Accelerated Biological Meta-Data Generation and Indexing on the Cray XD1

Eric Stahlberg[1], Joseph Fernando[2], Jeff Doak[3], Kevin Wohlever[2]
[1]*Ohio Supercomputer Center, 1224 Kinnear Road, Columbus, Ohio*
[2]*Ohio Supercomputer Center-Springfield, 1 Limestone Street, Springfield, Ohio*
[3] *Cray Inc., Mendota Heights, Minnesota*

ABSTRACT: *The volume and diversity of biological information to be integrated in comparative bioinformatics studies continues to grow. Increasingly, the information is unstructured and without appreciable annotation necessary to make the necessary associations for comparative analysis. The FPGA and the Cray XD1 in particular provide a means to rapidly generate the dynamic metadata information needed to enhance the value of unstructured and semi-structured data. Similar to efforts required of compression and encryption, the presentation will showcase efforts at the OSC center for data intensive computing to employ FPGA technology to the challenge of generalized biological sequence indexing as a foundation for comparative analysis and subsequent predictive inference.*

KEYWORDS: XD1, FPGA , bioinformatics, metadata, digital signature, hash function

1. Introduction

Bioinformatics continues to be an area of explosive growth in the volume of information generated through both instrumental and *in silico* simulations and studies. New species are joining the ranks sequenced genomes annually. The expanding use of automated experimental techniques, including DNA micro-arrays, gene chips and mass-spectrometry add ever increasing amounts of unstructured and semi-structured data to information reservoir on a daily basis. Much of the information has not yet been and may not ever be published or publicly registered. A fundamental challenge facing the management and efficient use of the combined wealth of public and private bioinformatics related content is an efficient means to identify and integrate similar sequences. Using a normalizing signature algorithm designed for maximum portability, ease of implementation and independence from choice of technology platform provides an important step to effectively integrating and maintaining coherency among the evermore pervasive sources of bioinformatics data. The interest in exploring the potential of the Cray XD1 with Field Programmable Gate Arrays (FPGA) in the context of a bioinformatics specific signature function provided the impetus for this project.

2. Motivations

The sources of bioinformatics information in general, and sequence data in particular, are becoming increasingly diverse. The clear message delivered by funding agencies such as the National Science Foundation and the National Institutes of Health to researchers is to share information more openly, thereby adding significant impetus to create even more sources of information. In the case of bioinformatics sequences, this includes registered, pre-registered and even unregistered sequence data. The problem created by such efforts is an inability to consistently identify, track and incorporate sequence information as it moves through various stages of maturity and availability.

2.1 Challenges Facing Sequence Integration

An illustrative example is provided in the experience building the ARABI-COIL database (http://www.coiled-coil.org)[1]. The aims of this project were to develop an integrated database of experimental and *in silico* computed information related to coiled-coil proteins in Arabidopsis thaliana. An initial problem faced was the inconsistent use of identifiers within initial datasets used as the basis for the database. This situation was further exacerbated when sequence updates were applied to the dataset where identifiers had evolved further. While all sequences were publicly registered, the lack of a

consistent fundamental identifier or signature connected to the sequence data itself added significant complexity to the application of updates and incorporation of computed results in the local database.

A further example of benefits derived from a normalizing sequence signature can be found in SAGE (Serial Analysis of Gene Expression) and MPSS (Massively Parallel Signature Searching) investigations. The application of SAGE [2] and MPSS [3] techniques to the study of gene expression generates hundreds of thousands to millions of short sequences (aka tags), typically ranging from 17 to 34 nucleotides in length. Each tag plays a critical role in the overall effectiveness of the study by its presence or absence, and is consequently handled as an independent sequence or element of information. It remains a very significant task to independently register each of these independent tags in a central database for the purposes of assigning an identifier.

