ΠΑΡΑΛΛΗΛΕΣ ΚΑΙ ΚΑΤΑΝΕΜΗΜΕΝΕΣ ΥΛΟΠΟΙΗΣΕΙΣ ΓΙΑ ΤΗΝ ΠΡΟΣΕΓΓΙΣΤΙΚΗ ΑΝΑΖΗΤΗΣΗ ΑΛΦΑΡΙΘΜΗΤΙΚΩΝ

ΠΑΝΑΓΙΩΤΗΣ ΜΙΧΑΗΛΙΔΗΣ

Πτυχιούχος, Τμήματος Εφαρμοσμένης Πληροφορικής, Πανεπιστήμιο Μακεδονίας, 1998

ΔΙΔΑΚΤΟΡΙΚΗ ΔΙΑΤΡΙΒΗ

Τμήμα Εφαρμοσμένης Πληροφορικής

Πανεπιστήμιο Μακεδονίας
Θεσσαλονίκη

Μάρτιος, 2004

Αφιέρωση

Στους γονείς μου

Ευχαριστίες

Από τη θέση αυτή επιθυμώ να εκφράσω τις ευχαριστίες μου σε όλους όσους με οποιονδήποτε τρόπο συνέβαλαν, ώστε να φθάσει στο τέλος της η ερευνητική αυτή προσπάθεια.

Ιδιαίτερα τον επιβλέποντα Καθηγητή κ. Κωνσταντίνο Μαργαρίτη, για την υπόδειξη του ερευνητικού θέματος, την παρακολούθηση και τις πολύτιμες παρατηρήσεις του καθώς και για τη βοήθεια του στην τελική παρουσίαση της εργασίας.

Τον Αναπληρωτή Καθηγητή κ. Κωνσταντίνο Ταραμπάνη και τον Επίκουρο Καθηγητή κ. Γεώργιο Ευαγγελίδη για τις πολύτιμες παρατηρήσεις τους στο κείμενο της διατριβής.

Τέλος, θέλω να ευχαριστήσω όλους όσους εργάζονται στο Εργαστήριο Παράλληλης και Κατανεμημένης Επεξεργασίας για την συμπαράσταση τους κατά τη διάρκεια της διατριβής μου.

ΠΑΡΑΛΛΗΛΕΣ ΚΑΙ ΚΑΤΑΝΕΜΗΜΕΝΕΣ ΥΛΟΠΟΙΗΣΕΙΣ ΓΙΑ ΤΗΝ ΠΡΟΣΕΓΓΙΣΤΙΚΗ ΑΝΑΖΗΤΗΣΗ ΑΛΦΑΡΙΘΜΗΤΙΚΩΝ

ΠΑΝΑΓΙΩΤΗΣ ΜΙΧΑΗΛΙΔΗΣ

ΠΕΡΙΛΗΨΗ ΤΗΣ ΔΙΔΑΚΤΟΡΙΚΗΣ ΔΙΑΤΡΙΒΗΣ

Τμήμα Εφαρμοσμένης Πληροφορικής

Πανεπιστήμιο Μακεδονίας

Θεσσαλονίκη

Μάρτιος, 2004

ΠΑΡΑΛΛΗΛΕΣ ΚΑΙ ΚΑΤΑΝΕΜΗΜΕΝΕΣ ΥΛΟΠΟΙΗΣΕΙΣ ΓΙΑ ΤΗΝ ΠΡΟΣΕΓΓΙΣΤΙΚΗ ΑΝΑΖΗΤΗΣΗ ΑΛΦΑΡΙΘΜΗΤΙΚΩΝ

ΠΑΝΑΓΙΩΤΗΣ ΜΙΧΑΗΛΙΔΗΣ

Πτυχιούχος, Τμήματος Εφαρμοσμένης Πληροφορικής, Πανεπιστήμιο Μακεδονίας, 1998

Υποψ. Διδάκτορας, Τμήματος Εφαρμοσμένης Πληροφορικής, Πανεπιστήμιο Μακεδονίας, 2004

Περίληψη

Σε αυτή τη διατριβή εξετάζει το πρόβλημα αναζήτησης αλφαριθμητικών που μπορεί να οριστεί γενικά ως η εύρεση των εμφανίσεων ενός προτύπου (pattern) σε μια μεγαλύτερη σε μέγεθος ακολουθία χαρακτήρων, η οποία λέγεται κείμενο (text). Το παραπάνω πρόβλημα μπορεί να γενικευτεί ώστε να επιτρέπονται να υπάρχει πεπερασμένος αριθμός από διαφορές ή σφάλματα ανάμεσα στο πρότυπο και στο κείμενο. Αυτό λέγεται προσεγγιστική αναζήτηση αλφαριθμητικών (approximate string searching). Στα τελευταία χρόνια υπάρχει μεγάλο ενδιαφέρον στο πρόβλημα της προσεγγιστικής αναζήτησης αλφαριθμητικών που προήλθε από την ραγδαία ανάπτυξη των πεδίων της υπολογιστικής μοριακής βιολογίας (computational molecular biology) και της ανάκτησης πληροφοριών (information retrieval) στο διαδίκτυο.

Ο στόχος της διατριβής είναι να αναπτύξουμε τεχνικές αναζήτησης από το χώρο της παράλληλης και κατανεμημένης επεξεργασίας (parallel and distributed processing) ώστε να επιταχύνουμε την αναζήτηση σε μεγάλο όγκο κειμένων, όπως αυτά που εμφανίζονται στον παγκόσμιο ιστό. Συνεπώς, παρουσιάζουμε υλοποιήσεις αλγορίθμων απλής και προσεγγιστικής αναζήτησης αλφαριθμητικών σε δύο κατανεμημένες αρχιτεκτονικές γενικού σκοπού, όπως η συστοιχία από ομοιογενείς σταθμούς εργασίας (cluster of workstations) ή η συστοιχία από ετερογενείς σταθμούς εργασίας (cluster of heterogeneous workstations). Επίσης, παρουσιάζουμε υλοποιήσεις των απαιτητικών αλγορίθμων για τα προβλήματα απλής και προσεγγιστικής αναζήτησης αλφαριθμητικών σε αρχιτεκτονικές ειδικού σκοπού όπως οι διατάξεις επεξεργαστών (processor arrays). Τέλος, παρουσιάζουμε μια ευέλικτη προγραμματιζόμενη

αρχιτεκτονική που υλοποιεί όλους τους αλγορίθμους απλής και προσεγγιστικής αναζήτησης αλφαριθμητικών.

Τα αποτελέσματα της παραπάνω έρευνας έδειξαν ότι οι τεχνικές και οι υλοποιήσεις που παρουσιάζουμε σε αυτήν διατριβή μπορούν να αντιμετωπίσουν ένα μεγάλο όγκο κειμένων αποτελεσματικά.

Περιεχόμενα

Κατάλογος Σχημάτων

Κατάλογος Πινάκων

Λίστα Αλγορίθμων

Κεφάλαιο 1

Εισαγωγή

Το πρόβλημα της αναζήτησης αλφαριθμητικών μπορεί να οριστεί γενικά ως η εύρεση των εμφανίσεων ενός προτύπου (pattern) σε μια μεγαλύτερη σε μέγεθος ακολουθία χαρακτήρων, η οποία λέγεται κείμενο (text).

Το παραπάνω πρόβλημα είναι ένα από τα παλαιότερα και διαχρονικά προβλήματα στην επιστήμη των υπολογιστών. Το πρόβλημα της αναζήτησης αλφαριθμητικών μπορεί να γενικευτεί επιτρέποντας ευελιξία (flexibility) στον ορισμό της αναζήτησης. Έτσι, τα πρότυπα μπορεί να μην είναι μόνο απλά αλφαριθμητικά αλλά μπορεί να περιέχουν περιορισμένες εκφράσεις (limited expressions) όπως 'αδιάφορου χαρακτήρα '(don't care), κλάσεις χαρακτήρων (character classes) και άρνηση ενός χαρακτήρα ή μιας κλάσης χαρακτήρων (complement). Επίσης, στον ορισμό της αναζήτησης επιτρέπεται να υπάρχει πεπερασμένος αριθμός από διαφορές ή σφάλματα ανάμεσα στο πρότυπο και την εμφάνιση του στο κείμενο. Αυτό λέγεται προσεγγιστική αναζήτηση αλφαριθμητικών (approximate string searching).

Τα τελευταία χρόνια υπάρχει ένα μεγάλο ενδιαφέρον στα προβλήματα αναζήτησης αλφαριθμητικών και ιδιαίτερα στο πρόβλημα της προσεγγιστικής αναζήτησης αλφαριθμητικών που προήλθε από την ραγδαία ανάπτυξη των πεδίων της υπολογιστικής μοριακής βιολογίας (computational molecular biology) ή βιοπληροφορικής (bioinformatics) και της ανάκτησης πληροφοριών (information retrieval) ιδιαίτερα σε σχέση με το διαδίκτυο (internet) και τους υπολογισμούς πλέγματος (grid computing). Μερικές καλές αναφορές για εφαρμογές της προσεγγιστικής αναζήτησης αλφαριθμητικών στο πεδίο της υπολογιστικής βιολογίας είναι οι [144, 132, 141, 123, 164] και στο πεδίο της ανάκτησης πληροφοριών οι

[163, 93, 133, 135, 70, 173, 41, 8].

Ο αριθμός των εφαρμογών για απλή ή προσεγγιστική αναζήτηση αλφαριθμητικών αυξάνεται καθημερινά. Ενδεικτικά αναφέρουμε την αναγνώριση γραφικού χαρακτήρα (hand-writing recognition) [89], την αχνίνευση ιού (virus detection) [71], τη συμπίεση εικόνας (image compression) [95], την εξόρυξη δεδομένων (data mining) [35], και την αναγνώριση προτύπων (pattern recognition) [48]. Σημαντικός αριθμός εφαρμογών αναφέρεται στο βιβλίο [141].

Σε αυτή την διατριβή δίνουμε περισσότερη έμφαση στο πεδίο της ανάκτησης πληροφοριών. Το πεδίο αυτό αντιμετωπίζει μια πρόκληση στο μεγάλο όγκο κειμένων του παγκόσμιου ιστού (world-wide-web). Έτσι, στον παγκόσμιο ιστό περιέχονται αδόμητες συλλογές κειμένων που αυξάνονται με εκθετικό ρυθμό κάθε χρόνο. Συνεπώς, η αναζήτηση ενός προτύπου ή μιας πληροφορίας σε μια τέτοια μεγάλη ποσότητα κειμένου γίνεται όλο και περισσότερο χρονοβόρα. Ο χρόνος εκτέλεσης για την αναζήτηση αλφαριθμητικών είναι πολλές φορές απαγορευτικός με την χρήση απλών και ταχύτερων αλγορίθμων σε ένα συμβατικό ή ακολουθιακό υπολογιστή. Επομένως, υπάρχει η ανάγκη να χρησιμοποιηθούν σύγχρονες και έξυπνες μέθοδοι αναζήτησης για να χειριστούν ένα τέτοιο μεγάλο όγκο δεδομένων αποτελεσματικά.

Μια σύγχρονη λύση στην προσπάθεια οι αλγόριθμοι αναζήτησης αλφαριθμητικών να είναι πιο γρήγοροι, αποτελεσματικοί και ευέλικτοι είναι να συνδυαστούν με μεθόδους από το χώρο της παράλληλης και κατανεμημένης επεξεργασίας (parallel and distributed processing). Συνεπώς, σε αυτή τη διατριβή αναπτύσσουμε υλοποιήσεις αλγορίθμων αναζήτησης σε παράλληλα και κατανεμημένα υπολογιστικά συστήματα και μοντέλα έτσι ώστε να αξιοποιηθούν αποτελεσματικά σε μεγάλες αδόμητες βάσεις κειμένων. Παρακάτω παρουσιάζονται τα σημαντικά σημεία της συμβολής μας στην έρευνα των υλοποιήσεων αλγορίθμων αναζήτησης αλφαριθμητικών σε διάφορες παράλληλες και κατανεμημένες αρχιτεκτονικές.

Πρώτον, παρουσιάζουμε μια ενιαία ταξινόμηση των ακολουθιακών αλγορίθμων σε βασικές κατηγορίες τόσο στο πρόβλημα της απλής αναζήτησης αλφαριθμητικών όσο και στο πρόβλημα της προσεγγιστικής αναζήτησης αλφαριθμητικών. Σε αυτή την ταξινόμηση καλύπτουμε καινούργιους αλγορίθμους που έχουν εμφανιστεί στα τελευταία χρόνια (όπως οι αλγόριθμοι που χρησιμοποιούν τις τεχνικές του bit-παραλληλισμού (bit-parallelism)) σε σχέση με παλαιότερες επισκοπήσεις από άλλους ερευνητές που δεν έχουν καλύψει τις τελευταίες εξελίξεις. Επίσης, παρουσιάσουμε συγκριτικά πειραματικά αποτελέσματα και μοντέλα απόδοσης όλων των αλγορίθμων. Η έρευνα αυτή ξεκίνησε το 1998 και δημοσιεύστηκε στα

περιοδικά [104, 110]. Η έρευνα μας συμπίπτει χρονικά με την αντίστοιχη έρευνα του Navvaro [129], αλλά είναι προφανώς ανεξάρτητη.

Δεύτερον, παρουσιάζουμε υλοποιήσεις ακολουθιακών αλγορίθμων απλής και προσεγγιστικής αναζήτησης αλφαριθμητικών σε μια παράλληλη και κατανεμημένη αρχιτεκτονική γενικού σκοπού, όπως η συστοιχία από ομοιογενείς σταθμούς εργασίας (cluster of workstations). Οι υλοποιήσεις των αλγορίθμων σε αυτή την νέα αρχιτεκτονική γίνεται με την βοήθεια του εργαλείου παράλληλου προγραμματισμού MPI (Message Passing Interface - Διεπαφή Περάσματος Μηνυμάτων). Επίσης, παρουσιάζουμε ένα μαθηματικό μοντέλο πρόβλεψης απόδοσης για τις παράλληλες υλοποιήσεις σε μια συστοιχία υπολογιστών. Η έρευνα για την υλοποίηση αλγορίθμων σε συστοιχίες από ομοιογενείς σταθμούς εργασίας έχει δημιοσιευτεί στα περιοδικά και συνέδρια [111, 103, 106, 107].

Τρίτον, αναπτύσσουμε παράλληλες υλοποιήσεις και ένα γενικό θεωρητικό μοντέλο πρόβλεψης απόδοσης σε μια εξελιγμένη αρχιτεκτονική γενικού σκοπού, όπως η συστοιχία από ετερογενείς σταθμούς εργασίας (cluster of heterogeneous workstations). Η έρευνα για την υλοποίηση αλγορίθμων σε συστοιχίες από ετερογενείς σταθμούς εργασίας έχει δημιοσιευτεί στις εργασίες [118, 119, 105, 108, 102, 109, 112, 120, 116, 117].

Τέταρτον, παρουσιάζουμε απεικονίσεις των πιο απαιτητικών αλγορίθμων για τα προβλήματα απλής και προσεγγιστικής αναζήτησης σε παράλληλες αρχιτεκτονικές ειδικού σκοπού όπως οι διατάξεις επεξεργαστών (processor arrays). Η δουλειά αυτή έχει δημιοσιευτεί στα συνέδρια [113, 115, 114].

Τέλος, αναπτύσσουμε μια ευέλικτη προγραμματιζόμενη αρχιτεκτονική υψηλής απόδοσης που μπορεί να υλοποιεί όλες τις κατηγορίες αλγορίθμων απλής και προσεγγιστικής αναζήτησης αλφαριθμητικών. Επίσης, προτείνουμε ένα θεωρητικό μοντέλο πρόβλεψης απόδοσης για την προγραμματιζόμενη αρχιτεκτονική. Αυτή η δουλειά δεν έχει ακόμα ανακοινωθεί σε συνέδριο αλλά έχει υποβληθεί σε περιοδικό.

