Πολυτεχνείο Κρήτης Τμήμα Ηλεκτρονικών Μηχανικών και Μηχανικών Ηλεκτρονικών Υπολογιστών Εργαστήριο Μικροεπεξεργαστών και Υλικού

Θέμα:

Σχεδίαση και υλοποίηση σε επεξεργαστή και FPGA απομακρυσμένης κλήσης συναρτήσεων για λογισμικό NCBI BLAST

Βασταρούχας Γιώργος

Διπλωματική εργασία

Επιτροπή

Δόλλας Απόστολος, Καθηγητής

Πνευματικάτος Διονύσιος, Καθηγητής

Παπαευσταθίου Ιωάννης, Αναπληρωτής Καθηγητής

Χανιά 2011

Στον πατέρα μου

Ευχαριστίες

Θα ήθελα να ευχαριστήσω πρώτα απ' όλα τον επιβλέποντα Καθηγητή Απόστολο Δόλλα για την εποικοδομητική συνεργασία σε ότι αφορά τη διπλωματική μου. Επίσης θα ήθελα να ευχαριστήσω τον Καθηγητή Διονύσιο Πνευματικάτο και τον Αναπληρωτή Καθηγητή Ιωάννη Παπαευσταθίου, για τη συμμετοχή τους ως μέλη της εξεταστικής επιτροπής.

Ένα μεγάλο ευχαριστώ στον Δρα. Ευριπίδη Σωτηριάδη για την καθοδήγηση του σε κάθε βήμα της πορείας αυτής της εργασίας από την αρχή μέχρι το τέλος καθώς επίσης και τον διδακτορικό φοιτητή Γρηγόρη Χρυσό για την πολύτιμη βοήθεια του.

Επίσης θέλω να ευχαριστήσω τους μεταπτυχιακούς φοιτητές Νίκο Παττακό για την υποστήριξη που παρείχε σε ότι αφορά το UNIX και τον Αργύρη Ηλία για τη βοήθεια του με ότι έχει σχέση με την πλατφόρμα Re.Do.FPGA.

Τις ευχαριστίες μου θα ήθελα να εκφράσω και στον υπεύθυνο του εργαστηρίου Μικροεπεξεργαστών και Υλικού κ. Μάρκο Κιμιωνή, καθώς και σε όλα τα μέλη του εργαστηρίου για την βοήθεια που μου παρείχαν και για το άψογο κλίμα που υπήρχε στις μεταξύ μας σχέσεις.

Για τη φιλία και τη στήριξη τους θα ήθελα να ευχαριστήσω και όλους τους φίλους και συναδέλφους που ήταν δίπλα μου.

Πάνω απ όλα θα ήθελα να ευχαριστήσω τα μέλη της οικογένειας μου που είναι δίπλα μου και με στηρίζουν στα όνειρα μου. Τη μητέρα μου Βάια και τον αδερφό μου Μιχάλη, και τέλος τον πατέρα μου Παναγιώτη που πλέον δεν είναι μαζί μας.

Περιεχόμενα

1 Εισαγωγή 9
1.1. Το πρόβλημα 9
1.2 Συνεισφορά διπλωματικής 12
1.3 Οργάνωση Διπλωματικής 12
2. Σχετική Έρευνα 13
2.1. Στοίχιση ακολουθιών 13
2.1.1. Επεξήγηση νουκλεονικών οξέων 13
2.1.2 Στοίχιση ακολουθιών 15
2.2 Αλγόριθμοι βιοπληροφορικής 18
2.3 Υλοποιήσεις αλγορίθμων βιοπληροφορικής σε FPGA 29
3 Μελέτη χαρακτηριστικών (Profiling) NCBI BLAST 33
3.1 Profiling NCBI BLAST 33
3.2 Ο αλγόριθμος σάρωσης της βάσης δεδομένων του NCBI BLASTn 44
3.3 Μοντελοποίηση της συνάρτησης ScanSubject 56
4 Υλοποίηση απομακρυσμένης κλήσης συναρτήσεων (Remote Procedure Call) μέσω της πλατφόρμας RE.DO.FPGA 65
4.1 Υλοποίηση της απομακρυσμένης κλήσης συναρτήσεων μέσω της πλατφόρμας Re.Do.FPGA 65
4.2 Περιγραφή της σχεδίασης και της υλοποίησης της συνάρτησης *s_BlastSmallNaScanSubject_8_4 σε Hardware* 71
4.3 Τελικό σύστημα απομακρυσμένης κλίσης συναρτήσεων μέσω της πλατφόρμας Re.Do.FPGA για το λογισμικό NCBI BLAST. 79
5 Επιβεβαίωση λειτουργίας, πειραματικά αποτελέσματα και αξιολόγηση συστήματος ..81
5.1 Επιβεβαίωση λειτουργίας 81
5.2 Πειραματικά αποτελέσματα 81
5.3 Αξιολόγηση Συστήματος 84
6 Συμπεράσματα και μελλοντικές επεκτάσεις 87
6.1 Συμπεράσματα 87
6.2 Μελλοντικές επεκτάσεις 88
7 Αναφορές 89
Παράρτημα 93

Κατάλογος εικόνων

Εικόνα 1 Όγκος δεδομένων βιοπληροφορικής ανά χρόνο [1]..... 10
Εικόνα 2 Νουκλεοτίδιο..... 14
Εικόνα 3 Η αλυσίδα του DNA..... 15
Εικόνα 4 Τα δύο είδη alignment [36]..... 17
Εικόνα 5 Βήματα του αλγορίθμου Needleman-Wunsch για στοίχιση των ακολουθιών ABCNYRQCLCRPM και AYCYNRCKCRBP [31]. (a) Αρχικοποίηση πίνακα , (b) και (c) γέμισμα πίνακα, (d) ο πίνακας ολοκληρωμένος, (e) και (f) οι καλύτερες στοιχίσεις με ίδιο σκορ..... 19
Εικόνα 6 Τελικός πίνακας αλγορίθμου Smith-Waterman [37] 20
Εικόνα 7 Βήματα του αλγορίθμου FASTA [38]..... 22
Εικόνα 8 Παράδειγμα υπολογισμού σκορ για τον αλγόριθμο BLAST χρησιμοποιώντας τον PAM 120 [33] ως πίνακα σκορ..... 24
Εικόνα 9 Λίστα με τα σημαντικότερα αποτελέσματα alignment του BLAST ως έξοδος..... 27
Εικόνα 10 Έξοδος του BLAST για μια στοίχιση..... 27
Εικόνα 11 Ποσοστιαία ανάλυση χρόνου εκτέλεσης ανά συνάρτηση για το sequence 3 αλλάζοντας τη βάση δεδομένων..... 36
Εικόνα 12 Ποσοστιαία ανάλυση χρόνου εκτέλεσης ανά συνάρτηση για το sequence 3 αλλάζοντας το μέγεθος λέξης (w-mer)..... 37
Εικόνα 13 Ποσοστιαία ανάλυση χρόνου εκτέλεσης ανά συνάρτηση αλλάζοντας μόνο το query..... 38
Εικόνα 14 Ποσοστιαία ανάλυση χρόνου εκτέλεσης ανά συνάρτηση για query μεγαλύτερο των 6.400 χαρακτήρων..... 39
Εικόνα 15 Οι Ποσοστιαία ανάλυση χρόνου εκτέλεσης ανά συνάρτηση για το sequence 3 αλλάζοντας τα Substitution Matrices..... 40
Εικόνα 16 Ποσοστιαία ανάλυση χρόνου εκτέλεσης ανά συνάρτηση για το sequence 3 αλλάζοντας το E-value..... 41
Εικόνα 17 Ποσοστά χρόνου κατά την εκτέλεση του αλγορίθμου NCBI BLASTn.... 42
Εικόνα 18 Παράδειγμα εισόδου σε FASTA μορφή..... 44
Εικόνα 19 Τα δύο άκρα του DNA 46
Εικόνα 20 Το τελικό query..... 47
Εικόνα 21 Σχέση πίνακα backbone με overflow 50
Εικόνα 22 Προσβάσεις στη βάση ανάλογα με τη διαφοροποίηση του stride 55
Εικόνα 23 Μοντελοποίηση συνάρτησης ScanSubject..... 58
Εικόνα 24 Μοντελοποίηση συνάρτησης NA_ACCESS_HITS..... 60
Εικόνα 25 Μοντελοποίηση συνάρτησης Retrieve_Hits..... 61
Εικόνα 26 Σύστημα Re.Do.FPGA 66
Εικόνα 27 Block diagram Re.Do.FPGA 66
Εικόνα 28 Η διεπαφή my_black_box και η επικοινωνία με το PCI-express 68
Εικόνα 29 Το συνολικό σύστημα Re.Do.FPGA..... 70
Εικόνα 31 Σχεδίαση του module overflow..... 75

Κατάλογος πινάκων

Πίνακας 1 Παράμετροι NCBI BLAST 34
ηΠίνακας 2 Queries που χρησιμοποιήσαμε για τις μετρήσεις 35
Πίνακας 3 Βάσεις δεδομένων που χρησιμοποιήσαμε για τις μετρήσεις 35
Πίνακας 4 Χρόνοι εκτέλεσης του BLASTn αλλάζοντας τις βάσεις δεδομένων 36
Πίνακας 5 Χρόνοι εκτέλεσης του BLASTn αλλάζοντας το μέγεθος λέξης (w-mer) . 37
Πίνακας 6 Χρόνοι εκτέλεσης του BLASTn αλλάζοντας τα queries 40
Πίνακας 7 Χρόνοι εκτέλεσης του BLASTn αλλάζοντας τα Substitution matrices 40
Πίνακας 8 Χρόνοι εκτέλεσης του BLASTn αλλάζοντας το E-value 41
Πίνακας 9 Indexing του query 48
Πίνακας 10 Ο πίνακας overflow που δημιουργήσαμε 49
Πίνακας 11 Οι συναρτήσεις του αλγόριθμου NCBI BLASTn για τη σάρωση της βάσης δεδομένων 52
Πίνακας 11 Συναρτήσεις NA_ACCESS_HITS ανάλογα με τον Lookup Table που χρησιμοποιήθηκε 59
Πίνακας 12 Συναρτήσεις τύπου Retrieve Hits βάση του Looku Table που χρησιμοποιήθηκε 60
Πίνακας 13 Στατιστικές μετρήσεις του BLASTn χρησιμοποιώντας διαφορετικά μεγέθη λέξεων [24] 62
Πίνακας 17 Σήματα εισόδου my_black_box 69
Πίνακας 18 Σήματα εξόδου my_black_box 69
Πίνακας 15 Διαστασιολόγηση backbone 74
Πίνακας 16 Διαστασιολόγηση backbone 74
Πίνακας 19 Χρήση πόρων FPGA στην αρχιτεκτονική της σχεδίασης 82
Πίνακας 20 Χρήση πόρων FPGA στην υλοποίηση του συστήματος 82
Πίνακας 21 Συχνότητα της σχεδίασης και του συστήματος μας 83
Πίνακας 22 Είσοδοι που χρησιμοποιήσαμε για τις μετρήσεις μας 83
Πίνακας 23 Βάσεις που χρησιμοποιήσαμε για τις μετρήσεις μας 83
Πίνακας 24 Χρόνος και speedup εκτέλεσης συνάρτησης 84
Πίνακας 25 Χρόνος και speedup εκτέλεσης συνάρτησης 84

1 Εισαγωγή

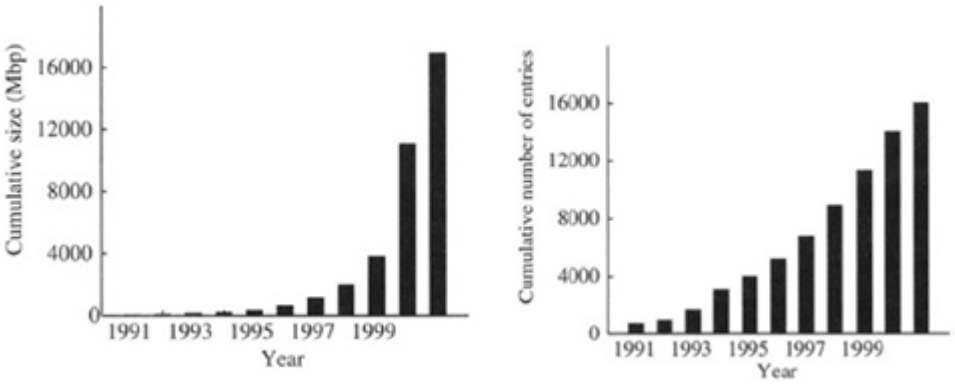
1.1. Το πρόβλημα

Έχουμε κατά καιρούς, στο εργαστήριο Μικροεπεξεργαστών και Υλικού, ασχοληθεί με αλγόριθμους της βιοπληροφορικής. Δεν είχαμε προσπαθήσει όμως μέχρι τώρα να συνδέσουμε έναν πολύπλοκο κώδικα από ένα γνωστό πρόγραμμα βιοπληροφορικής με μια δική μας σχεδίαση σε αναδιατασσόμενη λογική. Εξαιτίας της εμπειρίας του τμήματος μας με τον τομέα της βιοπληροφορικής αποφασίσουμε να διαλέξουμε ένα από τα πολλά βαριά προγράμματα της βιοπληροφορικής για να εφαρμόσουμε για πρώτη φορά τις ιδέες μας.

Ο τομέας της βιοπληροφορικής

Η κατανόηση της γλώσσας των ζωντανών κυττάρων είναι η αναζήτηση της σύγχρονης μοριακής βιολογίας. Από ένα αλφάβητο μόνο τεσσάρων γραμμάτων, τα οποία αντιπροσωπεύουν τις χημικές υπο-μονάδες του DNA αναδύεται το συντακτικό των διεργασιών της ζωής που το πιο περίπλοκο της στοιχείο είναι ο ίδιος ο άνθρωπος.

Το 1953 ήταν μια πολύ σημαντική χρονιά για τη μοριακή βιολογία. Τότε ήταν που οι Watson και Crick ανακάλυψαν τη δομή του DNA. Έκτοτε ο κλάδος της μοριακής βιολογίας βρίσκεται σε συνεχή πρόοδο. Με την ολοένα και αυξανόμενη ικανότητα μας να μεταχειριζόμαστε βιο-μοριακές ακολουθίες, προήλθε ένας τεράστιος όγκος δεδομένων ο οποίος αυξάνει με εκθετικούς ρυθμούς όπως φαίνεται και στην εικόνα 1 [1].



Εικόνα 1 Όγκος δεδομένων βιοπληροφορικής ανά χρόνο [1]

Κάποιες από τις βάσεις αυτές είναι και οι GenBank , EMBL, PIR, GSDB, DDBJ, EBI, and Swiss-Prot.

Παρά το γεγονός ότι βιολόγοι ερευνητές είναι οι δημιουργοί και οι χρήστες των δεδομένων αυτών, η μεγάλη πολυπλοκότητα και το μέγεθος των δεδομένων κάνουν απαραίτητη τη βοήθεια και άλλων επιστημών όπως τα μαθηματικά και την επιστήμη των υπολογιστών. Όσον αφορά τον κλάδο της τεχνολογίας, δεν προσφέρει μόνο την ωμή χωρητικότητα για την αποθήκευση και διαχείριση των δεδομένων αλλά και νέες έξυπνες μαθηματικές (αλγοριθμικές) μεθόδους για να επιτευχθούν καλά αποτελέσματα. Το αποτέλεσμα ήταν να δημιουργηθεί ένα νέο πεδίο το οποίο ονομάζεται υπολογιστική μοριακή βιολογία ή αλλιώς βιοπληροφορική (bioinformatics).

Ο κλάδος των bioinformatics ξεκίνησε τη δεκαετία του ’70 και σήμερα γνωρίζει μια πολύ μεγάλη άνθιση προσφέροντας περισσότερο ακριβή και ισχυρά εργαλεία στους βιολόγους. Η σύγκριση ακολουθιών, ιδιαίτερα με βάσεις πρωτεϊνών και DNA, είναι το πιο συχνό εγχείρημα των βιολόγων σήμερα.

Προβλήματα της βιοπληροφορικής

Αν μπορούσαμε να θέσουμε ένα μεγάλο εγχείρημα στον κλάδο των bioinformatics αυτό θα ήταν το εξής: Χρειαζόμαστε καλή οργάνωση και μια εύκολη και γρήγορη πρόσβαση στην πληροφορία. Όσο καλύτερα γίνεται η οργάνωση αυτή τόσο πιο

εύκολα μπορούμε να αντιμετωπίσουμε τις έξι μεγάλες κατηγορίες προβλημάτων του κλάδου των bioinformatics.

1) **Σύγκριση ακολουθιών**. Ουσιαστικά εννοούμε την ευθυγράμμιση των ακολουθιών δηλαδή το ποσοστό ομοιότητας δύο ή και περισσοτέρων ακολουθιών.
2) **Συναρμολόγηση κομματιών γονιδίων**. Οι βιολόγοι προσπαθούν να συναρμολογήσουν εξ' ολοκλήρου ένα ολόκληρο γονίδιο χρησιμοποιώντας μόνο τα κομμάτια αυτού.
3) **Πρόβλημα φυσικής χαρτογράφησης**. Μοιάζει με το προηγούμενο πρόβλημα αλλά είναι μεγαλύτερο σε έκταση.
4) **Φυλογενετικό δέντρο**. Αναπαράσταση του δέντρου της ζωής με σκοπό να κατανοήσουμε την εξέλιξη. Είναι πολύπλοκο πρόβλημα και οι βιολόγοι έχουν ξοδέψει εκατομμύρια ώρες υπολογισμών σε CPU.
5) **Αναδιατάξεις γονιδίων**. Έχει παρατηρηθεί ότι σε πολλούς οργανισμούς η διαφορά δεν έγκειται σε επίπεδο ακολουθιών αλλά στη σειρά με την οποία οι ακολουθίες αυτές εμφανίζονται μέσα στα γονίδια. Για τη δουλειά αυτή έχουν αναπτυχθεί ενδιαφέροντα μαθηματικά μοντέλα.
6) **Πρόβλεψη δομής πρωτεϊνών και δομής RNA**. Επειδή τα μόρια αναπτύσσονται σε τρεις διαστάσεις, και χάρη στην ανάπτυξη αυτή στηρίζεται πολλές φορές η λειτουργία του μορίου, οι ερευνητές προσπαθούν βασισμένοι σε μια συγκεκριμένη ακολουθία να προβλέψουν τον τρόπο με τον οποίο το μόριο αυτό θα αναπτυχθεί στις τρεις διαστάσεις.

Εμείς θα ασχοληθούμε κυρίως με το πρώτο πρόβλημα του κλάδου της βιοπληροφορικής, τη σύγκριση ή αλλιώς τη στοίχιση ακολουθιών το οποίο όμως βρίσκει εφαρμογές και στα υπόλοιπα προβλήματα.

1.2 Συνεισφορά διπλωματικής

- Στη διπλωματική αυτή εργασία μελετήσαμε σε βάθος την NCBI υλοποίηση του αλγορίθμου BLAST όπου:
    i. Εντοπίστηκαν οι συναρτήσεις που καταναλώνουν μέχρι και το 70% του χρόνου.
    ii. Βρήκαμε ότι οι συναρτήσεις αυτές αλλάζουν ανάλογα με τη διαστασιολόγηση του προβλήματος και το μέγεθος της λέξης (w-mer), αλλά έχουν την ίδια δομή και παρόμοια λειτουργικότητα ακόμη και για την NCBI υλοποίηση.
    iii. Η NCBI υλοποίηση δεν εξετάζει απαραίτητα χαρακτήρα προς χαρακτήρα τη βάση δεδομένων. Μπορεί να έχει βήμα 2, 3 ή και 4 και έτσι εξηγείται η εύρεση υπερσυνόλου αποτελεσμάτων στις TUC [26-30] υλοποιήσεις σε σχέση με αυτή του NCBI.
- Σχεδιάσαμε μια γενική αρχιτεκτονική σε hardware για την πλειονότητα των συναρτήσεων σάρωσης (ScanSubject).
- Υλοποιήσαμε μια συνάρτηση σάρωσης για w-mer=8 και βήμα= 4
- Τέλος κάναμε Remote Procedure Call μέσω της πλατφόρμας Re.Do.FPGA που αναπτύχθηκε από το Πολυτεχνείο Κρήτης. Τρέξαμε τον NCBI BLASTn και στο βαρύ του κομμάτι το οποίο και αντικαταστήσαμε κάναμε κλήση στην FPGA. Όταν το βήμα αυτό του αλγορίθμου τελείωσε συνέχισε να εκτελείται το software και πήραμε σωστά αποτελέσματα.

1.3 Οργάνωση Διπλωματικής

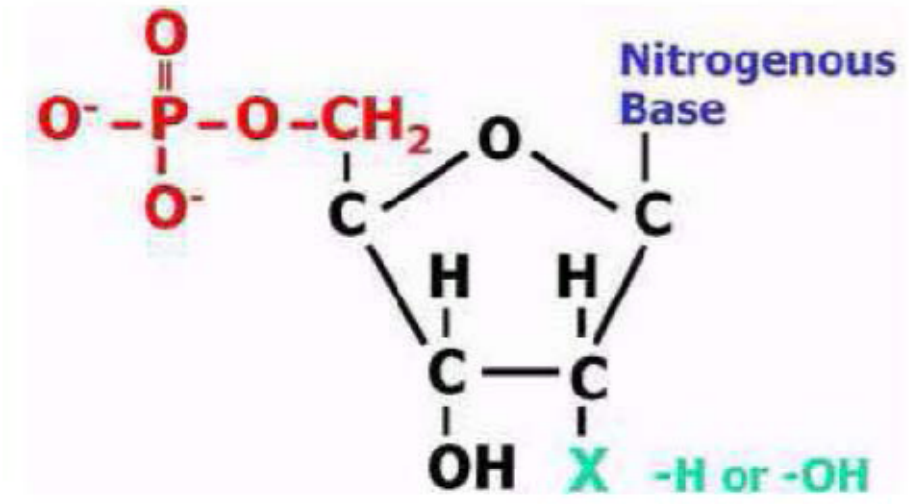
Στο κεφάλαιο 2 θα παρουσιάσουμε τη σχετική έρευνα που έχει γίνει κυρίως στο κομμάτι των αλγορίθμων του τομέα της βιοπληροφορικής. Στο κεφάλαιο 3 παρουσιάζουμε την NCBI υλοποίηση του BLASTn καθώς και αναλύουμε σε βάθος τις συναρτήσεις σάρωσης της βάσης. Στο κεφάλαιο 4 περιγράφουμε μια γενική υλοποίηση των συναρτήσεων σάρωσης της βάσης και στο κεφάλαιο 5 κάνουμε μια συνολική αποτίμηση του συστήματος μας. Στο 6 ακολουθούν τα συμπεράσματα που βγάλαμε μέσω της εργασίας μας και τις μελλοντικές της επεκτάσεις. Ακολουθούν οι αναφορές και το παράρτημα.

2. Σχετική Έρευνα

2.1. Στοίχιση ακολουθιών

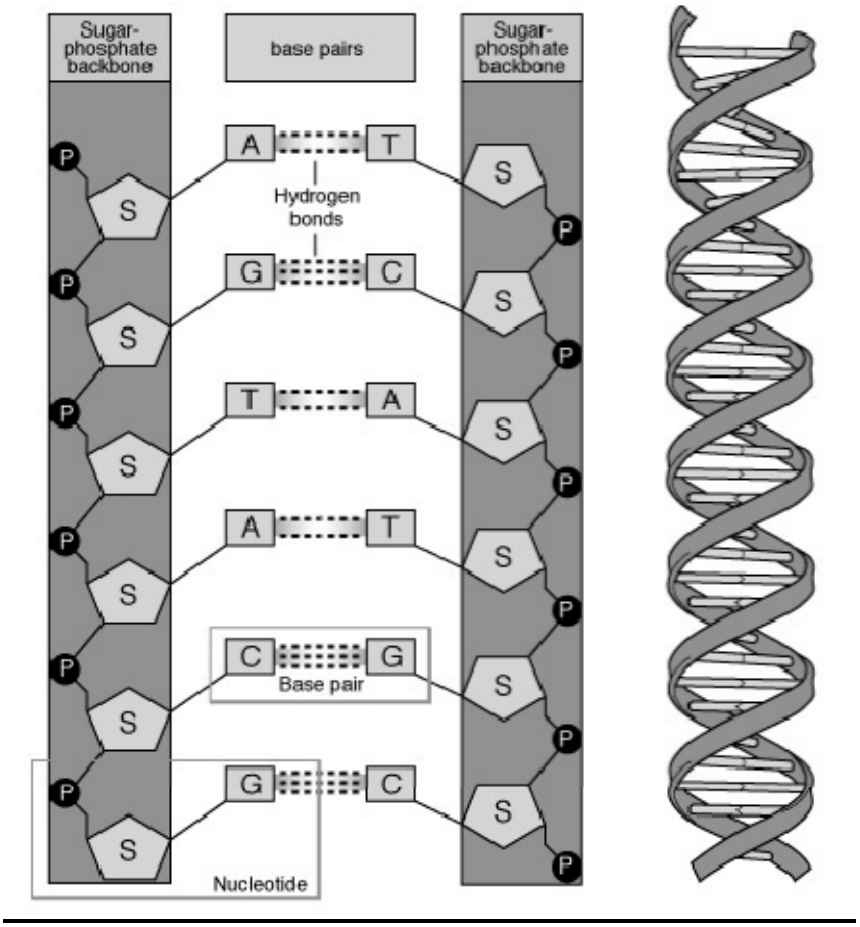
2.1.1. Επεξήγηση νουκλεονικών οξέων

Το DNA και το RNA είναι τα δύο είδη νουκλεονικών οξέων που σχετίζονται πολύ στενά με την γενετική πληροφορία. Είναι δηλαδή, τα στοιχεία εκείνα που έχουν να κάνουν με την κληρονομικότητα και τα ιδιαίτερα χαρακτηριστικά κάθε ατόμου. Η δομική μονάδα των νουκλεονικών οξέων είναι τα νουκλεοτίδια τα οποία αποτελούνται από μια πεντόζη που είναι ενωμένη με μία φωσφορική ομάδα και μία αζωτούχο βάση. Στο μόνο που διαφέρει το ένα νουκλεοτίδιο με το άλλο είναι η αζωτούχα βάση με την οποία είναι συνδεδεμένο. Στα νουκλεοτίδια του DNA η αζωτούχος βάση μπορεί να είναι: Αδενίνη (A Adenine), Γουανίνη (G Guanine), Θυμίνη (T Thymine), Κυτοσίνη (C Cytosine). Στο RNA μπορεί να συναντήσει κανείς όλες τις προηγούμενες, έκτος από τη Θυμίνη που τη θέση της καταλαμβάνει η Ουρακίλη (U Uracil) .



Εικόνα 2 Νουκλεοτίδιο

Τα νουκλεοτίδια συνδέονται μεταξύ τους με ομοιοπολικό δεσμό για να σχηματίσουν το μακρομόριο και μπορούν να εμφανίζονται στην αλυσίδα του DNA με οποιαδήποτε σειρά. Το σύμπλεγμα του σακχάρου και των φωσφορικών οξέων αποτελεί τη ραχοκοκαλιά του μορίου του DNA. Επειδή μία μόνο αλυσίδα του μορίου του DNA είναι αρκετά εύθραυστη συνήθως το DNA εμφανίζεται να σχηματίζει διπλές αλυσίδες. Ο κανόνας για τον σχηματισμό των αλυσίδων αυτών είναι η τήρηση της συμπληρωματικότητας των βάσεων. Τα επιτρεπτά ζεύγη είναι A-T και C-G. Στην περίπτωση σχηματισμού συμπληρωματικής αλυσίδας RNA, οι συνδυασμοί αυτοί γίνονται AU και CG, καθώς όπου υπάρχει θυμίνη στο DNA υπάρχει ουρακίλη στο RNA. Η συμπληρωματικότητα είναι το χαρακτηριστικό εκείνο του DNA που του παρέχει τη δυνατότητα του αυτοδιπλασιασμού (αντιγραφή του DNA). Μια ακόμα διαφορά στη δομή των DNA και RNA είναι ότι ενώ το RNA αποτελείται από μόνο μια αλυσίδα νουκλεοτιδίων, το DNA αποτελείται από δύο. Οι σημαντικότερες διαδικασίες που παρατηρούνται στο DNA είναι η αντιγραφή, η μεταγραφή, η μετάφραση και η μετάλλαξη. Το DNA αποτελεί το γενετικό υλικό όλων των κυττάρων και των περισσοτέρων ιών. Το γενετικό υλικό ενός κυττάρου αποτελεί το γονιδίωμα ή αλλιώς γένωμα του. Μια σειρά νουκλεοτιδίων του DNA αποτελούν ένα γονίδιο. Τα γονίδια περιέχουν τον κωδικό για την παραγωγή των πρωτεϊνών, που είναι τα μακρομόρια εκείνα που διεκπεραιώνουν τις περισσότερες εργασίες στο κύτταρο.



Εικόνα 3 Η αλυσίδα του DNA

2.1.2 Στοίχιση ακολουθιών

Τι είναι η στοίχιση ακολουθιών και ποια τα είδη της

Η **στοίχιση ακολουθιών** (sequence alignment), είναι μια διαδικασία κατά την οποία δύο ακολουθίες ή αλλιώς συμβολοσειρές τοποθετούνται η μία κάτω από την άλλη, με τέτοιον τρόπο που τα κοινά τους σύμβολα να είναι τοποθετημένα στην ίδια θέση. Σκοπός είναι να βρεθεί η "βέλτιστη στοίχιση", δηλαδή η στοίχιση στην οποία οι δύο ακολουθίες ταιριάζουν περισσότερο μεταξύ τους. Η διαδικασία αυτή χρησιμοποιείται ιδιαίτερα στη βιοπληροφορική (bioinformatics), όπου ως ακολουθίες χρησιμοποιούνται τμήματα DNA, RNA ή πρωτεϊνών.

