, Mark, Matt, Robin, and Tara,
for their encouragement.

Acknowledgements

First I would like to thank my thesis advisor and committee chair, Dr. Athanas for seeing the potential of, initiating, and supporting this research project. A special thanks goes to Dr. Dickerman of the Virginia Bioinformatics Institute who was our invaluable guide through the world of bioinformatics, and without which, the quality of the project would have suffered greatly. I would also like to thank my other committee members Dr. Michael Hsiao and Dr. Mark Jones.

For the development of the system, a debt of gratitude is owed to Anusha Dandapani for aiding in the software development, Kiran Puttegowda for development of the hardware, William Worek for his development of an initial system, and Jae Park for his expert guidance.

In regards to the writing of this thesis, a major thanks goes to my father, George Pappas, for his meticulous reviewing and editing of this document.

Finally, I would like to thank the ASPIRES (A Support Program for Innovative Research Strategies) program for financial support of the project, without which, none of this would have been possible.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

Bioinformatics is an area of study that applies computer technology to biological data. DNA sequencing endeavors such as the Human Genome Project, and the continuous advancement of genetic sequencing technology, are currently generating DNA sequence data at an exponential rate. The amount of data is increasing at a rate faster than that of the development of microprocessor technology; thus, the amount of data is outpacing the ability to process it. As a result, new faster methods of processing are needed.

A fundamental problem in the field of bioinformatics is the comparison of two sequences of biological data, such as DNA sequences or proteins. Moreover, the comparison of a sequence against a large set of sequences, such as a large database, in and effort to search for the most similar sequences in the set is important. A search of a biological sequence database can take a great deal of time for two main reasons. First, sequence comparison algorithms have a significant time complexity because they must allow for genetic mutations, such as character insertions, deletions, or substitutions. Second, the size of a sequence database can be very large, some on the order of billions of characters.

Many sequence database search systems use common general-purpose commercial computing architectures. For example, Celera built one of the most powerful non-government supercomputing facilities in the world, consisting of 800 interconnected Compaq, Alpha-based, computer systems [11]. The Virginia Bioinformatics Institute (VBI) recently purchased two supercomputers: a 196 CPU IBM Linux cluster; and a Sun Microsystems 15K [24]. However, there are many commercial and academic efforts in using hardware accelerators in sequence database searches. For example, Paracel's [30] GeneMatcher2 [29] uses custom ASIC processors that are optimized for sequence comparison; and TimeLogic's [35] DeCypher [36] and Compugen's [8] BioXL/H [9] use Field Programmable Gate Arrays (FPGAs) to perform sequence comparisons.

In this thesis, a sequence database search system, named HokieGene is presented. HokieGene is a multi-processor system that incorporates adaptive computing technology. The FPGAs are used to implement an enhanced version of a common sequence comparison algorithm, in an effort to gain significant increases in performance over other processor based, and custom hardware based systems.

1.1 Thesis Statement

The objective of the research project was to develop a high-speed biological sequence database search system utilizing adaptive and distributed computing technologies.

To achieve this objective, multiple networked computers were used as a computing platform. A software control system was developed to manage sequence databases, queries, and search processing over the multiple machines. The search processing was accomplished using both the processors of the individual machines, and an adaptive computing system, known as SLAAC, [37] which is comprised of an API and an adaptive computing hardware platform known as Osiris.

The author's specific contributions to the project are the following:

- Utilized the SLAAC API to create an interface to the Osiris adaptive computing hardware.
- Developed biological sequence database management software to preprocess databases to increase system efficiency.
- Developed a multi-user, batch oriented, distributed search system control software.

- Analyzed the performance of the search system.
- Analyzed the utility and performance of the SLAAC API.
- Compared the system to other common commercially available systems.

1.2 Thesis Organization

Chapter 2 presents the background information on biology and bioinformatics for the work presented in the thesis. Chapter 3 gives a detailed description of the sequence alignment algorithm that is discussed throughout the thesis. Chapter 4 presents the biological sequence database search system developed for the thesis. It includes background information on the technologies used, the design issues related to the system, and the system's hardware and software implementation details. Chapter 5 presents an analysis of the system, and a comparison of the system to other biological sequence database search systems. Chapter 6 presents related work that is currently being done, and that may be done in the future. Chapter 7 summarizes the research, the results, and the expected results of ongoing work.

Chapter 2

Background

2.1 Basic Biology

DNA (deoxyribonucleic acid) stores an organism's genetic information in the form of a long sequence of molecular subunits. The information is encoded using four bases: adenine, guanine, cytosine, and thymine (abbreviated as A, G, C, and T respectively). Nucleotides, each of which consists of a base plus molecules of sugar and phosphoric acid, make up a strand. Each base in one strand binds to a specific base in another strand, where A's bind to T's, and G's bind to C's, each of which is called a base-pair. The two strands with bound base pairs constitute a DNA molecule.

Of the total DNA sequence in an organism's genome, some regions encode the information needed to synthesize proteins, while other regions have other roles, such as controlling the timing of protein expression. For the purposes of this paper, we will only focus on protein production. A protein is sequence of molecules, similar to DNA in that it has a backbone of common molecules, to each of which an amino acid is attached. There are 20 different amino acids, each with a unique set of properties, such as size, charge, and tendency to interact with water.

When a cell produces a protein, the segment of DNA that encodes the protein is identified by "begin" and "end" codes. The cell copies the segment of DNA into a piece of RNA, which is similar to DNA except the base thymine is replaced by the base uracil (U). The RNA is then provided to the cellular machinery that produces the protein. This machinery uses the bases in groups of three, called codons, to determine the amino acid to be added to the protein.

Proteins have multiple functions such as building blocks, transports, catalysts, and signal transducers. These functions are determined by the protein's amino acid structure. For example, the 3-dimensional shape, which is critical to the protein's functions, is based on the chemical properties of the amino acids.

2.2 Bioinformatics

Bioinformatics is a field of science that brings together biology, computer science, and information technology, in an effort to increase our understanding of biology. As the field of bioinformatics has developed, it has been labeled with various other names. These names include computational biology, computational genomics, computational molecular biology, and biomedical informatics. Much of the motivation behind this field is drug development; currently to discover drugs and, in the future, to model the effects of drugs on biological systems. In addition, there are many other scientific motivations, such as determination of evolutionary trees of multiple species from their DNA. The field of bioinformatics includes many computationally challenging problems, many of which involve large amounts of data, very complex systems, or both.

One of the fundamental tasks in bioinformatics is the generation, storage, and dissemination of biological sequence data. Many laboratories and private companies are working to determine the genetic makeup of biological systems. For example, the Human Genome Project is an international effort to map all of the genes of a human. The processes and equipment used in these efforts are constantly improving and, as a result, the amount of data that is being generated is increasing at an exponential rate. This biological sequence data can be obtained from a variety of public and private databases. For example, the common public DNA database in the United States, named GenBank [25], publishes sequences with taxonomic, identification, bibliographic citations, and other information for each entry. Figure 2.1 shows the growth of the GenBank database in terms of the total number of base pairs and total number of sequences [25]. The total number of base pairs stored in GenBank has doubled approximately every 17 months over its lifetime, every 14 months in the last 10 years, and every 13 months in the last 5 years. This rate of growth is faster than the rate of the development of computer processors. As a result, the total amount of data is outpacing the ability to process it.

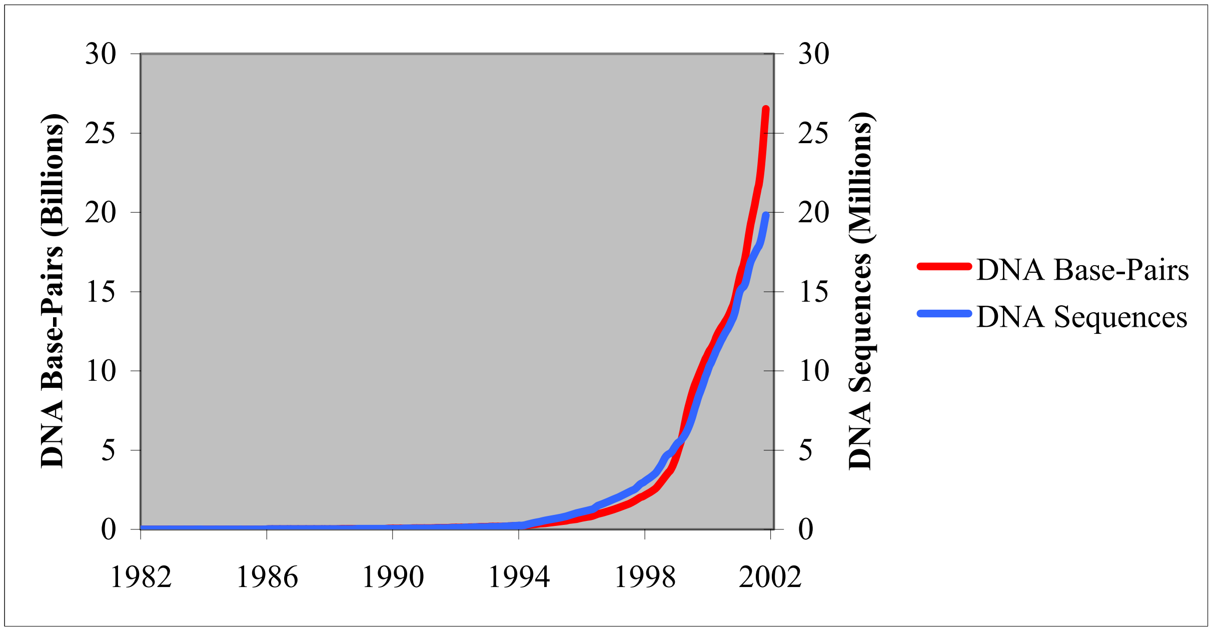

Figure 2.1. Growth of the GenBank DNA database.

Another fundamental problem in bioinformatics is the comparison of sequences of biological data in an effort to determine their degree of similarity. Algorithms, such as the Smith-Waterman algorithm, [32] are used to compare two sequences allowing for genetic mutations such as insertions, deletions, or substitutions. It is also useful to compare multiple sequences; however, the number of operations involved in non-heuristic algorithms is proportional to the product of the lengths of the sequences, and as a result, require a great deal of processing time. In addition, it is useful to determine the evolutionary history or phylogenic tree of a given a set of sequences. This problem is also computationally intensive.

Another problem is the prediction of the three-dimensional shape of a protein from its amino acid makeup, also known as protein folding. A protein with a given makeup will always fold into the same shape, a result of the chemical properties of the amino acids. Given the amino acid makeup and the chemical properties of the amino acids, it is possible to determine the shape of a protein, however the non-heuristic algorithms have an exponential time complexity based on the length of the protein, and as a result, cannot be solved in a reasonable amount of time for even medium length proteins.

2.3 Searching Sequence Databases

A common task in bioinformatics is the search of a database for the sequences with significant similarity to a query sequence. These search methods must balance two parameters: the sensitivity of the algorithm to the differences between sequences; and the time needed to search the entire database. These methods also incorporate a number of parameters so that a search can be tailored to the specific needs of the user.

2.3.1 Sequence Comparison

The similarity score of two sequences is based on the alignment of the two sequences that maximizes biological conservation. This similarity score is the based on the score of each pair-wise alignment, and the cost of gaps. The score of each pair-wise alignment can be simple, such as +1 for a match and –1 for a substitution or mismatch, or the score can be tailored to the particular nucleotides or amino acids in the character pair. A scoring matrix contains the similarity score for each pair of characters in the sequence alphabet. For example, an amino acid that is hydrophobic, or repels water molecules, is more likely to be replaced in a similar protein by another amino acid that is also hydrophobic, as opposed to one that is hydrophilic, or attracts water molecules. Thus the score of two hydrophobic amino acids will be higher than that of a hydrophobic and hydrophilic amino acid pair. Two common examples of scoring matrices are PAM matrices [12], which are based on evolutionary change statistics, and BLOSUM matrices [2], which are based on substitution frequency observations. Gaps, which are interpreted as the insertion or deletion of a character in a sequence, are always given a negative score. These negative scores are also referred to as a cost or penalty. One gap penalty calculation method is for each gap to result in an equal penalty, or linear gap penalties. Another more biologically realistic gap penalty computation is to assume that insertions or deletion in a sequence occur in groups; thus, a higher penalty is used for an initial gap, with a lower penalty is used for each subsequent gap, referred to as an affine gap penalty.

2.3.2 Contemporary Database Search Software

There are two main techniques used to search a sequence database. One technique is to compare a query sequence to every database sequence using comparison algorithms such as the Smith-Waterman or Hidden Markov model. This method can be very time consuming, yet generally produces the best results. Another technique is to first preprocess the database before any searches, such that a subset of database sequences that are similar to the query sequence can be found very quickly. The sequences in the subset of the database are then directly compared the query sequence. Two commonly used search systems that use this preprocessing technique are BLAST (Basic Local Alignment Search Tool) [3, 27] and FASTA [13, 31]. These systems preprocess the database by generating a data structure similar to a hash table. The table contains entries for all possible short sequences of a specified length. Each entry of the table contains a list of sequence identifiers. Each sequence identifier corresponds to a database sequence that contains the table entry sequence. When a search is performed, all database sequences that have a common short subsequence to the query sequence can be found very quickly using this table.

2.3.3 Contemporary Database Search Hardware

One common type of hardware used to run database searches is a standalone workstation or Personal Computer (PC). A majority of the computationally intensive operations of the search software can typically be executed in parallel; thus, the software is typically executed on distributed systems that consist of multiple processors or multiple machines. One type of distributed system is a collection of PC's networked together, commonly referred to as a Pile of PC's (PoP), or a Beowulf cluster. This type of system has the advantage of consisting of very inexpensive commercially available hardware. However, this hardware is not optimized for sequence comparison algorithms.

Two systems that use specialized hardware that is optimized for sequence comparison algorithms are Paracel's GeneMatcher2 and TimeLogic's DeCypher. The GeneMatcher2 system uses multiple custom Application Specific Integrated Circuits (ASICs) that implement multiple custom sequence comparison processors. The DeCypher system uses multiple Field Programmable Gate Arrays (FPGAs) to implement sequence comparison algorithms.

Chapter 3

Smith-Waterman / Needleman-Wunsch algorithms

The Smith-Waterman [32] and Needleman-Wunsch [28] algorithms are sequence alignment algorithms that determine the maximal alignment of two sequences based on certain parameters, and determine a score that represents the quality of the alignment and the degree of similarity of the two sequences. In an alignment, each character of a sequence can be aligned to another sequence in one of two ways: the character can be aligned with any character in the other sequence, or the character can be aligned with a gap inserted into the other sequence. In an example alignment shown in Figure 3.2, there are multiple character identity alignments, one alignment of a character G in sequence 1 to a gap in sequence 2, and another alignment of a character G in sequence 1 aligned to an A in sequence 2. The Needleman-Wunsch algorithm (often mistakenly referred to as the Smith-Waterman algorithm) computes the similarity of two entire sequences, or global alignment. The Smith-Waterman algorithm however, computes a score of the maximally similar subsequences of two sequences, or local alignment.