1.1 Οργάνωση της Διατριβής

Η διδακτορική διατριβή είναι διαθρωμένη ως εξής: Στο κεφάλαιο 2 παρουσιάζουμε μια επισκόπηση αλγορίθμων σύμφωνα με τέσσερις προσεγγίσεις για το πρόβλημα απλής αναζήτησης αλφαριθμητικών (exact string searching). Στην συνέχεια, αναφέρουμε τα πειραματικά αποτελέσματα των αλγορίθμων

σε σχέση με διάφορες παραμέτρους όπως είναι το μήκος του προτύπου και το μεγέθος του αλφαβήτου. Τέλος, στο κεφάλαιο αυτό παρουσιάζουμε ένα πειραματικό χάρτη όπου μπορούμε εύκολα να επιλέξουμε το ταχύτερο αλγόριθμο σύμφωνα με το μήκος του προτύπου και το μέγεθος του αλφαβήτου.

Στο κεφάλαιο 3 παρουσιάζουμε μια επισκόπηση πρόσφατων αλγορίθμων σύμφωνα με τέσσερις κατηγορίες για το πρόβλημα προσεγγιστικής αναζήτησης αλφαριθμητικών (approximate string searching). Στην συνέχεια, παρουσιάζουμε τα πειραματικά αποτελέσματα των αλγορίθμων σε σχέση με διάφορες παραμέτρους όπως είναι το μήκος του προτύπου, το μέγεθος του αλφαβήτου και ο αριθμός των διαφορών. Επίσης, στο ίδιο κεφάλαιο δίνουμε ένα πειραματικό χάρτη που δείχνει περιοχές υπεροχής των διαφορετικών αλγορίθμων.

Στο κεφάλαιο 4 παρουσιάζουμε το βασικό υπόβαθρο από την περιοχή της παράλληλης και κατανεμημένης επεξεργασίας.

Στο κεφάλαιο 5 προτείνουμε τέσσερις στρατηγικές παραλληλοποίησης για τα προβλήματα απλής και προσεγγιστικής αναζήτησης αλφαριθμητικών σε ένα ομοιογενές κατανεμημένο σύστημα (συστοιχία) υπολογιστών (cluster). Οι στρατηγικές αυτές παραλληλοποίησης βασίζονται στο μοντέλο συντονιστής - εργαζόμενος (master-worker) και υλοποιούνται με την βοήθεια της βιβλιοθήκης MPI (Message Passing Interface - Διεπαφή Περάσματος Μηνυμάτων). Στην συνέχεια παρουσιάζουμε ένα μαθηματικό μοντέλο πρόβλεψης για την συμπεριφορά της απόδοσης των στρατηγικών σε ένα ομοιογενές σύστημα. Επίσης, στο ίδιο κεφάλαιο αναφέρουμε τα πειραματικά και θεωρητικά αποτελέσματα των τεσσάρων στρατηγικών παραλληλοποίησης για τους βασικούς αλγόριθμους απλής και προσεγγιστικής αναζήτησης αλφαριθμητικών.

Στο κεφάλαιο 6 τροποποιούμε τις τέσσερις στρατηγικές παραλληλοποίησης και του μαθηματικού μοντέλου πρόβλεψης σε μια νέα αρχιτεκτονική, δηλαδή μια ετερογενή συστοιχία υπολογιστών (cluster of heterogeneous workstations). Στο ίδιο κεφάλαιο παρουσιάσουμε τα πειραματικά και θεωρητικά αποτελέσματα για τους βασικούς αλγόριθμους απλής και προσεγγιστικής αναζήτησης αλφαριθμητικών.

Στο κεφάλαιο 7 παρουσιάζουμε υλοποιήσεις απλών αλγορίθμων για τα προβλήματα απλής και προσεγγιστικής αναζήτησης αλφαριθμητικών σε διατάξεις επεξεργαστών (processor arrays) ειδικού σκοπού με τη βοήθεια του μοντέλου διασωλήνωσης (pipeline). Επίσης, στο ίδιο κεφάλαιο παρουσιάζεται υλοποίηση ενός απλού αλγορίθμου για το πρόβλημα της μακρύτερης κοινής υποακολουθίας (longest

common subsequence) που είναι παραλλαγή του προβλήματος προσεγγιστικής αναζήτησης αλφαριθμητικών.

Στο κεφάλαιο 8 προτείνουμε μια ενιαία προγραμματιζόμενη αρχιτεκτονική διάταξη επεξεργαστών (programmable array processor architecture) κατάλληλη για αποτελεσματική εκτέλεση μιας ποικιλίας αλγορίθμων αναζήτησης αλφαριθμητικών. Στην συνέχεια, παρουσιάζουμε ένα μαθηματικό μοντέλο απόδοσης για την προτεινόμενη προγραμματιζόμενη αρχιτεκτονική. Τέλος, στο ίδιο κεφάλαιο δίνουμε κάποιες ενδεικτικές εμπειρικές τιμές για την προγραμματιζόμενη αρχιτεκτονική πάνω σε ένα σύστημα Pentium με βάση το παραπάνω μοντέλο απόδοσης.

Στο κεφάλαιο 9 παρουσιάζουμε τα συμπεράσματα της διδακτορικής διατριβής και αναπτύσσουμε προτάσεις για την περαιτέρω έρευνα.

Η διατριβή κλείνει με τις αναφορές και παραρτήματα όπου παρουσιάζεται ο κώδικας των εφαρμογών σε C.

Κεφάλαιο 2

Αναζήτηση Αλφαριθμητικών: Επισκόπηση και Πειραματικά Αποτελέσματα

2.1 Εισαγωγή

Σε αυτήν την παράγραφο παρουσιάζουμε σύντομα ορισμένες βασικές έννοιες και στην συνέχεια δίνουμε το ορισμό του προβλήματος.

Έστω Σ ένα οποιοδήποτε πεπερασμένο σύνολο, το οποίο ονομάζεται αλφάβητο (alphabet). Τα στοιχεία του συνόλου Σ ονομάζονται γράμματα (letters) ή χαρακτήρες (characters). Ένα αλφαριθμητικό (string) πάνω στο Σ είναι μια πεπερασμένη ακολουθία γραμμάτων ή χαρακτήρων του Σ. Το μήκος (length) ενός αλφαριθμητικού s είναι το πλήθος των χαρακτήρων που το απαρτίζουν και συμβολίζεται με $|s|$. Επίσης συμβολίζουμε με s_j τον j χαρακτήρα του αλφαριθμητικού s όπου j είναι η θέση του στο s. Την ακολουθία χαρακτήρων $s_i s_{i+1} ... s_j$ μπορούμε να τη συμβολίζουμε για λόγους ευκολίας με $s_{i...j}$. Το κενό αλφαριθμητικό (empty string) είναι ένα ειδικό αλφαριθμητικό μήκους μηδέν και συμβολίζεται με το ελληνικό γράμμα ϵ.

Ένα αλφαριθμητικό y ονομάζεται προθεματικό (prefix) του x, αν το y προκύπτει από το x με τη

διαγραφή κανενός ή περισσοτέρων χαρακτήρων από το τέλος του x. Ένα αλφαριθμητικό y ονομάζεται μεταθεματικό (suffix) του x, αν το y προκύπτει από το x με τη διαγραφή κανενός ή περισσοτέρων χαρακτήρων από την αρχή του x. Ένα αλφαριθμητικό y ονομάζεται υποαλφαριθμητικό (substring) του x, αν το y προκύπτει από το x με τη διαγραφή κανενός ή περισσοτέρων χαρακτήρων από την αρχή και το τέλος του x. Τέλος, ένα αλφαριθμητικό y ονομάζεται υποακολουθία (subsequence) του x, αν το y προκύπτει από το x με τη διαγραφή κανενός ή περισσοτέρων χαρακτήρων από το x και το y δεν είναι ίσο με το x.

Η αναζήτηση αλφαριθμητικών (string matching) είναι ένα σημαντικό πρόβλημα στην επιστήμη υπολογιστών και εμφανίζεται σε πολλές περιοχές, όπως, επεξεργασία δεδομένων (data processing), ανάκτηση πληροφοριών (information retrieval), αναγνώριση ομιλίας (speech recognition), αναγνώριση εικόνας (image recognition) και υπολογιστική βιολογία (computational biology).

Το πρόβλημα της αναζήτησης αλφαριθμητικών ορίζεται ως εξής: Δίνεται ένα μικρό πρότυπο αλφαριθμητικό (pattern string) $p = p_1p_2...p_m$ μήκους m και ένα μεγάλο κείμενο αλφαριθμητικό (text string) $t = t_1t_2...t_n$ μήκους n, όπου $m, n > 0$ και $m \leq n$. Ζητείται να βρεθούν όλες οι εμφανίσεις του προτύπου p μέσα στο κείμενο t.

Από την άλλη μεριά, ενδιαφερόμαστε να προσαρμόσουμε το παραπάνω πρόβλημα σύμφωνα με τις απαιτήσεις της ανάκτησης πληροφοριών, όπως για παράδειγμα η αναζήτηση περιορισμένων εκφράσεων. Συγκεριμένα, οι περιορισμένες εκφράσεις περιέχουν τα σύμβολα 'αδιάφορου χαρακτήρα'(don't care), άρνηση ενός χαρακτήρα (complement) και κλάση χαρακτήρων (character class). Το σύμβολο του αδιάφορου χαρακτήρα παριστάνει την ταύτιση με ένα οποιοδήποτε χαρακτήρα. Το σύμβολο της άρνησης παριστάνει την ταύτιση με όλους τους χαρακτήρες εκτός από τον χαρακτήρα που δεν θέλουμε να εμφανίζεται. Τέλος, το σύμβολο της κλάσης χαρακτήρων συμβολίζει την ταύτιση που ορίζει η κλάση ή το διάστημα χαρακτήρων.

Η λύση σε αυτό το πρόβλημα διαφέρει αν ο αλγόριθμος πρέπει να είναι on-line (δηλαδή, το κείμενο δεν είναι γνωστό εκ των προτέρων) ή off-line (δηλαδή, το κείμενο μπορεί να υποστεί προεπεξεργασία). Σε αυτή την διατριβή, θα ασχοληθούμε με τους on-line αλγορίθμους για το παραπάνω πρόβλημα. Πάρα πολλές λύσεις έχουν σχεδιαστεί για το πρόβλημα αναζήτησης αλφαριθμητικών [2, 34, 153, 100]. Γενικά, ένας on-line αλγόριθμος αναζήτησης αλφαριθμητικών αποτελείται από δύο φάσεις: τη φάση της προεπερξεργασίας (preprocessing phase) του προτύπου p και τη φάση της αναζήτησης (search

phase) του προτύπου p στο κείμενο t. Κατά την διάρκεια της φάσης προεπεξεργασίας κατασκευάζεται μια δομή δεδομένων Q που το μέγεθος της είναι ανάλογο με το μήκος του προτύπου και οι λεπτομέρειες κατασκευής διαφέρουν σε διαφορετικούς αλγορίθμους. Η φάση της αναζήτησης χρησιμοποιεί την δομή δεδομένων Q και προσπαθεί να προσδιορίσει γρήγορα αν το πρότυπο εμφανίζεται μέσα στο κείμενο. Αυτή η φάση βασίζεται σε τέσσερις διαφορετικές προσεγγίσεις.

Έτσι, οι αλγόριθμοι αναζήτησης αλφαριθμητικών μπορούν να χωριστούν σε τέσσερις κατηγορίες:

- Κλασσικοί αλγόριθμοι (classical): αλγόριθμος Brute-Force [100], αλγόριθμος Knuth-Morris-Pratt [69], αλγόριθμος Simon [52], αλγόριθμος Colussi [26], αλγόριθμος Boyer-Moore [20] και οι παραλλαγές του αλγόριθμου Boyer-Moore, όπως αλγόριθμος Galil [42], αλγόριθμος Apostolico-Giancarlo [3], αλγόριθμος Turbo-BM [30], αλγόριθμος Reverse-Colussi [27], αλγόριθμος Boyer-Moore-Horspool [58], οι αλγόριθμοι του Sunday (Quick Search, Optimal Mismatch, Maximal Shift) [154], αλγόριθμος Boyer-Moore-Horspool-Raita [139] και αλγόριθμος Boyer-Moore-Smith [150].

- Αλγόριθμοι μεταθεματικού αυτομάτου (suffix-automata): αλγόριθμος Reverse Factor [80, 30] και αλγόριθμος Turbo Reverse Factor [30].

- Αλγόριθμοι Bit-παραλληλισμού (bit-parallelism): αλγόριθμος Shift-Or [7], αλγόριθμος Shift-And [167] και αλγόριθμος Backward Nondeterministic DAWG Matching [130].

- Αλγόριθμοι κατακερματισμού (hashing): αλγόριθμος Harrison [53] και αλγόριθμος Karp-Rabin [100].

Στο παρελθόν έχουν παρουσιαστεί διάφορα πειράματα για αλγόριθμους αναζήτησης αλφαριθμητικών [58, 149, 36, 5, 154, 61, 150, 7, 139, 81, 96, 130]. Σε αυτό το κεφάλαιο, αναφέρουμε πειραματικά αποτελέσματα για έντεκα γνωστούς και αντιπροσωπευτικούς αλγορίθμους από τις τέσσερις κατηγορίες: τον αλγόριθμο Brute-Force, τον αλγόριθμο Knuth-Morris-Pratt, τον αλγόριθμο Boyer-Moore, τον αλγόριθμο Turbo-BM, τον αλγόριθμο Boyer-Moore-Horspool, τον αλγόριθμο Quick-Search, τον αλγόριθμο Boyer-Moore-Smith, τον αλγόριθμο Reverse Factor, τον αλγόριθμο Shift-Or, τον αλγόριθμο Backward Nondeterministic DAWG Matching και τον αλγόριθμο Karp-Rabin.

Το κεφάλαιο αυτό είναι διαρθρωμένο ως εξής: στην επόμενη ενότητα παρουσιάζουμε συνοπτικά τους έντεκα αλγορίθμους που χρησιμοποιήσαμε στα πειράματα μας. Στην ενότητα 2.3 περιγράφουμε την πειραματική μας μεθοδολογία περιλαμβάνοντας τον υπολογιστικό περιβάλλον, τα δοκιμαστικά δεδομένα και τους τρόπους μέτρησης των αποτελεσμάτων. Στην ενότητα 2.4 παρουσιάζουμε τα πειραματικά μας αποτελέσματα σε μορφή πινάκων και γραφημάτων. Τέλος, στην ενότητα 2.5 δίνουμε έναν πειραματικό χάρτη ο οποίος δείχνει τον ταχύτερο αλγόριθμο αναζήτησης αλφαριθμητικών σύμφωνα με το μήκος του προτύπου και το μέγεθος του αλφαβήτου.

2.2 Αλγόριθμοι Αναζήτησης Αλφαριθμητικών

Σε αυτή την παράγραφο παρουσιάζουμε τους βασικούς ακολουθιακούς αλγόριθμους για την επίλυση του προβλήματος αναζήτησης αλφαριθμητικών. Όμως, για περισσότερες λεπτομέρειες και την κωδικοποίηση των αλγορίθμων, ο αναγνώστης μπορεί να ανατρέξει στην εργασία μας [100] και τις αρχικές αναφορές. Επίσης, στο παράρτημα παρουσιάζεται ο κώδικας των ακολουθιακών αλγορίθμων σε γλώσσα προγραμματισμού C.

2.2.1 Κλασσική Προσέγγιση

Οι κλασσικοί αλγόριθμοι αναζήτησης αλφαριθμητικών βασίζονται σε συγκρίσεις χαρακτήρων.

Ο αλγόριθμος Brute-Force (για συντομία, BF) [100], ο οποίος είναι απλούστερος, εκτελεί συγκρίσεις χαρακτήρων ανάμεσα σε ένα χαρακτήρα του κειμένου και σε ένα χαρακτήρα του προτύπου από αριστερά προς τα δεξιά. Σε οποιαδήποτε περίπτωση, μετά από ανεπιτυχή ταύτιση χαρακτήρα ή επιτυχή ταύτιση ολόκληρου του προτύπου, ο αλγόριθμος μετατοπίζει ακριβώς μια θέση προς τα δεξιά. Αυτός ο αλγόριθμος δεν απαιτεί την φάση της προεπεξεργασίας αλλά ούτε επιπλέον χώρο μνήμης. Ο αλγόριθμος BF έχει πολυπλοκότητα χρόνου $O(mn)$ στην χειρότερη περίπτωση. Επίσης, ο μέσος αριθμός συγκρίσεων χαρακτήρων είναι $n(1 + \frac{1}{(|\Sigma|-1)})$.