Η διαδικασία της στοίχισης, όταν συμβαίνει σε συμβολοσειρές μεγάλου μήκους (όπως αυτές που προκύπτουν από τα βιολογικά δεδομένα) είναι μια σχετικά δύσκολη διαδικασία. Στην πράξη χρησιμοποιείται πληθώρα αλγορίθμων, οι περισσότεροι από τους οποίους χρησιμοποιούν την φιλοσοφία του δυναμικού προγραμματισμού (dynamic programming)

Ο βασικός κανόνας της στοίχισης ακολουθιών είναι η τοποθέτηση των δύο (υπό εξέταση) ακολουθιών με τέτοιον τρόπο την μία κάτω από την άλλη, ώστε να ταιριάζουν όσο το δυνατόν περισσότερο. Από αυτό φαίνεται ότι μπορεί κάποια από τα σύμβολα των ακολουθιών να μην ταιριάζουν μετά τη στοίχιση. Το τι γίνεται τότε έχει να κάνει με τον τρόπο με τον οποίο μετράται το "ταίριασμα". Συνήθως γίνεται προσπάθεια να στοιχιστούν οι ακολουθίες με τέτοιον τρόπο, ώστε να μεγιστοποιηθεί κάποιο σκορ. Κάθε ζεύγος συμβόλων που ταιριάζουν μεταξύ τους παίρνει κάποιο θετικό σκορ (π.χ. +1), ενώ κάθε ζεύγος που αποτυγχάνει να ταιριάξει παίρνει κάποιο αρνητικό σκορ (π.χ. -1). Αν επιτρέπεται να εισαχθούν κενοί χαρακτήρες για αποφυγή αποτυχίας σε ένα σημείο, τότε κάθε κενό "κοστολογείται" και αυτό με αρνητικό σκορ (π.χ. -1). Δύο κενά δεν γίνεται να ταιριάζουν μεταξύ τους. Αθροίζοντας όλα τα επιμέρους σκορ υπολογίζεται το συνολικό σκορ της στοίχισης.

Είδη στοίχισης ακολουθιών

Υπάρχουν δύο είδη στοίχισης, η **ολική στοίχιση** (global alignment) και η **τοπική στοίχιση** (local alignment), οι οποίες χρησιμοποιούνται ανάλογα με τον τύπο του προβλήματος που αντιμετωπίζεται.

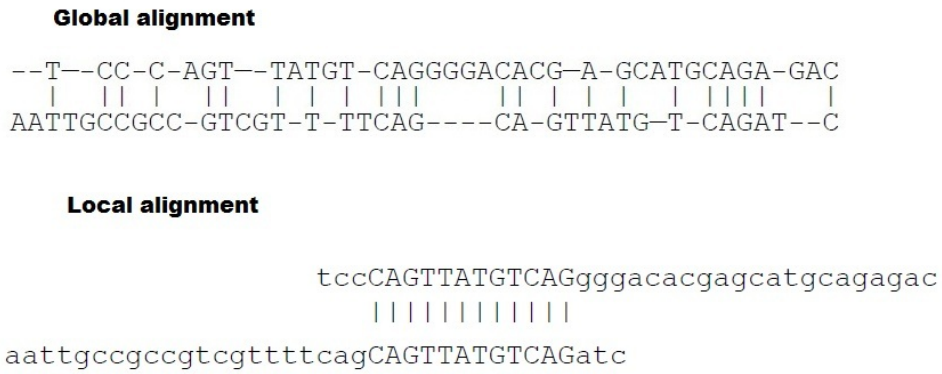
- *Ολική στοίχιση ακολουθιών*

Κατά την ολική στοίχιση ακολουθιών (global sequence alignment) γίνεται προσπάθεια να στοιχιστεί κάθε σύμβολο κάθε ακολουθίας. Οι δύο ακολουθίες στοιχίζονται σε ολόκληρο το μήκος τους με τον καλύτερο δυνατό τρόπο. Κάθε σύμβολο κάθε ακολουθίας, λοιπόν, αντιστοιχίζεται σε ένα σύμβολο της άλλης ή σε ένα κενό. Ένας γνωστός αλγόριθμος για ολική στοίχιση ακολουθιών είναι ο αλγόριθμος Needleman-Wunsch [2]. Καλό θεωρείται στην περίπτωση της ολικής στοίχισης η είσοδος να έχει περίπου το ίδιο μέγεθος με τη βάση με την οποία κάνουμε τη σύγκριση.

- *Τοπική στοίχιση ακολουθιών*

Κατά την τοπική στοίχιση ακολουθιών (local sequence alignment) γίνεται η καλύτερη δυνατή στοίχιση μεταξύ τμημάτων των δύο ακολουθιών. Δηλαδή επιτρέπεται κάποια κομμάτια που "χαλούν" τη στοίχιση να μείνουν εκτός. Ένα παράδειγμα που χρησιμοποιείται συχνά είναι ότι μπορούμε να χρησιμοποιήσουμε την τοπική στοίχιση ακολουθιών προκειμένου να βρούμε τις προτάσεις δύο κειμένων, οι οποίες παρουσιάζουν την περισσότερη ομοιότητα. Ένας γνωστός αλγόριθμος για τοπική στοίχιση ακολουθιών είναι ο αλγόριθμος Smith-Waterman [3].

Στην παρακάτω εικόνα βλέπουμε ένα παράδειγμα των δύο ειδών στοίχισης.



Εικόνα 4 Τα δύο είδη alignment [36]

Στην παραπάνω εικόνα βλέπουμε και σχηματικά σε τι διαφέρει η ολική με την μερική στοίχιση. Τα βιολογικά πορίσματα ποικίλουν μάλιστα αναλόγως με τον τρόπο με τον οποίο κάνουμε τη στοίχιση. Αναλυτικότερα τα αποτελέσματα που παίρνουμε από τις δύο διαφορετικές στοιχίσεις είναι τα εξής (επιτυχία *hit=+1, αποτυχία miss/gap=-1)*:

- *Global alignment*: Ομοιότητα κατά 23/48 (47%)
- *Local alignment*: Ομοιότητα κατά 12/12 (100%) όσον αφορά τη λέξη CAGTTATGTCAG με ένα κομμάτι της βάσης

2.2 Αλγόριθμοι βιοπληροφορικής

Στο κεφάλαιο αυτό θα αναφέρουμε τους βασικούς αλγορίθμους που χρησιμοποιούνται για την στοίχιση ακολουθιών (alignment)

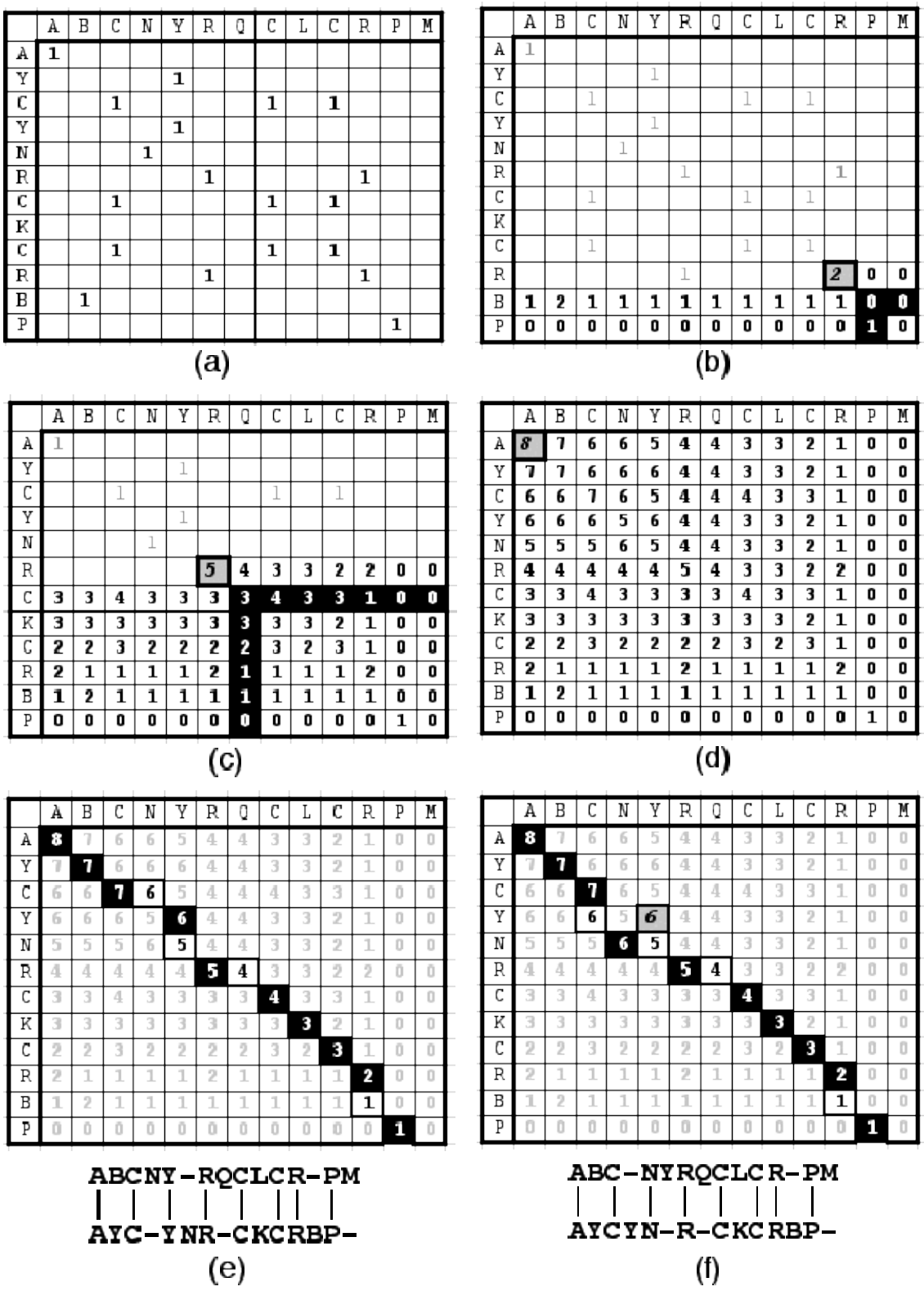
- Needleman-Wunsch
- Smith-Waterman
- FASTA
- BLAST

Needleman-Wunsch

Το 1970 οι Needleman καιWunsch ανέπτυξαν έναν αλγόριθμο βασισμένο στον δυναμικό προγραμματισμό[2]. Στον πυρήνα του αλγορίθμου για ολική στοίχιση ακολουθιών υπάρχει ένας πίνακας του οποίου οι γραμμές αντιστοιχούν στα σύμβολα της μίας ακολουθίας και οι στήλες στα σύμβολα της άλλης. Κάθε κελί αυτού του πίνακα αντιστοιχεί σε ένα ταίριασμα γραμμάτων των δύο ακολουθιών, και περιέχει δύο τιμές: Ένα σκορ (που υπολογίζεται με βάση ένα συγκεκριμένο σχήμα) και έναν δείκτη ο οποίος χρησιμεύει για το traceback. Κατά την εκτέλεση ακολουθείται μια αλληλουχία τριών φάσεων:

1. Αρχικοποίηση πίνακα (Εικόνα 5 a):
2. Γέμισμα του πίνακα με τιμές σκορ (Εικόνα 5 b, c)
3. Οπισθοχώρηση (traceback) και εύρεση καλύτερης στοίχισης (optimum alignment) (Εικόνα 5 e, f).

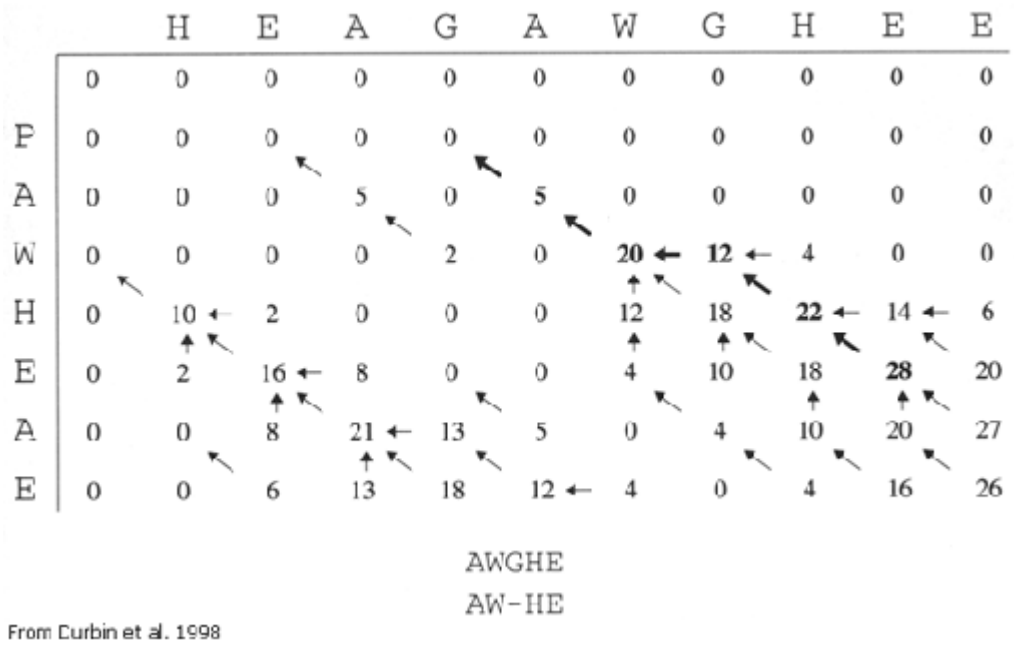
Η υπολογιστική πολυπλοκότητα του αλγορίθμου αυτού είναι O(m*n).



Εικόνα 5 Βήματα του αλγορίθμου Needleman-Wunsch για στοίχιση των ακολουθιών ABCNYRQCLCRPM και AYCYNRCKCRBP [31]. (a) Αρχικοποίηση πίνακα , (b) και (c) γέμισμα πίνακα, (d) ο πίνακας ολοκληρωμένος, (e) και (f) οι καλύτερες στοιχίσεις με ίδιο σκορ.

Smith-Waterman

Ο συγκεκριμένος αλγόριθμος παρουσιάστηκε πρώτη φορά από τον Temple Smith και τον Michael Waterman το 1981[3]. Είναι αλγόριθμος δυναμικού προγραμματισμού για τοπική στοίχιση ακολουθιών. Με άλλα λόγια ο αλγόριθμος καθορίζει όμοιες περιοχές ανάμεσα σε δύο ακολουθίες νουκλεοτιδίων ή πρωτεϊνών. Ως αλγόριθμος δυναμικού προγραμματισμού έχει την δυνατότητα να βρίσκει την βέλτιστη τοπική στοίχιση χρησιμοποιώντας ένα σύστημα σκοραρίσματος. Η βασική του διαφορά στο σημείο αυτό με τον αλγόριθμο Needleman-Wunsch είναι ότι στα κελιά με αρνητική τιμή του πίνακα σκορ (scoring matrix), έχει τοποθετήσει την τιμή μηδέν, πράγμα που κάνει τις τοπικές στοιχίσεις, με θετικό σκορ, ορατές. Η ιχνηλάτηση προς τα πίσω (back tracing) ξεκινά από το κελί του πίνακα με το μεγαλύτερο σκορ και συνεχίζει μέχρι να συναντήσει ένα κελί με σκορ μηδέν, και έτσι βρίσκει τη στοίχιση με το μεγαλύτερο σκορ. Στην παρακάτω εικόνα εναποθέτουμε ένα παράδειγμα.



Εικόνα 6 Τελικός πίνακας αλγορίθμου Smith-Waterman [37]

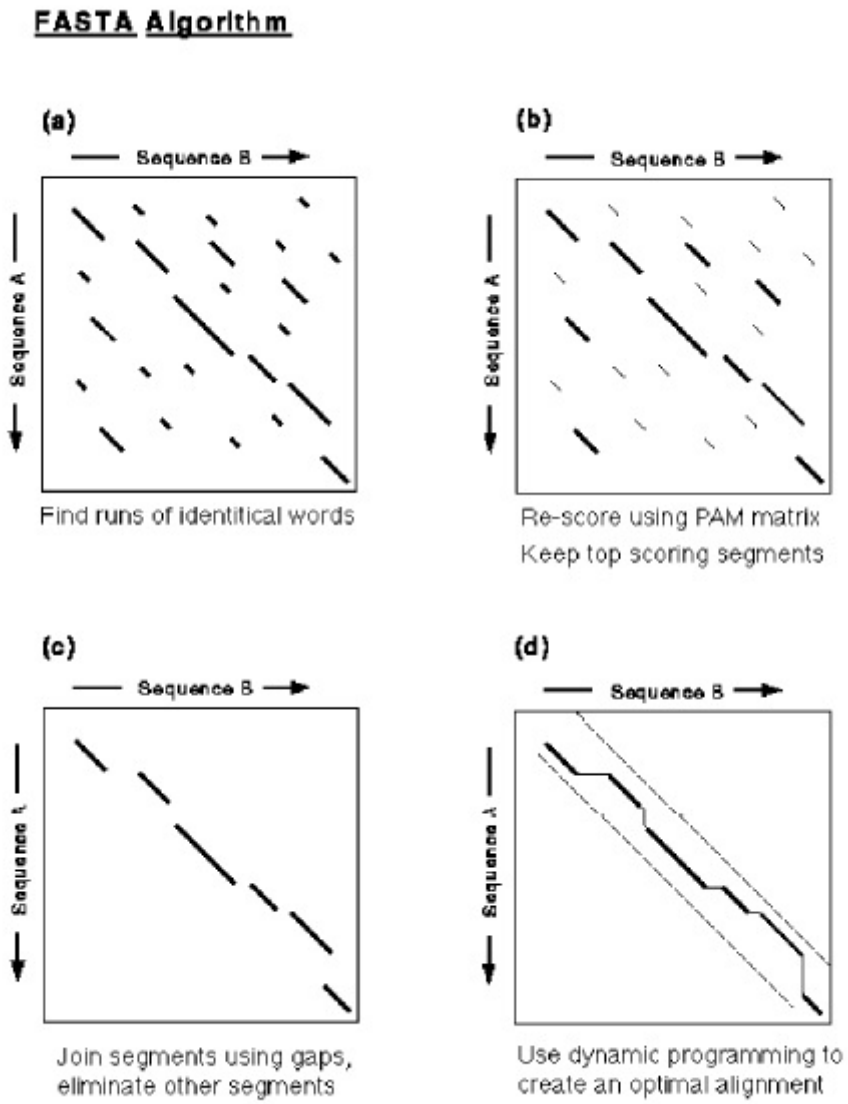
Ο αλγόριθμος Smith-Waterman είναι πολύ απαιτητικός σε πόρους χρόνου και μνήμης. Πιο συγκεκριμένα για να βρει τη στοίχιση δύο ακολουθιών με μέγεθος m και n χρειάζεται O(m*n) χρόνο. Αυτός είναι και ο λόγος που ο αλγόριθμος αυτός δεν χρησιμοποιείται στην πράξη και έχει αντικατασταθεί από άλλους αλγόριθμους όπως ο BLAST ο οποίος δεν έχει βέβαια την ικανότητα να βρίσκει την βέλτιστη στοίχιση αλλά παρόλα αυτά είναι πολύ αποδοτικός.

Μια υλοποίηση του αλγορίθμου σε μια Virtex- 4 FPGA από το Cray Inc μας έδειξε μια επιτάχυνση της τάξης του *100x* σε σύγκριση με έναν επεξεργαστή Opteron στα 2.2Ghz [32].

FASTA

FAST είναι μια οικογένεια προγραμμάτων τα οποία χρησιμοποιούνται για αναζήτηση ακολουθιών σε μια βάση δεδομένων βιο-μοριακών ακολουθιών. Το πρώτο από τα προγράμματα αυτά ήταν το FASTP το οποίο και παρουσιάστηκε το 1985 από τον David J. Lipman και τον William R. Pearson[4]. Το αρχικό FASTP σχεδιάστηκε για εύρεση ομοιοτήτων σε ακολουθίες πρωτεϊνών. Προσθέτοντας στο αρχικό πρόγραμμα FASTP τη δυνατότητα να κάνει αναζητήσεις DNA - DNA και μεταφρασμένη πρωτεΐνη – DNA, το 1988 παρουσιάστηκε το πρόγραμμα FASTA.

Ο αλγόριθμος FASTA ακολουθεί μια «ευριστική» (heuristic) προσέγγιση για την αναζήτηση ομοιοτήτων σε ακολουθίες νουκλεοτιδίων ή πρωτεϊνών, πράγμα που συμβάλει στην ταχύτητα με την οποία και εκτελείται ο αλγόριθμος αυτός.



Εικόνα 7 Βήματα του αλγορίθμου FASTA [38]

Αρχικά δημιουργεί κάποια ζευγάρια αντιστοιχίας λέξη προς λέξη (word-to-word matches) για ένα δοσμένο μέγεθος λέξης, και βρίσκει τα πιθανά σημεία στοίχισης προτού ξεκινήσει την αναζήτηση με έναν αλγόριθμο εύρεσης βέλτιστης λύσης τύπου Smith-Waterman ο οποίος καταναλώνει όμως πολύ χρόνο. Το μέγεθος της λέξης, το οποίο δίνεται από την παράμετρο *ktup (k-tuple)*, ελέγχει την ευαισθησία και την ταχύτητα του προγράμματος. Αυξάνοντας το μέγεθος λέξης μειώνεται ο αριθμός των αρχικών ζευγαριών (hit). Με τον τρόπο αυτό βρίσκει τις περιοχές γύρω από τις οποίες

αξίζει να επικεντρωθεί η αναζήτηση και ψάχνει στις περιοχές αυτές για νέα πιθανά ζευγάρια.

Πρέπει να υπολογιστούν τρία είδη σκορ τα οποία περιγράφουν την ομοιότητα της ακολουθίας. Αυτό γίνεται σε τέσσερα βήματα:

- Εύρεση περιοχών με τη μεγαλύτερη πυκνότητα ζευγαριών. Κρατάμε τις δέκα επικρατέστερες περιοχές. Οι περιοχές αυτές παίρνουν ένα σκορ η καθεμία σύμφωνα με το άθροισμα των ακολουθιών από τις οποίες απαρτίζονται. Το σκορ αυτό ονομάζεται initn.
- Σάρωση για δεύτερη φορά στις περιοχές αυτές χρησιμοποιώντας όμως τώρα έναν πίνακα σκορ για να κρατήσουμε μόνο τις ακολουθίες που μας δίνουν το μεγαλύτερο σκορ. (Εδώ μπορούμε να βρούμε και λέξεις με μέγεθος μικρότερο από αυτό του ktup, και επίσης λέξεις που έχουν και missmatch). Το σκορ αυτό ονομάζεται init1.
- Αν υπάρχουν στοιχίσεις με μεγαλύτερο σκορ από ένα κατώφλι, ελέγχουμε αν είναι γειτονικές και αν μπορούμε να τις ενώσουμε με κάποια κενά. Και στο βήμα αυτό χρησιμοποιείται ο πίνακας σκορ. Για κάθε κενό το πρόστιμο είναι -20.
- Στο βήμα αυτό χρησιμοποιείται ένας αλγόριθμος τύπου Smith-Waterman για να βρεθεί το βέλτιστο σκορ ,το οποίο και ονομάζεται opt, για κάθε ακολουθία.

BLAST

Ο όρος BLAST είναι ακρωνύμιο για την έκφραση Basic Local Alignment Tool δηλαδή Βασικό Εργαλείο Τοπικής Ευθυγράμμισης. Ο αλγόριθμος BLAST σχεδιάστηκε από τον Stephen Altschul, τον Eugene Myers, τον Warren Gish, τον David J. Lipman και τον Webb Miller στο ινστιτούτο NIH και εκδόθηκε το 1990[5]. Τα προγράμματα του BLAST είναι ανάμεσα στα πιο διαδεδομένα προγράμματα όσον αφορά την αναζήτηση σε βάσεις δεδομένων βιολογικού περιεχομένου (Με τον όρο βάση δεδομένων βιολογικού περιεχομένου βιοπληροφορικής εννοούμε ένα μεγάλο σύνολο από ακολουθίες).

Ο BLAST είναι ένας αλγόριθμος που χρησιμοποιείται για την σύγκριση βιολογικών ακολουθιών, που μπορούν να αφορούν από αμινοξέα διαφορετικών πρωτεϊνών μέχρι και τα νουκλεοτίδια από ακολουθίες DNA.

Με το εργαλείο αυτό (BLAST) ο ερευνητής μπορεί να συγκρίνει μια ακολουθία με άλλες που έχουν καταχωρηθεί σε βιβλιοθήκες ή βάσεις δεδομένων ακολουθιών ώστε να αναγνωρίσει ποιες είναι οι ακολουθίες οι οποίες έχουν το καλύτερο ποσοστό ομοιότητας. Επίσης μπορεί να ορίσει ένα κατώφλι (threshold) ώστε να μπορεί ο ίδιος να ρυθμίσει την «ευαισθησία της ομοιότητας».

Πριν εξηγήσουμε λεπτομερειακά τον τρόπο λειτουργίας του BLAST θα έπρεπε πρώτα να δώσουμε τις έννοιες κάποιον όρων οι οποίες θα μας βοηθήσουν στη συνέχεια.

- **Segment**: Με τον όρο αυτό αναφερόμαστε σε ένα κομμάτι της ακολουθίας μας
- **Segment pair**: Δοθεισών δύο ακολουθιών segment pair θεωρείται ένα ζευγάρι από segments ίδιου μήκους, ένα από την κάθε ακολουθία. Τα segment pairs δεν έχουν κενά ανάμεσα τους. Από το σημείο αυτό και μετά μπορούμε να υπολογίσουμε το σκορ της στοίχισης. Στο επόμενο παράδειγμα παρουσιάζεται ένα παράδειγμα από ένα segment pair. Ο υπολογισμός του σκορ γίνεται χρησιμοποιώντας έναν πίνακα σκορ (πχ PAM, BLOSUM).

K	A	L	M	R		
V	A	K	N	S		
−4	3	−4	−3	−1	→	Total: −9

Εικόνα 8 Παράδειγμα υπολογισμού σκορ για τον αλγόριθμο BLAST χρησιμοποιώντας τον PAM 120 [33] ως πίνακα σκορ.

- **Maximum Segment Pair (MSP)**: Με τον όρο MSP αναφερόμαστε στο segment pair το οποίο έχει επιτύχει το υψηλότερο σκορ. Το σκορ αυτό είναι ένα μέτρο ομοιότητας και μπορεί να υπολογιστεί πολύ εύκολα χρησιμοποιώντας τη μέθοδο του δυναμικού προγραμματισμού. Παρόλα αυτά ο αλγόριθμος BLAST μπορεί να κάνει μια πολύ καλή εκτίμηση του αριθμού αυτού πολύ γρηγορότερα από τη μέθοδο δυναμικού προγραμματισμού.

Σε γενικές γραμμές εάν μας δοθεί μια ακολουθία ως είσοδος (query) και ένα κατώφλι (S) ο αλγόριθμος BLAST μας επιστρέφει όλα τα segment pairs ανάμεσα στο query και τη βάση δεδομένων που έχουν σκορ μεγαλύτερο του κατωφλίου S. Το κατώφλι μπορεί να ποικίλει αναλόγως με τις απαιτήσεις του χρήστη. Όπως αναφέραμε πριν οι στοιχίσεις αυτές δεν παρουσιάζουν κενά. Αυτός είναι ένας από τους λόγους που ο BLAST είναι τόσο γρήγορος, επειδή για καλές στοιχίσεις με κενά καταναλώνεται περισσότερος χρόνος.

Για να υπολογίσει ο BLAST τα segment pairs με μεγάλο σκορ βρίσκει πρώτα μικρά segments pairs ανάμεσα στη βάση δεδομένων και την είσοδο τα οποία και αποκαλούμε ***seeds***. Αμέσως μετά ο BLAST επεκτείνει (extends) τα ζευγάρια (seeds) αυτά προς τις δύο κατευθύνσεις χωρίς κενά μέχρι να επιτύχει το μεγαλύτερο δυνατό σκορ για τις επεκτάσεις αυτές. Ο BLAST έχει συγκεκριμένα κριτήρια για να σταματήσει την επέκταση αν το σκορ πέσει κάτω από προκαθορισμένο όριο.

Είναι λογικό να πούμε ότι ο BLAST έχει 3 βασικά στάδια¨

1. Βρίσκει πρώτα όλα τα w-mers του query
2. Κάνει το seeding, δηλαδή ψάχνει για αρχικά hits
3. Κάνει extend τα seeds που βρήκε προς τις δύο κατευθύνσεις.

Τα παραπάνω βήματα βέβαια διαφέρουν σχετικά με το είδος των ακολουθιών που συγκρίνονται (Πρωτεΐνες ή DNA).