For the following, let S represent a sequence with length m, where S_i represents the character at position i of sequence S and $0 < i \leq m$, and let T represent another sequence, with length n, where T_j represents the character at position j of sequence T and $0 < j \leq n$. Let the function $s(a, b)$ represent the cost or score of aligning character a to character b, and let g represent the cost or penalty of inserting a gap. Finally let an alignment of subsequences $S_1 \ldots S_i$ and $T_1 \ldots T_j$ be represented by $H_{i,j}$.

For all i and j, the algorithm computes the score for subsequences $S_1 \ldots S_i$ and $T_1 \ldots T_j$, or $H_{i,j}$, where the score is the maximum of three possible alignments. In one alignment, S_i *and* T_j are aligned and the score of aligning S_i and T_j, or $s(S_i, T_j)$, is added to the alignment score $H_{i-1,j-1}$. In another alignment, S_i is aligned to a gap inserted into position j of T, and the gap cost is added to the alignment score $H_{i-1,j}$. Finally, T_i is aligned to a gap inserted into position i of S, and the gap cost is added to the alignment score $H_{i,j-1}$. More simply, each subsequence alignment is determined from the two characters, and the three smaller subsequence alignments.

Each subsequence alignment score is used in the calculation of three other alignment scores. This use of scores is known as dynamic programming, and these algorithms are known as dynamic programming algorithms. Dynamic programming is an algorithm development technique akin to "divide and conquer", where sub-problems have common sub-problems. Implementations of this algorithm generally store each value of $H_{i,j}$, in an m by n matrix so that each value $H_{i,j}$ does not have to be recomputed, as shown in Figure 3.1.

$$\begin{array}{c|cc}
 & T_{j-1} & T_j \\
\hline
S_{i-1} & H_{i-1,j-1} & H_{i-1,j} \\
S_i & H_{i,j-1} & H_{i,j}
\end{array}$$

Figure 3.1. Dynamic Programming Matrix.

3.1 Algorithm Details

The Needleman-Wunsch algorithm determines the global alignment of two sequences, where a match is scored as 0, mismatches and gaps result in positive scores, and the final score of the maximal global alignment is minimal. The final score is analogous to number of mutations to transform sequence *S* into sequence *T*, or the edit distance, and is equal to $H_{m,n}$, where $H_{i,j}$ is given in Equation 3.1, the function *s(a, b)* is equal to 0 if *a* is equal to *b*, and positive otherwise, and *g* is positive.

$$H_{i,j} = \min \begin{cases} H_{i-1,j-1} + s(S_i, T_i) \\ H_{i,j-1} + g \\ H_{i-1,j} + g \end{cases} \tag{3.1}$$

The Smith-Waterman algorithm determines the maximally similar subsequences of *S* and *T*. Character alignments that are considered good alignments are assigned positive scores, alignment considered poor alignments and gaps are assigned negative scores. This scoring policy is necessary for computing local alignments because no value of $H_{i,j}$ is allowed to go below zero, so that previous bad alignments do not spoil good local alignments. Bad alignments that score below zero are set to 0 to essentially reset the alignment calculation. The final score of the maximally similar subsequences is maximum value of the matrix *H*, where $H_{i,j}$ is given in Equation 3.2; *s(a, b)* is positive for good alignments, and negative for poor alignments; and *g* is negative.

$$H_{i,j} = \max \begin{cases} H_{i-1,j-1} + s(S_i, T_i) \\ H_{i,j-1} + g \\ H_{i-1,j} + g \\ 0 \end{cases} \tag{3.2}$$

Figure 3.2 shows an example Smith-Waterman alignment calculation where *s* is +4 for a match and –3 for a mismatch; *g* or gap penalty is equal to –6; and a final alignment score of 15. Note, that alignments $H_{1,1}$ and $H_{1,2}$ do not fall below zero; thus, they do not affect the maximal local alignment. Also note that in alignment $H_{3,5}$, a gap is inserted into *S*, and in alignment $H_{6,8}$, two different characters are aligned, to achieve the maximal local alignment.

Sequences:

S = GTACGCTGTTG

T = CACCTATTA

Alignment Matrix:

	G	T	**A**	**C**	**G**	**C**	**T**	**G**	**T**	**T**	G
C	0	0	0	4	0	4	0	0	0	0	0
A	0	0	**4**	0	1	0	1	0	0	0	0
C	0	0	0	↖ **8**	← **2**	5	0	0	0	0	0
C	0	0	0	4	5	↖ **6**	2	0	0	0	0
T	0	4	0	0	1	2	↖ **10**	4	4	4	0
A	0	0	8	2	0	0	4	↖ **7**	1	1	1
T	0	4	2	5	0	0	4	1	↖ **11**	5	0
T	0	4	1	0	2	0	4	1	5	↖ **15**	9
A	0	0	8	2	0	0	0	1	0	9	12

Alignment:

Sequence	Alignment
S	`GTACGCTGTTG`
	`  ||  ||X||`
T	` CAC CTATTA`

Figure 3.2. Smith-Waterman alignment of two sequences.

3.2 Smith-Waterman Gap Penalties

Smith-Waterman alignment gap penalties are a function of the length of a sequence of consecutive gaps. As shown in Equation 3.3, the maximal alignment $H_{i,j}$ is either an extension of $H_{i-1,j-1}$, or a consecutive sequence of gaps inserted into S or T, where the length of the gap sequence results the maximal alignment, and the value of the gap penalty is determined by the function W_k. One specific type of gap penalty is known as linear, where $W_k = c * k$, and c is a constant. The linear gap penalty of a sequence of gaps is proportional to its length, and simplifies Equation 3.3 to equal Equation 3.2.

$$H_{i,j} = \max\begin{cases} H_{i-1,j-1} + s(S_i, T_i) \\ \max\{H_{i-k,j} - W_k\}, k \geq i \\ \max\{H_{i,j-k} - W_k\}, k \geq j \\ 0 \end{cases} \tag{3.3}$$

The actual insertion or deletion genetic mutations are more likely to occur in groups; thus, gap penalties that have a greater cost for short gap sequences than linear penalties and smaller cost for long gap sequences than linear penalties, result in more accurate genetic alignments. A common solution is affine gap penalties, which have two parameters: an initial gap penalty, which is cost to begin a gap sequence; and a gap extension penalty, which is the cost to extend a gap sequence, and is less than the initial gap penalty. Affine gap penalties are defined by $W_k = uk + v$ where the initial gap penalty is defined as, $w_1 = u + v$ and the gap extension penalty is v. Affine gap penalties also have the advantage that there is a simplification such that they can be calculated in constant time.

3.3 Smith-Waterman Optimizations

Gap penalty calculations have a time complexity of *O(N)*; thus, increasing the Smith-Waterman time complexity to *$O(N^3)$*; however, affine gap penalties can be calculated in constant time [14]. Let the two gap penalty equations in Equation 3.3 be represented by *P* and *Q*, where $W_k = uk + v$ and $w_1 = u + v$. *P* and *Q* can be simplified, as shown in Equation 3.4. In this simplification, the gap penalties *P* and *Q* can be calculated in constant time, using previous values of *P* and *Q*, and saving values of *P* and *Q* for subsequent iterations. The lesson here is to mind your P's and Q's.

$$\begin{aligned} P_{i,j} &= \max\{H_{i-k,j} - (u * k + v)\}, k \geq i = \max\{H_{i-1,j} - w_1, P_{i-1,j} - u\} \\ Q_{i,j} &= \max\{H_{i,j-k} - (u * k + v)\}, k \geq j = \max\{H_{i,j-1} - w_1, Q_{i,j-1} - u\} \end{aligned} \tag{3.4}$$

The Smith-Waterman algorithm space requirement is *O(mn)* as a result of *m* by *n* calculation matrix. However, if the size of *m* and *n* are very large, the space requirement could exceed a computer system's space limit. In *H*, all values in a single row r are dependant only on the values in the row itself, and the previous row *r-1*. As a result, each row of *H* can be calculated using space for r and *r-1*.

Chapter 4

Search system

The biological database search system performs searches using the Smith-Waterman algorithm, utilizing adaptive and distributed computing technologies in an effort to achieve the fastest possible search times. The system consists of three major components: an optimized version of the algorithm designed for fast initial database searches implemented in adaptive computing hardware; a hardware-software interface; and distributed system control software.

The system performs a two-stage search. An initial search is executed with a version of the Smith-Waterman algorithm, implemented in adaptive computing hardware, to flag sequences whose comparison score exceeds a predetermined threshold. In the second search, the flagged sequences are searched using a conventional implementation of the Smith-Waterman algorithm. This two-stage approach was found to be significantly faster than a single stage search.

4.1 Adaptive Computing

Adaptive computing systems, also known as reconfigureable computing systems, are systems that are comprised of reprogrammable logic devices such as field programmable gate arrays. These systems are generally used to increase processing speed over microprocessors, while maintaining a certain level of flexibility. The following describes the adaptive computing technologies used in this project.

FPGAs are a type of programmable logic devices or (PLD), which are hardware devices that can be configured to perform a wide variety of tasks. FPGAs consist of basic units of logic, such as individual gates, look up tables or multiplexers. These units are all connected to an interconnection network. Each interconnect of the network is controlled by a single bit of memory, which is programmable by the user.

One of the FPGAs applicable to this project is the Xilinx Virtex II X2CV6000 [39, 40]. The unit of programmable logic in a Virtex II FPGA is referred to as a configurable logic block (CLB), where each CLB is made up of four smaller units referred to as slices. A CLB has connections to the general routing matrix, as well as connections to neighboring CLBs and connections between slices to facilitate circuits such as long carry chains and shift registers. Each slice consists of two four-input function generators, two flip-flops, multiplexers, and other common logic. Virtex II devices also incorporate other commonly used components such as blocks of RAM and multipliers.

Systems Level Applications of Adaptive Computing (SLAAC) [37] is a project at the Information Sciences Institute (ISI) East, a part of the University of Southern California, and is sponsored by the Defense Advanced Research Projects Agency. The goal of the SLAAC project is to develop an open, standard, and scalable adaptive computing platform. This platform consists of FPGA boards that are designed to reside in a host computer, and an application programmer's interface (API) for host software [4]. The platform provides for simplified communication and control between the board and host, with FIFO data input/output channels, board memory access, clock control, control register, and other functionality, available to both board implementations and host software.

The Osiris board is an FPGA board developed by the SLAAC project [5], and used in the HokieGene system. The board contains two Virtex II FPGAs: a Virtex II 1000 (XC2V1000), referred to as IF, which is used as an interface between the user FPGA and the host; and a Virtex II 6000 (XC2V6000) [40] referred to as the XP, which is a user programmable device available for implementing hardware designs.

The Osiris board contains two types of memory accessible by XP and two types of memory accessible by IF. The IF accessible memory, typically used for storing configurations, is made up of 4 MB of flash memory and 6 MB of asynchronous SRAM. The XP has access to ten SRAM memory modules, each a 512K × 36 ZBT RAM capable of running at 200 MHz, and two 256MB SDRAMs with an onboard SDRAM controller. Access to this board memory can be shared with the host computer. The Osiris Board streams data into, and out of, XP through 64-bit wide and 512 word deep, input and output FIFOs. This memory and FIFO functionality is accomplished via a 64-bit, 66 MHz, PCI connection with the host machine. Figure 4.1 shows a diagram of the Osiris Board Architecture.

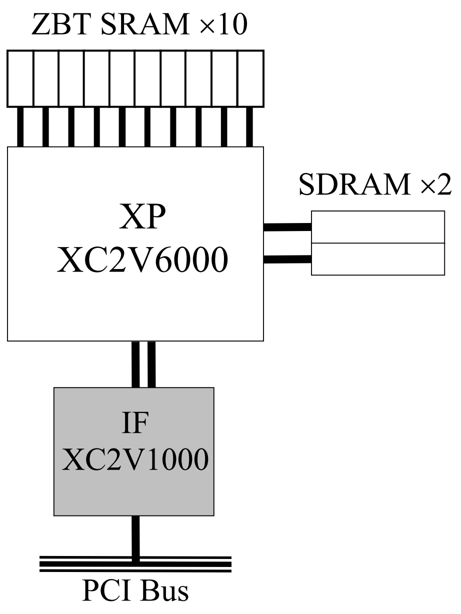

Figure 4.1. Osiris Board Architecture Diagram

The SLAAC API allows simplified development of host machine software. The API provides functionality to initialize a board with an FPGA implementation, and set frequencies of control clocks. The API also provides data transfer functionality. Data can be transferred to and from the board using the FIFOs or the board memory, with both blocking (execution is halted until the transfer is complete), and non-blocking (execution is not halted) modes. Finally, handshake, interrupt, and control register functionality are also provided.

4.2 Distributed Computing and MPI

Parallel computing or distributed computing utilizes multiple processors simultaneously to perform task. There are a wide variety of parallel computing machines, but a common type consists of several machines connected to a network, commonly referred to as a Pile of PC's (PoP), or a Beowulf cluster [41].

With any parallel computing architecture, there are the problems of communication between nodes and control of the system as a whole. MPI or Message Passing Interface is a commonly used library specification for communication and control for parallel processes [23, 34]. For example, the library provides functionality to send and receive data between individual machines, broadcast data to many machines, and collect data from many machines. Implementations of MPI for Beowulf clusters provide the libraries, compliers, tools, and ability to execute an application on the machines of the cluster from a single terminal.

4.3 Design Criteria/Overview

One system design consideration is system deployment. The system could potentially be implemented on a single machine and used by a single user. The system could also be implemented using multiple machines with one master and multiple slaves, and shared by multiple users. In addition, some of these machines may or may not contain the sequence search accelerator hardware. As a result, system flexibility and configuration control are required. The HokieGene software uses a configuration file that describes information such as each machine used in the system and the location of important directories.

A major design criterion is the size of sequence databases. The National Center for Biotechnology Information (NCBI) is a research program of the National Library of Medicine, which is within the National Institute of Health. NBCI maintains and distributes many common sequence databases. One of these is a DNA sequence database, referred to as NT [25], which contains all non-redundant GenBank sequences. The NT database contains approximately 1.3 million sequences, with 6.4 billion sequence characters, and a total file size of 6.2 GB, as of November 2002. As a result, it is not feasible to store all the searchable databases in memory; therefore, they must be stored on disk. When searching a database, the sequence data must be loaded from the disk into memory, requiring a significant amount of time. To minimize this time, each database is reformatted so that each sequence is stored using the minimal amount of space by using only the minimum number of necessary bits to represent each character, which generally results in a 4 to 1 size reduction of DNA databases. In addition, the database is divided into sections, and each section is placed on the local file system of a slave machine. Therefore, the slave machines each read relatively small amounts of data in parallel; thus, reducing the time to read a database.