Ο αλγόριθμος Knuth-Morris-Pratt (KMP) [69] ήταν ο πρώτος αλγόριθμος γραμμικού χρόνου αναζήτησης αλφαριθμητικών που διατυπώθηκε. Ο αλγόριθμος KMP εκτελεί συγκρίσεις χαρακτήρων από αριστερά προς τα δεξιά. Στην περίπτωση που έχουμε ανεπιτυχή ταύτιση χαρακτήρα, ο αλγόριθ-

μος χρησιμοποιεί τη γνώση των προηγούμενων χαρακτήρων που έχουν ήδη εξεταστεί προκειμένου να υπολογίζει την επόμενη θέση του κειμένου που θα αρχίζει η επόμενη σύγκριση. Το πλεονέκτημα του αλγόριθμου KMP είναι ότι δεν υλοποιεί οπισθοδρόμηση στους προηγουμένους χαρακτήρες του κειμένου. Η φάση της προεπεξεργασίας του αλγόριθμου KMP απαιτεί $O(m)$ χρόνο και χώρο. Η φάση της αναζήτησης απαιτεί $O(n)$ χρόνο στην χειρότερη και μέση περίπτωση.

Ο επόμενος αλγόριθμος είναι ο Boyer-Moore (BM) [20], ο οποίος είναι από τους πιο γνωστούς γρήγορους αλγορίθμους αναζήτησης αλφαριθμητικών στη θεωρία και στην πράξη. Ο αλγόριθμος BM εκτελεί συγκρίσεις χαρακτήρων ανάμεσα σε ένα χαρακτήρα του κειμένου και σε ένα χαρακτήρα του προτύπου από δεξιά προς τα αριστερά. Στην περίπτωση που έχουμε ανεπιτυχή ταύτιση χαρακτήρα ή πλήρη ταύτιση ολόκληρου του προτύπου, τότε χρησιμοποιεί δύο ευριστικές μετατόπισης (shift heuristics) για να μετατοπίζει το πρότυπο προς τα δεξιά. Οι δύο ευριστικές μετατόπισης λέγονται ευριστικό εμφάνισης (occurrence heuristic) και ευριστικό ταύτισης (match heuristic). Το μήκος της μετατόπισης είναι η μέγιστη μετατόπιση ανάμεσα στο ευριστικό εμφάνισης και στο ευριστικό ταύτισης. Περισσότερες λεπτομέρειες για τις δύο ευριστικές μετατόπισης αναφέρονται στην αρχική εργασία [20]. Οι ευριστικές μετατόπισης προεπεξεργάζονται σε $O(m+|\Sigma|)$ χρόνο και χώρο. Η φάση της αναζήτησης του αλγόριθμου BM απαιτεί $O(n+rm)$ χρόνο στην χειρότερη περίπτωση, όπου r είναι ο αριθμός των εμφανίσεων του προτύπου στο κείμενο. Τέλος, η μέση απόδοση του αλγόριθμου BM είναι υπογραμμική απαιτώντας περίπου $\frac{n}{m}$ συγκρίσεις χαρακτήρων κατά μέσο όρο.

Ο αλγόριθμος Turbo-BM (TBM) [30] είναι μια παραλλαγή του αλγόριθμου BM. Ο αλγόριθμος αυτός στηρίζεται στην ιδέα να αποθηκεύσουμε το υποαλφαριθμητικό (substring) του κειμένου που ταυτίστηκε με ένα μεταθεματικό (suffix) του προτύπου κατά τις τελευταίες συγκρίσεις χαρακτήρων. Αυτή η μέθοδος έχει δύο πλεονεκτήματα: α) ότι μπορεί να υπερπηδήσει πάνω από το αποθηκευμένο υποαλφαριθμητικό και β) ότι είναι ικανή να εκτελέσει μια turbo μετατόπιση. Περισσότερες λεπτομέρειες για την turbo μετατόπιση αναφέρονται στην αρχική εργασία [30]. Ο αριθμός των συγκρίσεων χαρακτήρων που εκτελείται από τον αλγόριθμο TBM φράσσεται προς τα πάνω από $2n$.

Ο αλγόριθμος Boyer-Moore-Horspool (BMH) [58] δεν χρησιμοποιεί το ευριστικό ταύτισης. Στην περίπτωση που έχουμε ανεπιτυχή ταύτιση χαρακτήρα ή ταύτιση ολόκληρου του προτύπου, το μήκος της μετατόπισης μεγιστοποιείται χρησιμοποιώντας μόνο το ευριστικό εμφάνισης για το χαρακτήρα του κειμένου που αντιστοιχεί στο τελευταίο δεξιότερο χαρακτήρα του προτύπου και όχι για τον χαρακτήρα

του κειμένου όπου εμφανίστηκε η ανεπιτυχή ταύτιση. Η φάση της προεπεξεργασίας του αλγόριθμου BMH απαιτεί $O(m + |\Sigma|)$ χρόνο και μειώνει τις απαιτήσεις χώρου από $O(m + |\Sigma|)$ σε $O(|\Sigma|)$. Τέλος, η φάση της αναζήτησης απαιτεί $O(mn)$ χρόνο στην χειρότερη περίπτωση αλλά έχει αποδειχτεί ότι ο μέσος αριθμός συγκρίσεων χαρακτήρων είναι $\frac{n}{|\Sigma|}$.

Ο αλγόριθμος Quick Search (QS) [154] του Sunday εκτελεί συγκρίσεις χαρακτήρων από αριστερά προς τα δεξιά. Στην περίπτωση που εμφανιστεί ανεπιτυχής ταύτιση χαρακτήρα υπολογίζει το μήκος της μετατόπισης χρησιμοποιώντας το ευριστικό εμφάνισης για τον πρώτο αμέσως χαρακτήρα του κειμένου μετά το τελευταίο χαρακτήρα του προτύπου. Ο χρόνος της προεπεξεργασίας και της αναζήτησης του αλγόριθμου QS είναι ίδιος όπως στον αλγόριθμο BMH.

Ο αλγόριθμος Boyer-Moore-Smith (BMS) [150] παρατήρησε ότι υπολογίζοντας την μετατόπιση για τον πρώτο χαρακτήρα του κειμένου μετά το τελευταίο χαρακτήρα του προτύπου δίνει μερικές φορές μεγαλύτερη μετατόπιση από ότι η μετατόπιση για το χαρακτήρα του κειμένου που αντιστοιχεί στο τελευταίο δεξιότερο χαρακτήρα του προτύπου. Συνεπώς, ο Smith υπολογίζει τη μέγιστη μετατόπιση ανάμεσα στις δύο εκείνες τιμές μετατόπισης. Η φάση της προεπεξεργασίας του αλγόριθμου BMS χρειάζεται $O(m + |\Sigma|)$ χρόνο και $O(|\Sigma|)$ χώρο. Επιπλέον, η φάση της αναζήτησης του αλγόριθμου αυτού απαιτεί πολυπλοκότητα χρόνου $O(mn)$ στην χειρότερη περίπτωση.

2.2.2 Αλγόριθμοι Μεταθεματικού Αυτομάτου

Η κατηγορία αυτή χρησιμοποιεί τη δομή δεδομένων μεταθεματικό αυτόματο (suffix automaton) που αναγνωρίζει όλα τα μεταθεματικά του προτύπου [34, 130]. Αυτή η δομή δεδομένων συχνά ονομάζεται DAWG (Deterministic Acyclic Word Graph).

Ο αλγόριθμος Reverse Factor (για συντομία, RF) [80, 30] σαρώνει τους χαρακτήρες του κειμένου από δεξιά προς τα αριστερά χρησιμοποιώντας το μικρότερο μεταθεματικό αυτόματο του αντίστροφου προτύπου. Η φάση της προεπεξεργασίας του αλγόριθμου RF απαιτεί γραμμικό χρόνο και χώρο ανάλογο με το μήκος του προτύπου. Η φάση της αναζήτησης του αλγόριθμου RF έχει τετραγωνική πολυπλοκότητα χρόνου στην χειρότερη περίπτωση αλλά κατά μέσο όρο ο χρόνος είναι βέλτιστος. Τέλος, ο αλγόριθμος αυτός εκτελεί $O(\frac{nlogm}{m})$ συγκρίσεις χαρακτήρων κατά μέσο όρο.

2.2.3 Αλγόριθμοι Bit-Παραλληλισμού

Η προσέγγιση του bit-παραλληλισμού [6, 7] χρησιμοποιεί εσωτερικό παραλληλισμό των δυαδικών ψηφίων (bits) της λέξης του υπολογιστή για να εκτελεστούν πολλές πράξεις ή λειτουργίες παράλληλα. Συμβολίζουμε w τον αριθμό των δυαδικών ψηφίων σε μια λέξη του υπολογιστή. Αυτή η τεχνική έχει γίνει ένας γενικός τρόπος για να προσομοιώνουμε απλά μη ντετερμινιστικά πεπερασμένα αυτόματα (Non-deterministic Finite Automata, NFA) αντί να μετατρέπουμε αυτά σε ντετερμινιστικά αυτόματα. Τα κύρια πλεονεκτήματα που εμφανίζει αυτή η προσέγγιση είναι η απλότητα της, η ευελιξία της και το ότι δεν απαιτεί ενδιάμεση μνήμη (no buffering).

Η βασική ιδέα του πρώτου αλγόριθμου Shift-Or (για συντομία, SO) [7] είναι να αναπαραστήσει την κατάσταση της αναζήτησης σαν ένα αριθμό και σε κάθε βήμα αναζήτησης κοστίζει ένα μικρό αριθμό από αριθμητικές και λογικές πράξεις, έτσι ώστε οι αριθμοί να είναι αρκετά μεγάλοι για να παραστήσει όλες τις πιθανές καταστάσεις της αναζήτησης. Η πολυπλοκότητα χρόνου της φάσης προεπεξεργασίας είναι $O((m+|\Sigma|)\lceil\frac{m}{w}\rceil)$ χρησιμοποιώντας $O(m|\Sigma|)$ χώρο, υποθέτοντας ότι το μήκος του προτύπου δεν είναι μεγαλύτερο από το μήκος λέξης του υπολογιστή. Τέλος, η πολυπλοκότητα χρόνου της φάσης της αναζήτησης είναι $O(n\lceil\frac{m}{w}\rceil)$ στη χειρότερη και μέση περίπτωση, όπου $\lceil\frac{m}{w}\rceil$ είναι ο χρόνος για να υπολογίζει την μετατόπιση ή μια απλή πράξη πάνω σε αριθμούς των m bits χρησιμοποιώντας το μήκος λέξης των w bits.

Πρόσφατα, ένας καινούργιος αλγόριθμος εμφανίστηκε που ονομάζεται Backward Nondeterministic DAWG Matching (BNDM) [130]. Ο αλγόριθμος αυτός προσομοιώνει ένα μη ντετερμινιστικό μεταθεματικό αυτόματο με την βοήθεια της τεχνικής bit-παραλληλισμού. Ο χρόνος της προεπεξεργασίας του αλγόριθμου BNDM είναι $O(m+|\Sigma|)$ για $m \leq w$ χρησιμοποιώντας $O(m|\Sigma|)$ επιπλέον χώρο. Ο χρόνος της αναζήτησης είναι $O(mn)$ στη χειρότερη περίπτωση και $O(\frac{nlogm}{m})$ στη μέση περίπτωση.

2.2.4 Αλγόριθμοι Κατακερματισμού

Ο αλγόριθμος Karp-Rabin (KR) [100] χρησιμοποιεί τεχνικές κατακερματισμού. Ο κατακερματισμός είναι μια απλή μέθοδος για να αποφύγει κανείς τον τετραγωνικό αριθμό συγκρίσεων χαρακτήρων στις περισσότερες πρακτικές περιπτώσεις. Η βασική ιδέα του αλγόριθμου KR είναι να υπολογίζει τη συνάρτηση κατακερματισμού (hashing function) για κάθε m-ομάδα υποαλφαριθμητικού του κειμένου

και να ελέγχει αν αυτή είναι ίση με την συνάρτηση κατακερματισμού του προτύπου. Η φάση της προεπεξεργασίας του αλγόριθμου KR απαιτεί $O(m)$ χρόνο ενώ η φάση της αναζήτησης απαιτεί $O(mn)$ χρόνο στη χειρότερη περίπτωση. Ο μέσος αριθμός συγκρίσεων χαρακτήρων του αλγόριθμου KR είναι $O(m+n)$.

Οι πολυπλοκότητες χρόνου και χώρου των διαφόρων αλγορίθμων παρουσιάζονται στον Πίνακα 2.1 για τη χειρότερη και μέση περίπτωση.

Πίνακας 2.1: Πολυπλοκότητες χρόνου και χώρου διαφόρων αλγορίθμων

Αλγόριθμος	Χρόνος Προεπεξεργασίας	Χρόνος Αναζήτησης		Χώρος Μνήμης
		Χειρότερη Περίπτωση	Μέση Περίπτωση	
BF		mn	$n(1+\frac{1}{(\lvert\Sigma\rvert-1)})$	1
KMP	m	n	n	m
BM	$m+\lvert\Sigma\rvert$	$n+rm$	$\frac{n}{m}$	$m+\lvert\Sigma\rvert$
TBM	$m+\lvert\Sigma\rvert$	$2n$		$m+\lvert\Sigma\rvert$
BMH	$m+\lvert\Sigma\rvert$	mn	$\frac{n}{\lvert\Sigma\rvert}$	$\lvert\Sigma\rvert$
QS	$m+\lvert\Sigma\rvert$	mn		$\lvert\Sigma\rvert$
BMS	$m+\lvert\Sigma\rvert$	mn		$\lvert\Sigma\rvert$
RF	m	mn	$n\frac{logm}{m}$	m
SO	$(m+\lvert\Sigma\rvert)\lceil\frac{m}{w}\rceil$	$n\lceil\frac{m}{w}\rceil$	$n\lceil\frac{m}{w}\rceil$	$m\lvert\Sigma\rvert$
BNDM	$m+\lvert\Sigma\rvert$	mn	$n\frac{logm}{m}$	$m\lvert\Sigma\rvert$
KR	m	mn	$m+n$	1

2.3 Πειραματική Μεθοδολογία

Σε αυτή την παράγραφο παρουσιάζουμε την πειραματική μεθοδολογία η οποία χρησιμοποιήθηκε στα πειράματα μας προκειμένου να συγκρίνουμε την απόδοση των αλγορίθμων αναζήτησης αλφαριθμητικών. Οι παράμετροι που περιγράφουν την απόδοση των αλγορίθμων είναι οι εξής:

1. Το μέγεθος του κειμένου,

2. Το μήκος του προτύπου και

3. Το μέγεθος του αλφαβήτου.

Όπως γνωρίζουμε, κανένας από τους αλγορίθμους δεν είναι βέλτιστος ή γενικά καλύτερος για όλες τις τρεις περιπτώσεις. Συνεπώς, ο κύριος στόχος της πειραματικής μας μελέτη είναι να συγκρίνουμε την πρακτική απόδοση των αλγορίθμων σε σχέση με το μήκος του προτύπου (μικρά και μεγάλα πρότυπα) κάτω από διαφορετικά μεγέθη αλφαβήτου (ή τύπων κειμένων) όπως το δυαδικό αλφάβητο, αλφάβητο μεγέθους 8, αγγλικό αλφάβητο και αλφάβητο DNA, τα οποία έχουν διαφορετικά χαρακτηριστικά.