Για πρωτεϊνικές ακολουθίες η λίστα με τις λέξεις που έχουν επιτύχει υψηλό σκορ, αποτελείται από όλες τις λέξεις μεγέθους ‘w’ (ονομάζονται w-mers), τα οποία έχουν πετύχει σκορ τουλάχιστον όσο το κατώφλι T που θέσαμε χρησιμοποιώντας έναν από τους πίνακες σκορ (substitution matrix) για να υπολογίσουμε το σκορ. W και T είναι οι βασικοί παράμετροι του προγράμματος. Η λίστα όλων των w-mers που έχουν βρεθεί μέσα στη βάση δεδομένων δεν είναι ανάγκη να περιέχει όλα τα w-mers του query. Παρόλα αυτά υπάρχει τρόπος να συγκαταλέξουμε όλα τα w-mers. Η πιο κοινή τιμή για το μέγεθος λέξης είναι 4 για αναζητήσεις πρωτεϊνών. Μπορούμε να σαρώσουμε τη βάση ψάχνοντας για hit βασισμένοι στη λίστα που δημιουργήσαμε προηγουμένως χρησιμοποιώντας δύο διαφορετικές μεθόδους. Η πρώτη μέθοδος είναι να ταξινομήσουμε πρώτα τη λίστα αυτή σε έναν hash table. Έτσι για κάθε λέξη της βάσης μεγέθους w, είναι πολύ εύκολο να βρούμε το index στον hash table και να το

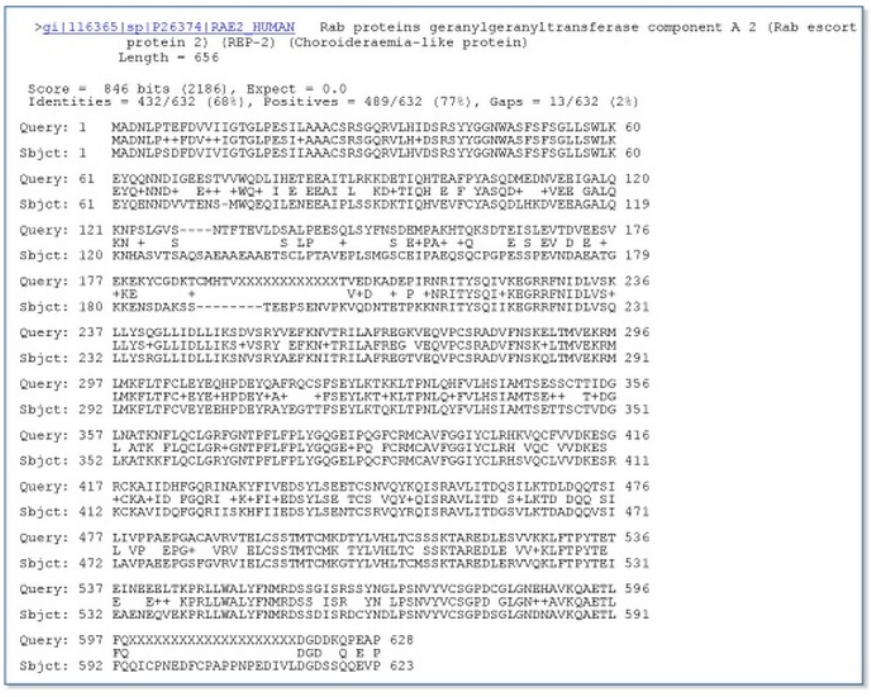
συγκρίνουμε με τις λέξεις εκεί. Η δεύτερη μέθοδος είναι να χρησιμοποιήσουμε ένα ντετερμινιστικό αυτόματο για να ψάξουμε για hits. Το αυτόματο αυτό ξεκινά από μια αρχική κατάσταση και μετά για κάθε χαρακτήρα της βάσης περνάει στην επόμενη κατάσταση. Αναλόγως την κατάσταση και τη μετάβαση αναγνωρίζεται μια λέξη από τη λίστα μας. Το τελικό στάδιο του αλγορίθμου για ακολουθίες πρωτεϊνών είναι απλό: Τα seeds που βρήκαμε πριν επεκτείνονται προς τις δύο κατευθύνσεις. Για να εξοικονομήσουμε χρόνο ο αλγόριθμος σταματά μόλις το σκορ πέσει κάτω από ένα κατώφλι T. Το segment pair το οποίο έχει επιτύχει το μεγαλύτερο σκορ κρατείται. Υπάρχει βέβαια η πιθανότητα να χαθούν κάποιες σημαντικές επεκτάσεις με την προσέγγιση αυτή, αλλά η πιθανότητα αυτή είναι αμελητέα.

Για ακολουθίες DNA η αρχική λίστα συγκρίνεται μόνο με τα w-mers του query. Ο τρόπος που σαρώνεται η βάση είναι πολύ διαφορετικός. Αυτό οφείλεται κυρίως στο γεγονός ότι για συγκρίσεις ακολουθιών DNA χρησιμοποιούμε ένα αλφάβητο που αποτελείται από τέσσερα γράμματα, οπότε χρειαζόμαστε μόνο 2 bits για την αναπαράσταση κάθε γράμματος, και θέλουμε συνολικά 1 byte για την αναπαράσταση 4 χαρακτήρων. Εκτός από το γεγονός ότι κάνουμε οικονομία χώρου μπορούμε για την αναζήτηση μας να χρησιμοποιήσουμε 1 byte οπότε η αναζήτηση μπορεί να γίνει πολύ γρήγορα. Η επέκταση γίνεται με τον ίδιο τρόπο με τις πρωτεΐνες.

Παρακάτω βλέπουμε τη μορφή των αποτελεσμάτων του BLAST. Στην πρώτη εικόνα βλέπουμε τη λίστα με τις στοιχίσεις που πέτυχαν το μεγαλύτερο σκορ, και στη δεύτερη βλέπουμε μια από τις στοιχίσεις.

Sequences producing significant alignments:		Score (bits)	E Value
(a)	(b)	(c)	(d)
gi|116365|sp|P26374|RAE2_HUMAN	Rab proteins geranylgeranylt...	1216	0.0
gi|21431807|sp|P24386|RAE1_HUMAN	Rab proteins geranylgerany...	879	0.0
gi|585775|sp|P37727|RAE1_RAT	Rab proteins geranylgeranyltra...	846	0.0
gi|13626886|sp|Q61598|GDIC_MOUSE	RAB GDP dissociation inhib...	127	5e-29
gi|729566|sp|P39958|GDI1_YEAST	SECRETORY PATHWAY GDP DISSOC...	127	5e-29
gi|13626813|sp|O97556|GDIB_CANFA	Rab GDP dissociation inhib...	126	1e-28
gi|13638229|sp|P50397|GDIB_MOUSE	RAB GDP dissociation inhib...	125	3e-28
gi|1707888|sp|P50398|GDIA_RAT	RAB GDP dissociation inhibito...	124	7e-28
gi|121108|sp|P21856|GDIA_BOVIN	Rab GDP dissociation inhibit...	124	7e-28
gi|21903424|sp|P50396|GDIA_MOUSE	Rab GDP dissociation inhib...	124	7e-28
gi|13626812|sp|O97555|GDIA_CANFA	RAB GDP dissociation inhib...	124	8e-28
gi|1707886|sp|P31150|GDIA_HUMAN	Rab GDP dissociation inhibi...	123	9e-28
gi|13638228|sp|P50395|GDIB_HUMAN	Rab GDP dissociation inhib...	122	2e-27
gi|1707891|sp|P50399|GDIB_RAT	RAB GDP DISSOCIATION INHIBITO...	121	5e-27
gi|1723467|sp|Q10305|YD4C_SCHPO	Putative secretory pathway ...	120	8e-27
gi|585776|sp|P32864|RAEP_YEAST	RAB proteins geranylgeranylt...	97	7e-20
gi|10720243|sp|O93831|RAEP_CANAL	RAB proteins geranylgerany...	74	9e-13
gi|2498411|sp|Q49398|GLF_MYCGE	UDP-galactopyranose mutase	35	0.63
gi|11135401|sp|Q9XBQ9|STHA_AZOVI	Soluble pyridine nucleotid...	34	1.0
gi|11135075|sp|O05139|STHA_PSEFL	Soluble pyridine nucleotid...	33	1.3
gi|11135195|sp|P57112|STHA_PSEAE	Soluble pyridine nucleotid...	33	1.8
gi|22257022|sp|Q8TZJ8|RLA0_PYRFU	Acidic ribosomal protein P...	33	2.1
gi|3915516|sp|P94488|YNAJ_BACSU	Hypothetical symporter ynaJ	32	3.4
gi|231788|sp|P30599|CHS2_USTMA	CHITIN SYNTHASE 2 (CHITIN-UD...	32	3.7
gi|2498412|sp|P75499|GLF_MYCPN	UDP-galactopyranose mutase	32	4.2
gi|547891|sp|P36225|MAP4_BOVIN	Microtubule-associated prote...	32	4.2
gi|586602|sp|P37747|GLF_ECOLI	UDP-galactopyranose mutase	32	4.6

Εικόνα 9 Λίστα με τα σημαντικότερα αποτελέσματα alignment του BLAST ως έξοδος



```
>gi|116365|sp|P26374|RAE2_HUMAN  Rab proteins geranylgeranyltransferase component A 2 (Rab escort
          protein 2) (REP-2) (Choroideraemia-like protein)
         Length = 656

 Score =  846 bits (2186), Expect = 0.0
 Identities = 432/632 (68%), Positives = 489/632 (77%), Gaps = 13/632 (2%)

Query: 1   MADNLPTEFDVVIIGTGLPESILAAACSRSGQRVLHIDSRSYYGGNWASFSFSGLLSWLK 60
           MADNLP++FDV++IGTGLPESI+AAACSRSGQRVLH+DSRSYYGGNWASFSFSGLLSWLK
Sbjct: 1   MADNLPSDFDVIVIGTGLPESIIAAACSRSGQRVLHVDSRSYYGGNWASFSFSGLLSWLK 60

Query: 61  EYQQNNDIGEESTVVWQDLIHETEEAITLRKKDETIQHTEAFPYASQDMEDNVEEIGALQ 120
           EYQ+NND+  E++ +WQ+ I E EEAI L  KD+TIQH E F YASQD+  +VEE GALQ
Sbjct: 61  EYQENNDVVTENS-MWQEQILENEEAIPLSSKDKTIQHVEVFCYASQDLHKDVEEAGALQ 119

Query: 121 KNPSLGVS----NTFTEVLDSALPEESQLSYFNSDEMPAKHTQKSDTEISLEVTDVEESV 176
           KN +   S          S LP   +     S E+PA+ +Q    E S EV D E +
Sbjct: 120 KNHASVTSAQSAEAAEAABTSCLPTAVEPLSMGSCEIPAEQSQCPGPESSPEVNDAEATG 179

Query: 177 EKEKYCGDKTCMHTVXXXXXXXXXXXXTVEDKADEPIRNRITYSQIVKEGRRFNIDLVSK 236
           +KE       *                V+D  + P +NRITYSQI+KEGRRFNIDLVS+
Sbjct: 180 KKENSDAKSS--------TEEPSENVPKVQDNTETPKKNRITYSQIIKEGRRFNIDLVSQ 231

Query: 237 LLYSQGLLIDLLIKSDVSRYVEFKNVTRILAFREGKVEQVPCSRADVFNSKELTMVEKRM 296
           LLYS+GLLIDLLIKS+VSRY EFKN+TRILAFREG VEQVPCSRADVFNSK+LTMVEKRM
Sbjct: 232 LLYSRGLLIDLLIKSNVSRYAEFKNITRILAFREGTVEQVPCSRADVFNSKQLTMVEKRM 291

Query: 297 LMKFLTFCLEYEQHPDEYQAFRQCSPSEYLKTKKLTFNLQHFVLHSIAMTSESSCTTIDG 356
           LMKFLTFC+EYE+HPDEY+A+   +FSEYLKT+KLTFNLQ+FVLHSIAMTSE++  T+DG
Sbjct: 292 LMKFLTFCVEYEEHPDEYRAYEGTTFSEYLKTQKLTFNLQYFVLHSIAMTSETTSCTVDG 351

Query: 357 LNATKNFLQCLGRFGNTPFLFPLVGQGEIPQGFCRMCAVFGGIYCLRHKVQCFVVDKESG 416
           L ATK FLQCLGR+GNTPFLFPLVGQGE+PQ FCRMCAVFGGIYCLRH VQC VVDKES
Sbjct: 352 LKATKKFLQCLGRYGNTPFLFPLVGQGELPQCFCRMCAVFGGIYCLRHSVQCLVVDKESR 411

Query: 417 RCKAIIDHFGQRINAKYFIVEDSYLSEETCSNVQYKQISRAVLITDQSILKTDLDQQTSI 476
           +CKA+ID FGQRI +K+FI+EDSYLSE TCS VQY+QISRAVLITD S+LKTD DQQ SI
Sbjct: 412 KCKAVIDQFGQRIISKHFIIEDSYLSENTCSRVQYRQISRAVLITDGSVLKTDADQQVSI 471

Query: 477 LIVPPAEPGACAVRVTELCSSTMTCMKDTYLVHLTCSSSKTAREDLESVVKKLFTPYTET 536
           L VP  EPG+  VRV ELCSSTMTCMK TYLVHLTC SSKTAREDLE VV+KLFTPYTE
Sbjct: 472 LAVPAEEPGSFGVRVIELCSSTMTCMKGTYLVHLTCMSSKTAREDLERVVQKLFTPYTEI 531

Query: 537 EINEEELTKPRLLWALYFNMRDSSGISRSSYNGLPSNVYVCSGPDCGLGNEHAVKQAETL 596
           E   E++ KPRLLWALYFNMRDSS ISR  YN LPSNVYVCSGPD GLGN++AVKQAETL
Sbjct: 532 EAENEQVEKPRLLWALYFNMRDSSDISRDCYNDLPSNVYVCSGPDSGLGNDNAVKQAETL 591

Query: 597 FQXXXXXXXXXXXXXXXXXXXDGDDKQPEAP 628
           FQ                  DGD  Q E P
Sbjct: 592 FQQICPNEDFCPAPPNPEDIVLDGDSSQQEVP 623
```

Εικόνα 10Έξοδος του BLAST για μια στοίχιση

Παρακάτω παρουσιάζονται τα διαφορετικά προγράμματα του BLAST και η χρησιμότητα τους όπως τα έχουμε βρει στο επίσημο site του NCBI:

blastn - συγκρίνει μια νουκλεοτιδική ακολουθία (DNA) με μια βάση νουκλεοτιδικών ακολουθιών (DNA). Η αναζήτηση γίνεται και στις δύο αλυσίδες. Είναι ένα πρόγραμμα βελτιστοποιημένης ταχύτητας, όχι όμως και ευαισθησίας.

blastp - συγκρίνει την ζητούμενη αμινοξική ακολουθία με μια βάση πρωτεϊνικών ακολουθιών (σύγκριση πρωτεΐνης με πρωτεΐνες).

blastx - συγκρίνει μια άγνωστη νουκλεοτιδική ακολουθία (DNA) μεταφρασμένη σε όλα τα πλαίσια ανάγνωσης (reading frames) με μια βάση πρωτεϊνικών ακολουθιών του NCBI. Χρησιμοποιείται για την έυρεση πιθανών μεταφρασμένων πρωτεϊνικών προϊόντων μιας άγνωστης νουκλεοτιδικής ακολουθίας.

tblastn - συγκρίνει την ζητούμενη πρωτεϊνική ακολουθία με μια βάση νουκλεοτιδικών ακολουθιών (DNA) του NCBI που μεταφράζεται δυναμικά σε όλα τα πλαίσια ανάγνωσης (reading frames).

tblastx - μετατρέπει μια νουκλεοτιδική ακολουθία (DNA) σε μια πρωτεϊνική ακολουθία σε όλα τα πλαίσια ανάγνωσης (reading frames) και μετά τη συγκρίνει με μια βάση νουκλεοτιδικών ακολουθιών του NCBI η οποία έχει μεταφραστεί σε όλα τα πλαίσια ανάγνωσης (reading frames).

BLAST2 - Ονομάζεται εξελιγμένο (advanced) BLAST. Εκτελεί στοιχίσεις που περιέχουν κενά (gapped alignments).

MEGABLAST - είναι ένα πρόγραμμα που χρησιμοποιεί έναν πλεονεκτικό ("greedy algorithm") αλγόριθμο (Miller *et al*, 2000), για αναζήτηση στοίχισης νουκλεοτιδικών ακολουθιών. Χρησημοποιείται για στοίχιση ακολουθιών με μικρές διαφορές και είναι 10 φορές γρηγορότερο από παρόμοια προγράμματα. Ενδείκνυται για σύγκριση μεταξύ μεγάλων ακολουθιών.

PSI-BLAST - (Position Specific Iterated BLAST) χρησιμοποιεί σταθερή αναζήτηση, στην οποία οι ακολουθίες που θα βρεθούν στον πρώτο γύρο αναζητήσεων

χρησιμοποιούνται για να χτίσουν ένα αποτελεσματικό μοντέλο για τους επόμενους κύκλους αναζητήσεων.

PHI-BLAST - (Pattern Hit Initiated BLAST) συνδυάζει το ταίριασμα ενός πρότυπου φυσιολογικής έκφρασης με μια συγκεκριμένη θέση που επαναλαμβάνεται στην πρωτεϊνική ακολουθία.

RPS-BLAST - συγκρίνει μια πρωτεϊνική ακολουθία ως προς την βάση Conserved Domain Database (CD-Search).

2.3 Υλοποιήσεις αλγορίθμων βιοπληροφορικής σε FPGA

Τα κομμάτι της κοινότητας των ακαδημαϊκών που ασχολείται με αναδιατασσόμενη λογική χρησιμοποίησε το πρόβλημα της σύγκρισης ακολουθιών DNA και την αναζήτηση σε μια βάση, για να δείξει πως βαριά υπολογιστικά προβλήματα μπορούν να λυθούν χρησιμοποιώντας FPGA. Η πλατφόρμα Splash 2 χρησιμοποιήθηκε τη δεκαετία του 1990 από τον Hoang [6][7] για να λύσει αυτά τα προβλήματα χρησιμοποιώντας τον αλγόριθμο Smith-Waterman. Αργότερα ο Guccione [8] χρησιμοποίησε την Jbits technology και σε συνεργασία με δύο τμήματα, τα Virginia Tech Configurable Computing Laboratory και το Nanyang Technological University, υλοποίησε για ακόμη μια φορά τον αλγόριθμο Smith-Waterman για την λύση του ίδιου προβλήματος[9][10].

Προηγούμενες προσπάθειες για τον αλγόριθμο BLAST

Λίγες ακαδημαϊκές προσπάθειες έγιναν για την υλοποίηση του αλγορίθμου BLAST μέχρι στιγμής. Η πρώτη ήταν ο RC_BLAST[10] όπου αν και υλοποιήθηκε το βαρύ κομμάτι του NCBI BLAST τελικά η συνολική απόδοση του συστήματος δεν φάνηκε γρηγορότερη από αυτή του software και τελικά δεν έγιναν περισσότερες προσπάθειες προς αυτή την κατεύθυνση.

Το 2005 έγινε ακόμη μια προσπάθεια για σύγκριση ακολουθιών DNA από το εργαστήριο CAAD στο Boston University. Πιο συγκεκριμένα υλοποίησανε τον αλγόριθμο BLAST για εισόδους μέχρι 800 χαρακτήρες[11] και αργότερα επεκτάθηκε[12], παρουσιάζοντας μεγάλη επιτάχυνση σε σύγκριση με την υλοποίηση σε software. Το πρόγραμμα αυτό ονομάστηκε Tree_BLAST και αργότερα το 2009 παρουσιάστηκε μια επέκταση του προγράμματος αυτού όπου φιλτράρεται η βάση[22]. Η μέθοδος αυτή είχε ήδη υλοποιηθεί και εκδοθεί από το Πολυτεχνείο Κρήτης το 2008[23].

Το Washington University παρουσίασε την αρχιτεκτονική Mercury BLAST[13][14][15], υλοποιώντας τον BLASTn και παρουσιάζοντας μια ικανοποιητική επιτάχυνση συγκριτικά με το software σε έναν υπολογιστή γενικής χρήσης. Το project του Mercury BLAST συνεχίζει ακόμη[16]. Ο Mercury BLAST χρησιμοποιεί φίλτρα BLOOM για την υλοποίηση ενώ εμείς στο Πολυτεχνείο Κρήτης τα χρησιμοποιήσαμε για την οργάνωσης της βάσης.

Μια πρόσφατη υλοποίηση του BLAST χρησιμοποιώντας αναδιατασσόμενη λογική παρουσιάστηκε από το IRISA και CNRS της Γαλλίας και το ICT του Πεκίνου[17]. Μια νέα υλοποίηση του αλγορίθμου BLAST με όνομα Multi-seed / Multi-Channel BLAST[18][19] από το Εθνικό Πανεπιστήμιο Τεχνολογίας της Κίνας μας δείχνει πολύ ενδιαφέροντα αποτελέσματα όσον αφορά μια γενική σχεδίαση για τον αλγόριθμο. Προσθέσαν μια FLASH μνήμη στο σύστημα αναδιατασσόμενης λογικής που έφτιαξαν και κατάφεραν να υλοποιήσουν τους BLASTx, TBLASTn, TBLASTx αλγόριθμους.

Παράλληλα με τις προσπάθειες που έγιναν από ιδρύματα έχουν υπάρξει και άλλες προσπάθειες από εταιρίες που ασχολούνται με τον τομέα της αναδιατασσόμενης λογικής. Η εταιρία Timelogic Inc. μας αναφέρει πολύ σημαντικά αποτελέσματα για το σύστημα της, το DeCypher[20] που υλοποιεί τον αλγόριθμο BLAST, χωρίς όμως να μας αναφέρει πολλές τεχνικές λεπτομέριες για την υλοποίηση αυτή, με αποτέλεσμα να μη μπορούν να συγκρίνουν με άλλους τα αποτελέσματα τους. Το UC Berkeley δημιούργησε το έναν υπερυπολογιστή, το BEE και χρησιμοποίησε τον BLAST ως ένα από τα προγράμματα του[21]. Και εδώ έχουμε ελλείψεις λεπτομεριών για την τεχνολογία που χρησιμοποιήθηκε και δεν μπορούμε να συγκρίνουμε τα αποτελέσματα τους με άλλα. Το 2007 η Silicon Graphics Inc. σε συνεργασία με την

εταιρία λογισμικού Mitrionics έκαναν μια τελευταία προσπάθεια για υλοποίηση του αλγορίθμου[21], όμως και πάλι η πολιτική της Timelogic Inc. δεν τους επέτρεψε να δημοσιοποιήσουν τις λεπτομέρειες.

Το Πολυτεχνείο Κρήτης έχει υλοποιήσει διάφορες αρχιτεκτονικές για τον αλγόριθμο BLAST σε αναδιατασσόμενη λογική από το 2005 έως σήμερα [26-30]

3 Μελέτη χαρακτηριστικών (Profiling) NCBI BLAST

Σε αυτό το κεφάλαιο περιγράφεται η αλγοριθμική ανάλυση του αλγορίθμου NCBI BLAST. Επίσης, παρουσιάζεται η απόδοση του αλγορίθμου για διάφορα πειράματα που πραγματοποιήθηκαν με διάφορα δεδομένα εισόδου καθώς επίσης και οι χρόνοι εκτέλεσης των πιο βαριών υπολογιστικά ρουτινών του αλγορίθμου.

Profiling είναι η διαδικασία κατά την οποία μπορούμε πειραματικά να μετρήσουμε την απόδοση ενός αλγορίθμου αλλά και την απόδοση των ρουτινών από τις οποίες αποτελείται ο αλγόριθμος. Μέσω του profiling μπορούμε να δούμε επίσης τη ροή του αλγορίθμου και να βρούμε και τα σημεία όπου ο αλγόριθμος καταναλώνει πολύ χρόνο (hotspots).

Για να βρούμε την πιο βαριά υπολογιστικά συνάρτηση του αλγορίθμου NCBI BLASTn κάναμε το profiling του αλγορίθμου. Ο σκοπός μας ήταν να αντικαταστήσουμε την πιο βαριά υπολογιστικά συνάρτηση με μία αντίστοιχη υλοποίηση στο hardware έτσι ώστε να μειώσουμε τον συνολικό χρόνο εκτέλεσης του αλγορίθμου

3.1 Profiling NCBI BLAST

Ο αλγόριθμος NCBI BLAST μπορεί να εκτελεστεί σε ένα μεγάλο εύρος από πλατφόρμες και λειτουργικά συστήματα. Η πλατφόρμα που χρησιμοποιήθηκε για τα πειράματα μας ήταν ένα PC με επεξεργαστή Intel Core 2 2.8 GHz, 1 GByte RAM με Ethernet PHY 1000/100/10 Mbps και με **UBUNTU 10.04 LTS** 32bit λειτουργικό σύστημα. Πραγματοποιήσαμε μεγάλο αριθμό πειραμάτων εισάγοντας διαφορετικές παραμέτρους κάθε φορά ώστε να εμβαθύνουμε στη μελέτη της απόδοσης του αλγορίθμου.

Η έκδοση του αλγορίθμου που χρησιμοποιήσαμε ήταν η **NCBI Blast 2.2.24** η οποία διανέμεται από την επίσημη ιστοσελίδα του NCBI. (http://www.ncbi.nlm.nih.gov/). Για την μέτρηση της απόδοσης του αλγορίθμου χρησιμοποιήθηκε το πρόγραμμα **INTEL VTUNE Amplifier XE 2011.**

Παρακάτω παρουσιάζονται οι βασικές παραμέτρους του αλγορίθμου που χρησιμοποιήθηκαν στα πειράματα μας. Στο παράρτημα παραθέτουμε όλες τις παραμέτρους που μας διατίθενται για το πρόγραμμα NCBI BLAST.

Παράμετροι	
p	Όνομα προγράμματος (string)
d	Η βάση που θα χρησιμοποιήσουμε
i	Η είσοδος μας, δηλαδή το query που αναζητούμε στη βάση (αρχείο FASTA)
o	Το τελικό αρχείο, η έξοδος μας. (default stdout)
W	Το μέγεθος λέξης με το οποίο κάνουμε το seed (integer)
g	Παράμετρος για gapped αναζήτηση (T/F)
F	Φίλτρα και μάσκες (T/F)
e	expectation value

Πίνακας 1 Παράμετροι NCBI BLAST

Σε όλα τα πειράματα επικεντρωθήκαμε στην ungapped αναζήτηση χωρίς φίλτρα και μάσκες. Επίσης, χρησιμοποιούσαμε την default τιμή για expectation value η οποία είναι ίση με 10.

Μια χαρακτηριστική αναζήτηση που κάναμε ήταν η εξής:

Blastall **–p** *blastn* **–d** *nt* **–i** *sequence1* **–o** *output* **–g** *F* **–F** *F*

Να τονίσουμε ότι το προεπιλεγμένο μέγεθος λέξης το οποίο δεν διευκρινίζεται είναι ίσο με 11.

Σετ δεδομένων

Τα σετ δεδομένων που χρησιμοποιήσαμε είναι: 5 διαφορετικά queries και 4 διαφορετικές databases. Τα δεδομένα αυτά κρίθηκαν ικανοποιητικά διότι καταλαμβάνουν όλο το εύρος των πιθανών μεγεθών των sequences που χρησιμοποιούνται από τους βιολόγους καθώς και τις πιο ευρέως χρησιμοποιούμενες γενετικές βάσεις δεδομένων.

Input name	Input Length (chars)
Sequence 1	700
Sequence 2	1400
Sequence3	3650
Sequence 4	5040
Sequence 5	7500

ηΠίνακας 2 Queries που χρησιμοποιήσαμε για τις μετρήσεις

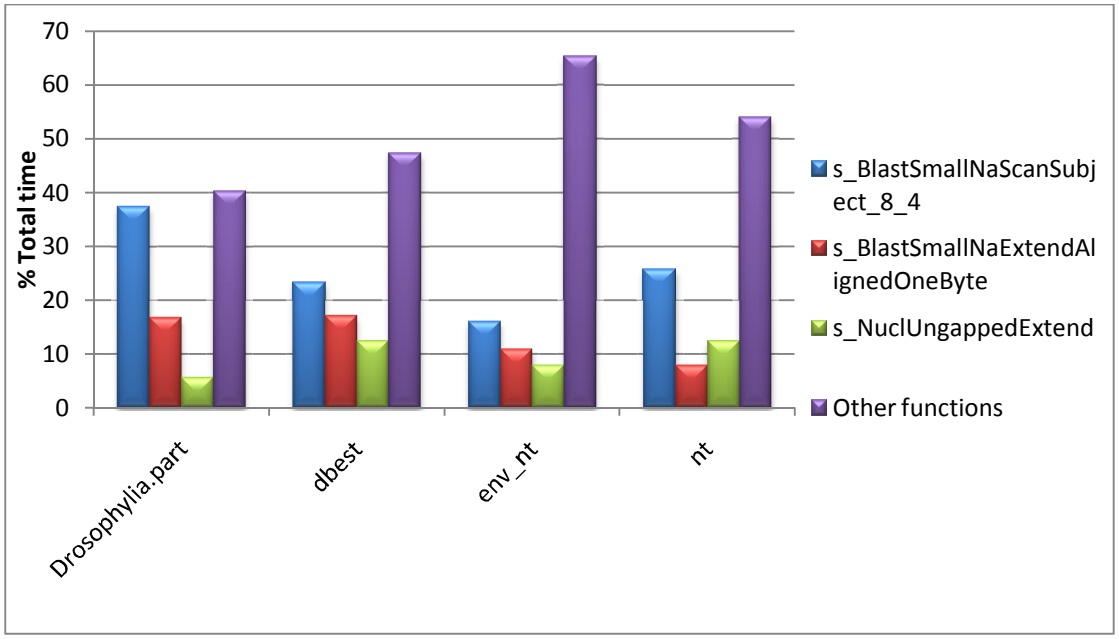
Database name	Database Length (chars)
Drosophylia.part	122.655.632
dbest	341.535.537
env_nt	13.869.239.821
nt	30.340.628.936

Πίνακας 3 Βάσεις δεδομένων που χρησιμοποιήσαμε για τις μετρήσεις

Τα queries και οι βάσεις δεδομένων που χρησιμοποιήθηκαν για τα πειράματα διατίθενται μέσω του NCBI και βρίσκονται σε FASTA μορφή, η οποία είναι κατάλληλη για την εκτέλεση του NCBI Blast αλγορίθμου.

Μετρήσεις

Αρχικά, χρησιμοποιήσαμε ως είσοδο query στον αλγόριθμο το sequence 3, που είναι μεγέθους 3650 χαρακτήρων, σε συνδυασμό με όλες τις παραπάνω γενετικές βάσεις δεδομένων. Το sequence 3 θεωρείται μια είσοδος μέσου μεγέθους ενώ το μέγεθος λέξης που χρησιμοποιήσαμε ήταν το προεπιλεγμένο (word_size=11). Τα αποτελέσματα που πήραμε παρουσιάζονται στην παρακάτω εικόνα:



Εικόνα 11 Ποσοστιαία ανάλυση χρόνου εκτέλεσης ανά συνάρτηση για το sequence 3 αλλάζοντας τη βάση δεδομένων

Επίσης, ο συνολικός χρόνος εκτέλεσης του αλγορίθμου για τα διαφορετικά πειράματα εμφανίζονται στον παρακάτω πίνακα.