A final motivation for defining a normalizing signature algorithm is found in the need to maintain coherency among several independently managed databases. Two predominant sources of human error manifesting in a lack of coherency among disparate data sources include correlating information and preparing information subsets for export. Augmenting the assigned identifier with a normalized signature provides an ability to integrate information more readily with a much higher degree of confidence and practical elimination of human error.

2.2 Solution Design Criteria

An algorithm for a normalized sequence signature and exchange identifier has been developed at OSC (Ohio Supercomputer Center) [4]. The algorithm accomplished several key design goals in the implementation. These are briefly summarized:

Portable – a priority has been placed on being highly portable and amenable to multiple native implementations, including Java, C/C++, Fortran, PERL and, in this case, field programmable gate arrays. The emphasis on portability encourages a pervasive implementation and adoption. The signature itself eliminates portability issues by disavowing the use of non-numeric characters in the signature, thereby providing consistent representation independent of implementation.

Normalizing – a priority has clearly been placed on accounting for conventions used for representing biosequence information in creating a convention-free signature.

Usable – the signature, while relatively brief, contains summary meta-data information about the sequence which may be employed to readily exclude sequences of disinterest. Maintaining a priority on human readable form and discernable fields aids the utility of the signature.

Evenly distributed – the goal of even distribution across the range of generated values is very important for digital signatures and hash functions.

Self-validating – the signature contains information to insure integrity of the signature itself.
Extensibility – the algorithm is defined to be extensible

The algorithm and supporting routines are part of a larger suite of bioinformatics software collectively known as the Ohio Biosciences Library (OBL) which complements the Cray BioLib and the Portable Cray BioLib. More information specific to this algorithm may be found at bioinformatics.osc.edu/obl.

The proven applicability and efficiency of the FPGA for similar functions, such as commonly employed signature and hash algorithms such as MD5 (Message Digest 5) [5] and SHA (Secure Hash Algorithm) provide a natural motivation for implementing the algorithm on the Cray XD1. Further motivation was found in the desire to explore the XD1 development environment.

3. Resources

The Cray XD1 cluster computing system was used as the host for this research effort. The goal was to leverage the unique features of Field Programmable Gate Array (FPGA) devices that are available on the XD1 for this bio-informatics application.

The XD1 that is available at OSC is a cluster with 36 Opteron processors working at 2.2 GHz in three chassis. Each chassis has six Symmetric Multiprocessor Processor (SMP) units, and each SMP has two Opteron processors. One of the chassis also contains six FPGA accelerator cards. Each accelerator card hosts a Xilinx Virtex II Pro 50 device with a -7 speed rating. All the SMPs are connected through a high speed modified infiniband based Rapid Array inter-connect with an effective bandwidth of 10.5 Gbps [6]. The FPGAs are connected to the SMP through a Rapid Array Processor (RAP). Data transfer between the SMP memory subsystem and the FPGA is 64-bits wide and has a bandwidth of 12.8 Gbps [7].

3.1 Software Tools

The XD1 at OSC hosts the Riviera SE mixed language HDL design and simulation environment. This environment [8] supports many Hardware Design Languages (HDL). In particular, it enables mixed language, VHDL and Verilog simulation. This environment was extensively used for design, development and debugging of circuits. The Xilinx ISE

6.3i development toolset [9] was extensively used to synthesize map, place and route the circuits developed.

4. **Implementation**

4.1 Algorithm Definition

The normalizing biosequence signature algorithm, also referred to as the BXID (for Biosequence eXchange ID) is based on several key elements of efficient hash functions and random number generation. The algorithm incorporates a mid-squares hash method to produce an even distribution across the range of values. The algorithm also incorporates a multi-state finite state machine to enhance randomness in the signature distribution despite frequent homology within supplied data. The algorithm employs a mapping function look-up table to accomplish efficient normalization and eliminate extraneous information not relevant to the sequence. Finally, the algorithm definition employs arithmetic operations of limited precision in lieu of bit manipulations to enhance portability of implementation and isolation from byte representation (big-endian vs. little-endian). Intermediate precision of results is designed to not exceed 31 bits, eliminating the need to accommodate sign issues in the arithmetic operations. The details of the algorithm are presented below:

Initialize ***seed*** = 255
Initialize uppermask = 2^23
Initialize lowermask = 2^8
Initialize maxsize = 32767
Initialize s_i = ***seed***, ***i*** an element of {0,1,2,3,4,5}
Initialize g or c character count, ***gc***, and length, ***l***, to zero
Initialize sequence type state variable, ***q*** = 0 (undefined state)
For each character, ***c***, in sequence
  Assign Index value, ***k***, A defined as position 1.
  k = index(***c***, ' ACTGUNXIQRYDOBSEFHJKLMPVWZ*')
  If ***k*** in range (***k*** > 0)
    Increment length**,** ***l***
    Update sequence type, ***q***
    Update ***gc*** count
    Update stage value, s_i, as follows
    i = mod(***l*** ,6)
    f = mod(***l*** + 1, 6)
    s_i = mod(***seed*** + s_i/2 + s_f /2 + ***k*** + ***k*** * mod(***l***, 1021)), maxsize)
    s_i = mod(s_i ^2, uppermask) / lowermask

Composite final hash function values
$h_1 = s_0 * 2^{16} + s_1$
$h_2 = s_2 * 2^{16} + s_3$
$h_3 = s_4 * 2^{16} + s_5$

Compute descriptor fields
length check digit, ***lc*** = ***l*** % 10
length magnitude, ***lm*** = min(floor($\log_{10}$ ***l***) , 9)
length magnitude multiplier, ***lmm*** = ***l*** / ***lm***
set current algorithm version ***v*** = 1
gc percentage, ***p*** = min((***gc*** * 100) / ***l*** , 99)
compute check digit for generated hash fields,
$x = \mathrm{mod}((\Sigma(i=1,3)\ \Sigma(k=0,4)\ \mathrm{mod}(h_i, 10^{2k+2})/10^{2k}),8)$

Composite descriptor fields in human identifiable form
$h_0 =$ ***q*** * 10,000,000 + ***p*** * 100,000 + ***lm*** * 10,000 + ***lmm*** * 1,000 + ***lc*** * 100 + ***v*** * 10 + ***x***

4.2 High-Level Code Description

The algorithm described above has been implemented using FORTAN, Java and C programming languages. In this implementation each character is processed serially. The input to the algorithm is a string of characters. Each input character is first converted to a normal index between 0 and 27, independent of case. Values outside the domain of relevance are ignored. Using the normalized character index value, a hashing function, the sum of *g* and *c* sequence characters, and seven metadata flags are computed.

Computing the hashing function is the major task of this algorithm. There are four hash values that are computed. Three of the four hash values are dependent upon the six intermediate hash values for each of the independent states. Two consecutive intermediate stage values are combined to make three higher precision hash values. The fourth signature element is computed based on the generated metadata flags and the sum of *g* and *c* sequence characters.

The intermediate hash value is computed based on the previous intermediate hash value, the hash value of the next intermediate state, the normalized character index, the number of valid sequence characters accumulated to that point and a ***seed*** that is set at initialization. The use of the mod function is integral to the function, ensuring intermediate values remain within accepted ranges.

S[n] = (***seed*** + S[n]/2 + S[(n+1)%6] /2 + ***k*** * (***l*** % 1021) * + ***k***) % ***maxsize***
S[n] = ((S[n] * S[n]) % ***uppermask***) / ***lowermask***
Where,
S[n] denotes the local hash value,
l denotes the number of valid characters encountered at that point in the sequence.
The initial value of ***seed*** is set to 255. The values ***maxsize***, ***uppermask*** and ***lowermask*** are set to 32767, 2^{23} and 2^8 respectively.
The initial values of all the six local hashes are set to zero.