Another consideration is how this system could be deployed and used. Due to the total cost of the system, multiple users would most likely share it on a network or over the Internet. It can probably be assumed that many users would submit queries at any given time. Some of these users will probably be searching the same databases. Due to the time it takes to read the database from disk, queries searching a common database will be batched together to minimize the number of times a database is loaded from disk. This will maximize the overall throughput of the system, but the individual user response time could vary greatly, depending on when the database of their query is scheduled. In addition, a web interface for submitting queries and displaying results would allow the system to easily shared over a network.

Finally, the initial-search hardware acceleration should be as fast and efficient as possible to minimize the total search time. The implementation of the Smith-Waterman algorithm in adaptive computing hardware uses multiple processing elements to calculate values of the dynamic programming matrix simultaneously. To achieve the fastest possible hardware implementation, a simplified form of the Smith-Waterman algorithm is used, because a simplified algorithm requires fewer hardware resources; thus, more processing elements can be implemented, and more simultaneous calculation can be performed.

The HokieGene system is intended to be implemented using one master machine, and multiple slave machines with multiple FPGA accelerator boards; however, it is also capable of running on a single machine. These machines use the x86 architecture, run the Linux operating system, and are connected together by a network, with a network file system. The master machine controls the overall operation of the system. It runs the user interface, receives and schedules user queries, and collects and formats the search results. The slave machines receive queries from the master, load the database to be queried in to memory, and perform the search using either the CPU or Osiris board. Figure 4.2 shows block diagram of search system implementation using multiple machines.

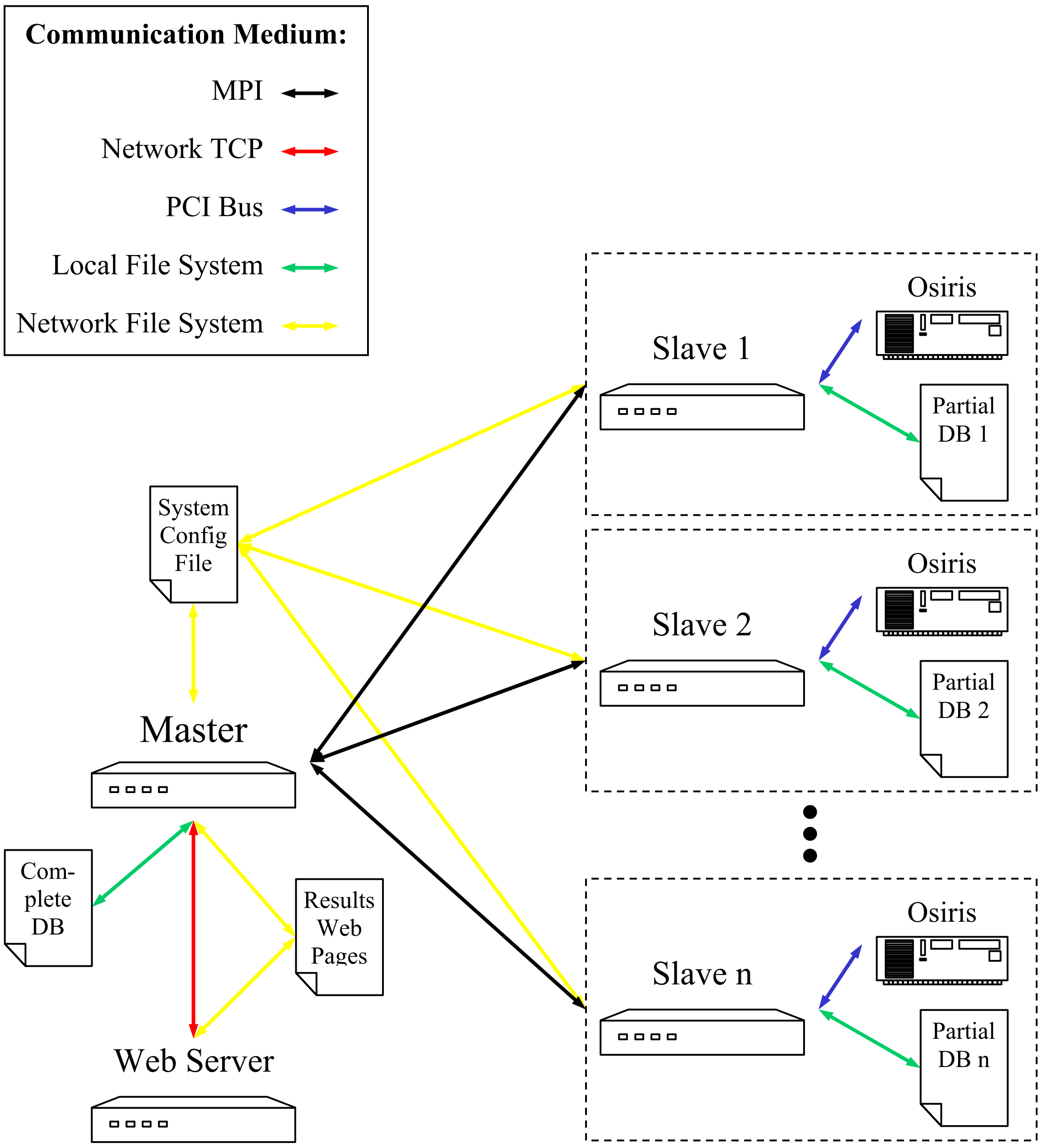

Figure 4.2. Search System Block Diagram.

4.4 Initial-Search Smith-Waterman Hardware Implementation

One of the major advantages of performing a task using an FPGA is that it can be configured to perform many operations simultaneously. The calculation of each element of the dynamic programming matrix in the Smith-Waterman algorithm is dependent only on the elements above and to the left. As a result, all the elements in a diagonal of the dynamic programming matrix can be calculated independently. This allows the Smith-Waterman algorithm to be implemented in parallel using an array of identical processing elements, where data is passed from one element to the next, known as a systolic array [17, 22]. In this approach each processing element, each iteration, calculates one element of the diagonal of the dynamic programming matrix, and only the necessary portions of the matrix are stored, with the rest discarded.

The hardware implementation consists of three types of components: a control component, which processes the input FIFO data and controls the hardware operation; two systolic arrays of processing elements, which implement the Smith-Waterman algorithm, one for the forward query sequence and the other for the reverse complement query sequence; and a results generator component, which processes the output of the two systolic arrays and sends the result data to the output FIFO. All components are controlled by a single, global clock signal, which is stopped if there is no data in the input FIFO, or if the output FIFO is full. The processing elements are also controlled by a global select signal, which is generated by the control component. The select signal sets the arrays in one of two modes of operation: a query mode, where the query sequence is processed by the control component and piped into the arrays, and a database mode, where the database sequence is processed by the control component and piped into the arrays.

4.4.1 Processing Element

Each processing element calculates a cell of the Smith-Waterman calculation matrix during each clock cycle. At the beginning of each clock cycle, a processing element receives the values $H_{i-1,j-1}$, and $H_{i,j-1}$, and a new sequence character from its neighboring processing element in the array. The processing element then compares the stored query sequence character to a database character and computes the cell value. Finally, the processing element outputs the values $H_{i,j}$, and $H_{i-1,j}$ and sequence character received to the next processing element in the array for the next clock cycle. If the array is in the query mode, the sequence character passed is a query sequence character, and is stored by the processing element. If the array is in the database mode, the sequence character passed is a database sequence character.

To reduce the resource requirement of each processing element, a simplified form of the Smith-Waterman algorithm was implemented. In this implementation linear gap penalties are used, and the scoring parameters are +1 for a match, -2 for a mismatch, and –1 for a gap. In addition, the implementation does not determine the similarity score, but whether the score in any of the processing elements exceeded a predetermined threshold. The threshold value implemented is 32, for two reasons: very little hardware resources are required to detect if a cell value exceeds 32 because only a single bit of the cell value needs to be monitored; and this threshold will only allow a small percentage of sequences to be flagged as exceeding the threshold, which reduces the final search time.

Each processing element has clock and select inputs, which are connected to the global clock and select signals respectively. Neighboring processing elements have six interconnections. For each connection, a processing element has an input and output, where the output of a processing element during a given clock cycle is processed by a neighboring processing element the following clock cycle. The processing element neighbor inputs include: the value $H_{i,j-1}$, named “C in”, which is used to calculate the cell; the value $H_{i-1,j-1}$, named “A in”, which is used to calculate the cell; a threshold flag, named “F in”, to indicate whether the previous cell scores exceeded the threshold; a reset flag, named “RST in”, which is used to indicates the end of a sequence and to reset the processing element to zero; a database sequence character, named "DB in"; and a query sequence character, named "Q in". The processing element neighbor outputs include: the value $H_{i,j}$, named "D out", which is required by the next processing element for its next cell calculation; the value $H_{i-1,j}$, named "A out", which is required by the next processing element for its next cell calculation; a threshold flag, named "F out", to indicate whether the processing element's cell score, or any previous cell scores, exceeded the threshold; a reset flag, named "RST out", which is used to indicate the end of a sequence and reset the next processing element to zero, and is enabled if the processing element was reset by a previous processing element; a database sequence character, named "DB out"; and a query sequence character, named "Q out".

4.4.2 Control Component

Input data is received from the input FIFO and is initially processed by a control component. In an effort to minimize the PCI bus bandwidth required to send sequence data to the hardware, two bits are used to represent each nucleotide. However, this leaves no other possible characters for control purposes, such as end of sequence. Therefore, the first data word received is a control data word, which indicates the type and length of sequence data that follows. The control component then reads the sequence data indicated by the control data word. After the sequence data words are processed, the processing elements are reset, and the next data word processed is a control word. When the control component receives a control data word with bit 63 set to 1, the sequence data that follows is 50 data words, or 1,600 query sequence characters. The select signal sets the systolic arrays to the query mode, and the query sequence characters are passed on to the systolic arrays. When the control component receives a control data word with bit 63 set to 0, the sequence data that follows are database sequence characters, where the number of sequence character data words is given by the value in bits 0 to 10 of the control word. Consequently, the maximum number of database sequence words that can be sent is 2^{11}-1 or 2047 words, or 65504 database sequence characters. The select signal then sets the systolic arrays to the database mode, and the database sequence characters are passed on to the systolic arrays.

The control component has a clock input, which is connected to the global clock signal, and a FIFO data input, which is connected to the input FIFO. The control component outputs are all connected to the first processing elements of the systolic arrays. They include: a reset flag, named "RST out", which is used to indicate the end of a sequence and to reset the first processing elements to zero, resulting in the reset of all the processing elements in the array; and database and query sequence characters, named "DB out" and "Q out" respectively, which are used to pass database and query sequence characters to the systolic arrays.

4.4.3 Result Generator

The results generator component monitors the output threshold flags of the last processing elements of the systolic arrays. When the last processing elements finish processing the last sequence character, the results generator generates a result data word that is output to the output FIFO, and resets itself. All bits between 0 and 31 of the result data word are set to 0 if the threshold flag never indicated a cell calculation exceeded the threshold for the forward query sequence array, and all bits between 32 and 63 of the result data word are set to 0 if the threshold flag never indicated a cell calculation exceeded the threshold for the reverse complement query sequence array.

The results generator component has a clock input, which is connected to the global clock signal, and a FIFO data output, which is connected to the output FIFO. The remaining results generator inputs are all connected to the last processing elements of the systolic arrays. They include: two threshold flags, named "F1 in" and "F2 in", which are used to monitor the output threshold flags of the last processing elements of the forward query sequence array and the reverse complement query sequence array, respectively; and two reset flags, named "RST1 in" and "RST2 in", which are used to determine the end of a sequence from the forward query sequence array and the reverse complement query sequence array, respectively, and used to determine when to generate the result data word, output the result data word, and to reset.

4.4.4 Complete Implementation

The control component outputs “RST out” and “DB out” are connected to the inputs “RST in” and “DB in”, respectively, of the first processing elements of the two systolic arrays. To generate the complement query sequence characters, the output “Q out” of the control component is connected to an inverter component. The inverter performs a bit-wise complement of a query character, where the bit-wise complement corresponds to the nucleotide complement. The output of the inverter is the complement sequence character, named “QC out”.

The outputs of each processing element *i*: “D out”, “B out”, “F out”, “RST out”, and “DB out”, are connected to the inputs of processing element *i+1*: “C in”, “A in”, “F in”, “RST in”, and “DB in”, respectfully. It is important to note that the output “Q out” of processing element *i* in the forward query sequence array is connected to the input “Q in” of processing element *i-1*, and that the output “Q out” of processing element *i* in the reverse complement query sequence array is connected to the input “Q in” of processing element *i+1*. Furthermore, the output “Q out” of the control component is connected to the input “Q in” of the last processing element of the forward query sequence array, and the output “QC out” of the inverter component, is connected to the input “Q in” of the first processing element of the reverse complement query array.

The reverse routing of the “Q in” and “Q out” signals of the forward query sequence array allow the query sequence to be passed into the array so that the first character of the query sequence ends up in the first processing element, or forward order. The forward routing of the “Q in” and “Q out” signals of the reverse complement query sequence array allow the complement query sequence to be passed into the array so that the first character of the complement query sequence ends up in the last processing element, or reverse order.

In addition, the inputs “C in”, “A in”, and “F in” of the first processing elements of the systolic arrays are connected to a 0 signal, to initialize the Smith-Waterman calculation. Finally, the outputs “F out” and “RST out” of the last processing elements of the two systolic arrays are connected to the inputs “F1 in”, “RST1 in”, “F2 in”, and “RST2 in”, respectively, of the results generator component. Figure 4.3 shows a schematic diagram of the entire hardware implementation, except for the clock signal, select signal, and some of the processing elements.