2.3.1 Υπολογιστικό Περιβάλλον

Τα πειράματα εκτελέστηκαν σε ένα Sun UltraSparc-1 με ταχύτητα χρονισμού 143 Mhz, με 64 Mb RAM και ένα σκληρό δίσκο μεγέθους 2.1 Gb. Το λειτουργικό σύστημα ήταν το Solaris 2.5. Κατά τη διάρκεια των πειραμάτων, ο υπολογιστής μας δεν εκτελούσε άλλες βαριές εργασίες. Οι δομές δεδομένων που χρησιμοποιήθηκαν στην μελέτη μας ήταν αποθηκευμένες στην κύρια μνήμη του υπολογιστή κατά την διάρκεια των πειραμάτων. Τέλος, οι αλγόριθμοι που παρουσιάσαμε προηγουμένως έχουν υλοποιηθεί στην γλώσσα προγραμματισμού ANSI C [67] με ομοιογενή τρόπο ώστε η σύγκριση τους να είναι αμερόληπτη, χρησιμοποιώντας τον μεταγλωττιστή *cc*. Για ορισμένους αλγόριθμους χρησιμοποιήσαμε τον κώδικα που παρουσιάζεται στις εργασίες [5, 47, 100].

2.3.2 Δοκιμαστικά Δεδομένα

Όπως γνωρίζουμε η απόδοση των αλγορίθμων αναζήτησης αλφαριθμητικών εξαρτάται και από τις στατιστικές ιδιότητες του προτύπου καθώς και του κειμένου από τα οποία προκύπτουν τα δοκιμαστικά πρότυπα. Για αυτό το λόγο τα πειράματα εκτελέστηκαν για τέσσερις διαφορετικούς τύπους κειμένου: δυαδικό αλφάβητο, αλφάβητο μεγέθους 8, αγγλικό αλφάβητο και αλφάβητο DNA.

Δυαδικό Αλφάβητο

Το αλφάβητο είναι $\Sigma = \{0, 1\}$. Το κείμενο αποτελείται από 150,000 χαρακτήρες και δημιουργήθηκε τυχαία. Αναζητούμε 50 τυχαία κατασκευασμένα πρότυπα για κάθε μήκος του προτύπου από 2 έως 100 χαρακτήρες.

Αλφάβητο Μέγεθους 8

Το αλφάβητο είναι $\Sigma = \{a, b, c, d, e, f, g, h\}$. Το κείμενο αποτελείται από 150,000 χαρακτήρες και δημιουργήθηκε τυχαία. Επιπλέον, αναζητούμε 50 τυχαία κατασκευασμένα πρότυπα για κάθε μήκος του προτύπου από 2 έως 100 χαρακτήρες.

Αγγλικό Αλφάβητο

Χρησιμοποιήσαμε ένα έγγραφο αγγλικής γλώσσας από μια ιστοσελίδα του διαδίκτυου. Το αλφάβητο αποτελείται από 70 διαφορετικούς χαρακτήρες. Το κείμενο αποτελείται από 148,188 χαρακτήρες και αναζητούμε 50 πρότυπα για κάθε μήκος από 2 έως 100 χαρακτήρες που επιλέχτηκαν τυχαία μέσα από το ίδιο κείμενο.

Αλφάβητο DNA

Το αλφάβητο DNA αποτελείται από τέσσερα νουλεοκτίδια a, c, g και t (αντιπροσωπεύει την αδενίνη (adenine), την κυτοσίνη (cytosine), την γουανίνη (guanine) και την θυμίνη (thymine) αντίστοιχα) που χρησιμοποιούνται για να κωδικοποιήσουμε το DNA. Συνεπώς το αλφάβητο είναι $\Sigma = \{a, c, g, t\}$. Το

κείμενο αποτελείται από 997,642 χαρακτήρες και αναζητούμε 50 πρότυπα για κάθε μήκος από 10 έως 100 χαρακτήρες. Τέλος, το κείμενο και τα πρότυπα είναι ένα κομμάτι από την DNA βάση δεδομένων GenBank, που διανέμεται από τους Hume και Sunday [61].

Πρέπει να σημειώσουμε ότι στην πειραματική μας μεθοδολογία δεν χρησιμοποιήσαμε μεγέθη κειμένου τα οποία θα απαιτούσαν προσπελάσεις στο δίσκο. Ο λόγος είναι ότι με τα μεγέθη της κεντρικής μνήμης στα σύγχρονα υπολογιστικά συστήματα είναι τέτοια που εξασφαλίζουν τη δυνατότητα εκτέλεσης on-line αλγορίθμων αναζήτησης σε κείμενα με ικανοποιητικά μεγέθη. Έτσι, σε αυτή την περίπτωση επιλέξαμε δύο μεγέθη κειμένου (1 MB για το αγγλικό αλφάβητο και 150 KB για τα υπόλοιπα αλφάβητα). Αντίστοιχα αποτελέσματα μπορούν να εξαχθούν με οποιοδήποτε κείμενο μέσα στα όρια της μνήμης του υπολογιστή που χρησιμοποιήσαμε.

2.3.3 Μετρήσεις της Απόδοσης

Για τη σύγκριση των αλγορίθμων αναζήτησης αλφαριθμητικών χρησιμοποιούμε δύο μετρήσεις, τον αριθμό συγκρίσεων χαρακτήρων και το χρόνο εκτέλεσης. Η μέτρηση του αριθμού συγκρίσεων χαρακτήρων είναι ίδια όπως χρησιμοποιήθηκε από τον Smith [150] και υπολογίζεται ως ο λόγος του αριθμού των χαρακτήρων που πραγματικά συγκρίναμε προς τον αριθμό των χαρακτήρων του κειμένου που έχουμε διαβάσει. Στα πειράματα μας ο αριθμός των χαρακτήρων που έχουμε διαβάσει είναι πάντα $n - m + 1$, αφού όλοι οι αλγόριθμοι υλοποιήθηκαν για να εντοπίσουν όλες τις εμφανίσεις του προτύπου μέσα στο κείμενο. Ο χρόνος εκτέλεσης είναι ο συνολικός χρόνος που απαιτείται από έναν αλγόριθμο για να αναζητήσει ένα πρότυπο μέσα στο κείμενο, περιλαμβάνοντας το χρόνο της προεπεξεργασίας. Ο χρόνος εκτέλεσης προκύπτει από την κλήση της συνάρτησης C clock() και μετριέται σε δευτερόλεπτα.

Προκειμένου να εξετάσουμε την επίδραση του μήκους του προτύπου, μετρήσαμε τον αριθμό συγκρίσεων χαρακτήρων και τον χρόνο εκτέλεσης για όλους τους αλγόριθμους που περιγράψαμε στην παράγραφο 2.2. Μετρήσαμε την επίδραση του μήκους του προτύπου μεταβάλοντας τις τιμές του m=2, 3, 4, 5, 6, 7, 8, 9, 10, 20, 30, 40, 60, 80 και 100. Στην περίπτωση του αλφαβήτου DNA χρησιμοποιήσαμε μεγαλύτερα πρότυπα επειδή στις βιολογικές εφαρμογές συναντώνται μόνο μεγάλα πρότυπα. Για αυτό τον λόγο, μετρήσαμε την επίδραση του μήκους του προτύπου μεταβάλοντας τις τιμές του m=10, 20, 30, 40, 50 και 100.

Τέλος, τα πειραματικά αποτελέσματα των αλγορίθμων προκύπτουν από τον μέσο όρο 50 εκτελέσεων με διαφορετικά πρότυπα για κάθε μήκος για να περιορίζουμε τις τυχαίες διακυμάνσεις.

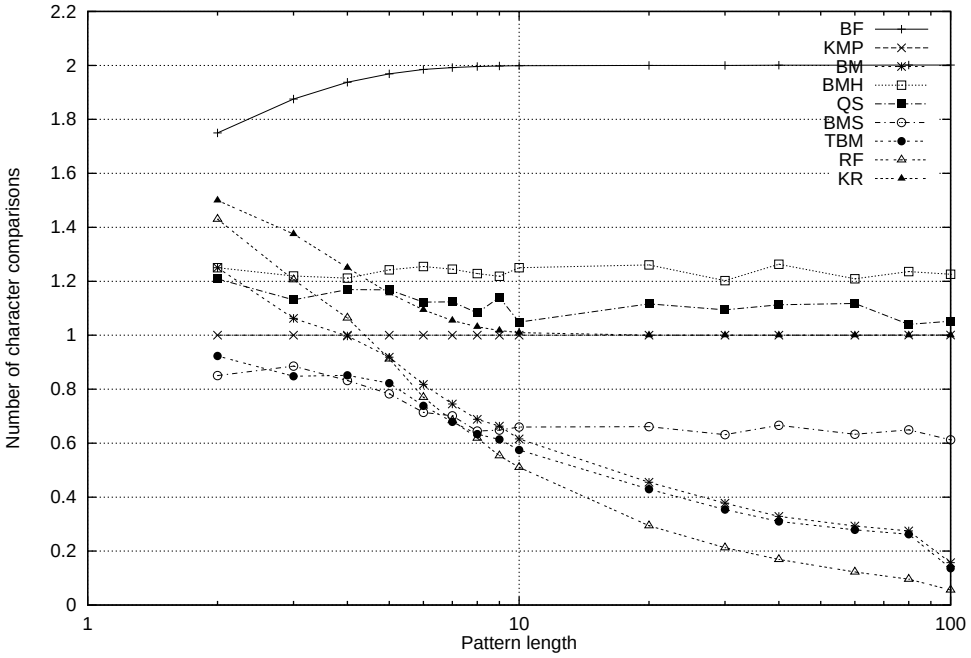
Για τους αλγορίθμους bit-παραλληλισμού όπως SO και BNDM μετρήσαμε μόνο τον χρόνο εκτέλεσης επειδή οι αλγόριθμοι αυτοί χρησιμοποιούν μόνο έμμεσες συγκρίσεις χαρακτήρων. Επίσης, η εκτέλεση των αλγορίθμων αυτών περιορίστηκε για μήκος του προτύπου μικρότερο από το μέγεθος της λέξης του υπολογιστή σε bits. Για αυτό τον λόγο, στην πειραματική μας μελέτη οι αλγόριθμοι SO και BNDM περιορίστηκαν σε $m \leq 31$.

2.4 Πειραματικά Αποτελέσματα

Στις προηγούμενες παραγράφους παρουσιάσαμε σύντομα τους δημοφιλείς αλγόριθμους αναζήτησης αλφαριθμητικών και την πειραματική μας μεθοδολογία. Σε αυτή την παράγραφο, παρουσιάζουμε τα πειραματικά αποτελέσματα για τους αλγόριθμους αναζήτησης αλφαριθμητικών σύμφωνα με τον αριθμό συγκρίσεων χαρακτήρων και το χρόνο εκτέλεσης. Τέλος, η απόδοση του κάθε αλγόριθμου απεικονίζεται σε συνάρτηση με το μήκος του προτύπου για κάθε είδος κειμένου.

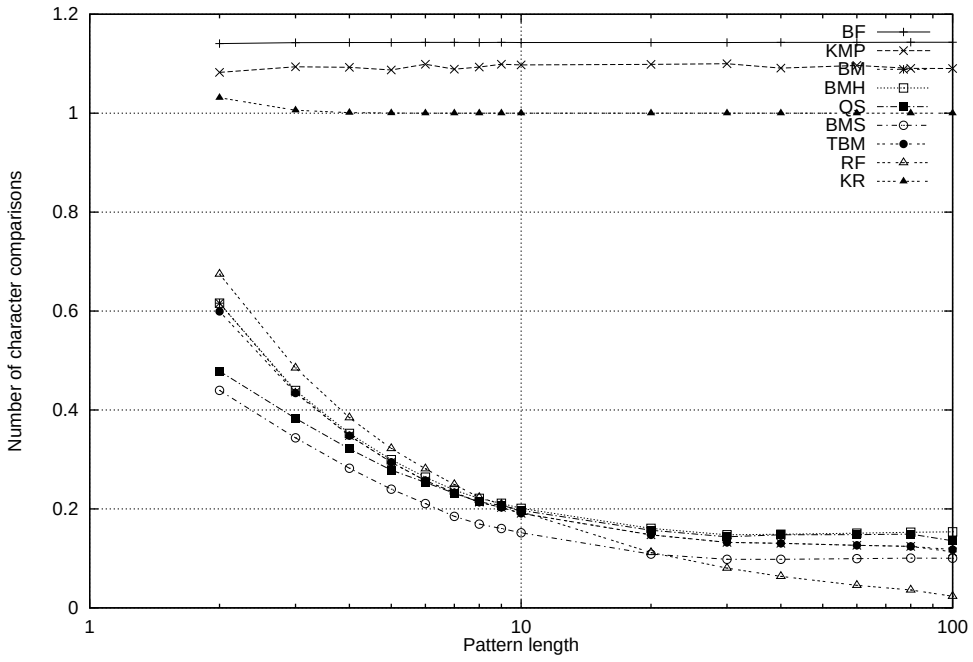
2.4.1 Αποτελέσματα για τον Αριθμό Συγκρίσεων Χαρακτήρων

Στα Σχήματα 2.1 έως 2.4 και στους Πίνακες 2.2 έως 2.5 παρουσιάζονται αποτελέσματα για τον αριθμό συγκρίσεων χαρακτήρων για το δυαδικό αλφάβητο, αλφάβητο μεγέθους 8, αγγλικό αλφάβητο και αλφάβητο DNA αντίστοιχα, σε συνάρτηση με το μήκος του προτύπου. Από τα αποτελέσματα προκύπτει ότι οι αλγόριθμοι KMP και KR δίνουν ακριβώς 1 σύγκριση ανά χαρακτήρα σε όλες τις περιπτώσεις. Επίσης, ο αλγόριθμος BF δίνει περίπου τον ίδιο αριθμό συγκρίσεων χαρακτήρων με τους αλγόριθμους KMP και KR για το αλφάβητο μεγέθους 8 και για το αγγλικό αλφάβητο. Όμως, ο αλγόριθμος BF απαιτεί περισσότερες συγκρίσεις χαρακτήρων για μικρά μεγέθη αλφαβήτου όπως είναι το δυαδικό ή αλφάβητο DNA. Με βάση τα πειραματικά αποτελέσματα είναι σαφές ότι για πρότυπα μεγαλύτερα από 10 χαρακτήρες, ο αριθμός συγκρίσεων χαρακτήρων του αλγόριθμου BF είναι κοντά στο 2, διπλάσιος από τον αριθμό που απαιτείται από τους αλγόριθμους KMP και KR για το δυαδικό αλφάβητο. Για το αλφάβητο DNA, ο αλγόριθμος BF απαιτεί 1,34 συγκρίσεις χαρακτήρων κατά μέσο όρο. Αυτό