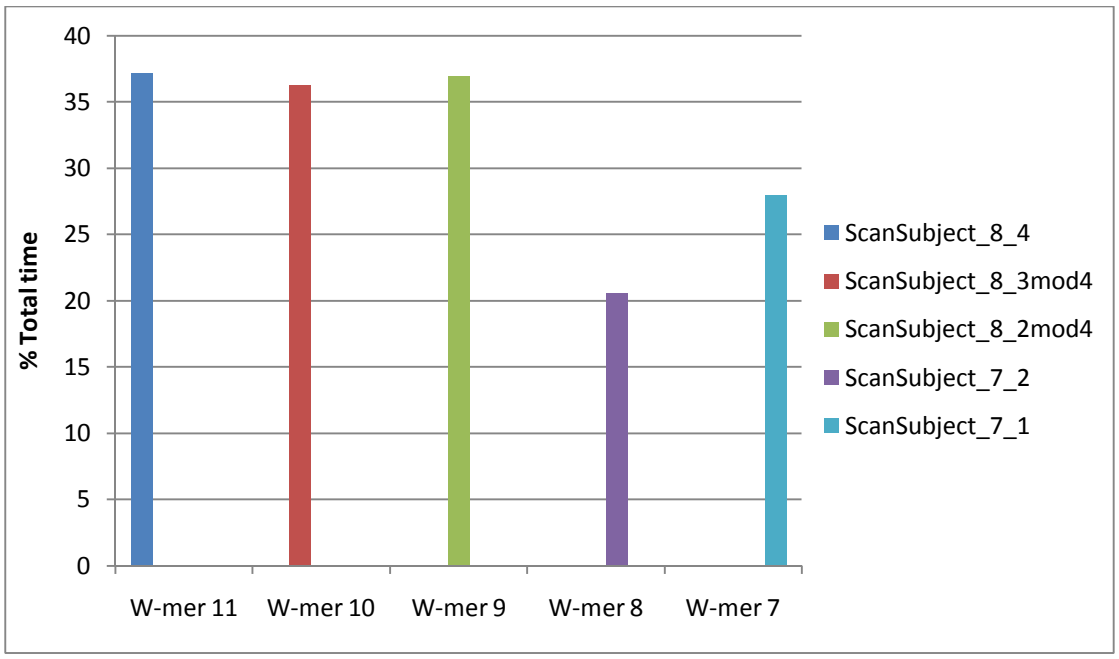
Database name	Sec	Μέγεθος database (chars)
Drosophylia.part	0,67	122.655.632
dbest	2,66	341.535.537
env_nt	75,76	13.869.239.821
nt	235,13	30.340.628.936

Πίνακας 4 Χρόνοι εκτέλεσης του BLASTn αλλάζοντας τις βάσεις δεδομένων

Παρατηρούμε ότι υπάρχει μια μεγάλη διαφορά στο χρόνο εκτέλεσης του προγράμματος όσο αυξάνεται το μέγεθος της βάσης που χρησιμοποιήσαμε. Αργότερα στην ανάλυση του αλγορίθμου θα εξηγήσουμε γιατί συμβαίνει αυτό. Επίσης από το διάγραμμα φαίνεται ότι η συνάρτηση που χρειάζεται για όλα σχεδόν τα πειράματα τον μεγαλύτερο χρόνο εκτέλεσης είναι η s_BlastSmallNaScan_Subject_8_4.

Στην δεύτερη σειρά πειραμάτων που πραγματοποιήσαμε, χρησιμοποιήθηκαν μία σταθερή γενετική βάση δεδομένων(dbest) και το sequence 1 ως είσοδο στον αλγόριθμο. Πραγματοποιήσαμε διάφορα πειράματα αλλάζοντας το μέγεθος λέξης με το οποίο κάνουμε την αναζήτηση. Παρατηρήσαμε ότι αλλάζοντας το μέγεθος της

λέξης άλλαζε και το σετ συναρτήσεων που χρησιμοποιούσε ο BLASTn αλγόριθμος. Αναγνωρίσαμε, όμως, ότι όλες αυτές οι συναρτήσεις είχαν μια κοινή ρίζα, ότι ανήκαν στην ίδια οικογένεια συναρτήσεων με το όνομα ScanSubject. Παρακάτω παρουσιάζεται η ποσοστιαία ανάλυση χρόνου εκτέλεσης ανά συνάρτηση. Στο σχήμα αυτό κρατήσαμε μόνο τις συναρτήσεις τύπου ScanSubject λόγω του ότι αν βάζαμε και τις υπόλοιπες θα είχαμε τουλάχιστον 11 συναρτήσεις και το σχήμα θα γινόταν δυσανάγνωστο.



Εικόνα 12 Ποσοστιαία ανάλυση χρόνου εκτέλεσης ανά συνάρτηση για το sequence 3 αλλάζοντας το μέγεθος λέξης (w-mer)

Ό χρόνος εκτέλεσης για διαφορετικά w-mer παρουσιάζεται στον παρακάτω πίνακα:

w-mer	Sec
11	0,65
10	1,05
9	2,10
8	2,03
7	3,72

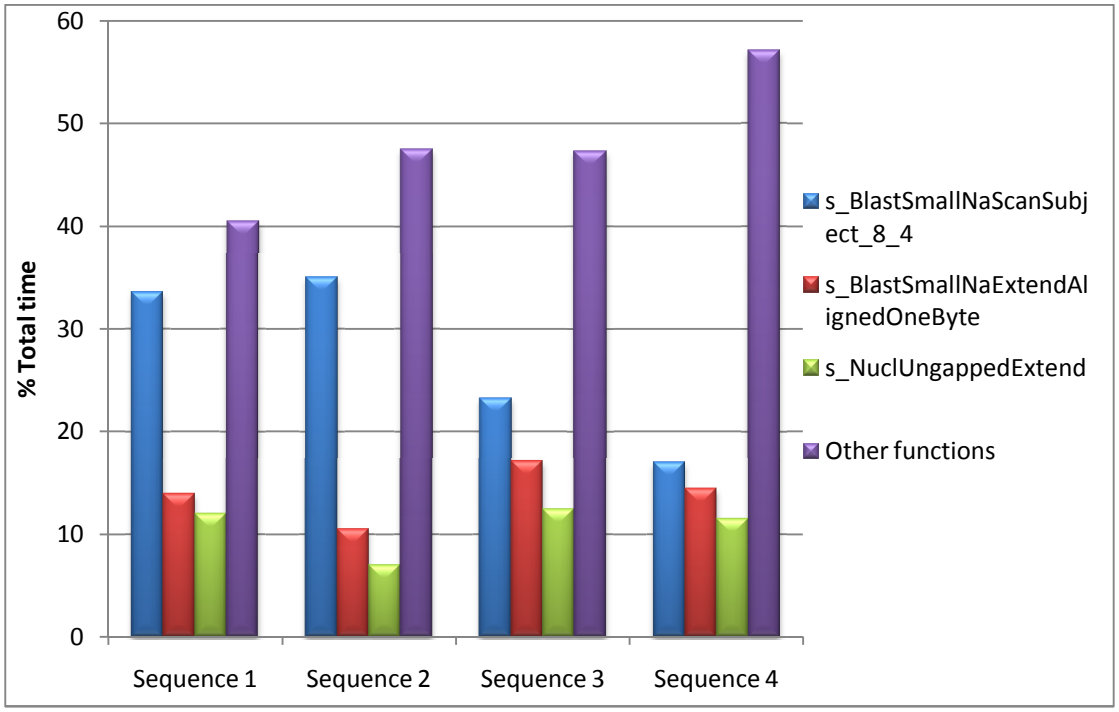
Πίνακας 5 Χρόνοι εκτέλεσης του BLASTn αλλάζοντας το μέγεθος λέξης (w-mer)

Παρατηρούμε και εδώ ότι ο χρόνος εκτέλεσης του προγράμματος αυξάνεται αρκετά μικραίνοντας το w-mer. Επίσης, σημαντικό συμπέρασμα είναι το γεγονός ότι ο

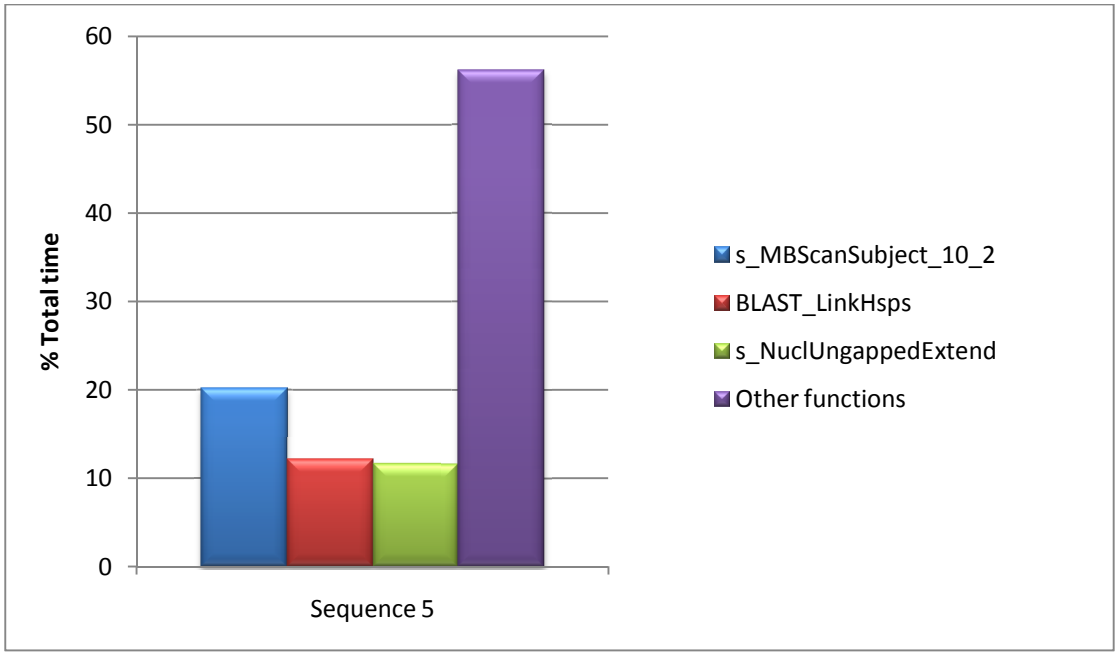
BLAST χρησιμοποιεί μια οικογένεια συναρτήσεων για τη σάρωση της βάσης. Βλέπουμε ότι η s_BlastSmallNaScanSubject_8_4 παρόλο που είναι η πιο γρήγορη συνάρτηση σε σύγκριση με τις υπόλοιπες της «οικογένειας» καταναλώνει το μεγαλύτερο ποσοστό του χρόνου του αλγορίθμου. Αυτό σημαίνει ότι είναι μια πολύ γρήγορη συνάρτηση πράγμα που επιβεβαιώνεται και από τους developers του NCBI [25].

Παρατηρήσαμε ότι μετά από κάποιο μέγεθος εισόδου (query) ο αλγόριθμος διαφοροποιούνταν ως προς τις συναρτήσεις που έτρεχαν. Πιο συγκεκριμένα, διαπιστώσαμε πειραματικά ότι για εισόδους μικρότερες των 6400 χαρακτήρων η συνάρτηση που καλούσε ο αλγόριθμος ήταν η s_BlastSmallNaScanSubject_8_4, ενώ για είσοδο μεγαλύτερη των 6400 χαρακτήρων (sequence5) η συνάρτηση που καλούσε ο αλγόριθμος ήταν η s_MBScanSubject_10_2 η οποία μας παραπέμπει σε συνάρτηση του MegaBlast.

Η τρίτη σειρά πειραμάτων περιελάμβανε την χρήση σταθερής γενετική βάσης δεδομένων (db_est) και εκτέλεση του αλγορίθμου για διαφορετικά queries. Τα αποτελέσματα όπως προέκυψψαν από την ανάλυση εμφανίζονται παρακάτω:



Εικόνα 13 Ποσοστιαία ανάλυση χρόνου εκτέλεσης ανά συνάρτηση αλλάζοντας μόνο το query



Εικόνα 14 Ποσοστιαία ανάλυση χρόνου εκτέλεσης ανά συνάρτηση για query μεγαλύτερο των 6.400 χαρακτήρων

Σε αυτή την σειρά πειραμάτων δεν παρατηρήσαμε κάποια σημαντική διαφορά στο χρόνο εκτέλεσης του αλγορίθμου, τουλάχιστον όχι τόση έντονη διαφορά όσο στο προηγούμενο πείραμα όπου αλλάζαμε τις βάσεις δεδομένων.

Η μόνη διαφορά που εμφανίστηκε ήταν ότι για την μεγάλη είσοδο query που χρησιμοποιήθηκε, ο αλγόριθμος χρησιμοποιεί ένα άλλο είδος συναρτήσεων οι οποίες είναι περισσότερο αργές σε σύγκριση με τις προηγούμενες. Αργότερα στην ανάλυση του αλγορίθμου θα μιλήσουμε για ποιο λόγο παρατηρείται το φαινόμενο αυτό.

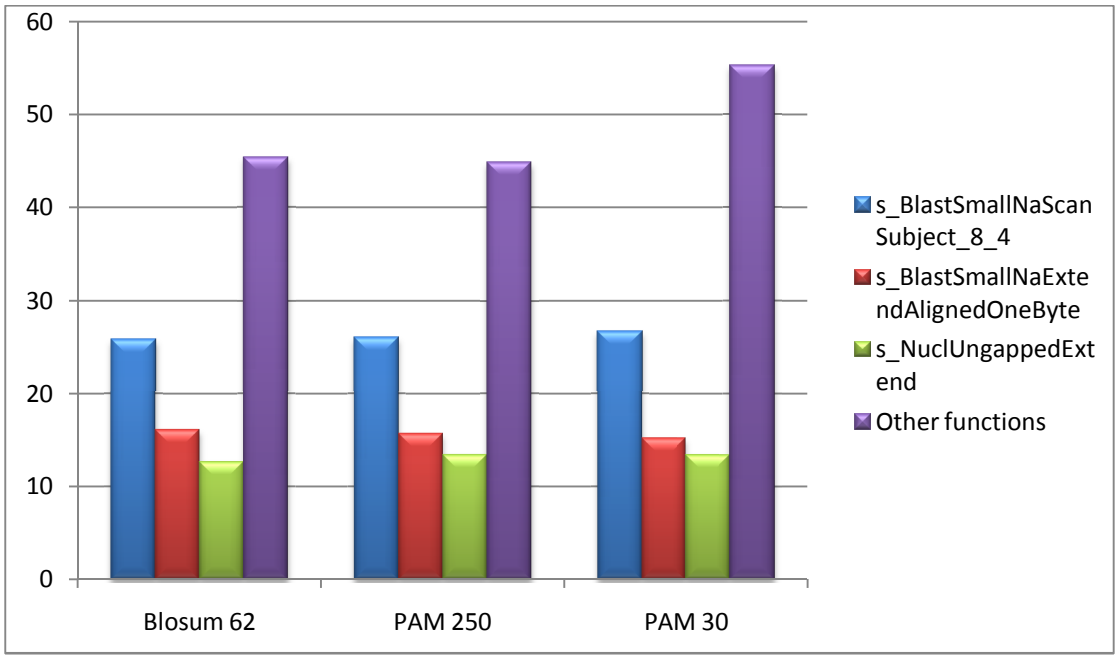
Παραθέτω παρακάτω τον πίνακα με τους χρόνους εκτέλεσης του αλγορίθμου.

Input	Total time (Sec)	Length (chars)
Sequence 1	1,18	700
Sequence 2	1,61	1400
Sequence 3	2,66	3650
Sequence 4	3,97	5040
Sequence 5	5,33	7000

Πίνακας 6 Χρόνοι εκτέλεσης του BLASTn αλλάζοντας τα queries

Επίσης αργότερα θα αναλύσουμε το γιατί ο αλγόριθμος χρησιμοποιεί διαφορετική συνάρτηση για μεγάλα queries και ποια είναι η βασική διαφορά ανάμεσα στις δύο αυτές συναρτήσεις.

Η τέταρτη σειρά πειραμάτων ήταν σχετικά με την αλλαγή του πίνακα αφαίρεσης (substitution matrix). Χρησιμοποιώντας σταθερή είσοδο και βάση παρατηρήσαμε ότι ο χρόνος εκτέλεσης του αλγορίθμου δεν παρουσίαζε κάποια σημαντική διαφορά ώστε να μπορούμε να πούμε ότι αξίζει να ασχοληθούμε με αυτό.



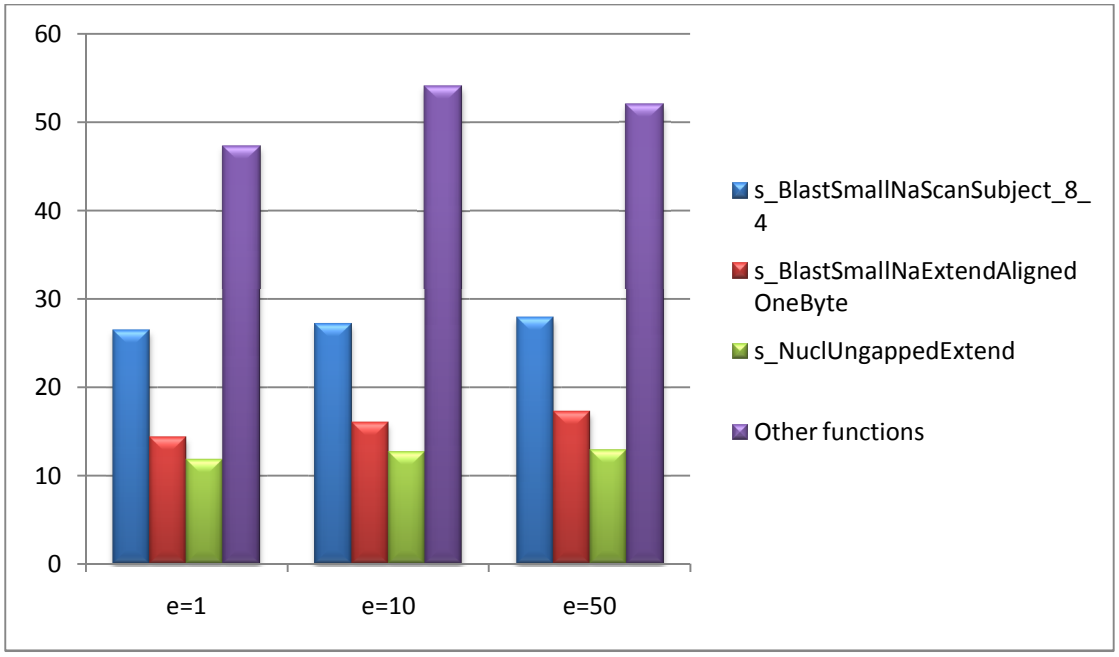
Εικόνα 15 Οι Ποσοστιαία ανάλυση χρόνου εκτέλεσης ανά συνάρτηση για το sequence 3 αλλάζοντας τα Substitution Matrices

Substitution Matrix	Sec
Blosum 62	0,63
PAM 250	0,60
PAM 30	0,61

Πίνακας 7 Χρόνοι εκτέλεσης του BLASTn αλλάζοντας τα Substitution matrices

Τέλος, κάναμε και κάποια πειράματα αλλάζοντας μόνο το E-value (expectation value). Σημαντικό είναι να σημειώσουμε ότι μειώνοντας το e-value παίρνουμε

λιγότερα αλλά περισσότερο σημαντικά αποτελέσματα και αυξάνοντας το e-value παίρνουμε περισσότερα αλλά λιγότερο σημαντικά αποτελέσματα.



Εικόνα 16 Ποσοστιαία ανάλυση χρόνου εκτέλεσης ανά συνάρτηση για το sequence 3 αλλάζοντας το E-value

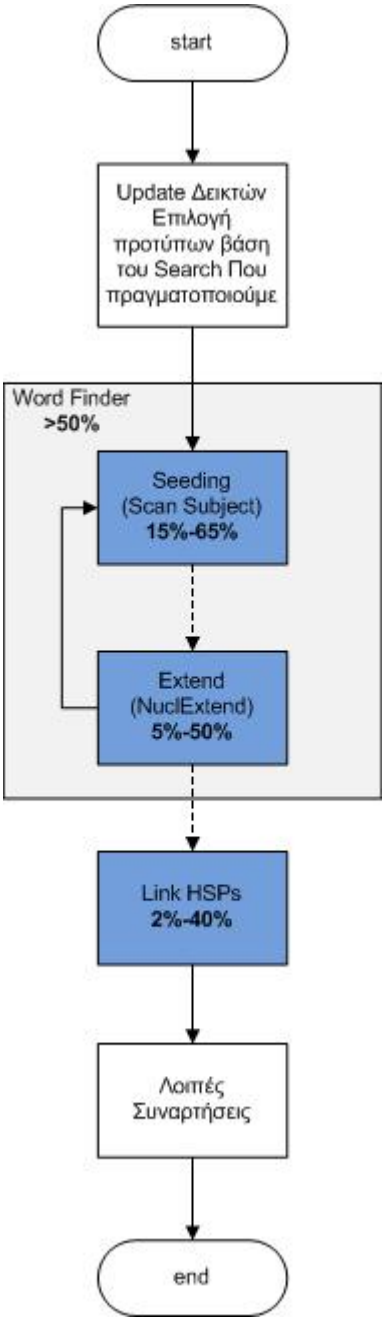
Και εδώ βλέπουμε ότι δεν έχουμε κάποια σημαντική διαφορά στο χρόνο που εκτελείται το πρόγραμμα αλλάζοντας το e-value και συμπεραίνουμε ότι το e-value δεν παίζει κάποιο σημαντικό ρόλο στον χρόνο εκτέλεσης του αλγορίθμου.

E-value	Sec
1	0,65
10	0,63
50	0,59

Πίνακας 8 Χρόνοι εκτέλεσης του BLASTn αλλάζοντας το E-value

Συμπεράσματα

Σε αυτό το σημείο θα παρουσιάσουμε συγκεντρωμένα τα συμπεράσματα από τα προηγούμενα πειράματα. Παρακάτω παρατείθεται μια εικόνα που δείχνει τα βασικά σημεία του αλγορίθμου και το ποσοστό χρόνου που καταναλώνεται σε κάθε σημείο του.



Εικόνα 17 Ποσοστά χρόνου κατά την εκτέλεση του αλγορίθμου NCBI BLASTn

Όπως βλέπουμε στην παραπάνω εικόνα η ρουτίνα Word Finder καταναλώνει συνεχώς πάνω από το 50% του χρόνου και δικαιολογημένα αφού είναι η «καρδιά» του αλγορίθμου. Μέσα στη ρουτίνα αυτή γίνονται οι δύο βασικές εργασίες του αλγορίθμου NCBI BLASTn που είναι οι εξής:

- **Seeding** το οποίο καταναλώνει 5%-65% του συνολικού χρόνου του αλγορίθμου.

- **Extension** το οποίο καταναλώνει 15%-50% του συνολικού χρόνου του αλγορίθμου

Πρέπει να αναφερθεί ότι οι δύο αυτές εργασίες εξαρτώνται από δύο παράγοντες που είναι, το μέγεθος της λέξης με το οποίο κάνουμε το seeding, και ο αριθμός των hits που πετυχαίνουμε στη φάση του seeding. Μεγάλο μέγεθος λέξης σημαίνει ότι θα έχουμε λιγότερα hits οπότε και δεν θα κάνουμε τόσο πολύ extension ενώ στην αντίθετη περίπτωση εκτός από πολλά hits θα έχουμε και πολύ περισσότερα extensions χωρίς πάντα το extension αυτό να είναι σημαντικό. Γι αυτό και τα ποσοστά του συνολικού χρόνου όσο αφορά τις δύο αυτές ρουτίνες (seeding-extension) είναι συμπληρωματικά με αθροιστικό χρόνο σταθερά πάνω από 50% του χρόνου που απαιτείται για να ολοκληρωθεί ο αλγόριθμος.

Κάποια άλλα συμπεράσματα που αντλήσαμε κάνοντας το profiling του αλγορίθμου NCBI BLASTn είναι τα εξής:

- Κυρίαρχο ρόλο στο χρόνο εκτέλεσης του αλγορίθμου παίζει το μέγεθος της βάσης. Μεγάλη βάση σημαίνει μεγάλο χρόνο αναζήτησης για seeds μέσα στη βάση και επίσης σημαίνει πολλά seeds τα οποία πρέπει να κάνει extend του βήματος 2.
- Όσο μικραίνουμε το μέγεθος της λέξης τόσο περισσότερες προσβάσεις κάνουμε στη βάση. Παρόλα αυτά ο αλγόριθμος χρησιμοποιείται συνήθως με μικρό φάσμα μεγέθους λέξης (συνήθως με 11).
- Υπάρχει μια οικογένεια συναρτήσεων για την διαδικασία σάρωσης της βάσης με όνομα ScanSubject. Οι συναρτήσεις αυτές έχουν την ίδια δομή και παρόμοια λειτουργικότητα αναλόγως με τη διαστασιολόγηση (μέγεθος query – word size) του προβλήματος. Η γρηγορότερη απ' όλες είναι η s_BlastSmallNaScanSubject_8_4 η οποία χρησιμοποιείται για queries μικρότερα των 6.400 χαρακτήρων και για μέγεθος λέξης ίσο με 11.
- Τέλος είδαμε ότι το e-value και τα substitution matrices δεν παίζουν μεγάλο ρόλο στον χρόνο εκτέλεσης του αλγορίθμου.

3.2 Ο αλγόριθμος σάρωσης της βάσης δεδομένων του NCBI BLASTn

Στο σημείο αυτό θα κάνουμε μια πιο αναλυτική περιγραφή του NCBI BLASTn αλγορίθμου καθώς και της συνάρτησης που χρησιμοποιείται για τη σάρωση της βάσης δεδομένων (για λόγους ευκολίας θα ονομάσουμε την οικογένεια συναρτήσεων που χρησιμοποιούνται για τη σάρωση της βάσης δεδομένων ως ScanSubject συνάρτηση).

Ακολουθία προς αναζήτηση (query):

Η περιοχή αναζήτησης του BLAST δέχεται διαφορετικούς τύπους μορφοποιημένης εισόδου και αυτόματα καθορίζει τη μορφή. Οι τρεις αυτές μορφοποιήσεις είναι οι εξής:

1. **FASTA format:** Η ακολουθία FASTA ξεκινά με μια γραμμή περιγραφής (single line description) που ακολουθείται από γραμμές μιας ακολουθίας μέχρι 70 χαρακτήρες νουκλεοτιδίων ανά γραμμή. Η γραμμή περιγραφής ξεχωρίζει από τις γραμμές ακολουθίας επειδή στην αρχή της γραμμής περιγραφής υπάρχει το σύμβολο (“>”). Στις FASTA ακολουθίες δεν επιτρέπονται κενές γραμμές. Ένα παράδειγμα ακολουθίας σε μορφή FASTA είναι το παρακάτω:

```
>AB013110
AGTCTATAAGAAGATTGACAGCCAAGAACACCACCACAATGAAGACCGCCGCTCTTGCACCGCTCT
TCTTCCTCCCCTCTGCCCTCGCCACGACGGTCTATCTCG
>X89714
TACCTCGCCAGTTACCTCTCCGCGACAGTGGTTAACGACGCGGTCGCGGGCCGCAGCGCAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAA
>Y08867
CTCAAAAAAGSCAGTAATGACTTGACTCAATGATGGTCAATAGTGGAATCTGGCCATGTGGTGATT
GCCGAGTTGTTRACAGGCTGSCCCCTTTCTCATCTTTCCT
>AY013315
GAGGACCGGCTGGTCCGGTGCAATGGCCATCGCCATCAATTGCTGCTGTGCAAG
```

Εικόνα 18 Παράδειγμα εισόδου σε FASTA μορφή

2. Σκέτη ακολουθία: Είναι μόνο οι γραμμές της ακολουθίας χωρίς τη FASTA γραμμή ορισμού.

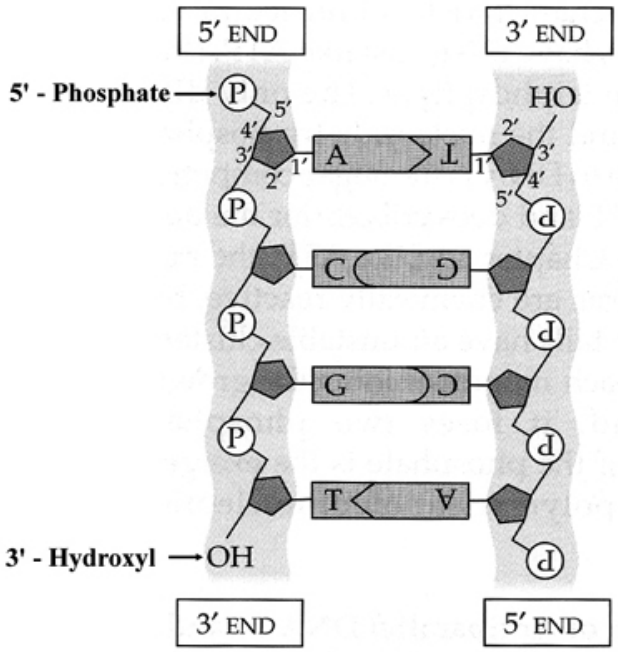
3. Προσδιοριστές: Είναι ένας αριθμός πρόσβασης στο NCBI Αυτοί οι προσδιοριστές ακολουθιών έχουν μια συγκεκριμένη σύνταξη που περιγράφεται μέσα στη βάση.

Εμείς επιλέξαμε να υλοποιήσουμε το πρόγραμμα BLASTn το οποίο συγκρίνει μια νουκλεοτιδική ακολουθία (DNA) με μια βάση νουκλεοτιδικών ακολουθιών (DNA).

Χρειάζεται να τονίσουμε ότι υπάρχουν δύο επιλογές εκτέλεσης του αλγορίθμου: i) gapped αναζήτηση και ii) ungapped. Στην ungapped λειτουργία τα hit seeds είναι hits χωρίς κενά.

Δημιουργία του Lookup table από μια είσοδο

Σε αντίθεση με τις πρωτεΐνες μια ακολουθία νουκλεοτιδικών βάσεων έχει το εξής χαρακτηριστικό. Έχει 2 έλικες (strides) πράγμα που σημαίνει ότι μπορεί να διαβαστεί και από τα 2 σκέλη. Κάθε σκέλος έχει διαφορετικά πάχη κατά μήκος (3’ και 5’). Κάθε έλικα μπορεί να διαβαστεί από το χοντρότερο (5’) προς το λεπτότερο (3’) άκρο.