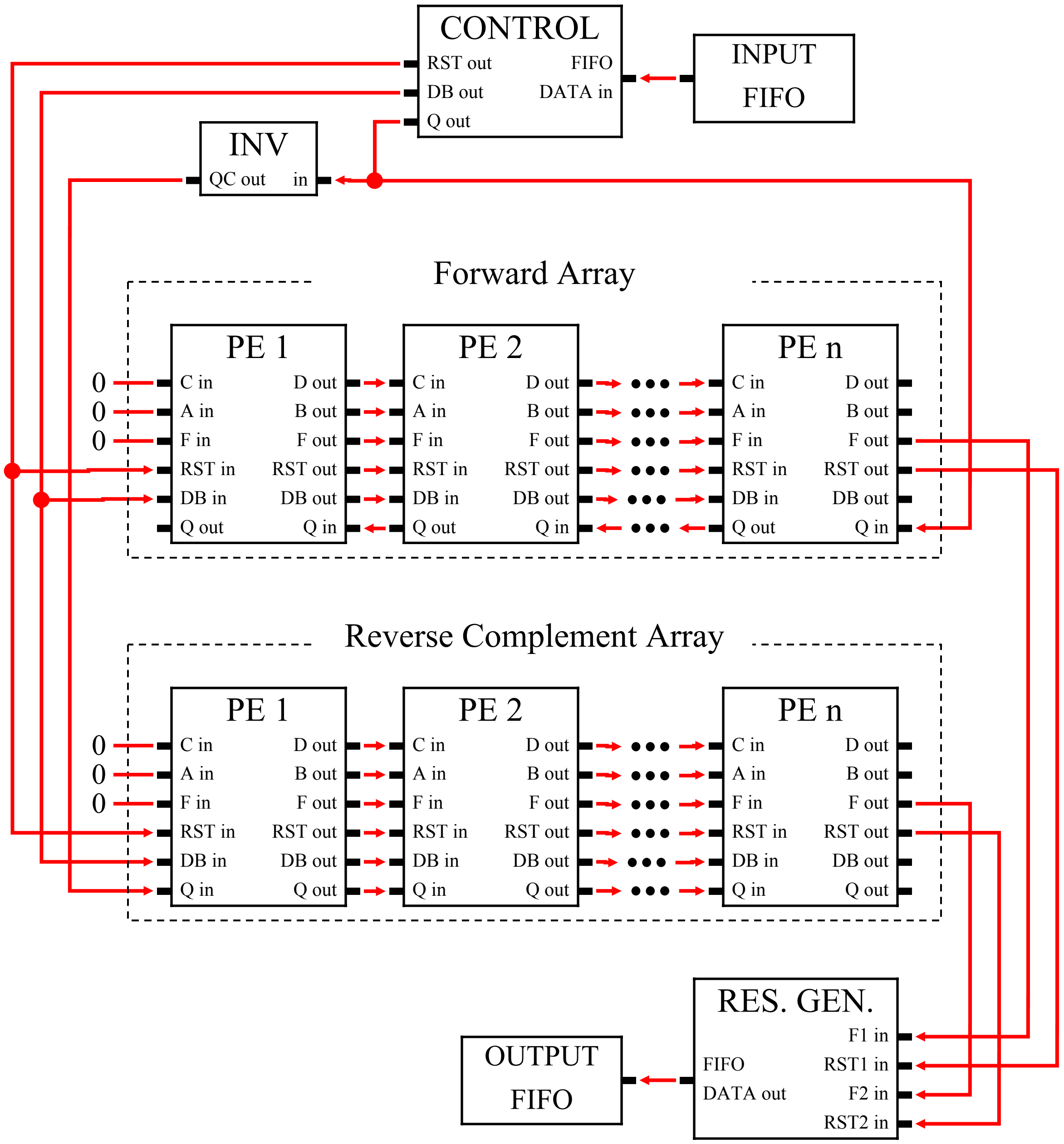

Figure 4.3. Initial Search Smith-Waterman Hardware Implementation Schematic.

4.5 Hardware-Software Interface

A component of the system software the hardware interface, which controls the operation of the hardware. The hardware interface component was implemented using the SLAAC API and multiple threads. Multiple threads were used for two reasons. First, the SLAAC API non-blocking FIFO calls did not work. Second, if the output FIFO becomes full, the hardware's clock stops to prevent a FIFO overflow. As a result, if a single threaded hardware interface component writes a large amount of data to the input FIFO using a blocking write-FIFO call, and a portion of that input data causes the output FIFO to become full, the clock is stopped. Since the clock is stopped, the hardware cannot continue processing the remaining data on the input FIFO; thus, the write-FIFO call never returns. The only way to restart the clock is to remove data from the output FIFO, but this is not possible because the software cannot return from the write-FIFO call. Thus, using a single thread and non-blocking write-FIFO calls can result in deadlock.

4.5.1 Initialization

The hardware interface component first initializes the hardware with the bitstream implementation file, and starts the clock. The hardware interface component then generates a data buffer to set the query sequence. The data buffer requires one data word for control, and a hardware query sequence of 50 data words, which corresponds to the number of processing elements in a systolic array. Bit 63 of the first data word in the buffer, i.e., the control word, is set to 1 to indicate a query sequence. The hardware query sequence data words are then filled with the query sequence data. If the query sequence data does not fill the entire data buffer, the remaining characters are set to the DNA character ‘G’. The reason the query sequence is padded with ‘G’ is because database sequences will probably be padded with the nucleotide character ‘A’ in an effort to fill complete database sequence data words. Thus, the possible subsequences of ‘A’s added to database sequences will not match to subsequences of ‘G’s in the hardware forward query sequence or subsequences of ‘C’s in the hardware reverse complement sequence.

The query data buffer is then written to the input FIFO to set the query sequence. Setting the query sequence also forces any initial junk data to be flushed from the systolic arrays. However, since the hardware query sequence is exactly the length of the systolic arrays, junk data can still exist at the outputs of the last processing elements of the systolic arrays. As a result, the query data buffer is written to the input FIFO a second time to assure that all the junk data has been flushed out of the systolic arrays. The output FIFO is then flushed of any junk output data. At this point, the hardware is in a known state.

Once the hardware is in a known state, database sequences can be written to the input FIFO. Since both query and database sequences processed by the systolic arrays result in a reset flag at the end of the sequence in the array, both generate a result data word. However, since the initial query sequence length is equal to the length of the systolic array, the resulting reset flag is stopped at the end of the array, and not yet processed by the results generator hardware component. Thus, the first database sequence written to the systolic array allows this query sequence reset flag to be processed, and a data word for the query sequence is generated. This data word must be read and discarded to achieve synchronization between the input sequences and the output data words.

4.5.2 Database Sequence Processing

Database sequences are written to the input FIFO using a sequence writer component that instantiates a separate thread to process the database sequences. The sequence writer component thread retrieves sequence objects from a database, and writes the database sequence data to the output FIFO until all the database sequences have been processed. The hardware interface component reads the resulting sequence comparison data from the output FIFO, and processes the sequences whose comparison score with the query sequence exceeded the threshold. The sequences being processed by the hardware are monitored using a processing queue component, which is shared by the sequence writer component and the hardware interface component. When the sequence writer component writes a database sequence to the input FIFO, the corresponding sequence object is added to the processing queue. When the hardware interface component receives a result data word from the output FIFO, the data word corresponds to the first sequence object in the processing queue.

The sequence writer component writes sequence data to the input FIFO using a non-blocking write FIFO component, which allows data to be written to the input FIFO in a separate thread. The non-blocking write FIFO component creates a thread on instantiation, which waits for data. When data is passed to the thread of the non-blocking write FIFO component, it writes the data using a blocking write FIFO call, and when done, continues waiting for data. The sequence writer component writes data to the input FIFO using two large data buffers, where one buffer is filled with sequence data, while the other is being written to the input FIFO by the non-blocking write FIFO component.

The sequence writer component retrieves each sequence from the database, and writes the sequence to the input FIFO. If the number of data words required to send the sequence is greater than the maximum value of the hardware database sequence control word count, the sequence is split into multiple smaller sequences, where the length of each smaller sequence is less than or equal to maximum hardware count. In addition, consecutive pairs of split sequences contain N common overlapped characters, where N is 2 times the threshold value. This overlap prevents a possible region of the database sequence, with significant similarity to the query sequence, from being split, and thus not flagged as a match by the hardware. For each sequence, or split portion of a sequence, its sequence object is added to the processing queue, and the sequence data is copied to one of the data buffers. If the sequence data does not fill a complete data word, the remaining characters of the data word are set to the nucleotide character 'A'. If the data buffer is full, it is sent using the non-blocking write FIFO component, and the next sequence data is added to the other data buffer. Finally, if the sequence includes unknown characters such as "N", a copy of the sequence is made in which the unknown characters are discarded.

After all of the database sequences have been written to the input FIFO, a dummy sequence must be written to the input FIFO because the reset flag of the last sequence in the systolic array cannot advance to the end of the systolic array without additional sequence data pushing it through the array. The number of dummy sequence characters sent must be more than the number of processing elements in the systolic array so that the last sequence's reset flag is pushed beyond the end of the systolic arrays, resulting in the generation of the sequence's result data word.

While the sequence writer component is writing sequence data to the input FIFO, the hardware interface component reads the result data words from the output FIFO. The hardware interface component firsts determines the number of expected data words from the number of sequence objects in the processing queue. The number of expected data words are then read from the output FIFO into a data buffer. For each data word and corresponding sequence object, if any of the bits 0 to 31 are set to one the sequence object is added to a list for sequences that exceed the threshold when compared with the forward query sequence; and if any of the bits 32 to 63 are set to one the sequence object is added to a list for sequences that exceed the threshold when compared with the reverse complement query sequence.

Since the sequence writer component adds a sequence object to the processing queue before the sequence object's data is written to the input FIFO, synchronization errors can be detected. If the processing queue is empty, but the output FIFO is not empty, then the data on the output FIFO has no corresponding sequence object; thus, a synchronization error must have occurred.

4.6 Database Format Software

To minimize the time to read a database from disk, each database is reformatted to use the minimal number of bits per character, and is divided into segments with the segments distributed among all of the slave machines, where each slave machine will read all of the sequence data in each segment into memory before processing a query. In addition, some FPGA board designs work more efficiently when sequences of similar lengths are grouped together; thus, the sequences can be sorted by the number of characters in the sequence. An application, written to perform these operations, reads in a sequence from the database and determines which alphabet to use to encode the sequence. It is important to note that these databases contain additional characters used to represent uncertain or unidentified characters. The possible alphabets include: DNA bases A, C, T, and G, which requires two bits per character and is applicable to most DNA sequences; DNA bases and N, where N represents an unknown character, which requires three bits per character; and DNA bases, N, and 10 other character that are used for specifying uncertain bases, which requires four bits per character. The sequence is then converted in to a binary sequence using the appropriate alphabet. For sorting, a group of sequences is read into memory using only a user-defined maximum of memory, sorted, then output to a temporary file. This process is repeated until all sequences have been sorted into temporary files. Theses temporary files are then merged into a temporary file. This final temporary file is then split into the multiple files required by the multiple machines.

The reformatting process takes a significant amount of time, approximately 25 minutes, for the NT database on a typical PC. As a result, the format application is only run when a database has changed, and before the HokieGene system is executed.

4.7 Search System Control Software

The HokieGene search system consists of many components for communications, configuration, user interface, and processing. All of these components were implemented using object oriented programming techniques. There are components used by all machines such as communications, and others used only by the master node such as the user interface. A majority of the system is written in C++; however, some components required the use of Java. The Java components are implemented as separate executables, but communicate directly with the main system using network socket communications. Figure 4.4 shows the component relationship of the system. Table 4.1 lists the name of the software components and their corresponding class names and source files.

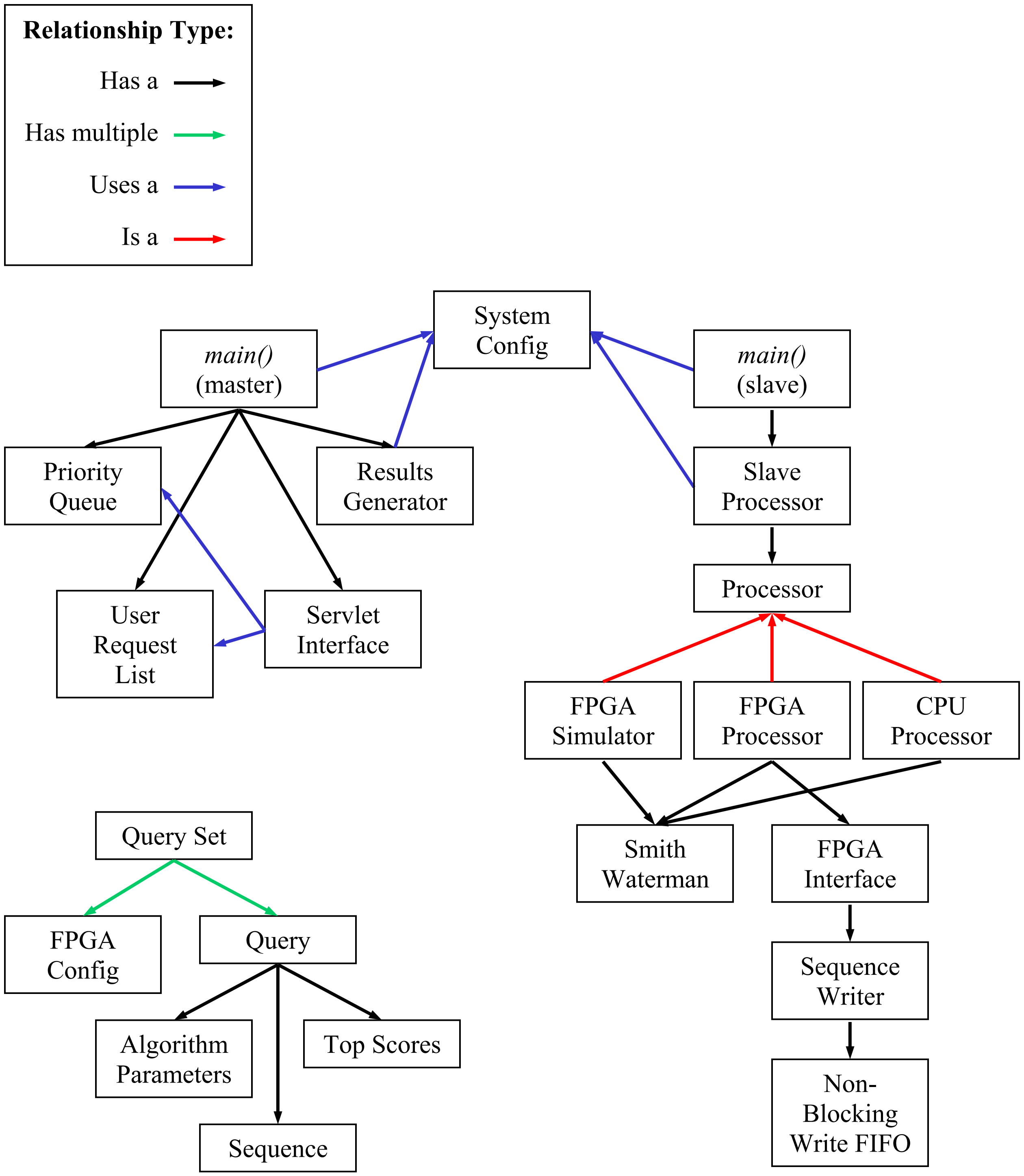

Figure 4.4. Search System Component Relationship Diagram.