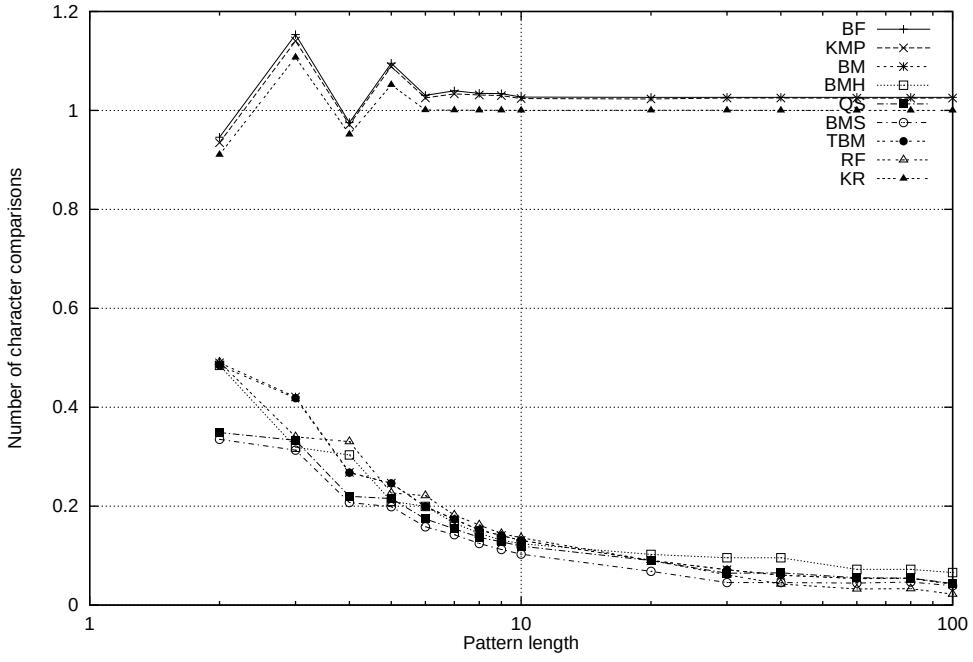


Σχήμα 2.1: Δυαδικό αλφάβητο

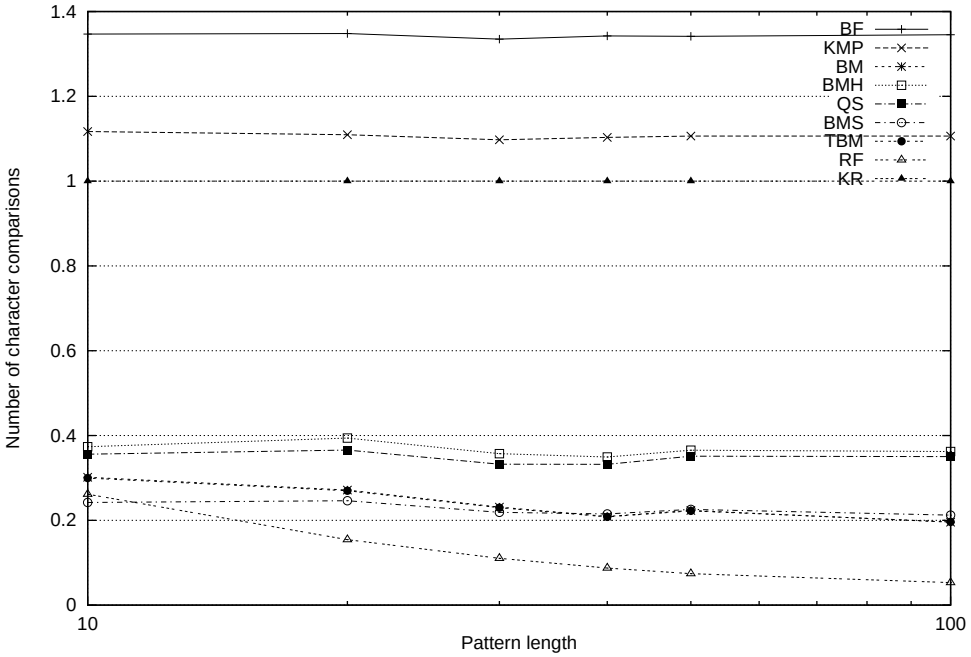
παρατηρείται επειδή όταν χρησιμοποιούνται μικρά μεγέθη αλφαβήτου καταλήγει να παρουσιάζει πολλές ταυτίσεις του προτύπου στο κείμενο και αυτό έχει σαν αποτέλεσμα να επιτείνει τον αριθμό συγκρίσεων χαρακτήρων μεγαλύτερο από 1. Όμως, όταν χρησιμοποιούνται μεγάλα μεγέθη αλφαβήτου το παραπάνω φαινόμενο εξαλείφεται σύμφωνα με τα Σχήματα 2.2 και 2.3.



Σχήμα 2.2: Αλφάβητο μεγέθους 8



Σχήμα 2.3: Αγγλικό αλφάβητο



Σχήμα 2.4: Αλφάβητο DNA

Πίνακας 2.2: Αριθμός συγκρίσεων χαρακτήρων για το δυαδικό αλφάβητο

m	BF	KMP	BM	BMH	QS	BMS	TBM	RF	KR
2	1.750004	1.000007	1.250156	1.250162	1.208933	0.850429	0.923206	1.430285	1.499949
3	1.875135	1.000014	1.062643	1.219574	1.131196	0.885437	0.848012	1.205877	1.375112
4	1.937597	1.000019	0.99704	1.211507	1.16929	0.832074	0.851674	1.063907	1.250036
5	1.968953	1.000027	0.918733	1.242546	1.168212	0.782943	0.822228	0.912138	1.155996
6	1.985074	1.000034	0.817716	1.254749	1.122865	0.714068	0.738341	0.769741	1.094103
7	1.992119	1.000039	0.745201	1.244882	1.123715	0.701102	0.679011	0.683943	1.054372
8	1.996129	1.000046	0.688121	1.228595	1.082625	0.644872	0.633306	0.618486	1.031142
9	1.998021	1.000053	0.662976	1.218788	1.14058	0.648325	0.613832	0.553692	1.017631
10	1.999020	1.00006	0.615933	1.250229	1.048682	0.659736	0.574773	0.510695	1.009614
20	1.999883	1.000126	0.454975	1.260394	1.116437	0.661115	0.429198	0.293805	1.000021
30	2.000081	1.000194	0.377558	1.202327	1.094088	0.631779	0.354037	0.212619	1
40	2.000906	1.00026	0.328498	1.263311	1.112723	0.66606	0.309631	0.16883	1
60	2.000806	1.000393	0.293197	1.209277	1.117767	0.632828	0.278643	0.122547	1
80	2.001165	1.000527	0.274593	1.235767	1.04045	0.649387	0.262522	0.09567	1
100	2.001170	1.000428	0.15729	1.225893	1.05212	0.612255	0.13646	0.0558	1
MO	1.967071	1.000148	0.642975	1.234533	1.115312	0.704827	0.563658	0.579869	1.099198

Πίνακας 2.3: Αριθμός συγκρίσεων χαρακτήρων για το αλφάβητο μεγέθους 8

m	BF	KMP	BM	BMH	QS	BMS	TBM	RF	KR
2	1.140649	1.082482	0.615808	0.615813	0.478422	0.439532	0.599131	0.674582	1.031357
3	1.142672	1.093926	0.436387	0.439241	0.383025	0.343549	0.434308	0.484915	1.005877
4	1.142858	1.092589	0.34825	0.352508	0.321288	0.282322	0.347923	0.384407	1.000959
5	1.142851	1.08735	0.29435	0.299585	0.278561	0.239789	0.294242	0.322069	1.000147
6	1.142999	1.09885	0.25708	0.264167	0.25374	0.210801	0.257024	0.281238	1.000025
7	1.143042	1.089026	0.233039	0.238104	0.231652	0.184943	0.233001	0.249258	1.000002
8	1.142850	1.093259	0.213613	0.221248	0.213952	0.169241	0.213557	0.223297	1.000002
9	1.143069	1.098822	0.20296	0.211367	0.207332	0.160332	0.202901	0.20939	1
10	1.142826	1.097637	0.191084	0.200971	0.197028	0.15178	0.191005	0.194314	1
20	1.142982	1.098593	0.147593	0.160721	0.156911	0.108738	0.147563	0.111766	1
30	1.142989	1.09997	0.132365	0.147843	0.143672	0.098227	0.132305	0.080353	1
40	1.143216	1.091098	0.130546	0.148864	0.14762	0.098028	0.13049	0.063557	1
60	1.143164	1.096612	0.126438	0.151015	0.14808	0.099525	0.126395	0.045576	1
80	1.143461	1.090527	0.124236	0.15292	0.149092	0.100521	0.124252	0.036088	1
100	1.143460	1.090428	0.11325	0.15365	0.13569	0.100256	0.117822	0.02355	1
MO	1.142873	1.093411	0.2378	0.250534	0.229738	0.185839	0.236795	0.225624	1.002558

Πίνακας 2.4: Αριθμός συγκρίσεων χαρακτήρων για το αγγλικό αλφάβητο

m	BF	KMP	BM	BMH	QS	BMS	TBM	RF	KR
2	0.945947	0.934728	0.489746	0.484534	0.348815	0.33509	0.484534	0.491121	0.910423
3	1.153414	1.140354	0.420683	0.319074	0.333132	0.313255	0.418214	0.340048	1.107222
4	0.975423	0.971058	0.26808	0.303567	0.21983	0.207258	0.267662	0.330358	0.95157
5	1.095298	1.088676	0.246358	0.209947	0.21523	0.198916	0.24601	0.226646	1.05163
6	1.030530	1.025256	0.199315	0.199341	0.174192	0.158094	0.199159	0.221283	1.000921
7	1.039575	1.033503	0.173535	0.166126	0.154276	0.142217	0.173438	0.18181	1.000627
8	1.034596	1.031199	0.152092	0.145059	0.136806	0.124448	0.152039	0.161933	1.000307
9	1.033859	1.029943	0.140771	0.131248	0.128023	0.112471	0.140735	0.144603	1.00033
10	1.026855	1.02403	0.13066	0.123912	0.119084	0.102977	0.130652	0.136166	1.000129
20	1.026384	1.023119	0.090241	0.10256	0.090146	0.068208	0.090234	0.090352	1.000136
30	1.026308	1.024997	0.07137	0.09563	0.064438	0.045643	0.071357	0.061233	1.000204
40	1.026305	1.024999	0.06	0.09562	0.065123	0.045633	0.061	0.042563	1
60	1.026450	1.0248	0.05421	0.07225	0.05521	0.044523	0.0533	0.03266	1
80	1.026449	1.0249	0.05424	0.07231	0.05456	0.04652	0.05423	0.033123	1
100	1.026451	1.02499	0.0423	0.06566	0.04361	0.03895	0.0423	0.02223	1
MO	1.032923	1.028437	0.172907	0.172456	0.146832	0.13228	0.172324	0.167742	1.001567

Πίνακας 2.5: Αριθμός συγκρίσεων χαρακτήρων για το αλφάβητο DNA

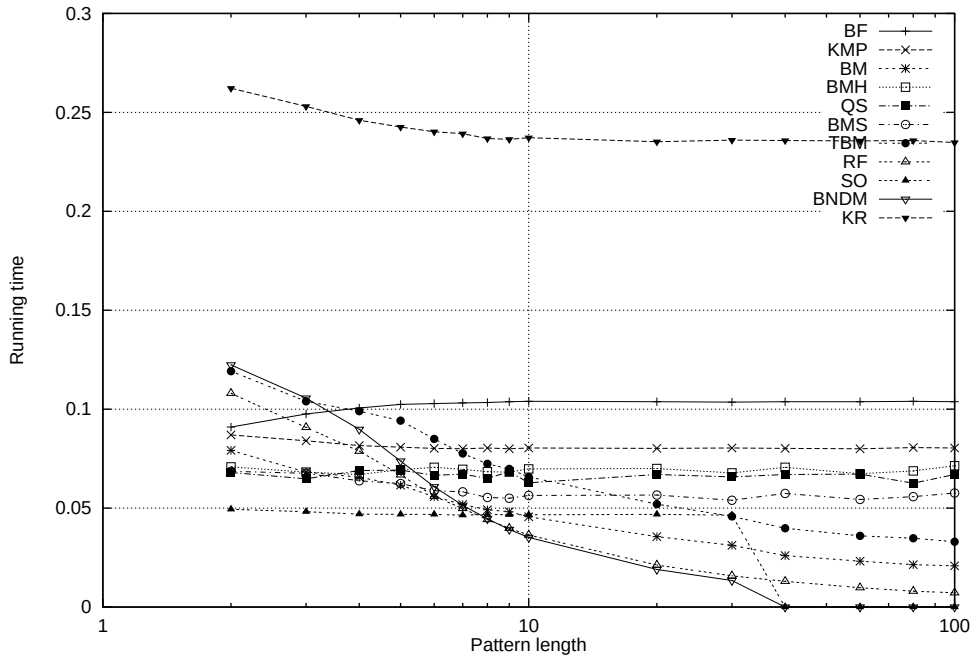
m	BF	KMP	BM	BMH	QS	BMS	TBM	RF	KR
10	1.346732	1.116931	0.301635	0.373474	0.356066	0.242178	0.299615	0.26149	1.000031
20	1.348057	1.109472	0.271363	0.39388	0.365869	0.246332	0.269731	0.154379	1.000009
30	1.335037	1.097344	0.231094	0.357166	0.332303	0.218819	0.229967	0.110341	1.000023
40	1.342449	1.103267	0.209129	0.349426	0.332189	0.215021	0.208034	0.087277	1.000018
50	1.34136	1.106258	0.223401	0.365487	0.351301	0.225939	0.222463	0.074325	1.000022
100	1.3453	1.106325	0.19552	0.362155	0.350235	0.212253	0.19623	0.05325	1
MO	1.343156	1.1066	0.23869	0.366931	0.347994	0.226757	0.237673	0.12351	1.000017

Ο αριθμός συγκρίσεων χαρακτήρων των αλγορίθμων της οικογένειας BM (όπως είναι BM, BMH, QS, BMS και TBM) και του αλγορίθμου μεταθεματικού αυτομάτου (όπως είναι RF) είναι γενικά μικρότερος από 1, με εξαίρεση στο δυαδικό αλφάβητο όπου οι αλγόριθμοι BMH και QS έχουν κατά μέσο όρο 1,25 και 1,1 συγκρίσεις χαρακτήρων αντίστοιχα. Επιπλέον, αυτό που πρέπει να σημειωθεί είναι ότι όταν χρησιμοποιείται το δυαδικό αλφάβητο ο αριθμός συγκρίσεων χαρακτήρων των αλγορίθμων της οικογένειας BM και του αλγόριθμου RF είναι σημαντικά υψηλότερος από οποιαδήποτε άλλο είδος κειμένου. Επίσης, πρέπει να παρατηρηθεί ότι ο αριθμός συγκρίσεων χαρακτήρων για όλους τους αλγόριθμους της οικογένειας BM και του αλγόριθμου RF μειώνεται σημαντικά καθώς το μήκος του προτύπου αυξάνεται. Συνεπώς, τα εμπειρικά αποτελέσματα υποστηρίζουν την θεωρητική ανάλυση ότι οι αλγόριθμοι της οικογένειας BM και του αλγόριθμου RF είναι υπογραμμικοί σε αριθμό συγκρίσεων χαρακτήρων. Έτσι, ο αριθμός συγκρίσεων χαρακτήρων μειώνεται πιο αργά καθώς το μήκος του προτύπου αυξάνεται επειδή για τα μεγάλα πρότυπα είναι υψηλότερη η πιθανότητα ο χαρακτήρας που προσκομίστηκε να εμφανίζεται κάπου μέσα στο πρότυπο και επομένως η απόσταση που το πρότυπο μπορεί να μετακινηθεί (αν εμφανίζεται ανεπιτυχής ταύτιση) μικραίνει. Επίσης, ο αριθμός συγκρίσεων χαρακτήρων όλων των αλγορίθμων της οικογένειας BM είναι πολύ κοντά με τα αποτελέσματα των υπόλοιπων αλγορίθμων και τείνει να σταθεροποιηθεί σε ένα σημείο εκτός από το δυαδικό αλφάβητο. Τέλος, για μεγάλα πρότυπα η διαφορά ανάμεσα στον αριθμό συγκρίσεων χαρακτήρων των αλγορίθμων της οικογένειας BM και στον αριθμό συγκρίσεων χαρακτήρων του αλγορίθμου μεταθεματικού αυτομάτου όπως είναι ο RF αυξάνει για όλα τα είδη κειμένου.

Στα Σχήματα και στους Πίνακες για όλα τα είδη κειμένου φαίνεται ότι οι αλγόριθμοι της οικογένειας BM και ο αλγόριθμος RF δίνουν καλύτερα αποτελέσματα. Συγκεριμένα, οι αλγόριθμοι της οικογένειας BM (όπως TBM και BMS) και ο αλγόριθμος RF είναι πολύ περισσότερο αποτελεσματικοί σε όρους συγκρίσεων χαρακτήρων από τους υπόλοιπους αλγόριθμους για μικρά και μεγάλα πρότυπα αντίστοιχα.