4.3 Parallel Computation and Modifications

As indicated earlier, the algorithm was developed targeting a sequential machine, where each sequence character is processed one after the other. However, it is

possible to process up to six characters in parallel, since this algorithm computes six local hash values.

The *l* value indicates the number of valid characters. Therefore, the computations need to be done serially. By filtering the input of invalid characters before entering the input, this algorithm could be parallelized and six characters processed simultaneously in six processing units. *l* is computed using a number that is a multiple of six and adding an offset based on the processing unit. *l* wraps around 1021, which is a prime number, in the original algorithm. This was changed to 1020, which is divisible by 6, in the FPGA implementation.

To improve the ease at which the algorithm could be implemented on an FPGA implementation, the ***maxsize*** was changed fro 32767 to 32768 (2^{15}). Also note that based on the above equation the valid bits of the intermediate hash functions are effectively 15.

There are six flags that are set based on the interpretation of the valid characters encountered. These flags characterize the sequence as one of the following: undefined, DNA with no unknown positions, DNA with unknown positions, RNA with no unknown positions, RNA with unknown positions, inconclusive (protein, DNA, RNA), and confirmed protein sequence. The gc percent composition is computed as a readily discernable element of the sequence. These flags and the summations are done in parallel to the intermediate hash computations.

Data Dependency

The intermediate hash value is circularly dependent on other local hash values by design. Local hash 0 depends on local hash 1 and local hash 5 depends on local hash 0. This dependency effectively reduces the parallelism to five. The sixth sequence character could be processed in a sixth processing unit or processed where local hash 0 is computed. Therefore, the latency to compute the latency to process six sequence characters would be double the latency of processing a single character. By overlapping the computations the latency could be reduced.

4.4 FPGA Implementation

The parallel version of this algorithm was implemented targeting the Cray XD1 FPGA programming environment. The compute intensive parts of the algorithms were mapped to FPGAs and the rest was computed using the Opteron. The six local hash and the three hash values that are dependent on the local hash values were computed on the FPGA. The metadata element that describes the global sequence character and the *gc* population were computed on the host Opteron. The major blocks that are implemented on the FPGA are given in Figure 1. The block that in computed on the Opteron is indicated by a dashed line.

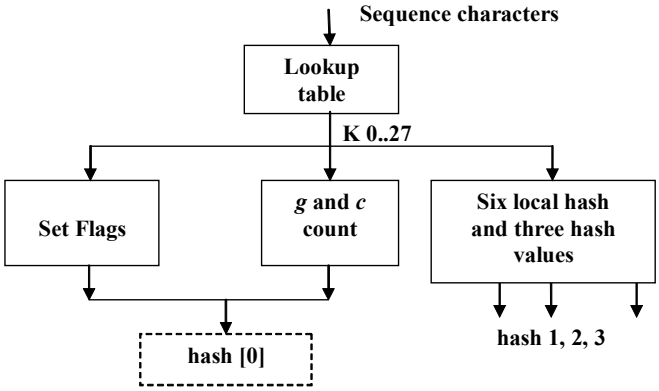

Figure 1. Overview of the Implementation

A register based interface is used to write the input sequence to the FPGA and to read the results from the FPGA. Six sequence characters and *l*, which is a multiple of six (without the offset) is written to a 64-bit register. The FPGA circuit computes the hash and the flags along with the sums of *g* and *c* sequence characters and fills write the results to two 64-bit registers. These registers can be read by the host and processed.
The local hash computations are done in four clock cycles in four different stages. The computations were divided into four stages to increase the clock speed and make it feasible to implement on the XD1. The *l* number and the offset are added and multiplied with the *normalized character index* in stage 1. The seed and the *normalized character index* is added to the resultant of stage 1 in stage 2. The previous local has values are added in stage 3 to the results of stage 2 and the results of stage 3 are multiplied with each other to get the square in stage 4. The bits 23 to 8 are extracted in stage 4 for squaring.