4.7.1 Communication / Data

Since there is a great deal of communication between the master node, slave nodes, and Java components, a common communication system was developed. First, a data buffer class was developed, modeled after the Java DataBuffer class. It allows an application to read and write a variety of data types, such as integers, doubles, and character strings, to a buffer or socket connection. Second, a common serialization interface was developed, modeled after the Java Serializable interface, to serialize an object into a data buffer object for simplified transfer of objects using a socket connection or MPI.

All of the components of the system operate on several units of data. A sequence object stores sequence data, in either ASCII or binary format, and stores sequence information such the sequence length and ID. A top scores object stores a list of the top scores generated by a database search. An algorithm parameter object stores the parameters used by a sequence comparison algorithm. A query object stores information regarding a user submitted query. It consists of a sequence object for the query sequence, an algorithm parameter object to specify the search algorithm, a top scores object for the results, and the ID of the user that submitted the query. An FPGA configuration object contains hardware implementation file information, and hardware query processing information. The major unit of data transferred between components is a query set object, which contains a list of query objects that will be used to search a single database, a list of FPGA configuration objects, and the name of the database.

Since the initial-search hardware has a finite number of processing elements, queries longer than the systolic array must be split into smaller portions, with each portion used as an individual query sequence during the initial search. For each long query sequence portion, a query set object contains a single FPGA configuration object. The FPGA configuration contains the starting point and length of the query sequence portion. Consecutive pairs of query sequence portions are set to have *N* common overlapped characters, where *N* is two times the threshold value of the hardware implementation. This overlap prevents a possible region of the query sequence, with significant similarity to a database sequence, from being split, and thus not flagged as a match by the hardware. The FPGA interface component uses an individual FPGA configuration object to determine the query sequence data to use in a search.

4.7.2 Configuration

To facilitate configuration management a system configuration object is used. A configuration file, with all necessary information about a system, is read and distributed to all the machines and components. The configuration file contains information on the master and all slave machines. For each machine, a local directory and the type of hardware in the machine, such as an Osiris board or only a processor, is specified. In addition, the configuration file specifies the number of machines in the system, the location of a common network directory that all machines have access to, the directory where result files should be placed, and the results web page link to be sent back to the user. The configuration file also specifies a name for the given configuration, so that files used and generated by a given configuration will not interfere with other configurations. Figure 4.5 shows a sample configuration file.

```
ConfigName:        # The keyword "ConfigName" is used to define the name of the
example            # configuration to identify the configuration and its associated files.
                   # Here the configuration name is defined as "example".

NetworkDir:        # The keyword "NetworkDir" is used to define the location of a
/shared/           # network directory accessible by all machines in the system.
                   # Here the network directory location is defined as "/shared/".

ResultsDir:        # The keyword "ResultsDir" is used to define the location where
/html/results/     # the output result files are to be written.  Here the results directory
                   # location is defined as "/html/results/".

ResultsLink:                    # The keyword "ResultsLink" is used to define
www.hokiegene.edu/results/      # the URL associated with the results directory.
                                # Here the results URL is define as
                                # "www.hokiegene.edu/results".

# All of the following machine definitions require three fields, machine name, local file
directory location, and the processor type defined as "c" for CPU processor, "o" for Osiris
processor, and "s" for simulated Osiris processor.

Master:                 # The keyword "Master" is used to define the master node of
master  /local/  c      # the system.  Here the master machine name is "master",
                        # its local directory is "/local", and it uses the CPU for
                        # processing.

NumSlaves: 3            # The keyword "NumSlaves" is used to define the number of
                        # slave machines used in the system.

Slaves:                 # The keyword "Slaves" is used to define a list of all of the
                        # slave machines used in the system.

slv1  /local/  o        # Here slave machine names are "slv1", "slv2", "slv3",
slv2  /local/  c        # their local directories are "/local/", and each uses an Osiris
slv3  /local/  s        # processor, a CPU processor, and an Osiris simulator
                        # processor, respectively.
```

Figure 4.5. Sample System Configuration File.

4.7.3 Master Node

The master node software consists of several components: a Servlet interface object to receive query data from the user interface; a user request list object to store query information submitted by each user; a priority queue object to schedule queries for processing; a results object to generate results for a user, and a control component. These components operate on several units of data: a user request object, which contains a list of query objects that represent queries submitted by a single user; a query object; and a query set object.

The Servlet interface object receives query data from the user interface by running a separate thread, where a socket connection with the Java Servlet executed by the web site is established. This connection and user are given an ID and stored for later use. The Servlet then transfers the information to the Servlet interface. The interface generates query objects based on the information specified by the user and adds the queries to the user request list and priority queue components. When results for the user's query are generated, the link and user ID is passed to the interface, and the link is sent to the to the appropriate Java Servlet.

The user request list object keeps track of which queries specific users have submitted by maintaining a list of user request objects. This is necessary because batch-processing queries may process a user's queries at different times. When a query object has been processed, it is flagged as done. Each time a batch is processed, the user request list is checked to see if any user requests contain only finished queries, and if so, the results for that user are generated.

Queries that are to search a common database are batched together into query set objects by the priority queue component. A query set is provided to the main control component when needed. In the current implementation, the batch that is processed next is determined by whichever query was submitted first. However, this will be updated to take into account the length of time a user has been waiting for results, the total number of pending queries, and the number of pending queries in each batch in an effort to minimize individual user response time, while minimizing the number of times a database must be read into memory and thus maximizing the total system throughput.

The results component generates the web pages for a given user request. It first generates a web page containing a list of each query and a link to the individual query results. Then a web page for each query is generated, listing the database sequences that scored the highest against the query. Eventually, each database sequence in this page will contain a link to display the optimal alignment of the query and database sequences. A tool distributed by NCBI called blast2seq [27], which takes two sequences and generates blast output, will generate this final alignment information so that users who are familiar with blast can easily understand the information.

The master control component initializes all components and controls the operation of the system. It first initializes the system configuration object with a command line argument specifying the configurations file name. The location of this configuration file is then sent to all nodes in the system. It then initializes the Servlet interface, priority queue, user request list, and results objects. When data is available, the master control gets a query set from the priority queue. It serializes and broadcasts the query set to all of the slave nodes. After the slave nodes process the query set, they send back the query set with the results. The master receives the query set from each slave, and resolves all of the results. All of the queries in the set are marked as finished, and the user request list is checked to see if any user requests are finished. If a user request is finished, it is given to the results component to generate the resulting web pages. The link to the pages and the user ID are then given to the Servlet interface to be sent back to the user's browser. The master control then waits for available data.

4.7.4 Slave Nodes

The slave node software consists of several components: sequence container objects, which read and store information in a database; processing objects, that handle how queries are processed; and a slave control component.

Processing objects perform the actual database search. Each of these takes a query set, and the sequence container that contains the database required by the query set, and processes each query in the query set. There are three different types of processing components: a CPU processor, an FPGA processor, and an FPGA simulator. Each of these processing components use a Smith-Waterman component that implements Smith-Waterman algorithm with the linear space optimization and the affine gap penalty optimization described in Section 3.3. A CPU processor component uses only the processor of the slave machine to search the database using the Smith-Waterman component. An FPGA processor component uses an FPGA interface component to interface with an Osiris board to perform an initial search of the database, using each FPGA configuration object in the query set object. The flagged sequences are then searched using the Smith-Waterman component. An FPGA simulator component uses only the processor of the slave machine to perform and initial search of the database using a simplified implementation of the Smith-Waterman algorithm that is similar to the algorithm implemented in hardware. The sequences that scored above the same threshold as the hardware threshold are then searched using the Smith-Waterman component.

The slave control component initially receives the location of the configuration file, and initializes the system configuration object. From the configuration, the control component determines what type of processor component should be implemented. The slave control component then waits until it receives a query set object from the master node. It then checks if the database required for the query set is in memory and, if not, uses a sequence container object to read the database into memory. The database that is read is the assigned portion of the formatted database located on the slave machine's local drive. Finally it processes the query set, and sends the resulting set back to the master node.

Component	Class/Function Name	File
Master:		
Master Control	N/A	main.cpp
Servlet Interface	servletInterface	servletInterface.h, servletInterface.cpp
Priority Queue	priorityQueue	priorityQueue.h, priorityQueue.cpp
User Request List	UserReqList	UserReqList.h, UserReqList.cpp
Web Page Results	N/A	Webpage.h, Webpage.cpp
Slave:		
Slave Control	N/A	main.cpp
Slave Processor	SlaveProcessor	SlaveProcessor.h, SlaveProcessor.cpp
Sequence Container	SeqContiner, SeqList, FileSeqDB, BinSeqDB, FastaSeqDB, GbSeqDB	SeqContainer.h, SeqContainer.cpp
Processor	Processor	Processor.h, Processor.cpp
CPU Processor	CPUProcessor	Processor.h, Processor.cpp
FPGA Processor	OsirisFpgaProcessor	OsirisFpgaProcessor.h, OsirisFpgaProcessor.cpp
FPGA Simulator	CpuFpgaSimProc	Processor.h, Processor.cpp
Smith-Waterman	SWAlgorithm	SWAlgorithm.h, SWAlgorithm.cpp
Data / Common:		
Query Set	QuerySet	QuerySet.h, QuerySet.cpp
Query	Query	Query.h, Query.cpp
Sequence	Sequence, SequenceBin, SequenceAscii	Sequence.h, Sequence.cpp
FPGA Configuration	Bitstream	Bitstream.h, Bitstream.cpp
Top Scores	TopScores	TopScores.h, TopScores.cpp
Algorithm Parameters	AlgoParms	AlgoParms.h, AlgoParms.cpp
Data Buffer	DataStream, DataInputStream, DataOutputStream	DataStream.h, DataStream.cpp
Serializable	Serializable	Serializable.h
System Configuration	SystemConfig	SystemConfig.h, SystemConfig.cpp

Table 4.1. List of Software Components and Corresponding Class Names and Files.

4.8 Overall Operation

The system is set up by first formatting the databases that will be used by the system. The resulting database portions are then distributed to the local drives of the slave machines. Next, the web server that implements the user interface is executed. Finally the system control software is executed.

On startup, the system control software reads the configuration file, and copies it to the network directory defined in the configuration files. It then sends the location of the copied configuration file to all of the slave nodes, and waits for a signal from the slave nodes. The slave nodes receive the configuration file location, read it, and signal the master.

When a user submits a query using the web interface, the query information is sent to the Servlet interface. For each of the user submitted queries, the Servlet interface thread generates a query object that contains the query information. The query objects are then passed to the user request list and priority queue components. The master control thread gets a query set object from the priority queue, and transfers it to all of the slave nodes. The slave nodes receive the query set and passes it to the slave processor component. The slave processor component then loads the database into memory. The database and query set are then passed on to a processor component. If the processor component is either a CPU or a FPGA simulator component, each query in the query set is processed sequentially. If the processor component is an FPGA, then each FPGA configuration object is processed sequentially. After the query set is processed, it is sent back to the master. The master collects all of the returned query objects from the slaves, and merges the results. The queries in the query set are marked as complete, and the user request list is checked for any user requests that are complete. Any completed user request objects are sent to the web page results component, which generates the results web pages. The URL of the results pages and the user id are passed to the Servlet interface, which sends the results link the appropriate user's web browser.

Chapter 5

Analysis

For the results obtained in the laboratory, only one Osiris Board was used due to their limited availability. In this experiment, three different platforms were used. A Dell PC with a 64 bit, 66 MHz PCI bus and a 1 GHz processor, which contained the Osiris board and was used as a slave node. A Dell PowerEdge 2650, with 2 Intel 1.4 GHz processors and 100 GB of local drive space was used as the master node. Eight Dell PowerEdge 1650 computers were used as slave nodes.

The convention for expressing the speed of the Smith-Waterman algorithm is the number of elements of the dynamic programming matrix calculated, or cell updates, per unit time. For simplicity, cell updates per second will be expressed by CU/s.

The DNA sequence database NT, or a portion thereof, was used extensively in this experiment. This database was chosen because it is one of the largest single DNA databases. As stated earlier, the NT database contains almost 1.4 million sequences and 6.4 billion characters. The sequences range in length from 6 characters to greater than 1.4 million. Figure 5.1 shows the distribution of sequence lengths for the NT database and Table 5.1 lists the cumulative distribution of sequence lengths of the NT database.

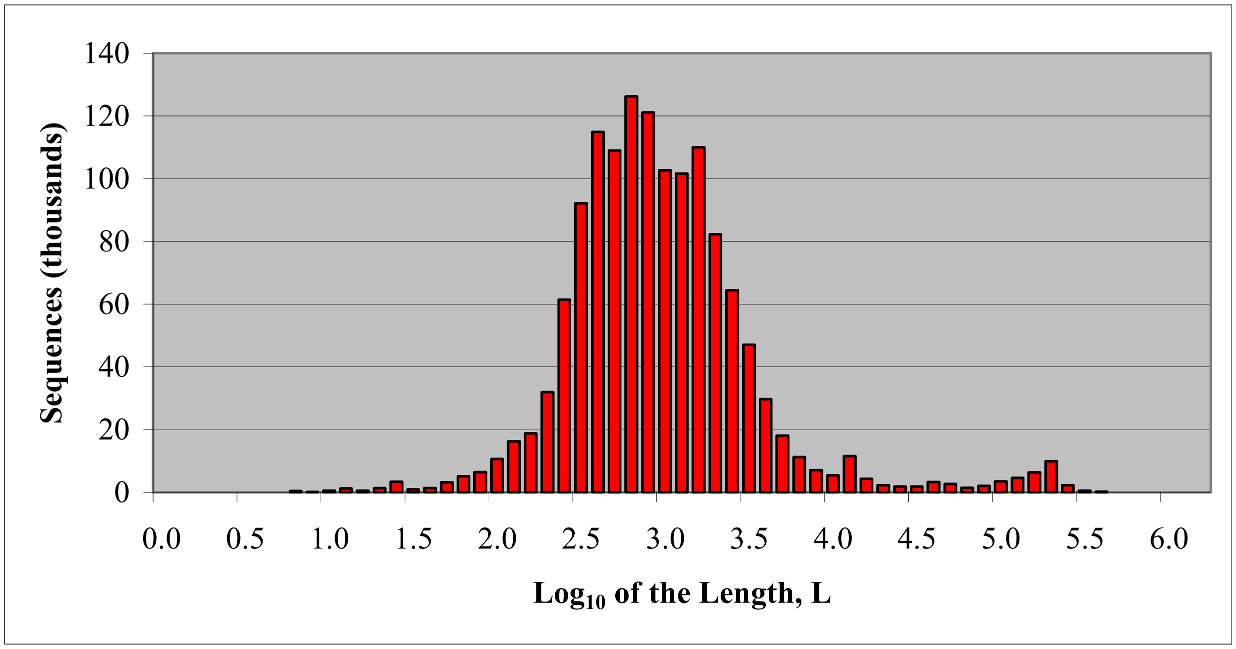

Figure 5.1. Sequence Length Distribution of the NT DNA Database.