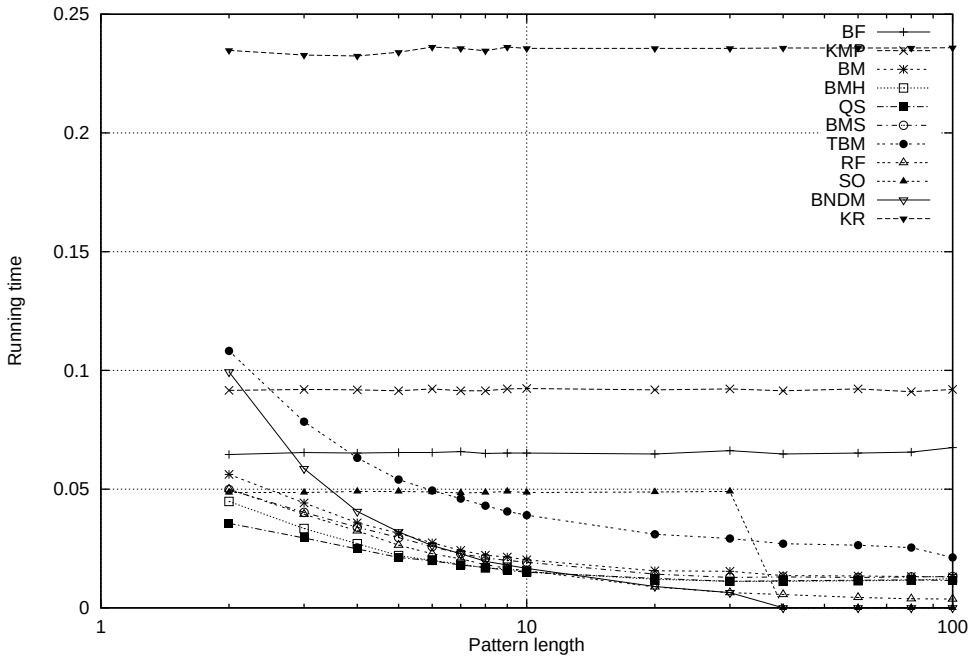
2.4.2 Αποτελέσματα για τον Χρόνο Εκτέλεσης

Στα Σχήματα 2.5 έως 2.8 και στους Πίνακες 2.6 έως 2.9 παρουσιάζονται αποτελέσματα για τον χρόνο εκτέλεσης για το δυαδικό αλφάβητο, αλφάβητο μεγέθους 8, αγγλικό αλφάβητο και αλφάβητο DNA αντίστοιχα, σε συνάρτηση με το μήκος του προτύπου. Από τα αποτελέσματα παρατηρούμε ότι για όλα τα είδη κειμένου ο αλγόριθμος KR απαιτεί πολύ περισσότερο χρόνο από τους υπόλοιπους αλγορίθμους.

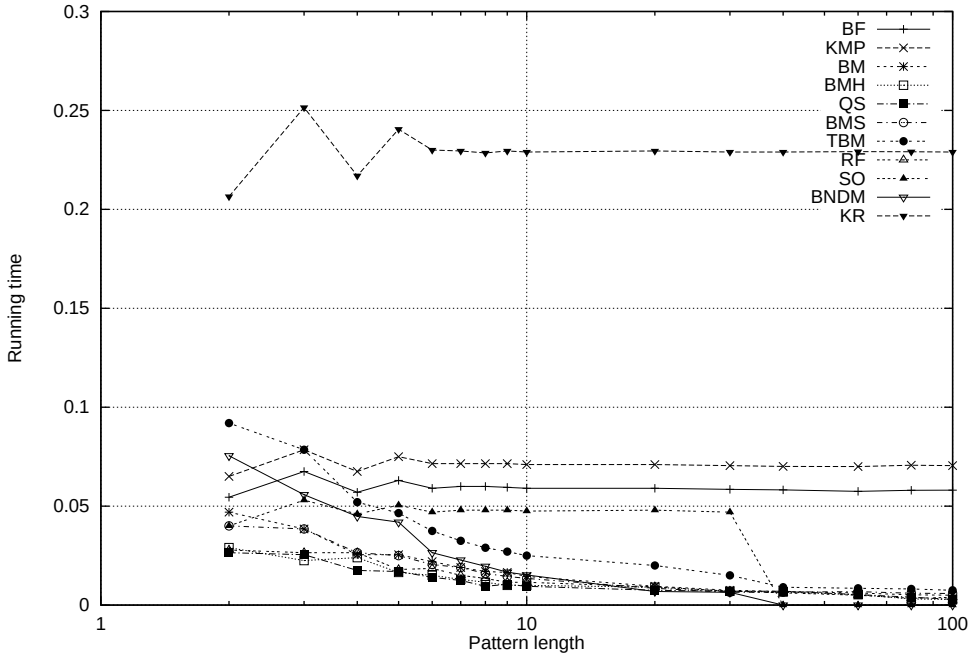


Σχήμα 2.5: Δυαδικό αλφάβητο

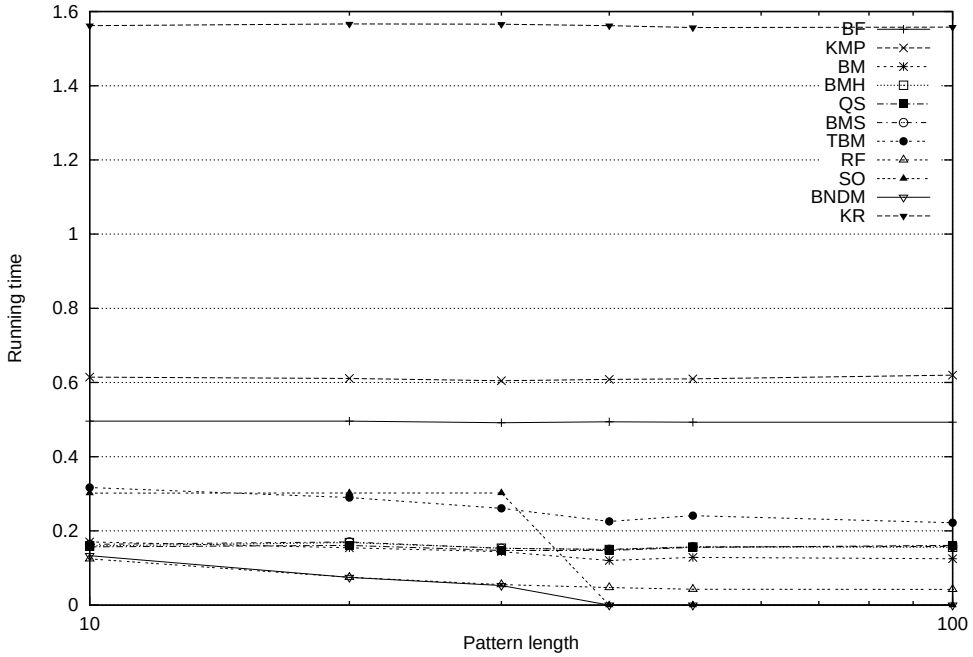
Αυτή η παρατήρηση είναι σύμφωνη με την αναμενόμενη συμπεριφορά αφού ο υπολογισμός των τιμών κατακερματισμού είναι υπολογιστικά ακριβός σε κύκλους μηχανής και συνεπώς αυξάνει τον χρόνο εκτέλεσης του αλγόριθμου KR. Άρα, ο αλγόριθμος αυτός δεν συνιστάται για εφαρμογές κειμένου.



Σχήμα 2.6: Αλφάβητο μεγέθους 8



Σχήμα 2.7: Αγγλικό αλφάβητο



Σχήμα 2.8: Αλφάβητο DNA

Πίνακας 2.6: Χρόνους εκτέλεσης για το δυαδικό αλφάβητο

m	BF	KMP	BM	BMH	QS	BMS	TBM	RF	SO	BNDM	KR
2	0.0910	0.087	0.0792	0.0708	0.068	0.0686	0.1192	0.108	0.0494	0.1224	0.2622
3	0.0976	0.084	0.0682	0.0682	0.0648	0.0676	0.104	0.0908	0.0482	0.1056	0.253
4	0.1006	0.0816	0.0656	0.067	0.069	0.0638	0.099	0.0788	0.047	0.0898	0.246
5	0.1024	0.0808	0.0616	0.0694	0.0692	0.0624	0.0942	0.0668	0.0468	0.0738	0.2426
6	0.1028	0.0802	0.0558	0.0706	0.0666	0.0588	0.085	0.056	0.0468	0.0608	0.2402
7	0.1032	0.08	0.0518	0.0696	0.0672	0.0582	0.0776	0.0498	0.0464	0.0514	0.2392
8	0.1034	0.0804	0.0492	0.0684	0.065	0.0554	0.0724	0.0442	0.0468	0.0446	0.2368
9	0.1038	0.08	0.0482	0.0682	0.0684	0.055	0.0698	0.0398	0.0466	0.0392	0.2364
10	0.1040	0.0804	0.0456	0.0698	0.0628	0.0564	0.0658	0.0364	0.0466	0.0352	0.2372
20	0.1038	0.0802	0.0356	0.07	0.067	0.0566	0.052	0.0212	0.0468	0.019	0.2352
30	0.1036	0.0804	0.0312	0.0678	0.0658	0.054	0.0458	0.0158	0.0464	0.0134	0.236
40	0.1038	0.0802	0.026	0.0706	0.067	0.0574	0.0398	0.013	-	-	0.2358
60	0.1038	0.08	0.0232	0.0674	0.0672	0.0544	0.036	0.0098	-	-	0.2356
80	0.1040	0.0806	0.0214	0.0688	0.0626	0.0558	0.0348	0.008	-	-	0.2358
100	0.1038	0.0804	0.0208	0.0714	0.067	0.0576	0.033	0.0072	-	-	0.2348
MO	0.102107	0.08108	0.04556	0.0692	0.066507	0.0588	0.06856	0.04304	0.047073	0.059564	0.240453

Πίνακας 2.7: Χρόνους εκτέλεσης για το αλφάβητο μεγέθους 8

m	BF	KMP	BM	BMH	QS	BMS	TBM	RF	SO	BNDM	KR
2	0.0646	0.0916	0.0562	0.0448	0.0356	0.0498	0.1082	0.05	0.0486	0.0994	0.2348
3	0.0654	0.092	0.0442	0.0334	0.0294	0.0402	0.0784	0.0394	0.0486	0.0586	0.2328
4	0.0652	0.0918	0.0358	0.027	0.0248	0.034	0.0632	0.0324	0.049	0.0406	0.2324
5	0.0654	0.0914	0.0314	0.022	0.0212	0.0296	0.054	0.0264	0.049	0.032	0.234
6	0.0654	0.0922	0.0274	0.02	0.0198	0.0258	0.0494	0.0228	0.0488	0.0262	0.2362
7	0.0658	0.0914	0.0242	0.0184	0.018	0.0228	0.046	0.0206	0.0484	0.0228	0.2356
8	0.0650	0.0914	0.0222	0.0168	0.017	0.021	0.043	0.0182	0.0486	0.0196	0.2346
9	0.0652	0.0922	0.0214	0.0164	0.0158	0.02	0.0406	0.0166	0.049	0.018	0.2362
10	0.0652	0.0924	0.0202	0.0152	0.015	0.0192	0.039	0.0156	0.0486	0.0166	0.2356
20	0.0648	0.0918	0.0156	0.012	0.0124	0.0142	0.031	0.0088	0.0488	0.009	0.2356
30	0.0662	0.0922	0.0154	0.0112	0.0112	0.0128	0.0292	0.0064	0.049	0.0064	0.2356
40	0.0648	0.0914	0.0136	0.0112	0.0114	0.013	0.027	0.0056	-	-	0.2358
60	0.0652	0.0922	0.0134	0.0114	0.0116	0.0128	0.0264	0.0044	-	-	0.2358
80	0.0656	0.091	0.0132	0.0116	0.0118	0.013	0.0254	0.0038	-	-	0.2358
100	0.0675	0.092	0.013	0.0123	0.0115	0.0132	0.0212	0.0037	-	-	0.2359
MO	0.06542	0.0918	0.02448	0.018913	0.017767	0.02276	0.045467	0.018313	0.048764	0.031745	0.235113

Πίνακας 2.8: Χρόνους εκτέλεσης για το αγγλικό αλφάβητο

m	BF	KMP	BM	BMH	QS	BMS	TBM	RF	SO	BNDM	KR
2	0.0545	0.065	0.047	0.02901	0.0265	0.04	0.092	0.0278	0.04	0.0755	0.2065
3	0.0675	0.0785	0.0385	0.0225	0.0255	0.0385	0.0785	0.0265	0.053	0.0558	0.2515
4	0.0570	0.0675	0.0255	0.024	0.0175	0.0265	0.052	0.0265	0.046	0.0448	0.217
5	0.0630	0.075	0.0255	0.0165	0.017	0.025	0.0465	0.018	0.0505	0.0419	0.2405
6	0.0590	0.0715	0.022	0.015	0.014	0.0205	0.0375	0.0185	0.047	0.0265	0.23
7	0.0600	0.0715	0.019	0.013	0.0125	0.019	0.0325	0.015	0.048	0.0228	0.2295
8	0.0600	0.0715	0.017	0.0115	0.0095	0.016	0.029	0.0135	0.048	0.0196	0.2285
9	0.0595	0.0715	0.0165	0.01	0.0105	0.0145	0.027	0.0115	0.048	0.01666	0.2295
10	0.0590	0.071	0.0145	0.01	0.0095	0.0135	0.025	0.0115	0.0475	0.0152	0.229
20	0.0590	0.071	0.0095	0.009	0.0075	0.0085	0.02	0.00921	0.048	0.0069	0.2295
30	0.0585	0.0705	0.0075	0.007	0.007	0.0065	0.015	0.007	0.047	0.0065	0.229
40	0.0582	0.0701	0.0069	0.00659	0.00699	0.006512	0.009	0.0062	-	-	0.229
60	0.0575	0.07	0.0061	0.00523	0.00532	0.00681	0.00852	0.005	-	-	0.2292
80	0.0580	0.0707	0.0052	0.0031	0.004	0.006	0.00812	0.00345	-	-	0.2291
100	0.0581	0.0705	0.005	0.00282	0.0029	0.00585	0.0075	0.00389	-	-	0.229
MO	0.05925	0.07105	0.017713	0.01235	0.011747	0.016911	0.032543	0.01357	0.047545	0.03019	0.22912

Πίνακας 2.9: Χρόνους εκτέλεσης για το αλφάβητο DNA

m	BF	KMP	BM	BMH	QS	BMS	TBM	RF	SO	BNDM	KR
10	0.4958	0.6146	0.1704	0.1596	0.157	0.1638	0.317	0.1248	0.3018	0.1334	1.5618
20	0.4958	0.6108	0.1546	0.1684	0.161	0.1702	0.2902	0.0752	0.3024	0.0746	1.5664
30	0.4916	0.605	0.144	0.1534	0.1472	0.153	0.261	0.0554	0.3024	0.0528	1.5658
40	0.494	0.6084	0.1204	0.1498	0.1474	0.1502	0.2256	0.0476	-	-	1.5618
50	0.4932	0.61	0.1286	0.1568	0.1556	0.157	0.2412	0.0428	-	-	1.557
100	0.4931	0.62	0.1252	0.156	0.161	0.1581	0.2221	0.0424	-	-	1.5578
MO	0.493917	0.611467	0.140533	0.157333	0.154867	0.158717	0.259517	0.0647	0.3022	0.086933	1.561767

Επιπλέον, από τα πειραματικά αποτελέσματα για όλα τα είδη κειμένου διαπιστώνεται ότι ο αλγόριθμος KMP είναι λίγο πιο αργός από τον αλγόριθμο BF για όλα τα μήκη του προτύπου με εξαίρεση το δυαδικό αλφάβητο. Αυτή η συμπεριφορά είναι σύμφωνη με την θεωρητική ανάλυση ότι ο αλγόριθμος KMP δεν είναι καλύτερος από τον αλγόριθμο BF κατά μέσο όρο. Επίσης, από τα αποτελέσματα φαίνεται ότι για όλα τα αλφάβητα οι αλγόριθμοι BF και KMP είναι σημαντικά πιο αργοί από τους αλγόριθμους της οικογένειας BM και τους αλγορίθμους bit-παραλληλισμού.

Ο χρόνος εκτέλεσης των αλγορίθμων BM και bit-παραλληλισμού όπως BNDM μειώνεται σημαντικά καθώς το μήκος του προτύπου αυξάνεται. Επιπλέον, πρέπει να σημειωθεί ότι όλοι οι αλγόριθμοι BM δίνουν περίπου ίδιους χρόνους εκτέλεσης για όλα τα είδη κειμένου με εξαίρεση το δυαδικό αλφάβητο. Επίσης, για μεγάλα πρότυπα, η διαφορά ανάμεσα στον χρόνο εκτέλεσης των αλγορίθμων BM και στον χρόνο εκτέλεσης του αλγόριθμου RF αυξάνει για όλες τις περιπτώσεις εκτός του αγγλικού αλφάβητου. Αυτή η διαφορά είναι προς όφελος του αλγόριθμου RF.