Εικόνα 19 Τα δύο άκρα του DNA

Παραδείγματος χάρη, έστω ότι μας δίνεται μια ακολουθία βάσεων νουκλεοτιδίων σε μορφή FASTA η οποία είναι η εξής :

AAGCCTATTTGAATAGGCTC

Να τονίσουμε εδώ ότι κάθε βάση νουκλεοτιδίων αναπαρίσταται με 2 bit ως εξής:

A: 00

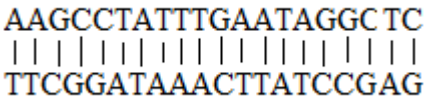
C: 01

G: 10

T: 11

Θα ήταν λογικό να πούμε ότι εάν ψάχνουμε για λέξεις 8 γραμμάτων (w-mer = 8) θα έχουμε 13 λέξεις μεγέθους 8. Στην πραγματικότητα όμως εάν κάποιος μας έχει στείλει μια ακολουθία DNA σημαίνει ότι εκτός από αυτό που μας έστειλε υπάρχει και η συμπληρωματική της. Η συμπληρωματική της ακολουθία είναι οι βάσεις

νουκλεοτιδίων με τις οποίες συνδέεται η ακολουθία που μας έχει δοθεί, όπως φαίνεται παρακάτω.



Εικόνα 20 Το τελικό query

Δηλαδή στέλνοντας μας την παραπάνω ακολουθία εμείς χρειάζεται να ψάξουμε και για την συμπληρωματική της μέσα στη βάση (database). Οπότε ψάχνουμε για 2 ακολουθίες.

> $1^{η}$ ακολουθία : Είναι η δοθείσα όπως ακριβώς είναι, δηλαδή AAGCCTATTTGAATAGGCTC
>
> $2^{η}$ ακολουθία : Είναι η συμπληρωματική της διαβάζοντας την **ανάποδα**,
>
> δηλαδή είναι η GAGCCTATTCAAATAGGCTT

Στο σημείο αυτό ο αλγόριθμος φτιάχνει έναν μονοδιάστατο πίνακα που περιέχει τις 2 παραπάνω ακολουθίες ($1^{η}$ και $2^{η}$).

Για το παράδειγμα μας θα δημιουργηθεί ο εξής πίνακας ο οποίος θα είναι και το πραγματικό μας query:

Index	0	1	2	3	4	5	6	7	8	9
Βάση	A	A	G	C	C	T	A	T	T	T
Binary	00	00	10	01	01	11	00	11	11	11

Index	10	11	12	13	14	15	16	17	18	19
Βάση	G	A	A	T	A	G	G	C	T	C
Binary	10	00	00	11	00	10	10	01	11	01

Index	20	21	22	23	24	25	26	27	28	29
Βάση	-	G	A	G	C	C	T	A	T	T

Binary	-	10	00	10	01	01	11	00	11	11

Index	30	31	32	33	34	35	36	37	38	39	40
Βάση	C	A	A	A	T	A	G	G	C	T	T
Binary	01	00	00	00	11	00	10	10	01	11	11

Πίνακας 9 Indexing του query

Με τον όρο index εννοούμε τη θέση στο query, και αυτό είναι επίσης ο αριθμός του w-mer. Στο συγκεκριμένο παράδειγμα θα έχουμε συνολικά 26 w-mers μεγέθους 8 χαρακτήρων (13 w-mers από index 0 ως 12 και άλλα 13 από index 21 ως 33).

Στη φάση αυτή όμως ο αλγόριθμος NCBI BLASTn φτιάχνει έναν άλλο πίνακα τον οποίο και ονομάζει backbone. Ο πίνακας backbone έχει βάθος ανάλογα με το μήκος λέξης την οποία εμείς ψάχνουμε. Αν θέσουμε ως **n** το μήκος της λέξης τότε ο πίνακας backbone θα έχει μέγεθος $\mathbf{2^{2n}}$ επειδή κάθε βάση αναπαρίσταται με 2 bit. Στην περίπτωση μας επειδή ψάχνουμε για λέξεις μεγέθους 8 βάσεων (16bit) ο πίνακας μας θα έχει βάθος $\mathbf{2^{16}}$**.**

Η κάθε θέση του πίνακα θα αντιπροσωπεύει και μια συγκεκριμένη λέξη 8 βάσεων. Δηλαδή στην πρώτη θέση η οποία είναι η θέση 0 αντιπροσωπεύεται η λέξη AAAAAAAA. Στη $2^{η}$ θέση θα αντιπροσωπεύεται η λέξη AAAAAAAC στην $3^{η}$ η λέξη AAAAAAAG και ου το καθ' εξής μέχρι να φτάσουμε στην τελευταία θέση που είναι η λέξη TTTTTTTT.

Πηγαίνοντας τώρα στο query ο NCBI BLASTn αυτό που κάνει είναι να παίρνει κάθε w-mer και να το βάζει το index του στη σωστή θέση στον πίνακα backbone.

Π.χ. Το $1^{ο}$ w-mer με index 0 είναι το: AAGCCTAT.

Σε binary αυτό μεταφράζεται ως εξής: 0000100101110010 (2419)

Οπότε στη θέση 2419 της μνήμης backbone θα πάει και θα γράψει το index του w-mer το οποίο είναι το **0**.

Με τον ίδιο τρόπο το στη θέση 38719 θα γραφεί το $3^{ο}$ w-mer (GCCTATTT) με αριθμό index **2**.

Στη θέση 23806 θα γραφεί το $4^{ο}$ w-mer(CCTATTTG) με αριθμό index **3** και ου το καθεξής.

Όμως είναι πολύ πιθανό να πετύχουμε κάποιο ίδιο w-mer σε παραπάνω από μια θέση όπως συμβαίνει και με το παράδειγμα μας, να έχουμε δηλαδή overflow. Στην περίπτωση μας τα w-mer με αριθμό index 11 και αυτό με αριθμό 32 είναι ουσιαστικά η ίδια λέξη (AATAGGCT). Το ίδιο ισχύει και με τα w-mer με αριθμό index 1 και 22 που έχουν την λέξη AGCCTATT.

Ο NCBI BLASTn δημιουργεί έναν πίνακα overflow βάθους το πολύ μέχρι 2^{16}. Όταν ο αλγόριθμος ολοκληρώσει την εργασία αυτή (το γέμισμα δηλαδή του πίνακα overflow) κρατάει μόνο τις θέσεις που χρειάζεται, όσες και να είναι αυτές, και στις υπόλοιπες θέσεις γράφει άλλες τιμές που σε εμάς είναι αδιάφορες.

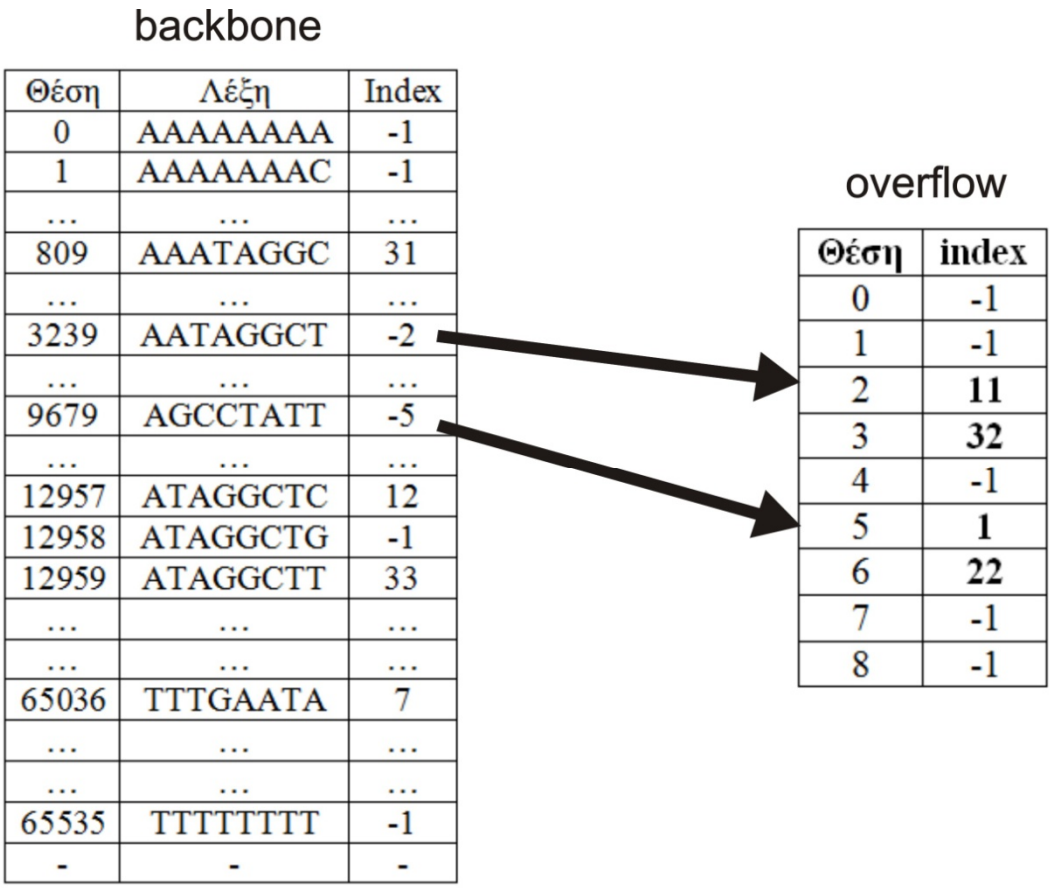
Στην περίπτωση μας ο πίνακας overflow που δημιουργείται έχει την παρακάτω μορφή:

Θέση	**index**
0	-1
1	-1
2	**11**
3	**32**
4	-1
5	**1**
6	**22**
7	-1
8	-1

Πίνακας 10 Ο πίνακας overflow που δημιουργήσαμε

Ο πίνακας overflow έχει πάντα τις 2 πρώτες θέσεις (0 και 1) κενές. Στην $3^{η}$ θέση υπάρχει το πρώτο index στο οποίο έχουμε overflow. Τρέχοντας τον πίνακα προς τα κάτω όσα indexes συναντάμε αντιστοιχούν στην ίδια λέξη, μέχρις ότου να συναντήσουμε τον separator ο οποίος έχει την τιμή **-1**.

Για να έχουμε μια πιο ολοκληρωμένη όμως εικόνα για το πώς σχετίζονται οι 2 πίνακες backbone και overflow ας να δούμε το παρακάτω σχήμα. (Η μεσαία στήλη μπαίνει για λόγους ευκολίας, στην πραγματικότητα δεν υπάρχει).



Εικόνα 21 Σχέση πίνακα backbone με overflow

Συνήθως ο πίνακας overflow μεγαλώνει ανάλογα με το query. Στο παράδειγμα μας όπου έχουμε ένα query μεγέθους 20 βάσεων βλέπουμε ότι έχουμε 2 περιπτώσεις overflow.

Παρατηρούμε στον παραπάνω πίνακα backbone ότι παίρνει 3 τύπους τιμών. Οι τιμές οι οποίες είναι μεγαλύτερες ή ίσες του 0 οι οποίες συμβολίζουν τη θέση στο query όπου βρίσκετε η λέξη που ψάχνουμε. Οι τιμές μικρότερες του -1 που δείχνουν στον πίνακα overflow όπου μπορούμε να βρούμε τις θέσεις τις οποίες ψάχνουμε. Και τέλος η τιμή -1 η οποία συμβολίζει ότι η ζητούμενη λέξη δεν υπάρχει πουθενά μέσα στο query.

Αυτοί οι 2 πίνακες (backbone και overflow) είναι τα βασικά αποτελέσματα του 1ου βήματος του αλγορίθμου NCBI BLASTn. Και οι 2 πίνακες περιέχονται μέσα στη δομή **LookupTableWrap** (αρχείο lookup_wrap.h) και έχουμε πρόσβαση σε αυτούς μέσω δεικτών.

Backbone: LookupTableWrap -> lut -> final_backbone (16bits)

Overflow: LookupTableWrap -> lut -> overflow (16bits)

Η δομή αυτή περιέχει επίσης την μεταβλητή **lut_type** η οποία μας υποδεικνύει το είδος του lookup table που θα χρησιμοποιήσουμε. Οι τιμές που παίρνει είναι οι εξής:

1) eAaLookupTable

2) eCompressedAaLookupTable

3) eIndexedMBLookupTable

4) eNaLookupTable

5) eSmallNaLookupTable

6) eMBLookupTable

7) ePhiLookupTable

8) eRPSLookupTable

Για τον αλγόριθμο BLASTn χρησιμοποιούμε τους lookup tables με αριθμό 4-5-6. Κυρίως τους 5 και 6. Από όσο μελετήσαμε τον αλγόριθμο συμπεράναμε ότι για query μικρότερα των 6.500 βάσεων αμινοξέων χρησιμοποιείται ο eSmallNaLookupTable ενώ για μεγαλύτερα query ο eMBLookupTable.

Ο αλγόριθμος με τον οποίο γίνεται η σάρωση της βάσης

Στο βήμα αυτό ο αλγόριθμος κάνει το seeding, δηλαδή βρίσκει τα αρχικά hits τα οποία αργότερα στο βήμα 3 θα γίνουν extend. Ο BLASTn χρησιμοποιεί στο στάδιο αυτό μια από τις παρακάτω συναρτήσεις:

Όνομα συνάρτησης	Μέγεθος seeding	Stride βήμα
s_BlastSmallNaScanSubject_4_1	4	1
s_BlastSmallNaScanSubject_5_1	5	1
s_BlastSmallNaScanSubject_6_1	6	1
s_BlastSmallNaScanSubject_6_2	7	2
s_BlastSmallNaScanSubject_7_1	7	1
s_BlastSmallNaScanSubject_7_2	8	2
s_BlastSmallNaScanSubject_7_3	9	3
s_BlastSmallNaScanSubject_8_1mod4	8	1
s_BlastSmallNaScanSubject_8_2mod4	9	2
s_BlastSmallNaScanSubject_8_3mod4	10	3
s_BlastSmallNaScanSubject_8_4	11	4
s_BlastNaScanSubject_8_4	11	4
s_MBScanSubject_9_1	9	1
s_ MBScanSubject_9_2	10	2
s_ MBScanSubject_10_1	10	1
s_ MBScanSubject_10_2	11	2
s_ MBScanSubject_10_3	12	3
s_ MBScanSubject_11_1mod4	11	1
s_ MBScanSubject_11_2mod4	12	2
s_ MBScanSubject_11_3mod4	13	3
s_BlastNaScanSubjectAny	Any	0
s_BlastSmallNaScanSubjectAny	Any	0

Πίνακας 11 Οι συναρτήσεις του αλγόριθμου NCBI BLASTn για τη σάρωση της βάσης δεδομένων

Αυτές οι συναρτήσεις κάνουν ουσιαστικά την ίδια λειτουργία. Είναι δηλαδή μια οικογένεια συναρτήσεων. Ο αλγόριθμος NCBI BLASTn επιλέγει μια από αυτές με την οποία κάνει το seeding. Πειραματικά είδαμε ότι κύριο λόγο για την επιλογή έχει το μέγεθος λέξης με το οποίο ζητάμε να γίνει το seeding. Από εκεί και πέρα σημαντικό ρόλο παίζει και το μέγεθος του query.

Για δική μας ευκολία μπορούμε να ονομάσουμε όλες τις παραπάνω ως μια γενική συνάρτηση με το όνομα ScanSubject.

Στις παραπάνω συναρτήσεις παρατηρούμε ότι μετά το όνομα ακολουθούν 2 αριθμοί. Ο πρώτος αριθμός μας δίνει το μέγεθος του w-mer σε χαρακτήρες βάσης και ο δεύτερος αριθμός μας δίνει το **stride**. Οι 2 αυτοί αριθμοί είναι σημαντικοί παράγοντες στην συμπεριφορά του NCBI Blast στο δεύτερο βήμα

Τώρα ας δούμε τη βασική δομή του 2^{ου} βήματος. Όπως είπαμε και πριν όλες οι παραπάνω συναρτήσεις είναι ουσιαστικά μια οικογένεια συναρτήσεων. Κάνουν περίπου την ίδια διεργασία οπότε παίρνουν και τα ίδια ορίσματα τα οποία είναι τα εξής:

- **Lookup_wrap:** Δείκτης τύπου LookupTableWrap. Εκεί βρίσκονται τα βασικά δεδομένα που δημιουργήθηκαν κατά την εκτέλεση του πρώτου βήματος του αλγορίθμου. Αυτά που χρησιμοποιούμε στο βήμα αυτό είναι οι 2 πίνακες, ο backbone και ο overflow στους οποίους όπως είπαμε και παραπάνω βρίσκονται οι θέσεις των επιμέρους w-mers ή αλλιώς τα indexes.

- **Subject:** Δείκτης τύπου BLAST_SequenceBlk. Δείχνει τη βάση πάνω στην οποία θα ψάξουμε για hits. Πολλές φορές οι μεγάλες βάσεις χωρίζονται σε sequences. Τότε ο δείκτης αυτός δείχνει κάθε φορά στην αρχή του κάθε sequence. Άλλες φορές έχουμε μεγάλες βάσεις οι οποίες δεν είναι χωρισμένες σε sequences. Τότε ο NCBI Blast «σπάει» μόνος του τη βάση σε μικρότερα κομμάτια και ο δείκτης αυτός δείχνει πάλι στην αρχή του κάθε κομματιού.

- **Scan range:** Δείκτης τύπου integer. Χρησιμοποιείται ως είσοδος και ως έξοδος. Ως είσοδος το scan range[0] μας δείχνει την αρχική θέση από την οποία θα ξεκινήσει η αναζήτηση στη βάση. Το scan range[1] μας δείχνει την θέση στην βάση στην οποία θα πρέπει να σταματήσει η αναζήτηση (υπάρχουν φορές που η αναζήτηση σταματάει πριν φτάσει στο scan range[1] αλλά θα αναφερθούμε σε αυτό αργότερα). Ως έξοδος το scan range[0] ανανεώνεται ώστε να γνωρίζουμε από ποιο σημείο θα ξεκινήσει η επόμενη αναζήτηση.

- **Offset_pairs:** Πίνακας στον οποίο τοποθετούνται οι θέσεις του query και της βάσης (database) έχει γίνει hit. Κάθε θέση στον πίνακα αυτόν αντιστοιχεί σε ένα hit. Η θέση του query αποθηκεύεται στο σημείο offset_pairs[i].qs_offsets.q_off ενώ η θέση της βάσης (database) στο offset_pairs[i].qs_offsets.s_off.

- **Max_hits:** Αριθμός integer ο οποίος μας δείχνει τον μέγιστο αριθμό hits που μπορούμε να έχουμε σε κάθε κλίση της συνάρτησης. Επίσης ο αριθμός αυτός μας υποδεικνύει το μέγεθος του πίνακα offset_pairs, δηλαδή τον αριθμό των hits των οποίο μπορεί να αποθηκεύσουμε στον πίνακα αυτό.

Πριν μιλήσουμε για την λειτουργία των συναρτήσεων αυτών ας δούμε πρώτα τον ορισμό του stride, τι σημαίνει και πως οι αλλαγές του επηρεάζουν το $2^{ο}$ βήμα του αλγορίθμου.

Stride:

Είναι ένας νεοτερισμός του NCBI Blast. Πρωτοεμφανίστηκε μετά την έκδοση NCBI Blast 2. Κυριολεκτικά σημαίνει βήμα. Ουσιαστικά μας δείχνει το ρυθμό με τον οποίο προσπελαύνουμε τη βάση (database). Όταν το stride είναι 1 τότε διαβάζουμε τη βάση ανά ένα χαρακτήρα κάθε φορά, όταν είναι 2 ανά δύο χαρακτήρες τη φορά και ου το καθεξής. Επίσης όταν αυξάνεται το stride μειώνεται και το μέγεθος της λέξης την οποία ψάχνουμε να βρούμε στην database. Αν δηλαδή ψάχνουμε για λέξεις μεγέθους 11 και χρησιμοποιούμε stride = 1 τότε ψάχνουμε στην database για λέξεις μεγέθους 11. Αν όμως κάνουμε το stride = 2 ψάχνουμε για λέξεις μεγέθους 10. Αν stride = 3 τότε ψάχνουμε για λέξεις μεγέθους 9 και ου το καθεξής.

Με την εισαγωγή του όρου stride μπορούμε αυξάνοντας το stride να μειώσουμε κατά πολύ τις προσβάσεις στην database**.**

Έστω έχουμε μια βάση με 10^6 χαρακτήρες και ζητάμε λέξεις μεγέθους 11. Τότε αν έχουμε:

- *Stride = 1* : Θα κάνουμε $(10^6 - 11) \; div \; 1 + 1 = 999.990$ προσβάσεις στη βάση ψάχνοντας για λέξεις μεγέθους 11

- Stride = 2 Θα κάνουμε (10^6 - 10) div 2 + 1 = 499.995 προσβάσεις στη βάση ψάχνοντας για λέξεις μεγέθους 10

Και ούτω καθεξής. Στον παρακάτω πίνακα φαίνονται οι προσβάσεις σε μια βάση μεγέθους 10^6 χαρακτήρων ανάλογα με το stride.

Stride	Προσβάσεις στη βάση
1	999.990
2	499.995
3	333.330
4	249.998

Εικόνα 22 Προσβάσεις στη βάση ανάλογα με τη διαφοροποίηση του stride

Παρατηρούμε ότι ο αριθμός των προσβάσεων μας στη βάση δεδομένων μειώνεται δραματικά αν εμείς αυξήσουμε έστω και ελάχιστα το stride. Μπορούμε σύμφωνα με τον παρακάτω πίνακα να μειώσουμε τις προσβάσεις μας στη μνήμη μέχρι και 75% από όσες είχαμε αν ψάχναμε τη βάση δεδομένων ελέγχοντας έναν χαρακτήρα τη φορά. Αυτή η ιδέα του stride μπορεί να μας βοηθήσει αρκετά στην προσπάθεια μας να πετύχουμε καλύτερους χρόνους κατά την εκτέλεση των διαφόρων προγραμμάτων βιοπληροφορικής αν αναλογιστούμε:

i. τα μεγάλα μεγέθη των βάσεων δεδομένων στον κλάδο των bioinformatics σήμερα και
ii. τον ρυθμό αύξησης των βάσεων αυτών ο οποίος αγγίζει και την γεωμετρική πρόοδο.

Το βασικό κομμάτι των συναρτήσεων αυτών είναι ένα while loop μέσα στο οποίο παίρνουμε ένα κομμάτι της βάσης (database) μεγέθους w-mer και ελέγχουμε μέσα στον πίνακα backbone (ή τον πίνακα overflow) εάν μπορούμε να το ταυτοποιήσουμε με κάποιο κομμάτι του query. Εάν πετύχουμε την ταυτοποίηση (έχουμε hit) τότε γράφουμε στον πίνακα offset pairs τις αντίστοιχες θέσεις του query και της database όπου έγινε το hit. Κατόπιν αυξάνουμε την μεταβλητή των hit τα οποία μετράμε μέχρι στιγμής και μετά ανανεώνουμε το scan_range[0] ώστε να πάρουμε το επόμενο

κομμάτι της database μεγέθους πάλι w-mer-length. Εάν τα hit τα οποία έχουμε κάνει ξεπερνούν τον αριθμό των max hits τότε η συνάρτηση σταματά και επιστρέφει τον συνολικό αριθμό hits που έχουμε πετύχει.

Σε αυτή την φάση θα μπορούσαμε να είχαμε τελειώσει το seeding αλλά με την εισαγωγή του όρου stride τα πράγματα αλλάζουν αρκετά. Εάν το stride είναι ίσο με 1 τότε με το τέλος των παραπάνω συναρτήσεων τα ζευγάρια που υπάρχουν μέσα στον πίνακα offset pairs θα θεωρηθούν ως κανονικά hits και ο αλγόριθμος θα συνεχίσει στο επόμενο βήμα.

Εάν όμως το stride μεγαλύτερο του 1 τότε το seeding δεν έχει τελειώσει. Τα ζευγάρια που υπάρχουν μέσα στον πίνακα offset pairs θεωρούνται initial hits και πρέπει να κάνουμε ένα αρχικό extend ώστε να μπορέσουμε να έχουμε ένα ολοκληρωμένο hit μεγέθους αντίστοιχου με αυτό που ζητάμε.

Ας πάμε πάλι σε ένα παράδειγμα για να κατανοήσουμε καλύτερα πως λειτουργεί το αλγόριθμος.

Έστω ότι ψάχνουμε για λέξεις μεγέθους 11 βάσεων αμινοξέων και έχουμε stride = 4. Αυτό μας υποδεικνύει ότι θα ψάξουμε στη βάση για λέξεις μεγέθους 8 βάσεων αμινοξέων. Αφού τελειώσουμε την αναζήτηση στη βάση τότε θα έχουμε έναν πίνακα (offset pairs) ο οποίος θα περιέχει initial hits μεγέθους 8 βάσεων αμινοξέων και όχι μεγέθους 11 που ζητάμε. Τότε ο αλγόριθμος θα εκτελέσει μια συνάρτηση η οποία θα ελέγξει αριστερά και δεξιά από τις θέσεις του initial hit για 3 ακριβώς hit. Εάν πετύχει 3 ακριβώς ταυτοποιήσεις από την κάθε μεριά τότε εκλαμβάνουμε το initial hit αυτό ως κανονικό hit και συνεχίζουμε στο επόμενο βήμα του αλγορίθμου που είναι το extend του αλγορίθμου. Εάν όμως αποτύχουμε και από τις 2 πλευρές του query να κάνουμε το extension αυτό, τότε το hit απορρίπτεται.

Στο σημείο αυτό έχει ολοκληρωθεί το seeding.

3.3 Μοντελοποίηση της συνάρτησης ScanSubject

Όπως έχει αναφερθεί και προηγουμένως (παράγραφος 3.2) υπάρχει μια οικογένεια συναρτήσεων και πιο συγκεκριμένα μια οικογένεια συναρτήσεων με σκοπό την σάρωση της βάσης η οποία «κοστίζει» πολύ στο χρόνο εκτέλεσης του αλγορίθμου. Για την διευκόλυνση της μελέτης ονομάσαμε την οικογένεια συναρτήσεων ως

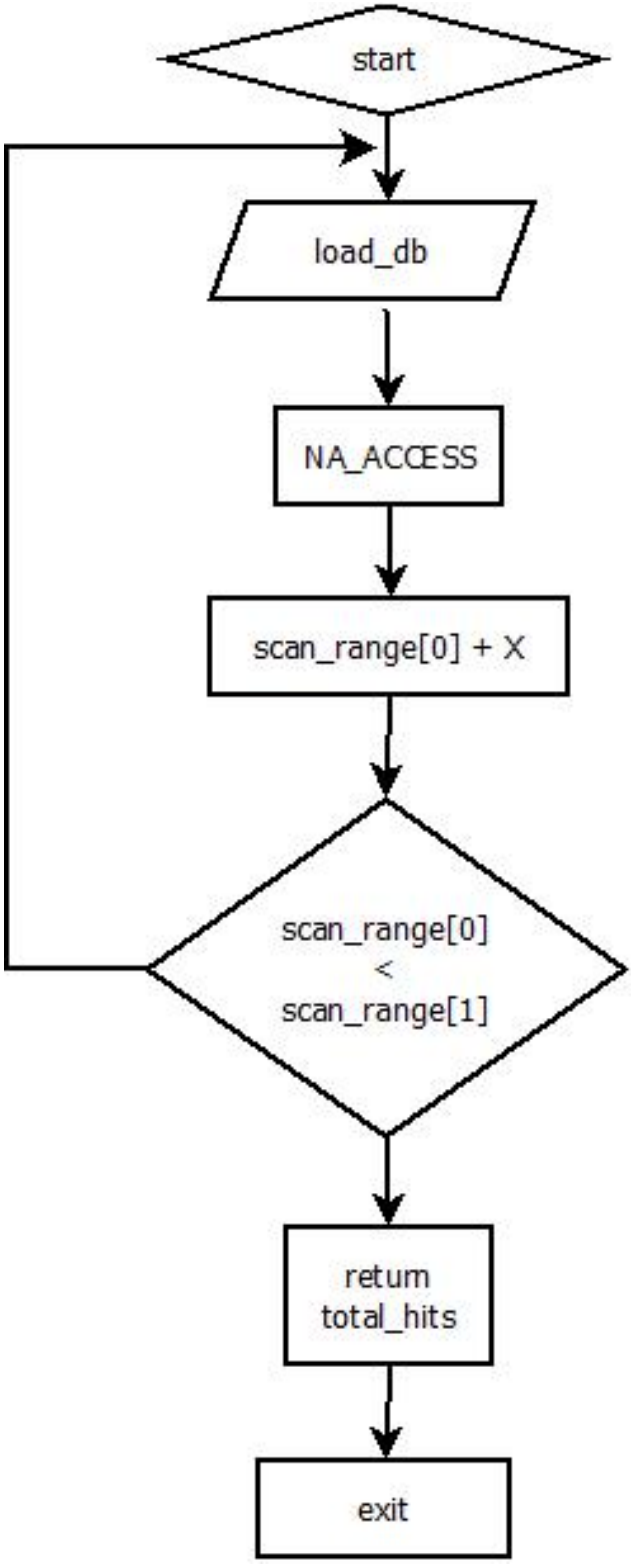
ScanSubject και στο σημείο αυτό θα προσπαθήσουμε να δώσουμε μια πιο συγκεκριμένη εικόνα για την λειτουργία της.

Όλες οι συναρτήσεις της οικογένειας ScanSubject δέχονται τα ίδια ορίσματα, κάνουν την ίδια διεργασία με πολύ μικρές διαφορές, και επιστρέφουν τον ίδιο τύπο αποτελεσμάτων.

Με λίγα λόγια η συνάρτηση ScanSubject κάνει τα εξής βήματα:

i. Φορτώνει ένα κομμάτι βάσης μεγέθους w-mer και γίνονται οι απαραίτητες αρχικοποιήσεις. (Εικόνα 22 load_db)
ii. Ψάχνει για κάποιο hit στον LookupTable. (Εικόνα 22 NA_ACCESS HITS)
iii. Επαναλαμβάνει την παραπάνω διαδικασία μέχρις ότου να φτάσει στο τέλος της βάσης που μελετάμε όπου και επιστρέφει τον συνολικό αριθμό από hits και έναν πίνακα με τα ζευγάρια θέσης ανάμεσα σε query και database. (Εικόνα 22 return_total_hits)

Όταν ξεκινά η συνάρτηση αρχικοποιούνται κάποιες μεταβλητές οι οποίες σχετίζονται με το μέγεθος λέξης που χρησιμοποιούμε, το σημείο της βάσης από το οποίο ξεκινάμε την αναζήτηση, το σημείο όπου πρέπει να την τελειώσουμε και άλλες τέτοιες αρχικοποιήσεις. Κατόπιν λαμβάνοντας υπόψη το stride γίνεται ένας σύντομος υπολογισμός για να βρεθεί ο μέγιστος αριθμός αναζητήσεων που μπορούν να γίνουν στη βάση. Αφού τελειώσουν όλες οι αρχικοποιήσεις που πρέπει να γίνουν ξεκινάει και το βασικό κομμάτι του αλγορίθμου που είναι η συνάρτηση NA _ACCESS_HITS η καλύτερα η οικογένεια συναρτήσεων NA_ACCESS_HITS η οποία βρίσκεται μέσα στη μεγάλη μας επανάληψη και καλείται συνεχώς μέχρι να τελειώσει το κομμάτι της βάσης το οποίο μελετάμε.