Percentage	**Maximum Length**
60	1,000
80	2,000
90	3,500
95	7,000
98	80,000
99	156,500
99.9	221,500
99.99	340,000
99.999	429,000

Table 5.1. NT DNA Database Sequences Length Cumulative Distribution.

The timing of operations were accomplished using the Unix function *times()*, which is included in the file "sys/times.h". The timing resolution of this function is approximately 10ms; however, some operations required a timing analysis with a resolution shorter than 10ms. In these instances, the operation was timed repeatedly, after which the average was taken to improve the timing resolution.

5.1 Database Management

By reformatting the database to use the minimum number of bits for each character in a sequence, and a single 4-byte ID to represent the sequence, a significant reduction in database size was achieved. The following shows the results of reformatting the database NT, which is one of the largest individual databases. The NT database contains nucleotide sequences; thus, three possible alphabets were used to represent each sequence. Table 5.2 lists the reduction statistics for NT, and shows that a large majority of the sequences use only the characters A C, T, and G, and thus can be reformatted using only two bits per character. The size of NT before reformatting is 6.20 GB, and after reformatting is 1.56 GB; thus, achieving approximately a 4-to-1 reduction in size. This reduction in size is also directly proportional to the reduction in time required for the database to be read from the disk.

Alphabet	Sequences	Characters	% Total Sequences	% Total Characters	Encoded Size	% Size
ACTG	1,195,947	6,049,683,058	88	94	1.42 GB	91
ACTGN	117,633	278,639,387	8.6	4.3	101 MB	6.3
ACTGN + 10 Unknowns	51,148	97,728,421	3.7	1.5	47.1 MB	2.9
Total	1,364,728	6,426,050,866	100	100	1.56 GB	100

Table 5.2. Sequence database NT Statistics Based on Alphabet.

By distributing portions of the reformatted database to each of the slave node's local drives, each node has to read a smaller amount of data into memory. This reduces the time to read the data, but presents a balancing problem. It is possible that different slave nodes could have different capabilities, such as a node with a FPGA board and a node without any hardware acceleration. In this situation, the node with the greater capability receives a larger portion of the database; thus, it must spend more time transferring the database into memory. The amount of time to process a set of queries is roughly proportional to the number of queries in the set;

however, the time to read the database is not. As a result, nodes with greater capabilities may take longer to both read the database portion and process small query sets than nodes with lesser capabilities. Solutions to this balancing problem are dependent on the number of queries submitted and number of possible search databases.

On average, data was read from the disk at approximately 30 MB/s, for disks with a theoretical limit of 40 MB/s. At that rate, the entire reformatted database can be read from one of these disks in approximately 50 to 60 seconds.

5.2 SLAAC Performance

The utility of the SLAAC API was somewhat disappointing because certain aspects of the software did not work as documented. Specifically, there were problems with the non-blocking read and write-FIFO methods, and the methods used to determine if a FIFO is full. It was not determined why the non-blocking read and write FIFO methods did not work.

With regard to the methods used to determine if a FIFO is full, it is suspected that the API incorrectly determines the amount of data in the FIFO. According to the documentation, the input FIFO is 256 words deep; thus, it should be possible to write 256 individual words to the input FIFO without a failure. However, when the Osiris board clock was stopped, and data was written to the input FIFO, the input FIFO failed before 256 words were written while the API reported that the input FIFO was not full. In one test, single words were written, to the input FIFO using individual FIFO-write calls, and the input FIFO failed after the 128th word. In another test, two word pairs were written to the input FIFO using individual FIFO-write calls, and the input FIFO failed after the 171st word. The results of all of the tests seemed to indicate that when an individual FIFO-write call is made, the number of words counted by the API is one greater than the actual number of words.

Despite the problems encountered, the performance of the SLAAC API and the Osiris board proved to meet the needs of the system. Figure 5.2 shows the performance of the input

FIFO when used in the system. The graph shows the total time to complete an individual FIFO-write call versus the amount of data written to the input FIFO. Interpolating the data using a linear least squares fit results in Equations 5.2.1 where, t is the time to complete the FIFO-write call in seconds, and s is the total amount of data sent in bytes. The least squares fit R^2 value was 0.9999955 indicating that the data closely fits Equations 5.2.1. As the size of the buffer approaches zero, the time to make the FIFO-write call approaches the constant term of Equation 5.1a, which is equal to 242 microseconds. This is approximately the initial setup time of the FIFO-write call. The linear term of Equation 5.1b, which is 2.41 MB/s, is equivalent to the actual transfer rate of data to the board.

Since the total FIFO-write call time is not proportional to the amount of data being sent, it is more efficient to send larger amounts of data with each call. Equation 5.2 is derived from Equations 5.1, and gives the data transfer rate taking into account the FIFO-write call setup time, or the good-put, where g is the good-put in bytes per second. Figure 5.3 shows a plot of Equation 5.2 for smaller amounts of data than were used to establish the equation. The plot shows that the good-put decreases at an increasing rate as the size of the buffer decreases.

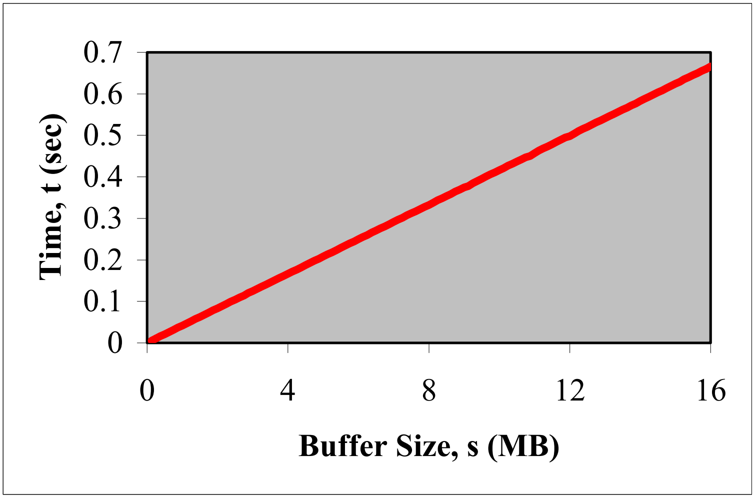

Figure 5.2. FIFO-Write Call Time vs. Amount of Data Sent Per Call.

$$\begin{aligned} t &= 3.97\times10^{-8}s + 2.42\times10^{-4} && \text{(a)} \\ s &= 2.52\times10^{7}t + 6.07\times10^{3} && \text{(b)} \end{aligned} \tag{5.1}$$

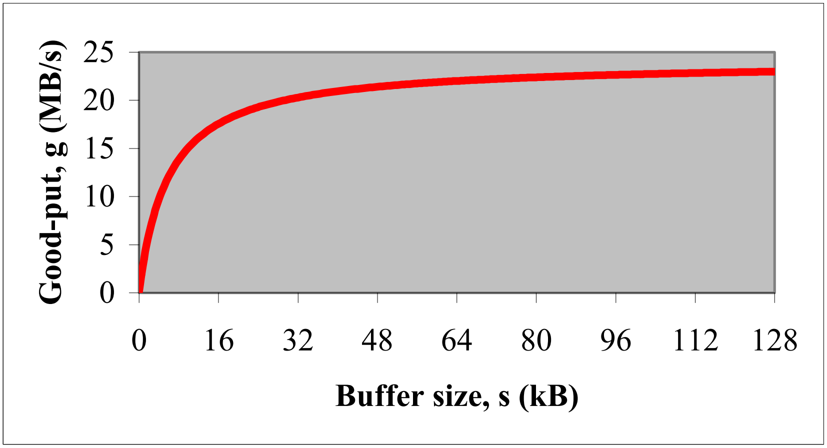

Figure 5.3. Extrapolated FIFO-Write Call Good-put vs. Amount of Data Sent Per Call.

$$g = \frac{1.00\,s + 37.6}{3.97\times10^{-8}s - 2.42\times10^{-4}} \tag{5.2}$$

5.3 Hardware and Hardware-Software Interface Performance

The maximum hardware clock speed achieved for the FPGA implementation was 120 MHz, and the maximum number of processing elements implemented was 3,200. To compute the theoretical speed and bandwidth of the hardware, it can be assumed that the processing time of the initial count data word and the bandwidth used by the result word are negligible. Database sequences are, on average, long enough to support this assumption and, it is further supported by the fact that the initial count data word and the result word are processed only once per sequence. For example, the NT DNA database contains approximately 30,000 sequences with a length greater than 65,000, accounting for about two-thirds of the entire database. The above assumptions also imply that all processing elements compute a single cell value each clock cycle; thus, the theoretical hardware calculation speed is:

$$\frac{3{,}200 \text{ PEs} \times 1 \text{ cell}/\text{PE}}{1 \text{ cycle}/\text{update}} \times 120 \text{ MHz} = 384 \text{ Billion CU}/\text{s}.$$

It can also be assumed that each character in a data word is processed in a single clock cycle, where a data word is 8 bytes and contains 32 characters. Thus, the theoretical bandwidth is:

$$\frac{8 \text{ bytes}}{32 \text{ cycles} \;/\; 120 \text{ MHz}} = 28.6 \text{ MB}/\text{s}.$$

Test searches were conducted using a portion of the NT DNA database that contained 2,962 relatively long sequences. The entire database portion, or 689,348,402 total characters, was sent to the board in approximately 6.40 seconds. Assuming query sequence lengths sum to 1,600, the hardware processing speed was 345 billion cu/s, which is approximately 90% of the theoretical value. During the test 21,495,808 data words were sent, and 15,449 were received, resulting in an overall throughput of 25.6 MB/s.

5.4 Initial-Search Algorithm Analysis

The overall speed of the system is dependant on the length of the query; the size of the database; the speed of the initial search; the speed of the final search; and the percentage of database sequences flagged as exceeding the threshold in the initial search that are searched in the final search. The percentage of database sequences flagged as exceeding the threshold in the initial search is dependant on the length of the query sequence and the average length of the database sequences.

Equation 5.3 [38] gives an approximation of the probability that a comparison of sequences *D* and *Q*, will result in a score *S(D,Q)* that exceeds a threshold *t*,

$$P\{S(D,Q) > t\} \approx 1 - e^{-ymnp^{t}} \tag{5.3}$$

where,

D is a database sequence,

Q is a query sequence,

S(D,Q) is the score of the comparison of sequences *D* and *Q*,

m is the length of the query sequence,

n is the length of the database sequence, and

y and *p* are constants that are determined by the algorithm used for the comparison.

Equation 5.4 gives a possible approximation of the percentage of database sequences flagged, *f*,

$$f \approx \frac{\sum_{i=1}^{N} P\{S(D_i,Q) > t\}}{N} = \frac{\sum_{i=1}^{N} 1 - e^{-ymn_i p^{t}}}{N} \approx 1 - e^{-kmn_{ave}} \tag{5.4}$$

where

D_i is a single database sequences with length n_i,

n_i is the length of a sequence i in the database.

n_{ave} is the average database sequence length, and

k is a constant for a given algorithm and given threshold.

This is a reasonable approximation for two reasons. First, Equation 5.3 is equivalent to the percentage of sequence comparisons exceeding the threshold t, given a query of length m, and a sequence database whose lengths are all n. Furthermore, the values y, p, and t, which are constant for a given algorithm and threshold in Equation 5.3, are represented by k in Equation 5.4; and the database sequences length n in Equation 5.3, is represented by the average database sequence length, n_{ave} in Equation 5.4. Second, Equation 5.4 makes intuitive sense in that as the average length of the database sequences or the length of the query sequence increases, the probability that the comparison score of two sequences will exceed the threshold increases; thus the percentage of database sequences flagged increases asymptotically to 1.

The value of k in Equation 5.4, was determined using the percentage flagged data generated from multiple database searches using multiple portions of the NT DNA database ranging from 12 sequences to 1,061 sequences, with average sequence lengths ranging from 1,389 to 125,505, and multiple random query sequences of lengths ranging from 256 to 12,032. Percentage flagged data was gathered separately for comparison of forward database sequences and reverse complement database sequences, and both sets of data should produce approximately the same values of k. It is important to note that the percentage of database sequences flagged for either the forward database sequences or reverse complement database sequences results in a different value of k because a search of both forward and reverse complement database sequences is essentially a different algorithm from searching either the forward of reverse complement individually. The forward database sequence search results are shown in Figure 5.4 as the percentage flagged versus the product of the average database sequence length and the query sequence length. Similar results for the reverse complement database sequence are shown in Figure 5.5. A linear regression using Equation 5.5 was used to determine the value of k.

$$\ln(1 - f) = -kmn_{ave} \tag{5.5}$$

The value of *k* for the forward database sequence search was 1.133×10^{-9}, and for the reverse complement database sequence search, 1.151×10^{-9}. The values of *k* for the two searches should be approximately equal because of the use of the same algorithm and threshold. Using both data sets the value of *k* is 1.42×10^{-9}. Due to the relatively small size of the database used to generate the percent flagged data, the percent flagged values vary on average, approximately 80% from the fitted curve.

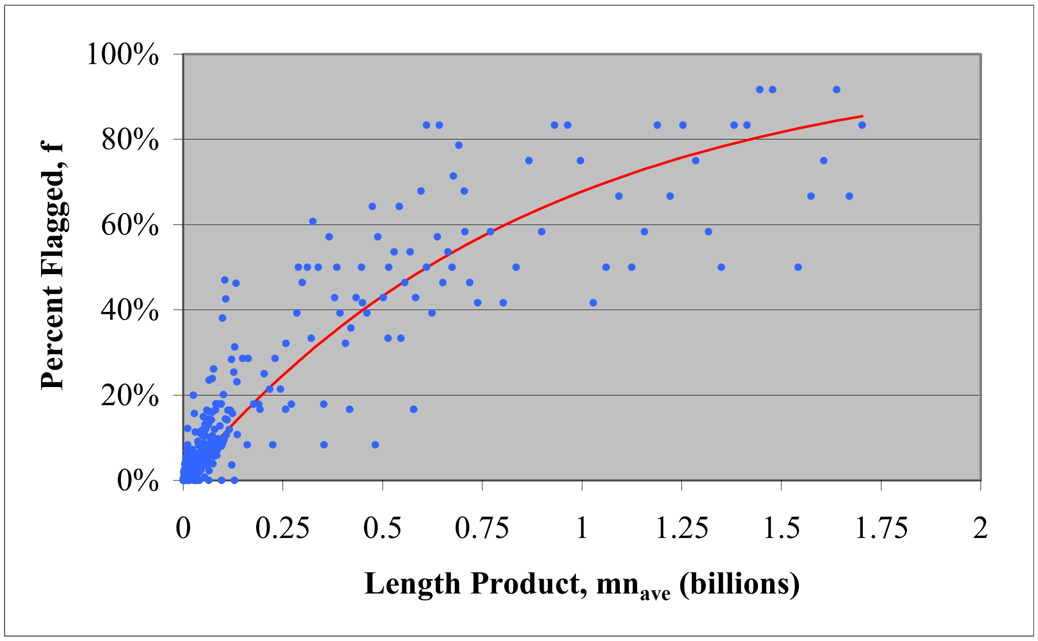

Figure 5.4. Percentage of Forward Database Sequences Flagged vs. the Product of the Query Sequence Length and the Average Database Sequence Length.