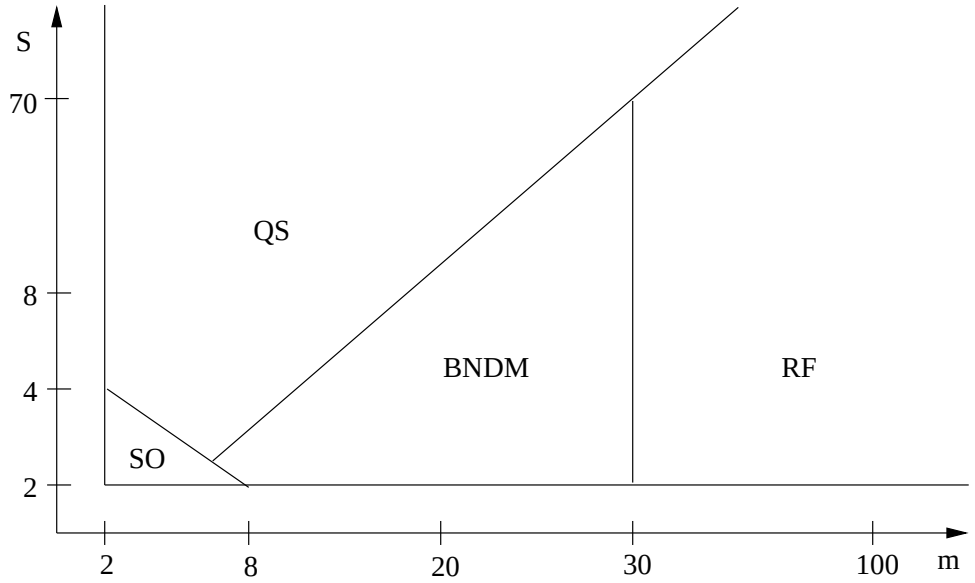
Ο αλγόριθμος SO έχει καλύτερη απόδοση από τους αλγόριθμους KR, KMP και BF για όλα τα μήκη του προτύπου. Επίσης, ο αλγόριθμος SO είναι ταχύτερος από τους αλγόριθμους TBM και BNDM μόνο για μικρά πρότυπα. Αυτή η παρατήρηση ισχύει για όλα τα αλφάβητα με εξαίρεση το δυαδικό αλφάβητο. Όμως, στο δυαδικό αλφάβητο ο αλγόριθμος SO είναι ταχύτερος από όλους τους αλγόριθμους BM και RF για μικρά πρότυπα.

Τέλος, από τα αποτελέσματα παρατηρούμε ότι στην πλειοψηφία των περιπτώσεων ο αλγόριθμος RF έχει μικρότερο χρόνο εκτέλεσης από τους αλγόριθμους της οικογένειας BM και τους αλγορίθμους bit-παραλληλισμού για μεγάλα πρότυπα. Επίσης, όλοι οι αλγόριθμοι της οικογένειας BM έχουν καλύτερους χρόνους εκτέλεσης για μικρά πρότυπα εκτός του δυαδικού αλφαβήτου.

2.5 Πειραματικός Χάρτης

Σε αυτή την ενότητα παρουσιάζουμε ένα χάρτη απόδοσης διαφορετικών αλγορίθμων αναζήτησης αλφαριθμητικών, δείχνοντας περιοχές που οι αλγόριθμοι είναι αποτελεσματικοί στην πράξη.

Στο Σχήμα 2.9 παρουσιάζεται ο χάρτης απόδοσης διαφορετικών αλγορίθμων. Από το Σχήμα φαίνεται ότι μόνο οι αλγόριθμοι QS, SO, BNDM και RF έχουν μια περιοχή υπεροχής στον χάρτη, ενώ



Σχήμα 2.9: Χάρτης πειραματικής απόδοσης για διαφορετικούς αλγορίθμους αναζήτησης αλφαριθμητικών

οι υπόλοιποι αλγόριθμοι είναι πιο αργοί.

Από τον χάρτη συμπεραίνουμε ότι ο αλγόριθμος QS είναι αποτελεσματικός καθώς το μέγεθος του αλφαβήτου και του προτύπου αυξάνεται. Επίσης, ο αλγόριθμος SO είναι ο ταχύτερος μόνο για μικρά πρότυπα και μικρά μεγέθη αλφαβήτου. Ο αλγόριθμος BNDM περιορίζεται σε μια μικρή περιοχή για μικρά μεγέθη αλφαβήτου και μεσαία πρότυπα. Τέλος, ο αλγόριθμος RF είναι καλύτερος για πολύ μεγάλα πρότυπα και μικρά/μεσαία μεγέθη αλφαβήτου.

Κεφάλαιο 3

Προσεγγιστική Αναζήτηση Αλφαριθμητικών: Επισκόπηση και Πειραματικά Αποτελέσματα

3.1 Εισαγωγή

Το πρόβλημα της προσεγγιστικής αναζήτησης αλφαριθμητικών (approximate string searching) είναι μια γενίκευση του προβλήματος της απλής αναζήτησης αλφαριθμητικών (exact string searching), το οποίο περιλαμβάνει την εύρευση υποαλφαριθμητικών του κειμένου που είναι κοντά ή πλησιέστερα προς το πρότυπο αλφαριθμητικό. Συγκεριμένα, το πρόβλημα της προσεγγιστικής αναζήτησης αλφαριθμητικών μπορεί να οριστεί ως εξής: Δίνεται ένα αλφάβητο Σ, ένα μικρό πρότυπο αλφαριθμητικό (pattern string) p μήκους m, ένα μεγάλο κείμενο αλφαριθμητικό (text string) t μήκους n με $m \ll n$, ένας ακέραιος αριθμός $k \geq 0$ και μια συνάρτηση απόστασης (distance function) d. Το πρόβλημα είναι να βρεθούν όλα τα υποαλφαριθμητικά s του κειμένου t τέτοιο ώστε $d(p, s) \leq k$.

Η απόσταση $d(p, s)$ ανάμεσα σε δύο αλφαριθμητικά p και s για ένα αλφάβητο Σ είναι το ελάχιστο κόστος ακολουθίας πράξεων έτσι ώστε να μετασχηματίσουμε το p σε s. Το κόστος της ακολουθίας των πράξεων είναι το άθροισμα του κόστους των ξεχωριστών απαιτούμενων πράξεων. Το κόστος μιας

πράξης θεωρείται ότι είναι ίσο με έναν θετικό αριθμό.

Σε εφαρμογές αναζήτησης αλφαριθμητικών οι πιο ενδιαφέρουσες πράξεις είναι: α) η αλλαγή ενός χαρακτήρα με ένα άλλο χαρακτήρα (ή αντικατάσταση), β) η διαγραφή ενός χαρακτήρα από ένα δοσμένο αλφαριθμητικό (ή διαγραφή) και γ) η εισαγωγή ενός χαρακτήρα σε ένα δοσμένο αλφαριθμητικό (ή εισαγωγή).

Υπάρχουν διάφορες συναρτήσεις απόστασης, αλλά στο κεφάλαιο αυτό χρησιμοποιούμε τις δύο πιο γνωστές συναρτήσεις που είναι η απόσταση Hamming (Hamming distance) και η απόσταση Levenshtein (Levenshtein distance). Η απόσταση Hamming ανάμεσα σε δύο αλφαριθμητικά ίδιου μήκους ορίζεται ως ο αριθμός των θέσεων που έχουν ανεπιτυχείς ταυτίσεις χαρακτήρων ανάμεσα σε δύο αλφαριθμητικά. Με άλλα λόγια, η απόσταση Hamming επιτρέπει μόνο αντικαταστάσεις. Το πρόβλημα της προσεγγιστικής αναζήτησης αλφαριθμητικών με το d να είναι απόσταση Hamming λέγεται αναζήτηση αλφαριθμητικών με k αποτυχίες (mismatches).

Η απόσταση Levenshtein ή απόσταση διόρθωσης (edit distance) ανάμεσα σε δύο αλφαριθμητικά που δεν είναι του ίδιου μήκους, είναι ο ελάχιστος αριθμός εισαγωγών, διαγραφών και αντικαταστάσεων που απαιτούνται για να μετασχηματιστεί το ένα αλφαριθμητικό στο άλλο. Αλγόριθμοι για τον υπολογισμό της απόστασης διόρθωσης ανάμεσα σε ένα ζεύγος αλφαριθμητικών παρουσιάζονται στις εργασίες [163, 98, 158]. Το πρόβλημα της προσεγγιστικής αναζήτησης αλφαριθμητικών με το d να είναι απόσταση Levenshtein ή απόσταση διόρθωσης λέγεται αναζήτηση αλφαριθμητικών με k διαφορές (differences). Τα δύο παραπάνω προβλήματα μαζί λέγονται προσεγγιστική αναζήτηση αλφαριθμητικών.

Οι λύσεις στα δύο παραπάνω προβλήματα διαφέρουν αν ο αλγόριθμος πρέπει να είναι on-line (δηλαδή, το κείμενο δεν είναι γνωστό εκ των προτέρων) ή off-line (δηλαδή, το κείμενο μπορεί να προεπεξεργαστεί). Σε αυτό το κεφάλαιο θα αναφέρουμε τους on-line αλγορίθμους για τα δύο προβλήματα. Υπάρχουν πάρα πολλοί αλγόριθμοι για το πρόβλημα της προσεγγιστικής αναζήτησης αλφαριθμητικών, όπως δείχνουν οι επισκοπήσεις [44, 2, 153, 9, 64, 128, 101]. Γενικά, ένας on-line αλγόριθμος προσεγγιστικής αναζήτησης αλφαριθμητικών αποτελείται από δύο φάσεις: τη φάση της προεπεξεργασίας του προτύπου p και τη φάση της αναζήτησης του προτύπου p μέσα στο κείμενο t. Η φάση της προεπεξεργασίας περιλαμβάνει τη συγκέντρωση πληροφορίας για το πρότυπο που μπορεί να χρησιμοποιηθεί για μια γρήγορη υλοποίηση των πράξεων στην φάση της αναζήτησης ή την κατασκευή ενός πεπερασμένου αυτομάτου που αναγνωρίζει όλα τα αλφαριθμητικά σε μια απόσταση k από το πρότυπο. Η φάση

της αναζήτησης περιλαμβάνει την σάρωση του κειμένου ή την κατασκευή ενός πίνακα προκειμένου να βρεθούν όλες οι προσεγγιστικές εμφανίσεις του προτύπου μέσα στο κείμενο. Γενικά, η φάση της αναζήτησης βασίζεται σε τέσσερις διαφορετικές προσεγγίσεις.

Συνεπώς, για το πρόβλημα αναζήτηση αλφαριθμητικών με k αποτυχίες, οι αλγόριθμοι χωρίζονται σε τέσσερις κατηγορίες:

- Κλασσικοί αλγόριθμοι (classical): αλγόριθμος Brute-Force, αλγόριθμος Landau-Vishkin [74], αλγόριθμος Galil-Giancarlo [43] και αλγόριθμος Tarhio-Ukkonen [157].
- Αλγόριθμοι ντετερμινιστικών πεπερασμένων αυτομάτων (deterministic finite automata): αλγόριθμος Partial-DFA [10, 127]
- Αλγόριθμοι απαρίθμησης (counting): αλγόριθμος Grossi-Luccio [50], αλγόριθμος BY φίλτρο [10], αλγόριθμος EM [38], αλγόριθμος Pevzner-Waterman [137] και αλγόριθμος Baeza-Yates-Perleberg [14].
- Αλγόριθμοι Bit-παραλληλισμού (bit-parallelism): αλγόριθμος Shift-Or [7], αλγόριθμος Dermouche [37] και αλγόριθμος Backward Nondeterministic DAWG Matching [130].

Όμοια, για το πρόβλημα αναζήτηση αλφαριθμητικών με k διαφορές, οι αλγόριθμοι χωρίζονται πάλι σε τέσσερις κατηγορίες:

- Αλγόριθμοι δυναμικού προγραμματισμού (dynamic programming): αλγόριθμος Sellers [145], αλγόριθμος CUTOFF [158], αλγόριθμος Landau-Vishkin [75, 76], αλγόριθμος Galil-Park [45], αλγόριθμος Ukkonen-Wood [161] και αλγόριθμος Chang-Lampe [24].
- Αλγόριθμοι ντετερμινιστικών πεπερασμένων αυτομάτων (deterministic finite automata): αλγόριθμος Ukkonen [158], αλγόριθμος Wu-Manber-Myers [168] και αλγόριθμος Partial-DFA [10, 73, 127].
- Αλγόριθμοι φιλτραρίσματος (filtering): αλγόριθμος Tarhio-Ukkonen [157], αλγόριθμος COUNT [50, 64, 126], αλγόριθμος Maximal Matches [160], αλγόριθμος Chang-Lawler [23], αλγόριθμος Takaoka [156], αλγόριθμος Suntinen-Tarhio [155] και αλγόριθμος Baeza et al με την τεχνική διαμερισμού (exact partitioning) [14].

- Αλγόριθμοι Bit-παραλληλισμού (bit-parallelism): αλγόριθμος Wu-Manber [167], αλγόριθμος Baeza-Yates-Navarro [11, 12, 13], αλγόριθμος Backward Nondeterministic DAWG Matching [130], αλγόριθμος Wright [166] και αλγόριθμος Myers [124, 125].

Είναι σαφές ότι με μια τέτοια ποικιλία διαφορετικών προσεγγίσεων στο ίδιο πρόβλημα είναι δύσκολο να επιλέξουμε έναν κατάλληλο αλγόριθμο για κάθε πρόβλημα της προσεγγιστικής αναζήτησης αλφαριθμητικών. Οι θεωρητικές αναλύσεις που υπάρχουν στη βιβλιογραφία είναι χρήσιμες αλλά είναι σημαντικό η θεωρία να ολοκληρώνεται με τις εκτενείς πειραματικές συγκρίσεις.

Έχουν ήδη αναφερθεί διάφορα πειράματα για το πρόβλημα της αναζήτησης αλφαριθμητικών με k διαφορές [64]. Οι Jokinen et al [64] σύγκριναν τον χρόνο εκτέλεσης επτά αλγορίθμων μόνο για το πρόβλημα αναζήτησης αλφαριθμητικών με k διαφορές. Συγκεριμένα, σύγκριναν δύο αλγορίθμους δυναμικού προγραμματισμού (Sellers [145] και CUTOFF [158]), δύο αλγορίθμους διαγώνιας μετάπτωσης (diagonal transition) (Galil-Park [45] και Ukkonen-Wood [161]) και τρεις αλγορίθμους φιλτραρίσματος (Tarhio-Ukkonen [157], Jokinen et al [64] και Maximal matches [160]). Όμως δεν υπάρχει πειραματική μελέτη για τους αλγορίθμους bit-παραλληλισμού και τους αλγορίθμους για το πρόβλημα αναζήτησης αλφαριθμητικών με k αποτυχίες. Σε αυτό το κεφάλαιο αναφέρουμε εκτενή πειράματα για τους χρόνους εκτέλεσης των πιο γνωστών και πρόσφατων αλγορίθμων για τα προβλήματα με k αποτυχίες και k διαφορές αντίστοιχα.

Το κεφάλαιο αυτό είναι διαρθρωμένο ως εξής: στην επόμενη ενότητα παρουσιάζουμε συνοπτικά τους αλγορίθμους για τα προβλήματα με k αποτυχίες και k διαφορές που χρησιμοποιήσαμε στα πειράματα μας. Στην ενότητα 3.3 περιγράφουμε την πειραματική μεθοδολογία μας περιλαμβάνοντας το υπολογιστικό περιβάλλον, τα δοκιμαστικά δεδομένα και τους τρόπους μέτρησης για την σύγκριση των αλγορίθμων. Στην ενότητα 3.4 παρουσιάζουμε τα πειραματικά μας αποτελέσματα σε μορφή πινάκων και γραφημάτων. Τέλος, στην ενότητα 3.5 δίνουμε έναν πειραματικό χάρτη ο οποίος δείχνει τον ταχύτερο αλγόριθμο αναζήτησης αλφαριθμητικών σύμφωνα με το μήκος του προτύπου, το αριθμό των σφαλμάτων και το μέγεθος του αλφαβήτου.