Stage 1: tmp_1 = ipos * K + offset * K
Stage 2: tmp_2 = tmp_1 + seed + K
Stage 3: tmp_3 = tmp_2 + localhash[n]/2 + localhash [n+1]/2
Stage 4: itmp_squared = tmp_3 * tmp_3
localhash [n] = itmp_squared(bits 22 to 8)

Note that the correct number of bits is used in stage 3, for the localhash[n] and localhash[n+1] and in stage 4, bits 22 to 8 of *itmp_squared* are assigned to localhash[n]. Three 18x18 bit multipliers are used in this implementation. In stage 1, *ipos* and offset could be added and then multiplied by K, reducing the multipliers to two. However, such an implementation would have a lower clock speed.

This implementation can be easily fully pipelined as the computations are done in multiple stages and the results

are registered. The major constraint to pipelined implementations is the inability to guarantee input data every clock cycle. A compromise would be to provide input data at selected clock cycles.

This implementation uses a clock frequency of 169 MHz, when a Xilinx 2 Pro device with a speed grade of -7 is targeted. Therefore, a fully pipelined implementation would have the ability to process over 1 billion sequence characters a second. The bandwidth between the FPGA and the host constrains this to a lower throughput.

The localhash processing unit was written in Verilog. The lookup table, flag generation and the *g* and *c* accumulators were written in VHDL.

Approximately, 2000 slices of the FPGA resources on a Xilinx 2VP50 device, which is about 8 % of the slices available, are used for this implementation. Included in this resource calculation, are the resources required for all other functional units such as the register interface, block RAM interface, QDR memory interface, etc. The algorithm itself would require no more than 500 slices.

5. Results

The strategy that was used to determine the correctness of the implementation was empirical. On each step of the design cycle, from developing the processing units to integrating with the register interface and developing the software interface, the circuits have been simulated and tested using a sample sequence string. The sample string was incorporated in an input data file, which is used by the simulator to read and write to specific register or memory locations. The outputs of the designated registers were checked for the output values.

After synthesizing the FPGA circuit, many randomly generated were used to check the hash values produced by the FPGA implementation and the software program. When the changes done to the FPGA were incorporated in the software program the results obtained by both methods were the same.

6. Limitations of the Implementation

One of the main limitations of the present implementation is that it is not fully pipelined, since it uses a register based interface. The register based interface design was chosen as there is no mechanism for the FPGA to interrupt the host processor and indicate the completion of tasks. The only method that is available is for the processor to poll on a register for a bit to be set by the FPGA fabric. This will reduce the available bandwidth. A FPGA Transfer Region (FTR) memory based communication interface between the FPGA and the host would alleviate the bandwidth constrains that exist, since it is able to better use the 'burst mode' for the transfer of data. In the burst mode, eight consecutive quad-words can be scheduled to be transferred either way instead of the single quad word at a time.

The FPGA can initiate a read or write FTR memory transfer. The host has to provide a pointer to the read buffer and the length. The FPGA can transfer the data and compute the hash values without any intervention by the host. Such an interface would be suitable for very long DNA sequences and would require a fully pipelined implementation.

The present implementation has been targeted for the Cray XD1 specifically. Presently, as is, this implementation is not portable to any other platform as it is quite integrated to the register interface. One could however, decouple the interface and make it more portable.

Presently, the HDL code is written in VHDL as well as in Verilog. This could be changed to a single language easily so that users who have access to single language design environments could also modify and simulate the code.

The XD1 API writes and reads to and from the registers using unsigned long data types. When a string of six characters have to be written to the registers each character has to be packed so that the order does not change because the Opteron is a little-endian machine. Basically, the first character of the string should be input to the first processing unit and not the sixth. This increases the work load on the host. This work could be eliminated by changing the order the characters are fed to the processing units. This would not be a factor if the host was a big-endian machine.