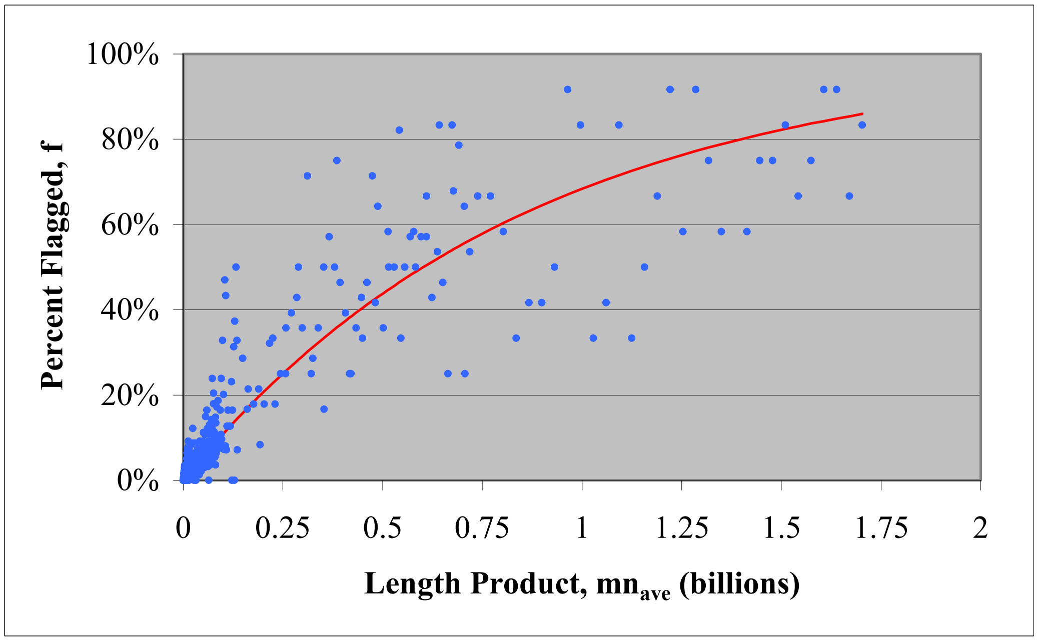

Figure 5.5. Percentage of Reverse-Complement Database Sequences Flagged vs. the Product of the Query Sequence Length and the Average Database Sequence Length.

Although the use of the initial search algorithm results in increased total search speed, to produce useful results, the algorithm should be configured such that the sequences that are flagged include all of the sequences that would be found to have significant similarity by a complete Smith Waterman algorithm search. The percentage of these sequences is referred to as the sensitivity of the initial search algorithm. In addition, to further increase the efficiency of the initial search algorithm, it should be configured to minimize the number of sequences flagged that would be found to have insignificant similarity by a complete Smith Waterman algorithm. The percentage of these sequences is referred to as the selectivity of the initial search.

To determine the sensitivity of the initial search algorithm, a database was searched using both a complete Smith-Waterman algorithm implementation and the initial-search algorithm. The searches used multiple databases and multiple queries, each of which varied in size. For each search, the percentage of the most similar sequences according to the complete Smith-Waterman algorithm flagged by the initial-search algorithm, or the percentage correct, was determined. The resulting data was sporadic and no significant pattern could be ascertained, except that as the percentage flagged increased, the percentage correct tended to increase. The percent correct varied from 0%, generally for short query sequences, to 100% for longer query sequences. On average, the percentage correct exceeded 50%. In addition, the percentage of least similar sequences flagged by the hardware was examined. In all the searches of the 25% least similar sequences, none were flagged by the initial search algorithm.

In order to increase the percentage correctness, future initial-search hardware implementations could incorporate lower thresholds for shorter query sequences, or different scoring parameters based on the scoring parameters submitted by the user.

5.5 System Analysis

The following analysis assumes a system using only one slave node with a single Osiris board. The search time of the system, t_t is the sum of the initial and final search time. The initial search time is a function of the query length and the size of the database, and approximated by Equation 5.6,

$$t_i = t_{is} + n_{split} \times (t_{hc} \times N_{dbc}) \qquad 5.6$$

where,

t_{is} is the initial-search startup time,

n_{split} is the number a times a query sequence must be split due to the limited number of processing elements,

t_{hc} is the initial-search hardware database character processing time, and

N_{dbc} is the total number of database characters.

The final search time is a function of the query length, the size of the database, and the percentage of the database flagged as exceeding the threshold by the initial search. An approximation of the final search time, t_f, is given by equation 5.7,

$$t_f = t_{fs} + \frac{2\, f\, N_{dbs}\, n_{ave}\, m}{s_f} \tag{5.7}$$

where,

t_{fs} is the final-search startup time,

f is the percent flagged,

N_{dbs} is the number of database sequences,

n_{ave} is the average database sequence length,

m is the query length, and

s_f is the speed of the final search in cell updates per second.

The factor of 2 in Equation 5.7 is required because the both the flagged forward database sequences and the flagged reverse complement query sequences are searched.

The total system speed is highly dependent on the length of the query sequence; thus, it is not possible to estimate a single value of the system speed. Figure 5.6 shows the estimated system speed as a function of the query sequence length. To compute an average system speed, assume several queries with lengths representative of queries submitted to search the NT database using the NCBI BLAST server [27]. These query lengths have an average length of approximately 400, and a standard deviation of approximately 700 [18]. The following calculation determines the average speed of the system over the sample query lengths. In addition, assume the NT DNA database is searched, which has an average sequence length of 4,709; the final search is performed by the CPU, which has a speed of approximately 50 million cu/s, and the initial-search hardware processes one sequence character every 9 nanoseconds. The average initial search time is approximately 67 seconds; the average percentage of database sequences flagged in the initial search is approximately 0.3%; the average final search time is approximately 1,200 seconds; and the average total search time is 1,250 seconds. The overall system speed, in cell updates per second, is the product of the total number of database characters and the query sequence length, divided by the total time. The average system speed is approximately 17 billion cu/s. This is approximately 350 times the speed of the processor alone.

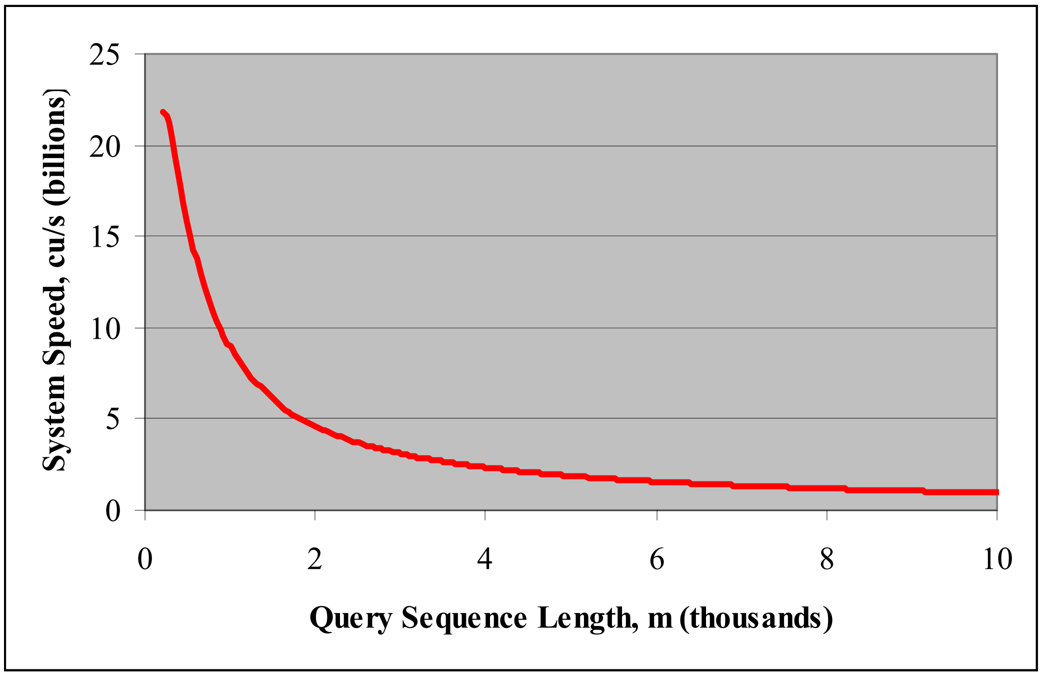

Figure 5.6. Estimated Total System Speed vs. Query Sequence Length.

It has been estimated that the final search implemented in hardware could run at approximately 1 billion cu/s. For the example described above, if it is assumed that the final search is implemented in hardware, the final search time would be approximately 60 seconds on average; the total search time would be approximately 125 seconds on average; and the total system speed would be approximately 50 billion cu/s on average.

5.6 System Comparison

Sequence database search systems can be executed in parallel with a near linear increase in speed for each additional processing node. As a result, comparing two sequence database search systems by only their overall speed is not useful because a system's performance can be increased linearly by adding more processing nodes. A useful comparison takes the total cost of the system into account. The cost of a system, in general, can be approximated by the cost of the system control component plus the additional cost of each node. However, the cost of the system control component becomes a small fraction of the total cost as the number of nodes of the system is increased; thus the total cost of the system can be approximated by the cost of the nodes for large-scale systems. Table 5.3 lists a system's node speed in cell updates per second, node cost, and the speed of the node per dollar for several systems. The list of systems include the Osiris initial-search implementation with no final-search algorithm implementation only to demonstrate the speed of the initial-search implementation as compared to other systems; and includes a proposed final-search hardware implementation. The data in the table shows that the use of the initial-search hardware implementation provides a significant speed-cost advantage of 3 to 10 times over other hardware-accelerated systems, and 10 to 30 times using both an initial-search and final-search hardware implementations.

System	**Speed per unit, cu/s (million)**	**Cost per unit, dollars (thousand)**	**Speed per dollar, cu/s/dollar (thousand)**
Pentium III 1.4 GHz	50	1.5	30
Paracel GeneMatcher2	15	50	300
TimeLogic DeCypher	25	220	110
Osiris (CPU final-search)	17,000	20	850
Osiris (initial-search only)	300,000	20	15,000
Osiris (proposed hardware final-search)	50,000	20	2500

Table 5.3. Speed-Cost comparison of Smith-Waterman Database Search Systems.

Chapter 6

Current / Future Work

6.1 Initial-Search Smith-Waterman Hardware Improvements

The initial-search Smith-Waterman hardware can be improved to increase its speed and efficiency. First, the maximum clock rate that could be achieved by the hardware implementation was 120 MHz, which was limited by the speed grade of the Virtex II FPGA used by the Osiris board. The clock rate could be increased to 180 MHz by using a Virtex II FPGA with a higher speed grade, which would increase the processing speed to more than 500 billion cell updates per second. Second, for each database sequence, the hardware generates a single result data word, which indicates whether the sequence comparisons exceeded the threshold. With the improved hardware up to 64 comparison results could be sent using a single data word; thus, reducing the bandwidth required. Finally, hardware implementations could be modified for specific queries using a system for dynamic modification of Virtex implementations called JBits. The implementation could be modified such that each query in a set could have an independent systolic array, and the score threshold used by the processing elements could be adjusted to a value optimal for each query.

6.2 Complete Smith Waterman Hardware Implementation

The initial-search Smith-Waterman algorithm hardware implementation, while extremely fast, is sub-optimal because it only compares DNA sequences, only calculates linear gap penalties, has static scoring parameters, has relatively small and imprecise scoring parameters; and only determines whether the similarity score exceeded a preset threshold. A full implementation, incorporating affine gap penalties, scoring matrices, larger sequence alphabets such as proteins, and user specified scoring parameters, would be useful. A full implementation would run at an estimated 1/100th of the speed of the initial-search hardware implementation, but would be significantly faster than a microprocessor implementation. It would be very useful for processing the subset of the database sequences flagged by the initial-search hardware implementation.

A complete Smith-Waterman algorithm implementation that calculates affine gap penalties would utilize the affine gap penalty optimization described in Chapter 3. Using the optimization, the Smith-Waterman matrix element equation becomes:

$$\begin{aligned} &H_{i,j} = \max\{H_{i-1,j-1} + s(S_i, T_i), Q_{i,j}, P_{i,j}, 0\} \\ &\text{where,} \\ &P_{i,j} = \max\{H_{i-1,j} - w_1, P_{i-1,j} - u\} \\ &Q_{i,j} = \max\{H_{i,j-1} - w_1, Q_{i,j-1} - u\} \end{aligned} \qquad (6.1)$$

Since the Smith-Waterman calculations would be performed using a systolic array of processing elements, it is useful to use a different notation. Instead of defining a cell calculation by its row and column in a matrix, it can be defined by the location of a processing element, and a processing element iteration. The value $H_{i,j}$, where i identifies the row, and j identifies the column, can be converted to $G_{a,b}$, where a identifies the processing element of the systolic array, and b identifies the iteration of the processing element. The transformations of the values required for a Smith-Waterman cell calculation are shown in Equation 6.2.

$$\begin{aligned} &H_{i,j} \Rightarrow G_{a,b} && \text{(a)} \\ &H_{i-1,j} \Rightarrow G_{a,b-1} && \text{(b)} \\ &H_{i,j-1} \Rightarrow G_{a-1,b-1} && \text{(c)} \\ &H_{i-1,j-1} \Rightarrow G_{a-1,b-2} && \text{(d)} \end{aligned} \tag{6.2}$$

Equation 6.2 (b) is the value calculated by the processing element during the previous iteration; Equation 6.2 (c) is the value calculated by the previous processing element during the previous iteration; and Equation 6.2 (d) is the value calculated by the previous processing element during two iterations previous. Using the transformations of Equation 6.2, the transformed Smith-Waterman cell equation is listed in Equation 6.3.

$$\begin{aligned} &G_{a,b} = \max\{G_{a-1,b-2} + s(S_a, T_b), Q_{a,b}, P_{a,b}, 0 \\ &\text{where,} \\ &P_{a,b} = \max\{G_{a,b-1} - w_1, P_{a,b-1} - u\} \\ &Q_{a,b} = \max\{G_{a-1,b-1} - w_1, Q_{a-1,b-1} - u\} \end{aligned} \tag{6.3}$$

Each iteration, each processing element calculates the value $G_{a,b}$. This value is transferred to the next processing element for the next iteration, which corresponds to $G_{a-1,b-1}$ for that processing element and that iteration. $G_{a,b}$ is also stored for the next iteration which corresponds to $G_{a,b-1}$ for the processing element the next iteration. The processing element transfers the stored value $G_{a,b-1}$ to the next processing element for the next iteration, which corresponds to $G_{a-1,b-2}$ for that processing element and that iteration. Each iteration, each processing element calculates the value $P_{a,b}$. This value is also stored for the next iteration which corresponds to $P_{a,b-1}$ for the processing element the next iteration. Each iteration, each processing element calculates the value $Q_{a,b}$. This value is transferred to the next processing element for the next iteration, which corresponds to $Q_{a-1,b-1}$ for that processing element and that iteration.