Εικόνα 23 Μοντελοποίηση συνάρτησης ScanSubject

Σε αυτό το σημείο να πούμε ότι η συνάρτηση ότι υπάρχουν 3 είδη συναρτήσεων τύπου NA_ACCESS_HITS τα οποία χωρίζουν μεταξύ τους βάση του Lookup Table που χρησιμοποιήθηκε. Οι συναρτήσεις είναι οι εξής:

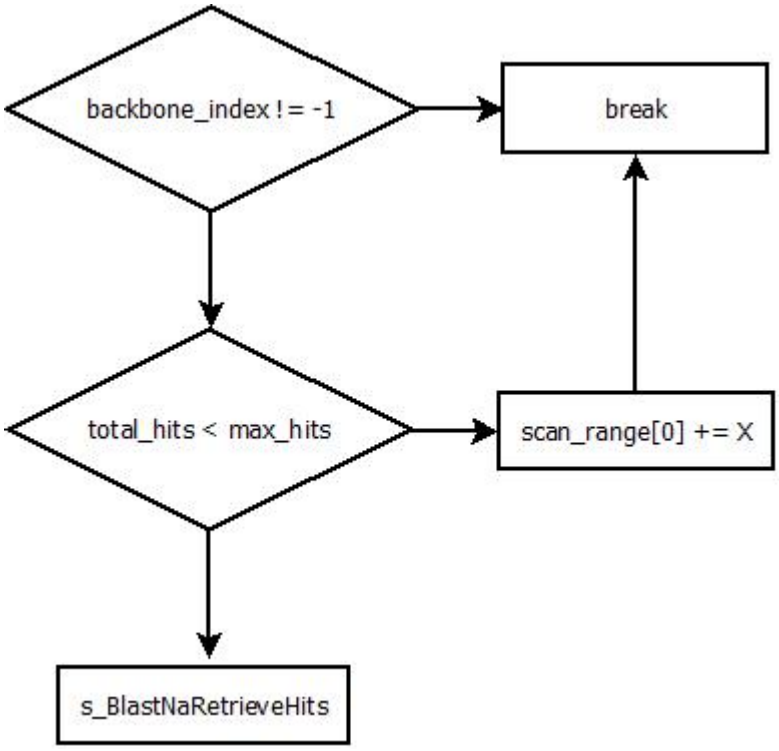
Όνομα συνάρτησης	Lookup Table
NA_ACCESS_HITS	NaLookupTable
SMALL_NA_ACCESS_HITS	SmallNaLookupTable
MB_ACCESS_HITS	MBLookupTable

Πίνακας 12 Συναρτήσεις NA_ACCESS_HITS ανάλογα με τον Lookup Table που χρησιμοποιήθηκε

Η συνάρτηση αυτή καλείται περισσότερες από μια φορά σε κάθε επανάληψη αναλόγως με το μέγεθος λέξης με τον οποίο έχει κατασκευαστεί ο πίνακας backbone και το ρυθμό που ελέγχουμε τη βάση (stride). Είναι μια inline συνάρτηση και ως όρισμα παίρνει έναν αριθμό 'x' που αντιπροσωπεύει το σημείο στο οποίο γίνεται αυτή τη στιγμή η αναζήτηση (πχ SMALL_NA_ACCESS_HITS(x)).

Οι συναρτήσεις αυτού του τύπου (NA_ACCESS_HITS) έχουν επίσης τη δυνατότητα να διακόψουν προσωρινά την συνάρτηση ScanSubject αν ο αριθμός των hits έχει ξεπεράσει τον επιτρεπόμενο αριθμό των max_hits δηλαδή τον μέγιστο αριθμό των hits που μπορούμε να αποθηκεύσουμε στον πίνακα offset_pairs.

Η λειτουργία που κάνει η συνάρτηση αυτή είναι να ελέγχει για το εάν υπάρχει κάποιο hit μεταξύ του query με τη βάση δεδομένων. Πιο συγκεκριμένα εάν στη θέση του πίνακα backbone (όπου υπάρχει αποθηκευμένο το query) υπάρχει η τιμή ' -1' πάει να πει ότι δεν έχουμε κάποιο hit και κάνουμε break από τη συνάρτηση NA_ACCESS_HITS. Σε αντίθετη περίπτωση, όταν δηλαδή η τιμή που υπάρχει στον πίνακα backbone είναι διαφορετική του '-1' τότε έχουμε ένα ή και περισσότερα hits. Για να βρεθεί ο ακριβής αριθμός των hits αλλά και των θέσεων στα οποία αυτά βρεθήκαν καλείται η συνάρτηση s_BlastNaRetrieveHits. Στο επόμενο σχήμα φαίνεται η μοντελοποίηση της συνάρτησης NA_ACCESS_HITS.



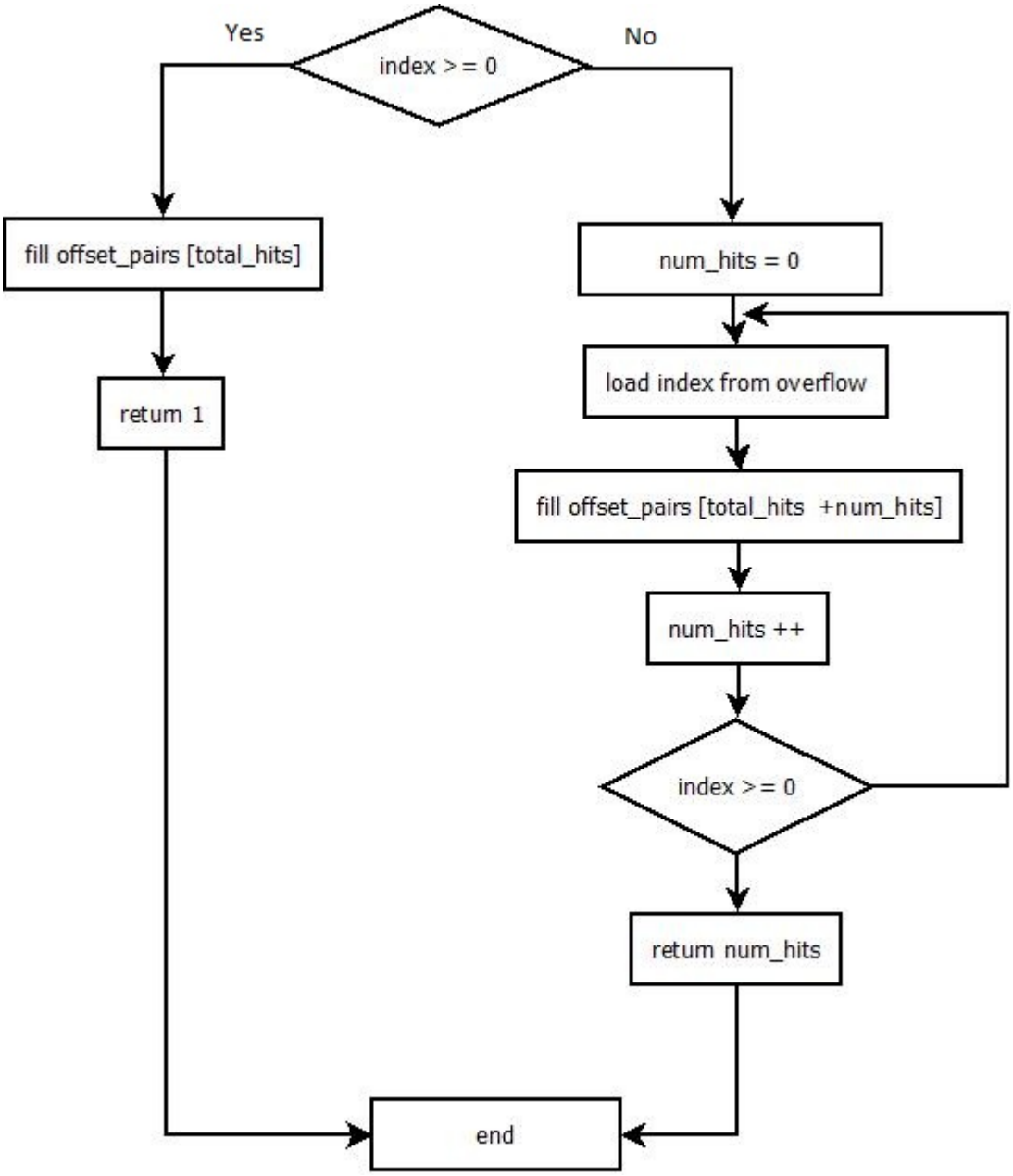
Εικόνα 24 Μοντελοποίηση συνάρτησης NA_ACCESS_HITS

Η συνάρτηση s_BlastNaRetrieveHits ανήκει και αυτή με τη σειρά της σε μια οικογένεια συναρτήσεων ανάλογα και αυτή με το είδος του Lookup Table που δημιουργήθηκε. Έχουμε και εδώ λοιπόν 3 συναρτήσεις οι οποίες είναι οι:

Όνομα συνάρτησης	Lookup Table
s_BlastNaRetrieveHits	NaLookupTable
s_BlastSmallNaRetrieveHits	SmallNaLookupTable
s_MBRetrieveHits	MBLookupTable

Πίνακας 13 Συναρτήσεις τύπου Retrieve Hits βάση του Looku Table που χρησιμοποιήθηκε

Στο σημείο αυτό αντλούμε τα «ωφέλιμα» δεδομένα τα οποία είναι τα εξής: Αριθμός hit στο συγκεκριμένο σημείο της βάσης και αντιστοιχία θέσης όπου αυτό συνέβη. Στο παρακάτω σχήμα (Εικόνα 24) φαίνονται οι δύο περιπτώσεις που αναφέραμε και προηγουμένως.



Εικόνα 25 Μοντελοποίηση συνάρτησης Retrieve_Hits

Πιο συγκεκριμένα αν το index που μας επιστρέψει ο πίνακας backbone είναι μεγαλύτερο ή ίσο του μηδενός τότε έχουμε ένα και μόνο hit. Στην περίπτωση αυτή πάμε στον πίνακα offset_pairs και στη θέση total_hits βάζουμε τα offsets της βάσης και της εισόδου. Πιο συγκεκριμένα έχουμε:

- offset_pairs[i].qs_offsets.q_off = index (offset εισόδου)
- offset_pairs[i].qs_offsets.s_off = soff (offset βάσης)

Αφού έχουμε αποθηκεύσει τις παραπάνω τιμές στον πίνακα offset_pairs η συνάρτηση μας επιστρέφει τον αριθμό των hits ο οποίος είναι ίσος με 1. Ο αριθμός αυτός προστίθεται στην μεταβλητή total_hits.

Σε περίπτωση overflow ο πίνακας backbone μας δίνει ένα index που είναι μικρότερο του -1. Κάνοντας το index αυτό θετικό παίρνουμε τη διεύθυνση στην οποία θα βρούμε τα indexes της εισόδου μας στον πίνακα overflow. Οπότε πηγαίνουμε στον πίνακα overflow και για κάθε θετικό αριθμό που διαβάζουμε αυξάνουμε την μεταβλητή num_hits και επαναλαμβάνουμε την παραπάνω διαδικασία αποθηκεύοντας αυτή τη φορά τα offset μας στην θέση offset_pairs [total_hits+num_hits] κάθε φορά. Τέλος επιστρέφουμε τον αριθμό num_hits οποίος και αυτός με τη σειρά του θα προστεθεί στην μεταβλητή total_hits.

Τα παραπάνω σχεδιαγράμματα ροής είναι γενικού τύπου και εφαρμόζονται σε όλες τις συναρτήσεις τύπου ScanSubject. Εμείς αποφασίσαμε να υλοποιήσουμε μια συνάρτηση από όλες. Η συνάρτηση που επιλέξαμε να υλοποιήσουμε είναι η s_BlastSmallNaScanSubject_8_4 η οποία είναι και η πιο γρήγορή σύμφωνα με τους ερευνητές του NCBI και είναι επίσης η συνάρτηση η οποία χρησιμοποιείται κατά κόρον από τους περισσότερους ερευνητές βιολογίας. Η συνάρτηση αυτή ψάχνει seeds 11 χαρακτήρων .Το μέγεθος λέξης (w-mer) που χρησιμοποιείται για την συνάρτηση αυτή είναι ίσο με 8 χαρακτήρες και το stride είναι ίσο με 4. Για την συνάρτηση μας χρησιμοποιήσαμε τον SmallNaLookupTable, την συνάρτηση SMALL_NA_ACCESS_HITS και τέλος την συνάρτηση s_BlastSmallNaRetrieveHits. Παρακάτω βλέπουμε έναν πίνακα[24] που μας δείχνει τον λόγο που οι περισσότεροι ερευνητές επιλέγουν τη συνάρτηση αυτή.

W	Time (s)	Words	Hits	Matches
8	15,9	44.587	118.941	130
9	6,8	44.586	39.218	123
10	4,3	44.585	15.321	114
11	3,5	44.584	7.345	106
12	3,2	44.583	4.197	98

Πίνακας 14 Στατιστικές μετρήσεις του BLASTn χρησιμοποιώντας διαφορετικά μεγέθη λέξεων [24]

Βλέπουμε ότι τα matches δηλαδή οι ταυτοποιήσεις είναι 106 για μέγεθος λέξης (w-mer) 11 και 130 για μέγεθος λέξης 8. Παρόλα αυτά ο απαιτούμενος χρόνος για μέγεθος λέξης 11 είναι 3,5 sec ενώ για 8 είναι 15,9.

Για την συνάρτηση αυτή παίρνουμε κάθε φορά ένα κομμάτι βάσης μεγέθους 16 bit (16bit = 8 χαρακτήρες βάσης). Αμέσως μετά την επεξεργασία του κομματιού αυτού βάσης ο αλγόριθμος κάνει ένα δεξί shift 4 χαρακτήρων και παίρνει τους επόμενους 8 χαρακτήρες βάσης (αυτό συμβαίνει εξαιτίας του stride που σε αυτή τη συνάρτηση είναι ίσο με 4). Να πούμε στο σημείο αυτό ότι η συνάρτηση SMALL_NA_ACCESS_HITS καλείται 8 φορές σε κάθε επανάληψη. Σε κάθε επανάληψη ο αλγόριθμος επεξεργάζεται συνολικά 72 bit βάσης (Παίρνει πρώτα μια ποσότητα 16 bit και τις επόμενες 7 φορές ποσότητες των 8 bit, έτσι έχουμε $16 + 7 * 8 = 72$ bit). Αφού επεξεργαστεί όλα αυτά τα δεδομένα ελέγχει αν το scan_range[0] είναι μικρότερο από το scan_range[1] και αν είναι τότε εκτελεί την επανάληψη άλλη μια φορά.

4 Υλοποίηση απομακρυσμένης κλήσης συναρτήσεων (Remote Procedure Call) μέσω της πλατφόρμας RE.DO.FPGA

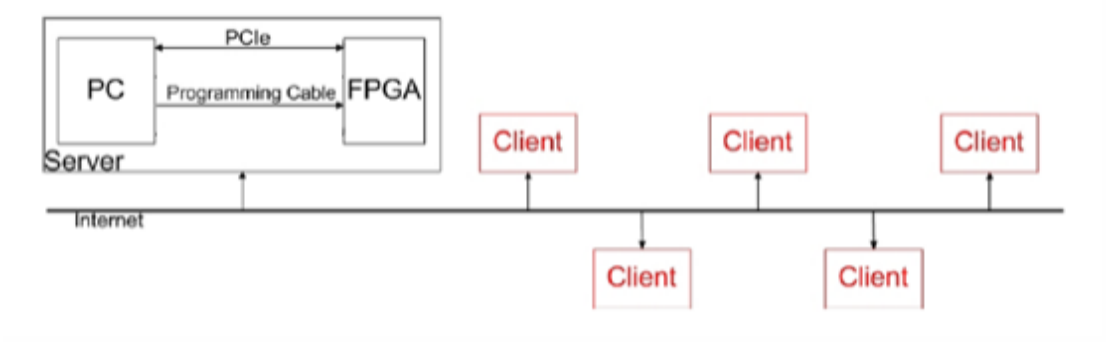
Στο κεφάλαιο αυτό θα αναλύσουμε και θα περιγράψουμε τον τρόπο με τον οποίο πραγματοποιήσαμε Remote Procedure Call μέσω της πλατφόρμας RE.DO.FPGA (Remote Download FPGA [34], παράλληλα με το λογισμικό του NCBI BLAST. Στη συνέχεια θα περιγράψουμε τη σχεδίαση της αρχιτεκτονικής που αναπτύξαμε στο hardware για την εύρεση των hits σύμφωνα με την λειτουργία της συνάρτησης *s_BlastSmallNaScanSubject_8_4* την οποία στη συνέχεια αντικαταστήσαμε.

Για την απομακρυσμένη χρήση του διαύλου PCI-express χρησιμοποιήσαμε την πλατφόρμα Re.Do.FPGA. Η πλατφόρμα Re.Do.FPGA είναι μια πλατφόρμα μέσω της οποίας μπορούμε εύκολα να χρησιμοποιήσουμε το πρωτόκολλο MTP (MHL Transfer Protocol) και επίσης τον οδηγό χρήσης του PCI-express [35] με σκοπό την εύκολη υλοποίηση πλήρων συστημάτων.

Η υλοποίηση της αρχιτεκτονικής σχεδίασης έγινε με τα εργαλεία XILINX ISE Design Suite 10.1 και XILINX EDK Design Suite 10.1.

4.1 Υλοποίηση της απομακρυσμένης κλήσης συναρτήσεων μέσω της πλατφόρμας Re.Do.FPGA

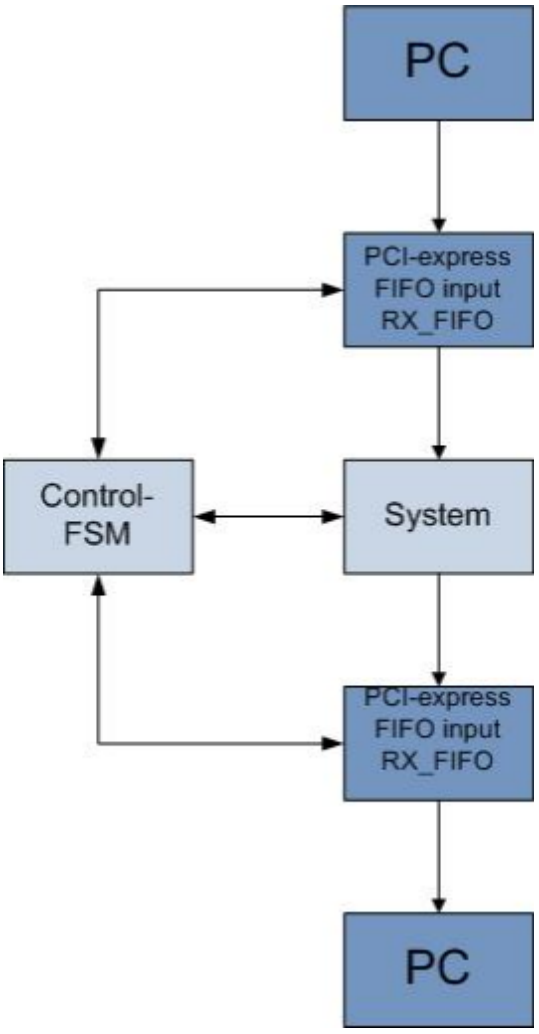
Η πλατφόρμα που χρησιμοποιήσαμε για την απομακρυσμένη χρήση του PCI-express είναι η Re.Do.FPGA η οποία έχει αναπτυχθεί από την ερευνητική ομάδα του Πολυτεχνείου Κρήτης [34]. Με το εργαλείο αυτό μας επιτρέπεται η χρήση της αναδιατασσόμενης συσκευής μέσω ενός μοντέλου εξυπηρετητή – πελάτη, από έναν απομακρυσμένο υπολογιστή όπως φαίνεται στο παρακάτω σχήμα.



Εικόνα 26 Σύστημα Re.Do.FPGA

Ο οδηγός του PCI- express που χρησιμοποιήσαμε έχει υλοποιηθεί από τον προπτυχιακό φοιτητή Ιωάννη Καρτσωνάκη στη διπλωματική του εργασία [35].

Πιο συγκεκριμένα, η λειτουργία του Re.Do.FPGA φαίνεται στο παρακάτω block diagram.



Εικόνα 27 Block diagram Re.Do.FPGA

Χρήση PCI-express

Μελετώντας την διπλωματική του Ιωάννη Καρτσωνάκη παρατηρούμε ότι η χρήση του PCI-express χωρίζεται σε δύο διαδικασίες. Την διαδικασία εγγραφής και αυτήν της ανάγνωσης.

- Διαδικασία εγγραφής

  Κατά την εκτέλεση του software για εγγραφή στο αναπτυξιακό μπορεί να γίνει με χρήση μόνο μιας συγκεκριμένης διεύθυνσης μνήμης. Η τιμή της διεύθυνσης αυτής είναι η εξής:

  - *write address: 0x024*

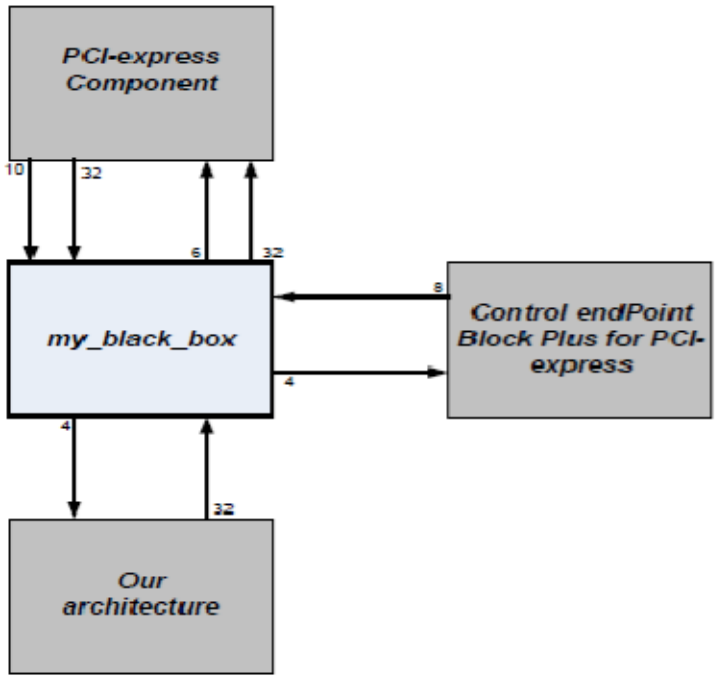
  Τα δεδομένα που στέλνονται είναι μεγέθους 32 bits. Τα δεδομένα εγγράφονται στο αναπτυξιακό μόνο όταν η διεύθυνση εγγραφής είναι η παραπάνω αλλιώς απορρίπτονται.

- Διαδικασία ανάγνωσης

  Για να γίνει ανάγνωση χρησιμοποιείται η μέθοδος της δειγματοληψίας. Αρχικά γίνεται έλεγχος για το αν υπάρχουν έγκυρα δεδομένα στη μνήμη (valid address) του αναπτυξιακού. Αν υπάρχουν έγκυρα δεδομένα τότε μόνο γίνεται ανάγνωση από την μνήμη (read address) μιας ποσότητας των 32 bits. Οι τιμές των παραπάνω διευθύνσεων είναι οι εξής:

  - *Valid address: 0x01c*
  - *Read address: 0x054*

Στην αρχιτεκτονική του PCI-express υπάρχει υλοποιημένη μια διεπαφή επικοινωνίας με τις σχεδιάσεις των χρηστών η οποία φέρει το όνομα my_black_box. Μέσω της διεπαφής αυτής μπορούμε να φέρουμε σε επαφή τη σχεδίαση μας με το εκάστοτε λογισμικό που χρησιμοποιούμε. Στο παρακάτω block diagram φαίνεται η λειτουργία της διεπαφής αυτής.



Εικόνα 28 Η διεπαφή my_black_box και η επικοινωνία με το PCI-express

Τα σήματα εισόδου και εξόδου καθώς και η χρησιμότητά τους φαίνονται στους παρακάτω πίνακες.

Σήματα εισόδου my_black_box	**Χρησιμότητα σημάτων**
PCIE_CLK	Ρολόι του PCI-express στα 62,5 MHz
RST	Reset συστήματος
FSL_RST	Επέμβαση χρήστη για αρχικοποίηση συστήματος εν ώρα λειτουργίας
FSL_CLK	Ρολόι microblaze
RX_engine_data	Δεδομένα εισόδου των 32 bits που λαμβάνουμε από το PCI-express
RX_engine_WR_EN	Σήμα ενεργοποίησης εγγραφής από το PCI-express
TX_engine_RD_PCIE	Σήμα ενεργοποίησης ανάγνωσης από το PCI-express

TX_fiflo_full	Σήμα ελέγχου για το αν η TX_FIFO είναι γεμάτη
TX_fifo_empty	Σήμα ελέγχου για το αν η TX_FIFO είναι άδεια
RX_fifo_full	Σήμα ελέγχου για το αν η RX_FIFO είναι γεμάτη
RX_fifo_empty	Σήμα ελέγχου για το αν η RX_FIFO είναι άδεια

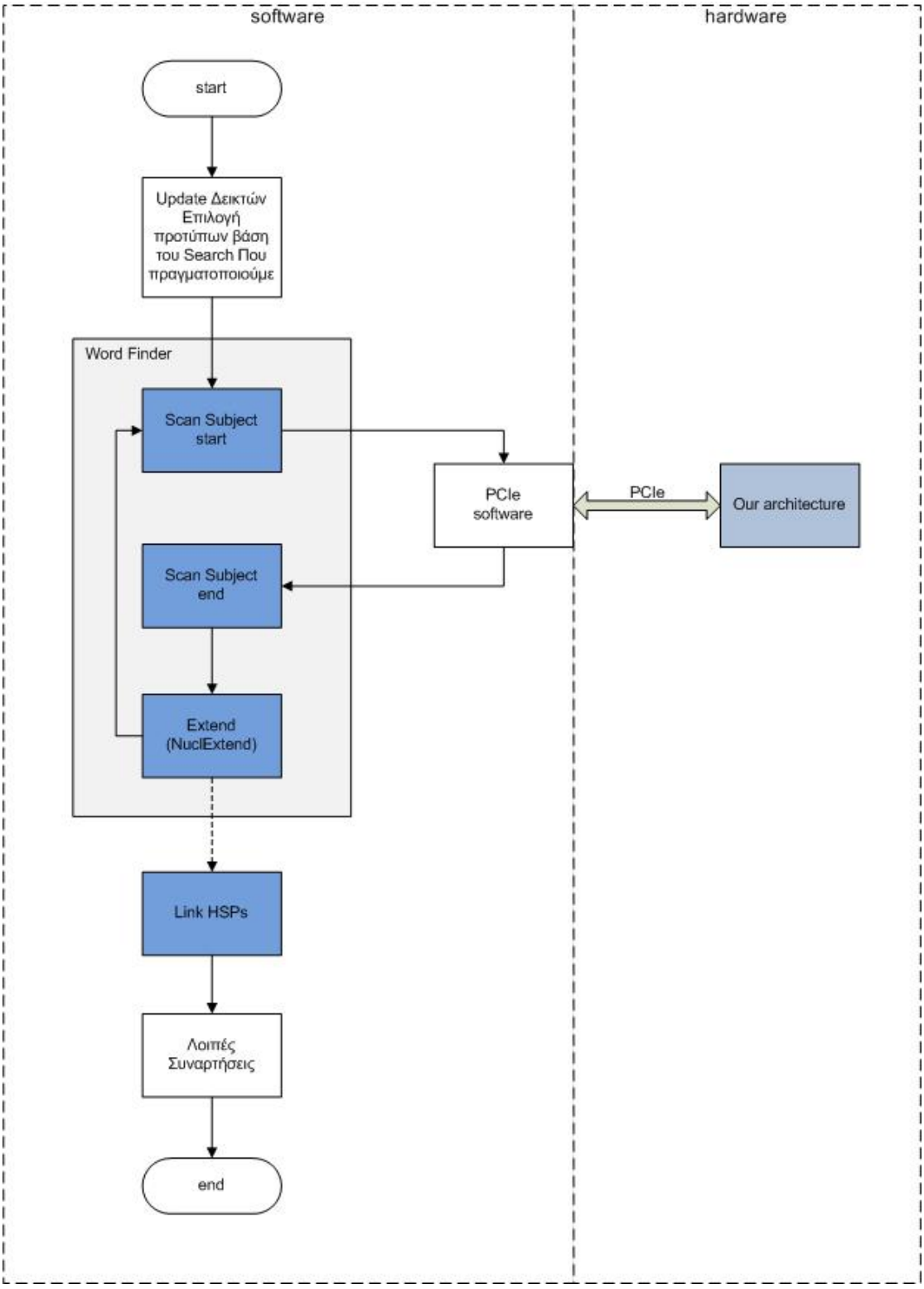
Πίνακας 15 Σήματα εισόδου my_black_box

Σήματα εξόδου my_black_box	**Χρησιμότητα σημάτων**
RX_engine_clk	Ρολόι ανάγνωσης της RX_FIFO από το PCI-express
RX_engine_RD_EN	Σήμα λήψης ενεργοποίησης για ανάγνωση από την RX_FIFO
TX_engine_data	Δεδομένα αποστολής των 32 bits προς το PCI-express
TX_engine_CLK	Ρολόι εγγραφής της TX_FIFO από το PCI-express
TX_engine_WR_EN	Σήμα ενεργοποίησης εγγραφής από την TX_FIFO
TX_engine_RD_EN	Σήμα ενεργοποίησης ανάγνωσης από την TX_FIFO
FIFO_RST	Αρχικοποίηση των FIFO του συστήματος

Πίνακας 16 Σήματα εξόδου my_black_box

Αρχικός στόχος της υλοποίησης μας ήταν η **"**ένωση**"** του αλγόριθμου NCBI Blast με την πλατφόρμα της FPGA μέσω της χρήσης του PCIe και του εργαλείου Re.Do.FPGA. Αρχικά, πειραματιστήκαμε τοποθετώντας μία κλήση προς την πλατφόρμα της FPGA στο σώμα της συνάρτησης *s_BlastSmallNaScanSubject_8_4*. Στη σχεδίαση της FPGA υλοποιήσαμε έναν πειραματικό μετρητή τον οποίο τον συνδέσαμε με την διεπαφή my_black_box. Με τις κατάλληλες αλλαγές στο λογισμικό του NCBI BLAST καταφέραμε να πραγματοποιήσουμε την απομακρυσμένη κλήση συναρτήσεων, δηλαδή να παγώνουμε τη ροή του προγράμματος του Blast αλγορίθμου, να στέλνουμε τιμές στη σχεδίαση μας στην FPGA μέσω του PCIe, και αφού στείλουμε τιμές πίσω στο software να συνεχιστει η κανονική εκτέλεση. Τελικά, η ορθή λειτουργία του συστήματος επιβεβαιώθηκε από τα σωστά αποτελέσματα που πήραμε πίσω. Πλέον είχαμε υλοποιήσει σωστά την απομακρυσμένη κλίση συναρτήσεων για λογισμικό του NCBI BLAST.