Performance

The algorithm was able to sustain a processing rate of 6 million sequence characters per second on the XD1**.**

Algorithm Effectiveness

Several studies were conducted on the Cray X1 to validate the overall effectiveness of the algorithm prior to implementation on the XD1. In these studies, randomly generated nucleotide and protein sequences of varying length were independently generated and evaluated for distribution and collision frequency.

The studies indicate an overall effective distribution of results for the algorithm. In the largest case, 500 million randomly protein sequences of varying length were created and analysed for collision frequency. In this case, no collisions were detected. In the case of exclusively nucleotide sequences, collisions were detected more frequently, with a relatively small number collisions

detected within 100 million randomly generated sequences of varying length, as a result of increased intra-sequence homology. The hash value distribution remained even for all cases considered. Full details of validation studies are available [4].

7. Conclusions

The Cray XD1 with field programmable gate array support provided an effective platform for implementing the biosequence signature algorithm. With minor adjustments to the algorithm, an efficient implementation was enabled, utilizing only a small portion of the available programmable logic on the FPGA. While tremendous processing rates appear to be within reach on the FPGA for this algorithm, communication bottlenecks readily become apparent, thereby limiting the true potential in the current implementation.

Future Efforts

OSC has developed an interface that used the FPGA transfer region (FTR) to communicate between the FPGA and the host. Preliminary results indicated that this interface has a bandwidth of approximately 800MB each way, consistently, using a round trip testing program. This interface is at least two orders of magnitude faster than the register interface in use at present. We intend to use this FTR memory interface to accelerate the computations.

Even with the limitations, the FPGA implemented signature module holds important potential for on and off FPGA integration with other sequence comparison algorithms, search algorithms and on-the-fly content analysis.

8. Acknowledgements

The authors would like to acknowledge the critical role of the Department of Energy for this effort in terms of equipment and staff time support. We would like to further acknowledge both OSC and Cray, Inc. for supporting staff efforts in the development, implementation and validation of the algorithm. The authors would also like to acknowledge the efforts of Pete Carswell at OSC for developing a reference C implementation of the algorithm used in this study. We would also like to acknowledge the National Science Foundation Project 2010 for funding the project for which this algorithm was developed and employed.

9. References

1. Rose A, Manikantan S, Schraegle SJ, Maloy MA, Stahlberg EA, Meier I: **Genome-wide identification of Arabidopsis coiled-coil proteins and establishment of the ARABI-COIL database**. Plant Physiology 2004a, **134**:927-939 (2004)

2. Velculescu VE, Zhang L, Vogelstein B and Kinzler KW, **Serial Analysis of Gene Expression**, Science, **270**:484-487 (1995)

3. Brenner S, *et al*. **Gene expression analysis by massively parallel signature sequencing (MPSS) on microbead arrays**. Nat. Biotechnol. **18**, 630−634 (2000)

4. Stahlberg EA, Doak J. *manuscript in process*

5. Rivest R, **The MD5 Message-Digest Algorithm**, RFC 1321, MIT LCS and RSA Data Security, Inc., April 1992

6. Why use RDMA, referenced from http://www.osc.edu/~/dennis/rdma/rdma.html

7. Cray XD1 Product Description, Cray Inc, 2004.

8. Riviera Overview, referenced from http://www.aldec.com

9. XST User Guide, referenced from http://www.xilinx.com/support/sw_manuals/xilinx6/dpwnload

About the Authors

Eric Stahlberg is senior researcher at OSC specializing in algorithm development and integration challenges in life sciences. Eric can be reached at OSC with the email: eas@osc.edu

Joseph Fernando is a senior researcher at the Ohio Supercomputer Center Sprinfield site specializing in FPGA algorithms and implementations. Joseph can be reached with the email: fernando@osc.edu

Kevin Wohlever is director of the Ohio Supercomputer Center Springfield site. Kevin may be contacted at kevin@osc.edu

Jeff Doak is the on-site Cray analyst at the Ohio Supercomputer Center Springfield. Jeff may be reached by email: jdoak@cray.com