The calculation of $s(S_a, T_b)$ can be accomplished using a substitution scoring matrix, such as a PAM matrix [12], stored using blocks of onboard RAM. A typical protein substitution matrix uses an alphabet of 24 characters, which can be represented using five bits per character. It also consists of 576 score values, which can typically be represented using at least eight bits. The Virtex II family of FPGAs incorporates multiple blocks of configurable RAM. These blocks can be configured to store 1,024 18-bit values, accessed by a 10-bit address. Thus, an entire substitution matrix could be stored in a single block RAM, and accessed by the two 5-bit characters values as the address. In addition, these block RAM modules are dual-port; thus, they can be shared and accessed simultaneously by two processing elements. The Virtex II XC2V6000 FPGA [40] used by the Osiris board contains 144 of these block RAMs, thus it could be possible to implement up to 288 processing elements.

The calculations of $Q_{a,b}$ and $P_{a,b}$ each require two 2-input addition components, and one 2-input maximum value component. The calculation of $G_{a,b}$ requires the values $s(S_a, T_b)$, $Q_{a,b}$ and $P_{a,b}$, one 2-input addition component, and one 4-input maximum value component. Figure 6.1 shows the block diagram of a single processing element.

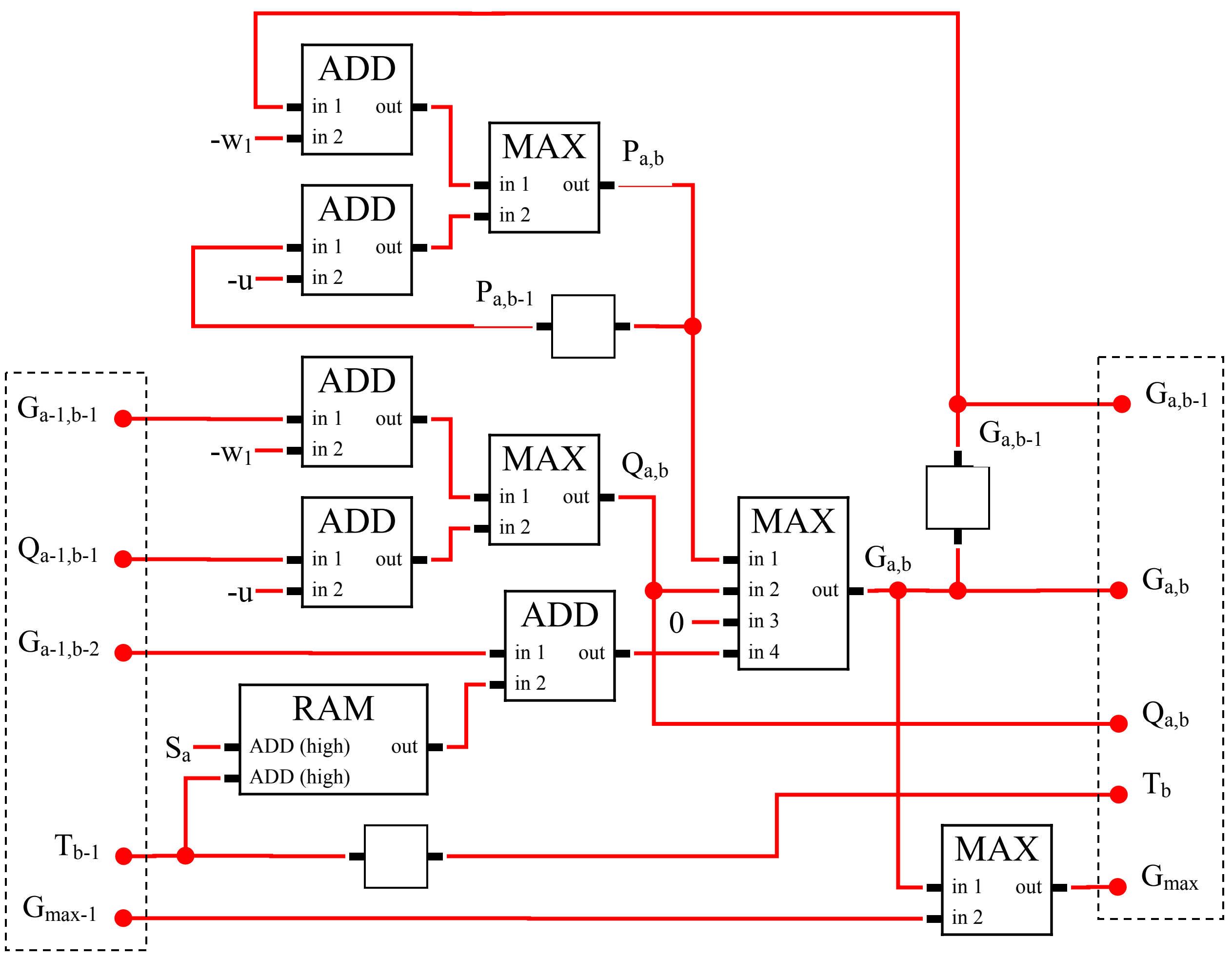

Figure 6.1. Complete Smith-Waterman Hardware Implementation Block Diagram.

Chapter 7

Conclusion

In this thesis, the design, implementation, and analysis of a biological sequence database search system that utilizes adaptive and distributing computing technology, was presented. The system is unique, differing from other processor and custom hardware based systems in that it performs a fast initial database search using an optimized adaptive computing hardware implementation of the Smith-Waterman algorithm, and a final search of the portion of the database flagged by the initial search. In addition, the system was designed to be executed in parallel on multiple machines, and to batch-process user submitted queries to increase the speed and efficiency of the system. Finally, a supporting application was developed to format sequence databases to reduce the file size and allow multiple machines to search individual portions.

Overall, the system performance relative to its cost was significantly better than similar processor and custom hardware based systems. As a result, this system would be useful to biological sequence laboratories and researchers who require fast sequence database searches. However, the systems usefulness is limited by the fact that it implements only the Smith-Waterman algorithm. Future implementations of the system that implement additional search algorithms, such as BLAST [3, 27], and additional capabilities, such as phylogenic tree determination, would significantly increase the overall usefulness of the system.

Bibliography

[1] Russ B. Altman, “Computer Applications in Molecular Biology,” <http://smi-web.stanford.edu/projects/helix/bmi214/primer.pdf> (2003).

[2] S.F. Altschul, “Amino Acid Substitution Matrices from an Information Theoretic Perspective,” *Journal of Molecular Biology*, vol. 219, pp. 555-565, 1991.

[3] S.F. Altschul, W. Gish, W. Miller, E.W Myers, and D.J. Lipman, “Basic local alignment search tool,“ *Journal of Molecular Biology*, vol. 215, pp. 403-410, 1990.

[4] Peter Bellows, Brian Schott, and Li Wang, “Host Programming with SLAAC: The SlaacBoard APIs,” USC Information Sciences Institute, 2.0.0 ed., 15 July 2002.

[5] Peter Bellows, Brian Schott, and Li Wang, “Osiris Board Architecture and VHDL Guide,” USC Information Sciences Institute, 2.0.0 ed., 15 July 2002.

[6] Peter Bellows, Brian Schott, and Li Wang, “The SLAAC Board Debugger,” USC Information Sciences Institute, 2.0.0 ed., 15 July 2002.

[7] Peter Bellows, Brian Schott, and Li Wang, “The SLAAC Board Programming Environment,” USC Information Sciences Institute, 2.0.0 ed., 15 July 2002.

[8] “Compugen homepage,” <http://www.cgen.com/>, (2002).

[9] Compugen, “BioXL/H,” <http://www.cgen.com/products/bioxl.htm>, (2003).

[10] “Celera Genomics homepage,” <http://www.celera.com/>, (2003).

[11] Celera Genomics, “Technology Platforms,” Corporate Information, <http://www.celera.com/company/home.cfm?ppage=overview&cpage=platforms>, (2003).

[12] M.O. Dayhoff, R.M. Schwartz, and B.C. Orcutt, “A Model of Evolutionary Change in Proteins,” *Atlas of Protein Sequence and Structure*, vol. 5, pp. 345-352, 1978.

[13] “FASTA Programs at the U. of Virginia,” <http://fasta.bioch.virginia.edu/>, (2002).

[14] O. Gotoh, “An improved algorithm for matching biological sequences,” *Journal of Molecular Biology*, vol. 162, pp. 705–708, 1982.

[15] S. Guccione and E. Keller, “Gene Matching Using JBits,” 2002.

[16] D. T. Hoang, “Searching genetic databases on splash 2,” *Proceedings 1993 IEEE Workshop on FPGAs for Custom Computing Machines*, pp. 185–191, IEEE Computer Society Press, 1993.

[17] D. T. Hoang, and D. P. Lopresti, “FPGA implementation of systolic sequence alignment,” *Field-Programmable Gate Arrays: Architectures and Tools for Rapid Prototyping*, pp. 183–191, Springer-Verlag, Berlin, 1992.

[18] D. L. Hoffman, “A comparison of the BioSCAN algorithm on Multiple architectures,” Department of Computer Science, University of North Carolina, Chapel Hill, NC, TR93-050, 1993, <http://www.dlhoffman.com/~hoffman/papers/TR93-050.ps>.

[19] R. Hughey, “Massively Parallel Biosequence Analysis,” *Technical Report UCSC-CRL-93-14*, University of California, Santa Cruz, 1993.

[20] R. J. Lipton and D. P. Lopresti, “A Systolic Array for Rapid String Comparison," *1985 Chapel Hill Conference on VLSI*, pp. 363–376, Computer Science Press, Rockville, MD, 1985.

[21] D. J. Lipman, W. R. Person, “Rapid and sensitive protein similarity searches,” *Science*, vol. 227, pp. 1435-1441, 1985.

[22] D. P. Lopresti, “P-NAC: A systolic array for comparing nucleic acid sequences,” *IEEE Computer*, vol. 20, pp. 98–99, July 1987.

[23] Mathematics and Computer Science Division at Argonne National Laboratory, “The Message Passing Interface (MPI) standard,“ <http://www-unix.mcs.anl.gov/mpi/>, (2003).

[24] Kevin Miller, “Real horsepower,” *Roanoke Times*, 2 July 2002, <http://www.roanoke.com/roatimes/news/story132905.html>, (2003).

[25] “National Center for Biotechnology Information,” 9 January 2003, <http://www.ncbi.nlm.nih.gov/>.

[26] “National Human Genome Research Institute,” 2003, <http://www.genome.gov/>.

[27] “NCBI BLAST homepage”, 29 January 2001, <http://www.ncbi.nlm.nih.gov/BLAST/>

[28] S.B Needleman, and C.D. Wunsch, “A General Method Applicable to the Search for Similarities in Amino Acid Sequence of Two Proteins,” *Journal of Molecular Biology*, 48, pp. 443-453, 1970.

[29] Paracel Inc., “The genematcher2 system datasheet,” 2002. <http://www.paracel.com/products/pdfs/gm2 datasheet.pdf>.

[30] “Paracel, Inc. homepage,” 2002, <http://www.paracel.com/>.

[31] W. R. Pearson, D. J. Lipman: “Imported tools for biological sequence comparison,” *Proceedings of the National Academy of Sciences*, vol. 85, pp. 2444-2448, 1988.

[32] T. Smith and M. Waterman, “Identification of common molecular subsequences,” *Journal of Molecular Biology*, vol. 147, pp. 195–197, 1981.

[34] Marc Snir, Steve Otto, Steven Huss-Lederman, David Walker, Jack Dongarra, “MPI: The Complete Reference, Volume 1, The MPI Core, Second Edition,” MIT Press, Boston Massachusetts, 1998.

[33] Stanford University, “Representations and Algorithms for Computational Molecular Biology,” 4 June 2002, <http://smi-web.stanford.edu/projects/helix/bmi214/>.

[35] “TimeLogic Corp., homepage, “ 2002, <http://www.timelogic.com/>.

[36] TimeLogic Corp., “DeCypher,” 2002, <http://www.timelogic.com/decypher_intro.html>.

[37] USC Information Sciences-East, “SLAAC Project Home Page,” 4 June 2001, <http://www.east.isi.edu/projects/SLAAC/>.

[38] M.S. Waterman and M. Vingron, "Rapid and Accurate Estimates of Statistical Significance for Sequence Database Searches," *Proceedings of the National Academy of Sciences*, vol 91, pp. 4625-4628, 1994.

[39] "Xilinx Inc. homepage," 2003, <http://www.xilinx.com/>.

[40] Xilinx Inc., "Virtex™-II Platform FPGA Data Sheet", 26 September 2002, <http://www.xilinx.com/partinfo/ds031.pdf>.

[41] "Beowulf.org", 2002, <http://www.beowulf.org/>.

Vita

Nicholas P. Pappas was born on January 23, 1979 in Fairfax, Virginia and raised in Falls Church, Virginia. After graduating with Honors from McLean High School in 1997, he was awarded several college scholarships and went on to attend Virginia Tech. He majored in Computer Engineering, minored in Math and History, and received a Magna Cum Laude Bachelor's degree with Honors from Virginia Tech in 2001. Nicholas then continued at Virginia Tech for his Master's degree in Computer Engineering. As a graduate student he worked in the Configurable Computing Machine Laboratory developing a biological sequence database search system. During his time at Virginia Tech he was involved in several organizations and activities, most notable of which was Tau Beta Pi, the Engineering Honor Society, where he held several positions, including President, for two years.