Στο παρακάτω σχήμα φαίνεται το συνολικό σύστημα που σχεδιάσαμε σε αυτό το στάδιο της εργασίας.



Εικόνα 29 Το συνολικό σύστημα Re.Do.FPGA

4.2 Περιγραφή της σχεδίασης και της υλοποίησης της συνάρτησης *s_BlastSmallNaScanSubject_8_4 σε Hardware*

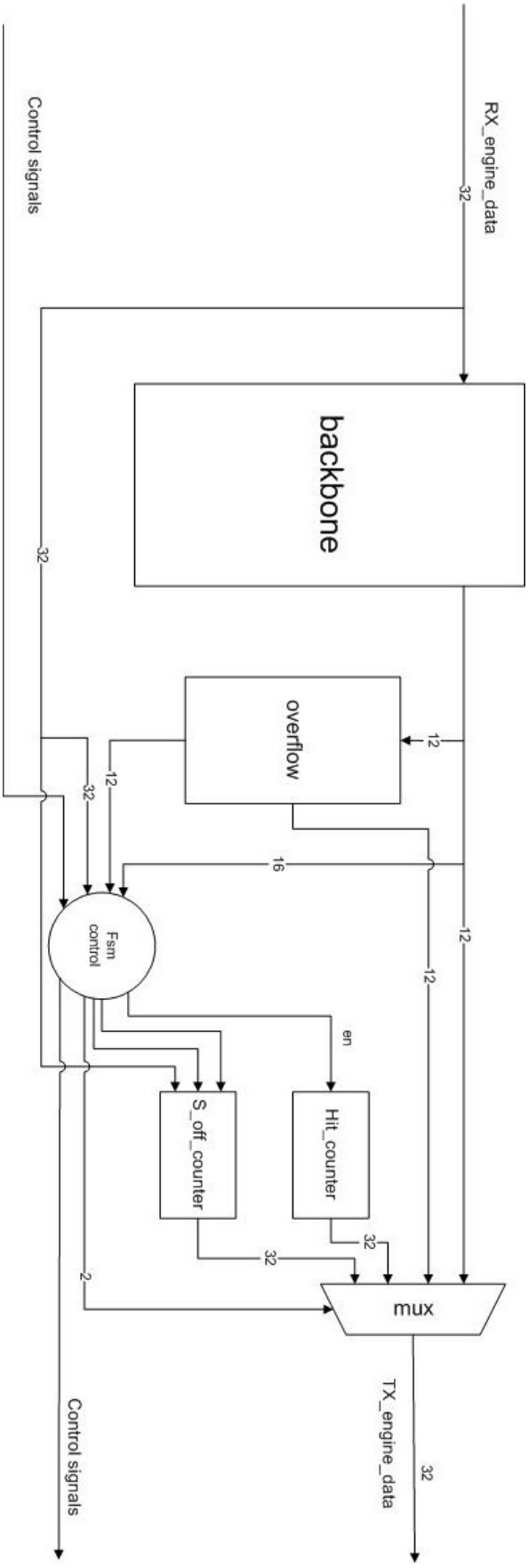
Σε αυτή την παράγραφο θα περιγράψουμε την αρχιτεκτονική της σχεδίασης μας σε hardware και η οποία υλοποιεί την συνάρτηση του software *s_BlastSmallNaScanSubject_8_4*.

Η συνάρτηση όπως έχουμε ήδη αναφέρει, πριν ξεκινήσει να παίρνει τμήματα βάσης ως είσοδο πρέπει να κάνει κάποιες αρχικοποιήσεις. Στην σχεδίαση μας χρειάζεται μόνο να αρχικοποιήσουμε το scan_range[0] που μας υποδεικνύει το αριστερό άκρο της του κομματιού της βάσης μέσα στο οποίο θα κάνουμε την αναζήτηση για hits.

Μετά την αρχικοποίηση έρχονται τα καθαρά δεδομένα (λέξεις βάσης) που είναι 16 bits κάθε φορά. Τα 16 bits αυτά αντιπροσωπεύουν μια λέξη 8 χαρακτήρων (A, C, G, T) . Η αντιστοιχία όπως έχουμε πει και προηγουμένως γίνεται με τον εξής τρόπο: *A => "00", C => "01", G => "10", T => "11"* .

Τα δεδομένα μπαίνουν ως είσοδος από το σήμα RX_engine_data στο module Backbone όπου γίνεται η σύγκριση με το query. Αν βρεθεί ένα hit τότε στην έξοδο, που είναι το σήμα TX_engine_data, βγάζουμε σε 2 κύκλους τις αντίστοιχες θέσεις από τη βάση δεδομένων και το query όπου είχαμε match. Το module hit counter μετρά τον συνολικό αριθμό των hit που έχουμε πετύχει και το module s_off_counter μετρά την θέση της βάσης δεδομένων όπου είχαμε hit. Αν τα hit είναι περισσότερα από ένα τότε το Overflow unit μας δίνει με σειρά όλες τις θέσεις του query όπου είχαμε hit. Η FSM ελέγχει τη συνολική λειτουργία του κυκλώματος. Για τη διασφάλιση της σωστής λειτουργίας του συστήματος η υλοποίηση της αρχιτεκτονικής μας έγινε "bit to bit", ώστε να στέλνουμε πίσω στο software όλες τις πληροφορίες που χρειάζεται για να μην παρουσιάσει κάποιο σφάλμα.

Παρακάτω φαίνεται το block diagram της σχεδίασης της συνάρτησης *s_BlastSmallNaScanSubject_8_4* σε hardware.



Εικόνα 30 Η σχεδίαση της συνάρτησης *s_BlastSmallNaScanSubject_8_4* σε hardware

Παρακάτω αναλύουμε τα modules που σχεδιάστηκαν καθώς και κάποιες άλλες χρήσιμες πληροφορίες.

Μνήμες: Χρησιμοποιήσαμε 2 μνήμες όπου έχουμε αποθηκευμένα τα indexes του της εισόδου. Η μνήμη backbone έχει βάθος 65.536 θέσεις (4^8 θέσεις επειδή αναζητάμε λέξεις μεγέθους 8) Επίσης χρησιμοποιήσαμε άλλη μια μνήμη για το overflow η οποία έχει βάθος στην περίπτωση μας 4096 θέσεων. Με μια τέτοια μνήμη μπορούμε να εξυπηρετήσουμε μια είσοδο μεγέθους το πολύ 2047 χαρακτήρων. Οι μνήμες έχουν δημιουργηθεί από το εργαλείο *Core Generator* του Xilinx ISE και τα αρχεία αρχικοποίησης (.coe) έχουν δημιουργηθεί με ένα script γραμμένο σε γλώσσα C.

Στο σημείο αυτό θα πούμε λίγα λόγια για την διαστασιολόγηση των μνημών. Αναλόγως με το w-mer με το οποίο ψάχνουμε για κάποιο hit υπολογίζουμε και το βάθος του πίνακα. Το βάθος μνήμης υπολογίζεται από το αποτέλεσμα $\mathbf{4^{w\text{-}mer}}$. Πιο αναλυτικά έχουμε:

w-mer	**Βάθος μνήμης**
4	4^4
5	4^5
6	4^6
7	4^7
8	4^8
9	4^9
10	4^{10}
11	4^{11}

Πίνακας 17 Διαστασιολόγηση backbone

Το πλάτος σε bit της μνήμης πρέπει να χωράει τα indexes της εισόδου (query). Πρέπει να είναι μεγαλύτεροο από *query * 2 +1*.

Πλάτος μνήμης σε bit	Query length
10	<511
11	<1023
12	<2047
13	<4095
14	<8191
15	<16383
16	<32767

Πίνακας 18 Διαστασιολόγηση backbone

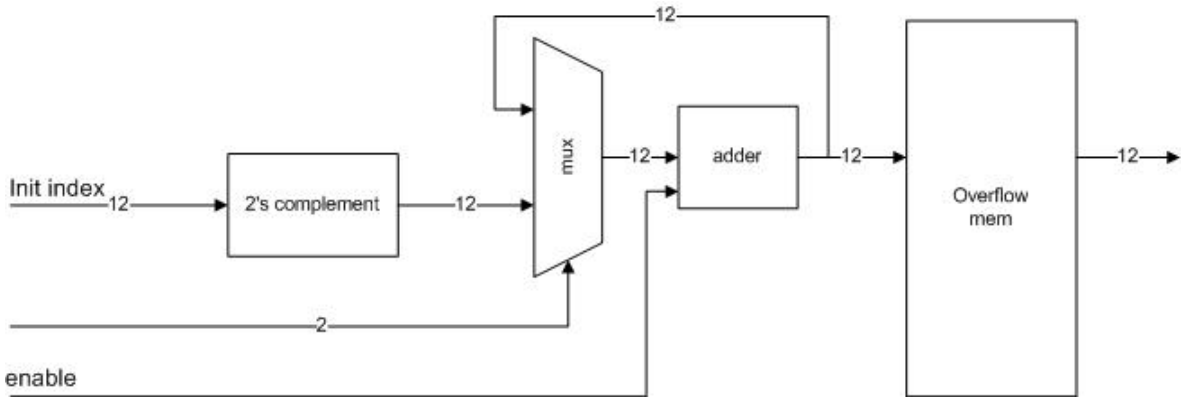
Backbone module

Περιέχει μια μνήμη μεγέθους 4^8. Το backbone module δέχεται την είσοδο από το PCI-express η οποία είναι 32 bit. Από τα 32 ωφέλιμη πληροφορία περιέχουν τα 16 μόνο bit στα δεξιά τα οποία είναι και το τμήμα της βάσης που εξετάζουμε για hit. Μας επιστρέφει το index το οποίο μπαίνει ως είσοδος στο overflow module γιατί σε περίπτωση που μας επιστραφεί αριθμός μικρότερος του '-1' αυτός θα χρησιμοποιηθεί ως διεύθυνση για τη μνήμη overflow αλλά και στο FSM control module για να γνωρίζουμε σε ποια κατάσταση θα γίνει η επόμενη μετάβαση.

Overflow module

Περιέχει μια μνήμη μεγέθους 2^{12} με όνομα overflow_mem. Το overflow module δέχεται μια τιμή από το backbone module η οποία γίνεται η αρχική διεύθυνση της μνήμης overflow_mem και από εκεί και πέρα προσπελαύνουμε τη μνήμη μέχρι αυτή

να μας δώσει μια αρνητική τιμή. Τότε σταματάμε την προσπέλαση και συνεχίζουμε την εκτέλεση του αλγορίθμου.



Εικόνα 301 Σχεδίαση του module overflow

Hit counter module

Το hit counter module είναι ένας μετρητής που αρχικοποιείται με την τιμή 0 κάθε φορά. Σε κάθε hit που γίνεται είτε από το backbone module είτε από το overflow module ο μετρητής αυξάνει κατά ένα. Όταν ο αλγόριθμος φτάσει στο τέλος το module αυτό μας επιστρέφει τον συνολικό αριθμό hits που έχουν γίνει.

Soff counter module

Το Soff counter module κρατάει το σημείο της βάσης το οποίο εξετάζουμε. Η πρώτη τιμή που λαμβάνουμε στη σχεδίαση μας κρατείται σε έναν καταχωρητή μέσα στο module αυτό και από εκεί και πέρα για κάθε νέα βάση που παίρνουμε αυτός ο μετρητής αυξάνεται κατά 4 (stride=4).

Mux module

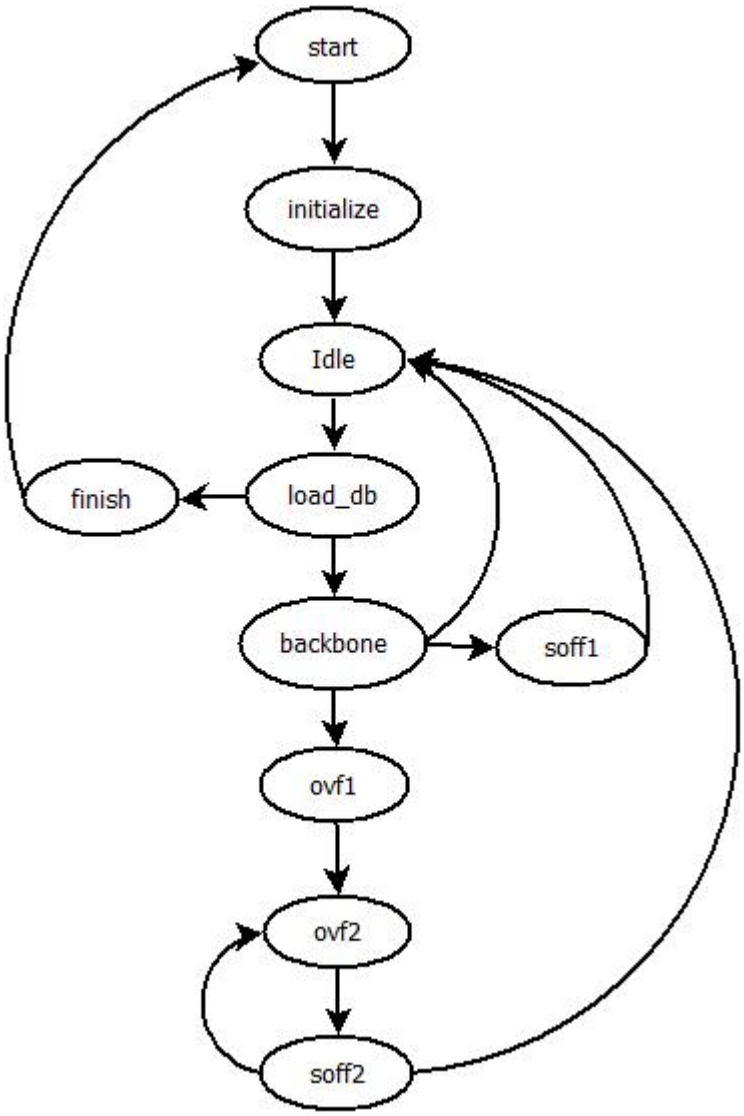
Με το module αυτό ελέγχουμε την έξοδο μας, δηλαδή τα δεδομένα που γράφουμε στην TX_fifo. Παίρνει ως εισόδους τα δεδομένα που μας επιστρέφει το module backbone, τα δεδομένα του module overflow, τα δεδομένα του soff counter και αυτά του hit counter και ανάλογα με τον έλεγχο από την FSM στέλνει την αντίστοιχη έξοδο στο PCIe. Οι τιμές τις οποίες επιστρέφει ο πολυπλέκτης είναι:

- Έξοδος από το backbone module (Θέση όπου έγινε το hit στο query)
- Έξοδος από το overflow module (Θέση όπου έγινε το hit στο query)
- Έξοδος από το soff_counter module (Θέση όπου έγινε το hit στη βάση δεδομένων)
- Έξοδος από το hit_counter module (Ο αριθμός των hit. Είναι η θέση του πίνακα όπου βρίσκονται τα ζευγάρια των hits)

Στο σημείο αυτό καλό θα ήταν να αναφερθούμε στη λειτουργία του module control-FSM που υλοποιήσαμε.

FSM control module

Το module αυτό είναι μια μηχανή πεπερασμένων καταστάσεων. Ρυθμίζει τη ροή των δεδομένων μας και μας εγγυάται τη σωστή επικοινωνία με το control unit του PCI-express μέσω του οποίου επικοινωνούμε με τον «έξω κόσμο».



Εικόνα 32 Οι καταστάσεις του module control_FSM

Έχει συνολικά 10 καταστάσεις:

- *start:* Η αρχική μας κατάσταση. Μένουμε σε αυτή την κατάσταση μέχρι να εμφανιστεί κάποια τιμή στη fifo εισόδου (RX_Fifo) δηλαδή μέχρι να ενεργοποιηθεί το σήμα rx_fifo_empty. Σε αυτή την περίπτωση δίνουμε τιμή στο σήμα RX_engine_Read_EN<=1 και περνάμε στην επόμενη κατάσταση την initialize.
- *Initialize:* Εδώ η τιμή έχει περάσει στο σύστημα μας (αργεί έναν κύκλο από τη στιγμή που θα σηκώσουμε το RX_engine_Read_EN). Αρχικοποιούμε τον καταχωρητή που βρίσκεται στο module soff_counter, σταματάμε να διαβάζουμε από την fifo εισόδου RX_engine_Read_EN<=0 και περνάμε στην κατάσταση Idle

- *Idle:* Στην κατάσταση αυτή περιμένουμε να εμφανιστεί μια τιμή στη fifo εισόδου (RX_Fifo). Όταν το σήμα rx_fifo_empty = 0 σηκώνουμε το RX_engine_Read_EN και περνάμε στην επόμενη κατάσταση, την load_db.
- *Load_db:* Στην κατάσταση αυτή παίρνουμε την τιμή από τη RX_fifo. Αν η τιμή αυτή είναι ίση με -1 πηγαίνουμε στην κατάσταση finish όπου και επιστρέφουμε τον συνολικό αριθμό των hit. Σε περίπτωση που η τιμή αυτή είναι διάφορη του -1 τότε περνάμε στην κατάσταση backbone.
- *Backbone:* Στην κατάσταση αυτή ο πίνακας backbone μας έχει επιστρέψει μια τιμή (backbone_data). Εδώ έχουμε 3 περιπτώσεις. Αν backbone_data=-1 τότε έχουμε miss και επιστρέφουμε στην κατάσταση Idle. Αν backbone_data>-1 τότε έχουμε hit σηκώνουμε το σήμα TX_engine_WR_EN για να γράψουμε τιμή στη TX_fifo (fifo εξόδου) και κάνουμε το σήμα select του mux module ίσο με '00'. Μετά περνάμε στην κατάσταση soff1. Αν Αν backbone_data<-1 τότε έχουμε overflow και περνάμε στην κατάσταση ovf1.
- *Ovf1:* Εδώ έχουμε δώσει τιμή στον καταχωρητή που χρησιμοποιούμε για διευθυνσιοδότηση της μνήμης overflow_mem. Στον επόμενο κύκλο θα έχουμε πάρει αποτέλεσμα.
- *Ovf2*: Εδώ έχουμε πάρει αποτέλεσμα από τη μνήμη overflow mem, το οποίο και γράφουμε στην TX fifo. Ταυτόχρονα αυξάνουμε και τον καταχωρητή διεύθυνσης κατά ένα.
- Soff1: Στην κατάσταση αυτή γράφουμε στην TX_fifo το σημείο της βάσης στο οποίο έγινε το hit. Μετά επιστρέφουμε στην κατάσταση Idle.
- Soff2: Όμοια με την κατάσταση Soff1. Η μόνη διαφορά είναι ότι στο σημείο αυτό έχουμε ήδη πάρει νέα τιμή από τη μνήμη overflow mem η οποία εάν είναι μεγαλύτερη του 0 συνεχίζουμε να παίρνουμε τιμές από το overflow module. Αν η τιμή που πάρουμε γίνει μικρότερη του 0 τότε επιστρέφουμε στην κατάσταση Idle.
- Finish: Στην κατάσταση αυτή επιστρέφουμε τον τελικό αριθμό των hit. Επόμενη κατάσταση είναι η κατάσταση start όπου και περιμένουμε να αρχίσει πάλι ο αλγόριθμος.

Όλο μας το σύστημα καθώς και οι TX_FIFO και RX_FIFO έχουν συγχρονιστεί με το PCIE_CLK (RXCLK<= PCIECLK , TCLK<= PCIECLK). Το ρολόι μας δηλαδή όπως φαίνεται και στον παραπάνω πίνακα είναι ρυθμισμένο στα 62.5 MHz. Αυτό

σημαίνει ότι έχουμε περιορισμό στην απόδοση του συστήματος μας γιατί η επικοινωνία με τον «έξω κόσμο» γίνεται με τη συχνότητα του ρολογιού του PCI-express η οποία είναι αρκετά χαμηλή (62.5 MHz).

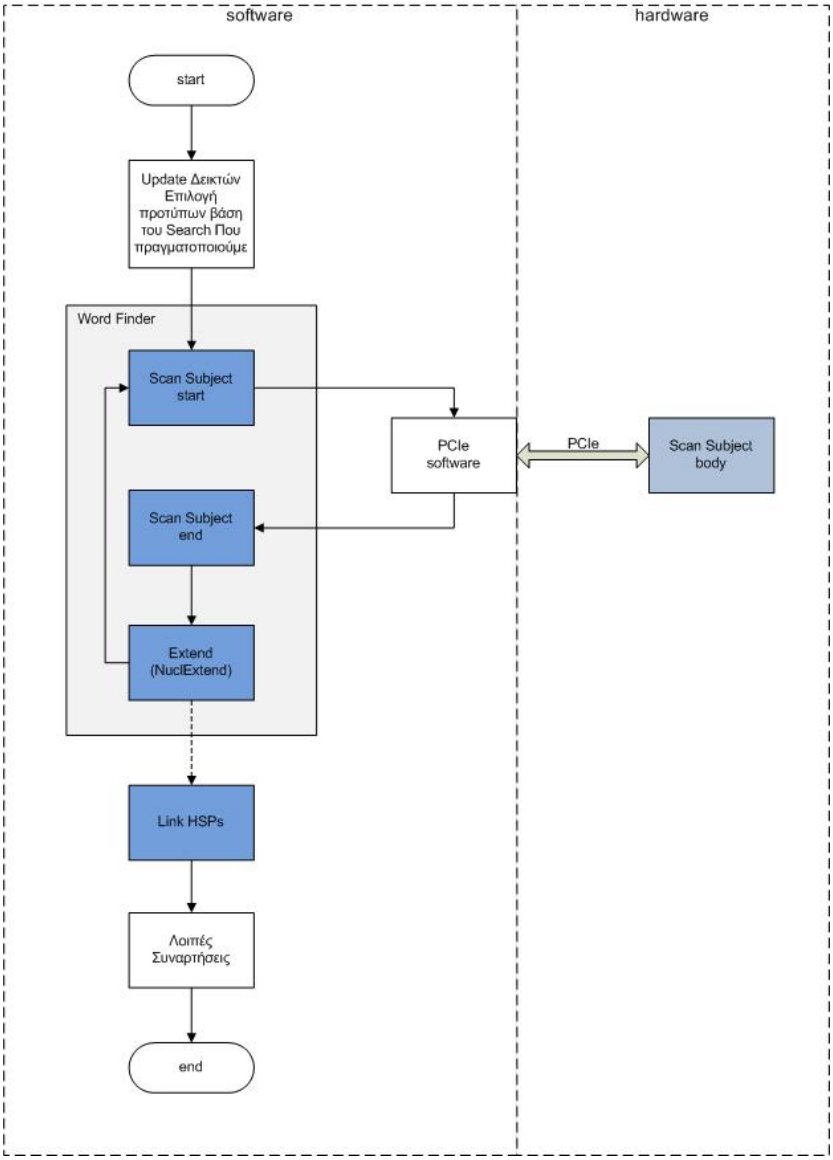
Όταν το σήμα εισόδου TX_engine_RD_PCIE είναι ίσο με το μηδέν τότε και το σήμα TX_engine_RD_EN παίρνει την ίδια μηδενική τιμή. Το ίδιο ισχύει και όταν παίρνει τη τιμή ένα. Εάν το σήμα RX_engine_RD_EN είναι ίσο με μηδέν τότε αυτό σημαίνει ότι δεν υπάρχουν δεδομένα για εγγραφή στην FIFO εισόδου και άρα παραμένουμε στην ίδια κατάσταση με τα σήματα TX_engine_WR_EN και RX_engine_RD_EN να είναι επίσης ίσα με το μηδέν. Εάν ενεργοποιηθεί τότε πάμε στην επόμενη κατάσταση και μέσα σε αυτήν ενεργοποιούμε τα παραπάνω σήματα εξόδου.

4.3 Τελικό σύστημα απομακρυσμένης κλίσης συναρτήσεων μέσω της πλατφόρμας Re.Do.FPGA για το λογισμικό NCBI BLAST.

Στο σημείο αυτό συνδέσαμε τη σχεδίαση μας με τον υπόλοιπο αλγόριθμο. Για λόγους ευκολίας ονομάσαμε Scan Subject body όλο το σύστημα που σχεδιάσαμε στο hardware και το οποίο υλοποιεί ακριβώς την συνάρτηση του software που αντικαταστάθηκε.

Το σημείο το οποίο αφαιρέσαμε από τον Blast αλγόριθμο είναι το σημείο στο οποίο γίνεται η αναζήτηση στη βάση για seeds. Μέσω ενός κατάλληλου software που προσθέσαμε, αρχικοποιούμε το σύστημα μας, και στη συνέχεια στέλνουμε τιμές στην σχεδίαση μας για επεξεργασία. Μετά στέλνουμε τα δεδομένα πίσω στο software ώστε να συνεχίσει ο αλγόριθμος. Η υλοποίηση έγινε για τη συνάρτηση *s_BlastSmallNaScanSubject_8_4* η οποία είναι η πλέον χαρακτηριστική της οικογένειας συναρτήσεων *ScanSubject.*

Παρακάτω στο σχήμα 26 φαίνεται το συνολικό σύστημα μας. Αριστερά φαίνεται το software και δεξιά το hardware που χρησιμοποιήσαμε.



Εικόνα 33 Διεπαφή BLASTn

Τα δεδομένα τα οποία μεταφέρουμε στη σχεδίαση μας μέσο του διαύλου PCIe είναι τα εξής:

- **Input:** Δεδομένα της βάσης δεδομένων καθώς και τιμές για την αρχικοποίηση του συστήματος.
- **Output:** Τις θέσεις στις οποίες έχουμε hit στη βάση (s_off) και query (q_off) αντίστοιχα, τον αριθμό του τρέχοντος hit και τέλος τον συνολικό αριθμό των hits τα οποία πετύχαμε στη συγκεκριμένη φάση της αναζήτησης.

5 Επιβεβαίωση λειτουργίας, πειραματικά αποτελέσματα και αξιολόγηση συστήματος

Στο κεφάλαιο αυτό περιγράφεται η διαδικασία επιβεβαίωσης λειτουργίας του συστήματος μας καθώς και η απόδοση του. Η υλοποίηση της αρχιτεκτονικής σχεδίασης έγινε με τα εργαλεία XILINX ISE Design Suite 10.1 και XILINX EDK Design Suite 10.1. Το τελικό σύστημα υλοποιήθηκε σε ένα board της Xilinx ML505 με Virtex 5 LX110T FPGA. Τα αποτελέσματα του τελικού συστήματος μας συγκρίθηκαν και πιστοποιήθηκαν με τα αντίστοιχα της επίσημης διανομής του NCBI BLAST.

5.1 Επιβεβαίωση λειτουργίας

Κατά τη διαδικασία της υλοποίησης της αρχιτεκτονικής, προσομοιώσαμε πρώτα τη λειτουργία της συνάρτησης χρησιμοποιώντας τον Xilinx ISE Simulator για να διαπιστώσουμε τη σωστή λειτουργία της συνάρτησης που υλοποιήσαμε. Τα δεδομένα που χρησιμοποιήσαμε στη φάση αυτή ήταν μια βάση δεδομένων που φτιάξαμε εμείς και κάποια μικρά queries φτιαγμένα από εμάς. Τα ίδια δεδομένα χρησιμοποιήσαμε και στον αλγόριθμο του NCBI BLASTn και πήραμε πίσω τα ίδια αποτελέσματα.

Στη συνέχεια υλοποιήσαμε την αρχιτεκτονική και τη συνδέσαμε με το PCI-e. Για να διαπιστώσουμε τη σωστή λειτουργία χρησιμοποιήσαμε τα ίδια δεδομένα και αφού επιδιορθώσαμε όλα τα bug που μας παρουσιάστηκαν ξεκινήσαμε να παίρνουμε μετρήσεις.

5.2 Πειραματικά αποτελέσματα

Αρχικά περιγράφουμε τους πόρους που χρησιμοποιήθηκαν για την υλοποίηση στην FPGA. Παρατηρούμε ότι η σχεδίαση μας δεν καταλαμβάνει πολλούς πόρους, όμως αξίζει να σημειωθεί ότι το PCI-express σαν αρχιτεκτονική στο EDK χρησιμοποιεί 16

από τις 148 διαθέσιμες BRAM/FIFO που σημαίνει ότι από μόνο του καταλαμβάνει περίπου ένα ποσοστό της τάξεως του 10% της FPGA.

Device utilization summary	Used	Available	Utilization
Number of slice registers	36	69.120	1%
Number of slice LUT's	136	69.120	1%
Number of Occupied Slices	67	17.280	1%
Number of bonded IOBs	75	640	11%
Number of Block RAM/FIFO	41	148	28%
Number of BUFG/BUFGCTRLs	1	32	3%

Πίνακας 19 Χρήση πόρων FPGA στην αρχιτεκτονική της σχεδίασης

Device utilization summary	Used	Available	Utilization
Number of slice registers	3.566	69.120	5%
Number of slice LUT's	17.493	69.120	25%
Number of Occupied Slices	5.812	17.280	33%
Number of bonded IOBs	6	640	1%
Number of Block RAM/FIFO	66	148	44%
Number od BUFG/BUFGCTRLs	4	32	12%

Πίνακας 20 Χρήση πόρων FPGA στην υλοποίηση του συστήματος

Παρατηρούμε , επίσης, πως το critical resource του συστήματος μας είναι τα block RAM/FIFO με ποσοστά 28% και 44% στην αρχιτεκτονική της σχεδίασης και στην υλοποίηση του συστήματος μας αντίστοιχα. Αυτό συμβαίνει επειδή η σχεδίαση μας είναι πολύ μικρή σε σχέση με τα δεδομένα τα οποία διαχειριζόμαστε.

Συχνότητα ρολογιού

Η συχνότητα του ρολογιού που χρησιμοποιήθηκε στο PCI-e είναι 62,5 MHz σε αντίθεση με την Post Place and Route σχεδίαση όπου η αντίστοιχη συχνότητα έφτασε τα 126,16 MHz. Όσον αφορά τη συνολική μας σχεδίαση χρησιμοποιήσαμε τη συχνότητα του ρολογιού που χρησιμοποιήθηκε στο PCI-e που είναι και το πιο αργό

ρολόι. Στον παρακάτω πίνακα φαίνεται η συχνότητα των ρολογιών που χρησιμοποιήθηκαν στην προσομοίωση και στο hardware αντίστοιχα.

	Συχνότητα (MHz)
Σχεδίαση Post Place and Route	126,16
Σύστημα Σχεδίαση και PCI-express	62,5

Πίνακας 21 Συχνότητα της σχεδίασης και του συστήματος μας

Για τη διαπίστωση της σωστής λειτουργίας χρησιμοποιήσαμε 2 διαφορετικές εισόδους και 3 διαφορετικές βάσεις δεδομένων. Τα query εξαιτίας του τρόπου με τον οποίο χρησιμοποιούνται δεν επηρεάζουν πολύ την απόδοση του αλγορίθμου. Όπως αναφέραμε και προηγουμένως 2 είναι οι βασικές αιτίες που κάνουν τον αλγόριθμο να καθυστερεί:

1. Το μέγεθος της λέξης με το οποίο κάνουμε το seeding
2. Το μέγεθος της βάσης δεδομένων που χρησιμοποιούμε.

Όλα τα υπόλοιπα προσθέτουν απειροελάχιστο χρόνο στη συνολικό χρόνο του αλγορίθμου.

Query	**Size**
1	480 chars
2	860 chars

Πίνακας 22 Είσοδοι που χρησιμοποιήσαμε για τις μετρήσεις μας

Database	**Size**
Dataset 1	44.603.624 chars
Dataset 2	89.100.036 chars
Dataset 3	133.703.660 chars

Πίνακας 23 Βάσεις που χρησιμοποιήσαμε για τις μετρήσεις μας

Ο μέσος χρόνος εκτέλεσης της συνάρτησης ανά βάση στον υπολογιστή για κάθε διαφορετική είσοδο, ο χρόνος εκτέλεσης του αλγορίθμου σε αναδιατασσόμενη λογική καθώς και ο χρόνος software hardware φαίνεται στους παρακάτω πίνακες.

Query 1:

Database	Software	Hardware		Software-Hardware co-design	
	Time (s)	Time (s)	Speed-up	Time (s)	Speed-up
1	0,09	0,26	0,34 x	1,56	0,04 x
2	0,18	0,53	0,33 x	3,13	0,04 x
3	0,25	0,80	0,31 x	4,77	0,04 x

Πίνακας 24 Χρόνος και speedup εκτέλεσης συνάρτησης

Query 2:

Database	Software	Hardware		Software-Hardware co-design	
	Time (s)	Time (s)	Speed-up	Time (s)	Speed-up
1	0,11	0,26	0,42 x	1,56	0,06 x
2	0,20	0,54	0,37 x	3,14	0,06 x
3	0,29	0,82	0,35 x	4,78	0,06 x

Πίνακας 25 Χρόνος και speedup εκτέλεσης συνάρτησης

5.3 Αξιολόγηση Συστήματος

Μέσα στους στόχους αυτής της εργασίας ήταν η μελέτη σε μεγαλύτερο βάθος της NCBI υλοποίησης του αλγορίθμου BLAST. Δοκιμάσαμε τον αλγόριθμο με διάφορα datasets (κεφ 3) βρήκαμε ότι οι υπολογιστικά βαριές συναρτήσεις τύπου ScanSubject (22 διαφορετικές υλοποιήσεις του ίδιου αλγορίθμου) καταναλώνουν ένα αρκετά σημαντικό ποσοστό του συνολικού χρόνου εκτέλεσης του προγράμματος (από 17% έως 70%).

Επίσης σύμφωνα με τους developers του NCBI BLAST για ένα γρήγορο και τυπικό alignment συνίσταται η χρήση μεγέθους λέξης ίσο με 11 (όταν επιλέγουμε να κάνουμε το αρχικό seeding με μέγεθος λέξης ίσο με 11 τότε επιλέγεται η συνάρτηση s_BlastSmallNaScanSubject_8_4 την οποία και υλοποιήσαμε). Ο λόγος είναι ότι

χρησιμοποιώντας αυτό το μέγεθος λέξης το πρόγραμμα τρέχει πολύ πιο γρήγορα χωρίς να έχουμε αξιόλογες αποκλίσεις στα αποτελέσματα.

Έχοντας τα παραπάνω υπόψη πήραμε την απόφαση να υλοποιήσουμε την συνάρτηση s_BlastSmallNaScanSubject_8_4 για να μελετήσουμε καλύτερα τον τρόπο με τον οποίο οι developers του NCBI BLAST πέτυχαν αυτό το speedup μέσω software και αφετέρου δε να δοκιμάσουμε την απόδοση του BLAST στην αναδιατασσόμενη λογική υλοποιώντας ένα πλήρες σύστημα.

Όπως βλέπουμε στους πίνακες (κεφ 5.2) τα αποτελέσματα που παίρνουμε από τη σχεδίαση μας δεν είναι ικανοποιητικά από άποψη επιτάχυνσης του αλγορίθμου. Όσον αφορά το hardware αυτό ισχύει επειδή:

- Για τη σωστή διασφάλιση του συστήματος κάναμε κάποιες θυσίες σε χώρο. Πιο συγκεκριμένα επειδή αποφασίσαμε να υλοποιήσουμε την συνάρτηση s_BlastSmallNaScanSubject_8_4 του NCBI BLAST και όχι τον αλγόριθμο της συνάρτησης. Έτσι περιοριστήκαμε πολύ σε χώρο αφού δεν μπορούσαμε σε έναν register 32 bit (παραδείγματος χάρη τον register που κρατάει την τιμή του subject offset) να καταχωρήσουμε 2 και 3 δεδομένα.
- Το software επωφελείται της μνήμης cache όπου και μπορεί και καλεί γρήγορα τις τιμές που βρίσκονται μέσα στον πίνακα backbone και να τις συγκρίνει με το κομμάτι βάσης που έχει φορτώσει και αυτό στην ίδια μνήμη. Ο συνδυασμός της συνάρτησης που φορτώνει τη βάση στην cache μνήμη του επεξεργαστή με τη συγκεκριμένη συνάρτηση είναι πολύ βολικός όσον αφορά τους σημερινούς επεξεργαστές για αναζητήσεις των αρχικών hits.

Όσον αφορά τη λειτουργία του software με το hardware τα αποτελέσματα οφείλονται στο ότι:

- Η συχνότητα του PCI-express είναι στα 62,5 MHz. Σε άλλη περίπτωση θα είχαμε πιο γρήγορο σύστημα.
- Ο οδηγός του PCI-express λειτουργεί με Polling και όχι με Interrupts στην επικοινωνία μεταξύ εισόδου/εξόδου. Δηλαδή το πρόγραμμά μας κάθε φορά κάνει ανάγνωση μνήμης από το PC και αυτό μας καθυστερεί πολύ.

- Ο driver του PCI-express είναι υλοποιημένος ώστε να δέχεται μόνο 16Kbyte σαν είσοδο. Αυτό μας περιορίζει πολύ λόγω του ότι δεν μπορούμε να εισάγουμε μεγάλες βάσεις στο σύστημα μας.
- Κάθε φορά μέσω του PCI-express γυρνάμε 32 bits δεδομένα. Αυτό έχει ως αποτέλεσμα να πρέπει να γίνεται κάθε φορά η επεξεργασία των δύο 16 bits αποτελεσμάτων στο software, άρα έχουμε κάποια απώλεια χρόνου και σε αυτό το μέρος της υλοποίησης.
- Χρειάζεται να διαβάζουμε κάθε φορά αποτελέσματα από τον οδηγό του PCI-express ακόμα κι αν δεν έχουμε πετύχει κάποιο hit πράγμα που είναι και το πιο πιθανό.

Κάτι τελευταίο που αξίζει να αναφέρουμε στο κεφάλαιο αυτό είναι και το εξής: Η συνάρτηση που υλοποιήσαμε δίνεται από τους developers του NCBI BLASTn ως η γρηγορότερη συνάρτηση επειδή βολεύει πολύ την 64-bit αρχιτεκτονική των νέων επεξεργαστών. Μπορούνε και γεμίζουν πολύ ομοιόμορφα την cache μνήμη του επεξεργαστή με όλα τα datasets που χρειάζονται για το πρόβλημα (βάση δεδομένων – πίνακες overflow – backbone και query) ώστε να μην ξοδεύουνε πολύ χρόνο για τις μεταξύ τους πράξεις.

6 Συμπεράσματα και μελλοντικές επεκτάσεις

6.1 Συμπεράσματα

Στην παρούσα διπλωματική εργασία κάναμε επιτυχώς Remote Procedure Call τρέχοντας την NCBI υλοποίηση του αλγορίθμου BLASTn μέσω της πλατφόρμας Re.Do.FPGA.

Μελετώντας τον αλγόριθμο αυτό σε μεγαλύτερο βάθος μπορούμε να έχουμε πλέον μια καλύτερη εικόνα για ότι αφορά την NCBI υλοποίηση του και βρήκαμε τι φταίει για τα αποτελέσματα που βγάζαμε σε κάποιες TUC υλοποιήσεις του. Πιο συγκεκριμένα βρήκαμε ότι ο BLAST σαρώνει τη βάση με μια οικογένεια συναρτήσεων με όνομα ScanSubject. Η επιλογή της συνάρτησης γίνεται βάση του w-mer και το μέγεθος του query. Βρήκαμε επίσης τον τρόπο με τον οποίο οι ερευνητές του NCBI BLAST πραγματοποιούν τόσο γρήγορα την αναζήτηση μέσα στη βάση δεδομένων (χρήση της παραμέτρου stride).

Αυτό που γνωρίζαμε μέχρι τώρα ως seeding γίνεται πλέον σε δύο φάσεις αν το stride είναι μεγαλύτερο του μηδενός:

i. Γίνεται πρώτα το seeding με μέγεθος λέξης μικρότερο από αυτό που ζητάμε
ii. Αν υπάρχει hit τότε και μόνο τότε ελέγχουμε και τους υπόλοιπους χαρακτήρες για να δούμε αν όντως έχουμε ένα seed.

Με βάση αυτή τη διαδικασία έχουμε μείωση μέχρι και 75% σε συγκρίσεις και σε προσβάσεις στη μνήμη (για stride ίσο με 4) και το κυριότερο χωρίς να χάνουμε πληροφορία.

Σχεδιάσαμε μια γενική αρχιτεκτονική για τις συναρτήσεις σάρωσης της βάσης δεδομένων και υλοποιήσαμε μια συγκεκριμένη για w-mer 8 και βήμα 4.

Η πλατφόρμα Re.Do.FPGA δεν είναι αποδοτική όταν γίνεται συνεχής κλίση συναρτήσεων λόγο του μεγάλου χρόνο που καταναλώνεται σε I/O.

6.2 Μελλοντικές επεκτάσεις

Σημαντικές επεκτάσεις και αλλαγές που μπορούν να πραγματοποιηθούν είναι:

- Αλλαγές όσον αφορά τον οδηγό του PCI ως προς τον τρόπο λειτουργίας του driver (αύξηση της συχνότητας, χρήση Interrupts, αύξηση των δεδομένων για upload).
- Εμπλουτισμό της πλατφόρμας Re.Do.FPGA με εργαλεία όπως το chipscope ώστε να έχουμε μεγαλύτερο έλεγχο για την σχεδίαση μας.
- Υλοποίηση του συστήματος σε μια άλλη πλατφόρμα που να μπορεί να δείχνει στη μνήμη. Η πλατφόρμα του PCI-express για streaming δεδομένα έχει αποδειχτεί και σε προηγούμενες διπλωματικές εργασίες ότι μας φορτώνει μένα μεγάλο χρόνο σε I/O.
- Εμπλουτισμός του αλγορίθμου TUC BLAST με νέες ιδέες όπως αυτή του stride.

7 Αναφορές

1. Lesk, A. M. (2002) “Introduction to bioinformatics”
2. B. Needleman, and C. Wunsch, “A General Method Applicable to the Search for Similarities in the Amino Acid Sequence of Two Proteins,”, J. Mol. Biol., vol. 48, pp 443-453, 1970
3. T. Smith, and M. Waterman, “Identification Of Common Molecular Subsequences,” Elsevier J. Mol. Biol., vol. 147, pp 195-197, 1981
4. W. Pearson, and D. Lipman, “Improved tools for biological sequence analysis” Proceedings of the National Academic Science of the USA, vol 85, pages 2444–2448, 1988.
5. S. Altschul, W. Gish, W. Miller, and E. Myers, “Basic Local Alignment Search Tool” Elsevier J. Mol. Biol., vol. 215, pp 403-410, 1990
6. D. Hoang et. al. “FPGA Implementation of Systolic Sequence Alignment”, Proceedings of the 2nd International Workshop on Field-Programmable Logic and Applications, Lecture Notes in Computer Science 705, pp 183-191, 1992.
7. D. Hoang “Searching Genetic Databases on Splash 2”, Proceedings IEEE Workshop on FPGAs for Custom Computing Machines (FCCM), pp 185-191, 1993.
8. S. Guccione and E. Keller “Gene Matching Using JBits”, Proceedings of the 12th International Conference on Field-Programmable Logic and Applications, Lecture Notes In Computer Science; Vol. 2438, pp 1168-1171, 2002.
9. K. Puttegowda et. Al. “A Run-Time Reconfigurable System for Gene-Sequence Searching”, Proceedings, 16th International Conference on VLSI Design pp 561 – 566, New Delhi 2003.
10. T. Oliver, B. Schmidt, D. Maskel “"Reconfigurable Architectures for Bio-sequence Database Scanning on FPGAs", IEEE Transactions on Circuits and Systems II, Vol, 52, No, 12, pp, 851-855, 2005.
11. K. Muriki, K. Underwood, and R. Sass, “RC-BLAST: Towards an open source hardware implementation,” In Proceedings of the International Workshop on High Performance Computational Biology (2005).

12. M. Herbordt, J. Model, B. Sukhwani, Y. Gu, and T. VanCourt, “Single pass streaming BLAST on FPGAs”, Parallel Computing, vol. 33, issue 10-11 (Nov, 2007), pp 741-756, 2007

13. P. Krishnamurthy, J. Buhler, R. Chamberlain, M. Franklin, K. Gyang, and J. Lancaster, "Biosequence Similarity Search on the Mercury System," In Proc. of the IEEE 15th Int'l Conf, on Application-Specific Systems, Architectures and Processors, September 2004, pp, 365-375
14. J. Lancaster, J. Buhler, R. Chamberlain, “Acceleration of Ungapped Extension in Mercury BLAST”, 7th workshop on media and streaming processors, Barcelona, Spain, November 12, 2005
15. A. Buhler et al. "Mercury blastn: faster dna sequence comparison using a streaming hardware architecture", RSSI, 2007

16. Washington University, "Method and apparatus for performing biosequence similarity searching" International Patent WO/2006/096324, 2006
17. D. Lavenier, L. Xinchun, G. Georges, “Seed-based Genomic Sequence Comparison using a FPGA/FLASH Accelerator”, in Proceedings of IEEE International Conference on Field Programmable Technology, 2006,(FPT ‘06), pp, 41 - 48, 2006.
18. F. Xia, Y. Dou and J. Xu, “Families of FPGA-Based Accelerators for BLAST Algorithm with Multi-seeds Detection and Parallel Extension”, Bioinformatics Research and Development, Second International Conference, BIRD 2008, pp, 43-57, Vienna, Austria, July 7-9, 2008.
19. F. Xia, Y. Dou, J. Xu, "FPGA-Based Accelerators for BLAST Families with Multi-Seeds Detection and Parallel Extension," The 2nd International Conference on Bioinformatics and Biomedical Engineering, 2008, ICBBE 2008,, pp,58-62, 16-18 May 2008.
20. http://www.timelogic.com/benchmark_blast.html
21. C. Chang “BLAST Implementation on BEE2“Electrical Engineering and Computer Science University of California at Berkeley (2005), http://bee2.eecs.berkeley.edu
22. P. Afratis, E. Sotiriades, G. Chrysos, S. Fytraki, and D. Pnevmatikatos, “A rate-based prefiltering approach to BLAST acceleration,” in Proc,IEEE Conference on Field Programmable Logic and Applications, 2008.
23. E. Sotiriades, C. Kozanitis, G. Chrysos, A. Dollas “Rapid Phototyping of a System-on-a-Chip for the BLAST Algorithm Implementation”, Proceedings, 17th International IEEE Workshop on Rapid System Prototyping RSP-2006, pp, 223-229, Chania, Greece, 14-16 June, 2006, Computer Society Press.

24. Karlin, S. & Altschul, S.F. (1990) "Methods for assessing the statistical significance of molecular sequence features by using general scoring schemes." Proc. Natl. Acad. Sci. USA 87:2264-2268
25. Jason Papadopoulos “The developer’s Guide to BLAST” p29-30
26. Euripides Sotiriades, Apostolos Dollas “A General Reconfigurable Architecture for the BLAST algorithm”, The Journal of VLSI Signal Processing Systems for Signal, Image, and Video Technology, Special Issue on Computing Architectures and Acceleration for Bioinformatics Algorithms, Kluwer Academic Publishers Volume 48, Issue 3 Pages: 189 – 208, September, 2007.
27. Panagiotis Afratis, Konstantinos Galanakis, Euripides Sotiriades, Georgios-Grigorios Mplemenos, Grigorios Chrysos, Yiannis Papaefstathiou, Dionisios Pnevmatikatos “Design and Implementation of a Database Filter for BLAST Acceleration” Design Automation & Test in Europe (DATE 2009), Nice France, pp166-171, April 24-29 2009.
28. Euripides Sotiriades, Christos Kozanitis, Grigorios Chrysos, Apostolos Dollas “Rapid Phototyping of a System-on-a-Chip for the BLAST Algorithm Implementation”, Proceedings, 17th International IEEE Workshop on Rapid System Prototyping RSP-2006, pp, 223-229, Chania, Greece, 14-16 June, 2006, Computer Society Press.
29. Euripides Sotiriades, Christos Kozanitis, Apostolos Dollas, “FPGA based Architecture of DNA Sequence Comparison and Database Search”, Proceedings 20th International Parallel and Distributed Processing Symposium, IPDPS 2006, p 193, ,at the 13th Reconfigurable Architectures Workshop Rhodes, Greece, 25-29 April, 2006.
30. Euripides Sotiriades, Christos Kozanitis, Apostolos Dollas, Some Initial Results on Hardware BLAST Acceleration with a Reconfigurable Architecture, Proceedings 20th International Parallel and Distributed Processing Symposium, IPDPS 2006, p 251 , at the 5th IEEE International Workshop on High Performance Computational Biology (HiCOMB2006), Rhodes, Greece, 25-29 April, 2006.
31. Christopher L. Liner and Robert G. Clapp. Nonlinear pairwise alignment of seismic traces p.174 November 11, 2002.
32. Olaf O. Storaasli, Dave Strenski, Performance Evaluation of FPGA-Based Biological Applications
33. Dayhoff, M.O., Schwartz, R. and Orcutt, B.C. (1978). "A model of Evolutionary Change in Proteins". Atlas of protein sequence and structure (volume 5, supplement 3 ed.). Nat. Biomed. Res. Found.. pp. 345–358. ISBN 0912466073
34. Nikodimos Georgiadis, Apostolos Dollas, Yiannis Papaefstathiou, Dionisios Pnevmatikatos “ Πλατφόρμα πειραματικής ανάπτυξης για αναδιατασσόμενη λογική

βασισμένη σε μοντέλο εξυπηρετητή-πελάτη" at Technical University of Crete, Chania , June 2010.

35. Yiannis Kartsonakis, Apostolos Dollas, Yiannis Papaefstathiou, Dionisios Pnevmatikatos, Dimitrios Meidanis, "Ανάπτυξη οδηγού λειτουργικού συστήματος ανοιχτού λογισμικού (Linux) και VHDL κώδικα για σειριακή επικοινωνία υψηλής ταχύτητας (PCIe) ηλεκτρονικού υπολογιστή με την αναδιατρασσόμενη συσκευή Virtex 5 Xilinx" at Technical University of Crete, Chania.
36. http://en.wikipedia.org/wiki/Sequence_alignment
37. http://www-cs-faculty.stanford.edu/~eroberts/courses/soco/projects/2000-01/computers-and-the-hgp/smith_waterman.html
38. http://www.med.nyu.edu/rcr/rcr/course/fasta.gif

Παράρτημα

Παράμετροι NCBI BLAST

-p

Program Name [String]

-d

Database [String] default = nr

-i

Query File [File In] default = stdin

-e

Expectation value (E) [Real] default = 10.0

-m

alignment view options: 0 = pairwise, 1 = query-anchored showing identities, 2 = query-anchored no identities, 3 = flat query-anchored, show identities, 4 = flat query-anchored, no identities, 5 = query-anchored no identities and blunt ends, 6 = flat query-anchored, no identities and blunt ends, 7 = XML Blast output, 8 = tabular, 9 tabular with comment lines [Integer] default = 0

-o

BLAST report Output File [File Out] Optional default = stdout

-F

Filter query sequence (DUST with blastn, SEG with others) [String] default = T

-G

Cost to open a gap (zero invokes default behavior) [Integer] default = 0

-E

Cost to extend a gap (zero invokes default behavior) [Integer] default = 0

-X

X dropoff value for gapped alignment (in bits) (zero invokes default behavior) blastn 30, megablast 20, tblastx 0, all others 15 [Integer] default = 0

-I

Show GI's in deflines [T/F] default = F

-q

Penalty for a nucleotide mismatch (blastn only) [Integer] default = -3

-r

Reward for a nucleotide match (blastn only) [Integer] default = 1

-v

Number of database sequences to show one-line descriptions for (V) [Integer] default = 500

-b

Number of database sequence to show alignments for (B) [Integer] default = 250

-f

Threshold for extending hits, default if zero blastp 11, blastn 0, blastx 12, tblastn 13 tblastx 13, megablast 0 [Integer] default = 0

-g

Perfom gapped alignment (not available with tblastx) [T/F] default = T

-Q

Query Genetic code to use [Integer] default = 1

-D

DB Genetic code (for tblast[nx] only) [Integer] default = 1

-a

Number of processors to use [Integer] default = 1

-O

SeqAlign file [File Out] Optional

-J

Believe the query defline [T/F] default = F

-M

Matrix [String] default = BLOSUM62

-W

Word size, default if zero (blastn 11, megablast 28, all others 3) [Integer] default = 0

-z

Effective length of the database (use zero for the real size) [Real] default = 0

-K

Number of best hits from a region to keep (off by default, if used a value of 100 is recommended) [Integer] default = 0

-Y

Effective length of the search space (use zero for the real size) [Real] default = 0

-S

Query strands to search against database (for blast[nx], and tblastx) 3 is both, 1 is top, 2 is bottom [Integer] default = 3

-T

Produce HTML output [T/F] default = F

-l

Restrict search of database to list of GI's [String] Optional

-U

Use lower case filtering of FASTA sequence [T/F] Optional default = F

-y

X dropoff value for ungapped extensions in bits (0.0 invokes default behavior) blastn 20, megablast 10, all others 7 [Real] default = 0.0

-Z

X dropoff value for final gapped alignment in bits (0.0 invokes default behavior) blastn/megablast 50, tblastx 0, all others 25 [Integer] default = 0

-R

PSI-TBLASTN checkpoint file [File In] Optional

-n

MegaBlast search [T/F] default = F

-L

Location on query sequence [String] Optional

-A

Multiple Hits window size, default if zero (blastn/megablast 0, all others 40 [Integer] default = 0

-w

Frame shift penalty (OOF algorithm for blastx) [Integer] default = 0

-t

Length of the largest intron allowed in tblastn for linking HSPs (0 disables linking) [Integer] default = 0

Κώδικας που χρησιμοποιήθηκε από το NCBI BLAST με τις αλλαγές μας για να μπορέσουμε να στείλουμε τιμές στο PCI-express

s_BlastSmallNaScanSubject_8_4

```
static Int4 s_BlastSmallNaScanSubject_8_4
                          ( const LookupTableWrap  * lookup_wrap,
                               const BLAST_SequenceBlk * subject,
                                  BlastOffsetPair * NCBI_RESTRICT
                                                    offset_pairs,
                                Int4 max_hits, Int4 * scan_range)
{

    BlastSmallNaLookupTable *lookup =
               (BlastSmallNaLookupTable *) lookup_wrap>lut;
    const Int4 kLutWordLength = 8;
    const Int4 kLutWordMask = (1 << (2 * kLutWordLength)) - 1;
    Int4 num_words = (scan_range[1] - scan_range[0]) / 4 + 1;
    Uint1 *s = subject->sequence + scan_range[0] / 4
    Int4 total_hits = 0;
    Int2 *backbone = lookup->final_backbone;
    Int2 *overflow = lookup->overflow;
    Int4 init_index;
    Int4 index;

    ASSERT(lookup_wrap->lut_type == eSmallNaLookupTable);
    ASSERT(lookup->lut_word_length == 8);
    ASSERT(lookup->scan_step == 4);
    max_hits -= lookup->longest_chain;
```

```
    init_index = s[0];
    switch (num_words % 8) {
    case 1: s -= 7; scan_range[0] -= 28; goto byte_7;
    case 2: s -= 6; scan_range[0] -= 24; goto byte_6;
    case 3: s -= 5; scan_range[0] -= 20; goto byte_5;
    case 4: s -= 4; scan_range[0] -= 16; goto byte_4;
    case 5: s -= 3; scan_range[0] -= 12; goto byte_3;
    case 6: s -= 2; scan_range[0] -=  8; goto byte_2;
    case 7: s -= 1; scan_range[0] -=  4; goto byte_1;
    }

    while (scan_range[0] <= scan_range[1]) {

        init_index = init_index << 8 | s[1];
        index = backbone[init_index & kLutWordMask];
        *(Xilinx_p) = init_index;
        //SMALL_NA_ACCESS_HITS(0);
byte_1:
        init_index = init_index << 8 | s[2];
        index = backbone[init_index & kLutWordMask];
        *(Xilinx_p) = init_index;

        //SMALL_NA_ACCESS_HITS(4);
byte_2:
        init_index = init_index << 8 | s[3];
        index = backbone[init_index & kLutWordMask];
        *(Xilinx_p) = init_index;

        //SMALL_NA_ACCESS_HITS(8);
byte_3:
        init_index = init_index << 8 | s[4];
        index = backbone[init_index & kLutWordMask];
        *(Xilinx_p) = init_index;

        //SMALL_NA_ACCESS_HITS(12);
byte_4:
        init_index = init_index << 8 | s[5];
        index = backbone[init_index & kLutWordMask];
        *(Xilinx_p) = init_index;
```

```
        //SMALL NA ACCESS HITS(16);
byte_5:
        init_index = init_index << 8 | s[6];
        index = backbone[init_index & kLutWordMask];
        *(Xilinx_p) = init_index;

        //SMALL NA ACCESS HITS(20);
byte_6:
        init_index = init_index << 8 | s[7];
        index = backbone[init_index & kLutWordMask];
        *(Xilinx_p) = init_index;

        //SMALL NA ACCESS HITS(24);
byte_7:
        init_index = init_index << 8 | s[8];
        //s += 8;
        index = backbone[init_index & kLutWordMask];
        *(Xilinx_p) = init_index;

        //SMALL NA ACCESS HITS(28);
        //s_off += 32;


      while ( *(p+valid_addr) == 0 ){
      total_hits++;
      offset_pairs[total_hits].qs_offsets.q_off=,*(p+rd_addr);
       if *(p+valid_addr==0)
      offset_pairs[total_hits].qs_offsets.s_off=,*(p+rd_addr);
      }

    }

    return total_hits;
}
```

NA ACCESS HITS

```
/** access the small-query lookup table */
```

```
#define SMALL_NA_ACCESS_HITS(x)                                         \
    if (index != -1) {                                                  \
        if (total_hits > max_hits) {                                    \
            scan_range[0] += (x);
\
            break;                                                      \
        }                                                               \
        total_hits += s_BlastSmallNaRetrieveHits(offset_pairs,  \
                                                              index,
                                                scan_range[0] + (x),  \
                                                       total_hits,    \
                                                       overflow);     \
    }
```

s_BlastSmallNaRetrieveHits

```
static NCBI_INLINE Int4 s_BlastSmallNaRetrieveHits(
                          BlastOffsetPair * NCBI_RESTRICT
offset_pairs,
                          Int4 index, Int4 s_off,
                          Int4 total_hits, Int2 *overflow)
{
    if (index >= 0) {
        offset_pairs[total_hits].qs_offsets.q_off = index;
        offset_pairs[total_hits].qs_offsets.s_off = s_off;
        return 1;
    }
    else {
        Int4 num_hits = 0;
        Int4 src_off = -index;

        index = overflow[src_off++];
      do {
      offset_pairs[total_hits+num_hits].qs_offsets.q_off =index;
       offset_pairs[total_hits+num_hits].qs_offsets.s_off = s_off;
       num_hits++;
```

```
        index = overflow[src_off++];
      } while (index >= 0);
         return num_hits;
    }
}
```