3.2 Αλγόριθμοι Προσεγγιστικής Αναζήτησης Αλφαριθμητικών

Σε αυτή την ενότητα δίνουμε την περιγραφή των προβλημάτων της αναζήτησης αλφαριθμητικών με k αποτυχίες και της αναζήτησης αλφαριθμητικών με k διαφορές. Όμως, για περισσότερες λεπτομέρειες και την κωδικοποίηση των αλγορίθμων, ο αναγνώστης μπορεί να ανατρέξει στην εργασία μας [101] και τις αρχικές αναφορές. Επίσης, στο παράρτημα παρουσιάζεται ο κώδικας των ακολουθιακών αλγορίθμων σε γλώσσα προγραμματισμού C. Αρχίζουμε να περιγράφουμε πρώτα τους βασικούς αλγορίθμους από κάθε κατηγορία για το πρόβλημα αναζήτησης αλφαριθμητικών με k αποτυχίες. Τέλος, σε όλους τους αλγορίθμους που ακολουθούν υποθέτουμε ότι το πρότυπο και το κείμενο αποθηκεύονται στους πίνακες $p[1..m]$ και $t[1..n]$.

3.2.1 Αναζήτηση Αλφαριθμητικών με k Αποτυχίες

Ορισμός Προβλήματος

Δίνεται ένα αλφάβητο Σ, ένα μικρό πρότυπο αλφαριθμητικό $p = p_1p_2...p_m$ μήκους m, ένα μεγάλο κείμενο $t = t_1t_2...t_n$ μήκους n, σε ένα αλφάβητο Σ μεγέθους $|\Sigma|$, όπου $m, n > 0$ και $m << n$ και ένας ακέραιος μέγιστος αριθμός αποτυχιών $k \geq 0$. Να βρεθούν όλες οι θέσεις j του κειμένου ώστε ανάμεσα στο p και στο t να έχουν k διαφορετικούς χαρακτήρες. Λέμε ότι υπάρχει μια προσεγγιστική εμφάνιση του p στην θέση j του t.

Κλασσική Προσέγγιση

Οι κλασσικοί αλγόριθμοι αναζήτησης αλφαριθμητικών βασίζονται σε συγκρίσεις χαρακτήρων.

Ο αλγόριθμος Brute-Force (για συντομία, BF) που είναι ο απλούστερος, εκτελεί συγκρίσεις χαρακτήρων ανάμεσα σε υποαλφαριθμητικό κειμένου και σε ολόκληρο το πρότυπο από αριστερά προς τα δεξιά και μετράει τον αριθμό θέσεων που αυτά δεν ταυτίζονται. Αν βρεθούν περισσότερες από k, τότε μετατοπίζει την έναρξη του υποαλφαριθμητικού κειμένου ακριβώς μια θέση προς τα δεξιά. Όταν φτάσουμε στο τέλος του προτύπου αναφέρουμε μια προσεγγιστική εμφάνιση. Αυτός ο αλγόριθμος δεν απαιτεί την φάση της προεπεξεργασίας. Τέλος, ο αλγόριθμος BF έχει πολυπλοκότητα χρόνου $O(mn)$

στην χειρότερη περίπτωση.

Ο αλγόριθμος Landau-Vishkin (LV) [74] ήταν ο πρώτος αποτελεσματικός αλγόριθμος για αυτό το πρόβλημα. Η προσέγγιση τους είναι παρόμοια με τον αλγόριθμο Knuth-Morris-Pratt [69], όπου ένας πίνακας προκύπτει από την προεπεξεργασία του προτύπου καθώς το κείμενο εξετάζεται από αριστερά προς τα δεξιά και ήδη γνωστή πληροφορία αξιοποιείται για να μειώσει τον αριθμό συγκρίσεων χαρακτήρων που απαιτούνται. Η φάση της προεπεξεργασίας έχει $O(kmlogm)$ χρόνο και η φάση της αναζήτησης έχει $O(kn)$ χρόνο. Ο επιπλέον χώρος μνήμης που απαιτείται από τον αλγόριθμο LV είναι $O(k(m+n))$. Ενώ ο αλγόριθμος αυτός είναι γρήγορος, ο χώρος που απαιτείται δεν είναι αποδεκτός για πρακτικούς σκοπούς. Στα πειράματα μας συμπεριλαμβάνουμε τη βελτιωμένη έκδοση του αλγορίθμου LV χρησιμοποιώντας ένα παράθυρο μεγέθους $O(m^2)$ για να επεξεργαστεί το κείμενο αντί του $O(mn)$ πίνακα όπως προτείνεται στην αρχική εργασία [74]. Συνεπώς, αυτός ο αλγόριθμος μειώνει το χώρο μνήμης σε $O(km)$ ο οποίος είναι αποδεκτός στην πράξη.

Ο επόμενος αλγόριθμος είναι ο Tarhio-Ukkonen (TU) [157] ο οποίος βασίζεται στον αλγόριθμο Boyer-Moore-Horspool (BMH) για την απλή αναζήτηση αλφαριθμητικών [58]. Η φάση της προεπεξεργασίας του αλγορίθμου TU έχει $O(m + k|\Sigma|)$ χρόνο και $O(k|\Sigma|)$ χώρο. Η φάση της αναζήτησης του αλγορίθμου TU χρειάζεται $O(mn)$ χρόνο στην χειρότερη περίπτωση. Όμως, ο μέσος χρόνος εκτέλεσης είναι $O(kn(\frac{k}{|\Sigma|} + \frac{1}{(m-k)}))$ για τυχαία αλφαριθμητικά.

Προσέγγιση Απαρίθμησης

Αυτή η προσέγγιση δεν χρησιμοποιεί συγκρίσεις χαρακτήρων όπως οι κλασσικοί αλγόριθμοι αλλά χρησιμοποιεί αριθμητικές πράξεις όπως για παράδειγμα, χρησιμοποιεί μετρητές για κάθε θέση του κειμένου.

Σε αυτή την κατηγορία, παρουσιάζουμε μόνο τον αλγόριθμο Baeza-Yates-Perleberg (για συντομία, BYP) [14] ο οποίος είναι πολύ πρακτικός και απλός για το πρόβλημα αναζήτησης αλφαριθμητικών με k αποτυχίες και του οποίου η απόδοση είναι ανεξάρτητη από τις τιμές του k. Αυτός ο αλγόριθμος έχει $O(n)$ χρόνο στη χειρότερη περίπτωση αν όλοι οι χαρακτήρες στο p είναι διαφορετικοί και $O(n+R)$ χρόνο στη χειρότερη περίπτωση αν υπάρχουν όμοιοι χαρακτήρες στο p, όπου R είναι ο συνολικός αριθμός των διατεταγμένων ζεύγων θέσεων στις οποίες p και t ταυτίζονται. Υποθέτοντας τους χαρακτήρες να

είναι ίσων πιθανοτήτων, ο μέσος χρόνος εκτέλεσης είναι $O((1 + \frac{m}{|\Sigma|})n)$, ανεξάρτητα από τον αριθμό των διαφορετικών χαρακτήρων του προτύπου. Τέλος, ο χρόνος εκτέλεσης για την προεπεξεργασία είναι $O(2m + |\Sigma|)$ και οι απαιτήσεις χώρου είναι $O(m + |\Sigma|)$.

Προσέγγιση Bit-Παραλληλισμού

Από αυτή την κατηγορία περιλαμβάνουμε στα πειράματα μας τον αριθμητικό αλγόριθμο Shift-Or (για συντομία, SO) [7]. Αυτός ο αλγόριθμος χειρίζεται τις αποτυχίες μετρώντας k από αυτές με ένα μετρητή μεγέθους $log_2 k$ αλλά δεν χειρίζεται τις αποτυχίες για το πρόβλημα με k διαφορές δηλαδή τις πράξεις διαγραφής και εισαγωγής. Επίσης, όσο μεγαλύτερος είναι ο αριθμός των bits που χρειάζεται για να παραστήσει τις ξεχωριστές καταστάσεις αναζήτησης, τόσο μικρότερο είναι το μήκος των προτύπων. Στην πειραματική μας μελέτη ο αλγόριθμος SO διαχειρίζεται μέχρι το πολύ m =8 χαρακτήρες αν το μήκος της λέξης του υπολογιστή είναι 32 bits και για την αναπαράσταση μιας κατάστασης απαιτούνται 4 bits.

Οι πολυπλοκότητες χρόνου και χώρου των διαφόρων αλγορίθμων για την επίλυση της απόστασης Hamming παρουσιάζονται στον Πίνακα 3.1 για την χειρότερη και μέση περίπτωση.

Πίνακας 3.1: Πολυπλοκότητες χρόνου και χώρου για αναζήτηση αλφαριθμητικών με k αποτυχίες

Αλγόριθμος	Χρόνος Προεπεξεργασίας	Χρόνος Αναζήτησης		Χώρος Μνήμης
		Χειρότερη Περίπτωση	Μέση Περίπτωση	
BF	-	mn	kn	1
LV	$kmlog^m$	kn	kn	km
TU	$m + k\|\Sigma\|$	mn	$kn(\frac{k}{\|\Sigma\|} + \frac{1}{(m-k)})$	$k\|\Sigma\|$
BYP	$2m + \|\Sigma\|$	n	$(1 + \frac{m}{\|\Sigma\|})n$	$m + \|\Sigma\|$
SO	$(\|\Sigma\| + m)log\frac{k}{w}$	$mnlog\frac{k}{w}$	$mnlog\frac{k}{w}$	$(\|\Sigma\| + m)log\frac{k}{w}$

3.2.2 Αναζήτηση Αλφαριθμητικών με k Διαφορές

Ορισμός Προβλήματος

Δίνεται ένα αλφάβητο Σ, ένα μικρό πρότυπο αλφαριθμητικό $p = p_1p_2...p_m$ μήκους m, ένα μεγάλο κείμενο $t = t_1t_2...t_n$ μήκους n, σε ένα αλφάβητο Σ μεγέθους $|\Sigma|$, όπου $m, n > 0$ και $m << n$ και ένας ακέραιος μέγιστος αριθμός διαφορών $k \geq 0$. Να βρεθούν όλες οι θέσεις j του κειμένου ώστε η απόσταση διόρθωσης (δηλαδή ο αριθμός των διαφορών) ανάμεσα στο p και σε κάποιο υποαλφαριθμητικό του t που τελειώνει στο t_j να είναι k. Τότε λέμε ότι υπάρχει μια προσεγγιστική εμφάνιση του p στην θέση j του t.

Προσέγγιση Δυναμικού Προγραμματισμού

Η προσέγγιση του δυναμικού προγραμματισμού είναι μια κλασσική λύση η οποία έχει προταθεί από πολλούς ερευνητές ανεξάρτητα και κυρίως από τους Wagner και Fischer [163] για τον υπολογισμό της απόστασης διόρθωσης ανάμεσα σε δύο αλφαριθμητικά. Οι αποστάσεις ανάμεσα σε όλο και μεγαλύτερα προθέματα των αλφαριθμητικών υπολογίζονται διαδοχικά από τις προηγούμενες τιμές μέχρι να προκύψει το τελικό αποτέλεσμα. Αργότερα, ο Sellers [145] μετέτρεψε την παραπάνω κλασσική λύση σε αλγόριθμο αναζήτησης προκειμένου να βρεθούν όλες οι προσεγγιστικές εμφανίσεις του προτύπου μέσα στο κείμενο. Ο αλγόριθμος αυτός έχει χρόνο εκτέλεσης $O(mn)$ στην χειρότερη και μέση περίπτωση. Υπάρχουν πολλά αποτελέσματα που βελτιώνουν τον αλγόριθμο SEL και εκμεταλλεύονται τις γεωμετρικές ιδιότητες του πίνακα δυναμικού προγραμματισμού (δηλαδή, οι τιμές στα γειτονικά κελιά διαφέρουν κατά 1) [158] προκειμένουν να υπολογίζουμε τις kn αντί τις mn γραμμές. Για παράδειγμα, ο Ukkonen [158] ανέπτυξε ένα αλγόριθμο που λέγεται CUTOFF του οποίου ο μέσος χρόνος εκτέλεσης είναι $O(kn)$, υπολογίζοντας μόνο ένα μέρος του πίνακα δυναμικού προγραμματισμού.

Στην συνέχεια, αναπτύχθηκαν καινούργιοι αλγόριθμοι που βασίζονται πάνω στην προσέγγιση διαγώνιας μετάπτωσης. Η βασική ιδέα των αλγορίθμων διαγώνιας μετάπτωσης είναι το γεγονός ότι οι διαγώνιοι του πίνακα δυναμικού προγραμματισμού είναι συνάρτηση μονοτονικά αύξουσα. Ο αλγόριθμος βασίζεται πάνω στον υπολογισμό των θέσεων όπου οι τιμές που βρίσκονται στους διαγώνιους αυξάνονται σε σταθερό χρόνο. Υπάρχουν τέσσερις αλγόριθμοι που βασίζονται στην προσέγγιση διαγώνιας

μετάπτωσης: Brute-Force, Landau-Vishkin [75, 76], Galil-Park [45] και Ukkonen-Wood [161]. Στα πειράματα μας συμπεριλαμβάνουμε τον αλγόριθμο Galil-Park (για συντομία, GP). Η φάση της προεπεξεργασίας του αλγόριθμου GP απαιτεί $O(m^2)$ χρόνο και χώρο, ενώ η φάση της αναζήτησης είναι $O(kn)$ χρόνο στην χειρότερη ή μέση περίπτωση. Αυτός ο αλγόριθμος χρησιμοποιεί αναφορά τριάδων (reference triple) που παριστάνει ταύτιση υποαλφαριθμητικών του προτύπου και του κειμένου όπως στον αλγόριθμο Landau-Vishkin.

Τέλος, ο αλγόριθμος Chang-Lampe, (CL) [24] είναι μια παραλλαγή του δυναμικού προγραμματισμού και είναι πολύ αποτελεσματικός και πρακτικός αλγόριθμος. Αυτή η προσαρμογή του απλού δυναμικού προγραμματισμού βασίζεται στην προσέγγιση 'τεμαχισμού στήλης ' (column partition) και έχει μέσο χρόνο εκτέλεσης $O(\frac{kn}{\sqrt{|\Sigma|}})$. Ο χρόνος εκτέλεσης για την προεπεξεργασία και οι απαιτήσεις χώρου μνήμης για αυτόν τον αλγόριθμο είναι $O(m|\Sigma|)$.

Προσέγγιση Ντετερμινιστικού Πεπερασμένου Αυτομάτου

Παρόλο που αυτή η προσέγγιση είναι παλιά έχει λάβει λίγη προσοχή. Αυτή η προσέγγιση βασίζεται στην έκφραση του προβλήματος σε όρους ενός αυτόματου. Η βασική ιδέα είναι να μετατρέψει το γενικό αυτόματο σε ένα ντετερμινιστικό αυτόματο που μειώνει τον αριθμό των καταστάσεων και τις απαιτήσεις μνήμης.

Ο Ukkonen σχεδίασε ένα αλγόριθμο ο οποίος πρότεινε την ιδέα ενός ντετερμινιστικού πεπερασμένου αυτόματου (DFA) [158]. Όμως, αυτός ο αλγόριθμος είχε το μειονέκτημα ότι είχε μεγάλο αριθμό καταστάσεων του αυτομάτου. Αυτό είχε σαν αποτέλεσμα μεγάλες απαιτήσεις χρόνου και χώρου γεγονός που περιορίζει την πρακτικότητα του αλγορίθμου.

Αργότερα, οι Wu et al μελέτησαν ξανά το παραπάνω πρόβλημα [168]. Έτσι, αυτοί χρησιμοποίησαν την τεχνική των τεσσάρων Ρώσων [4] και παρουσίασαν έναν αλγόριθμο με πολυπλοκότητα χρόνου $O(\frac{kn}{logn})$ που είναι μια λογαριθμική βελτίωση σε σχέση με τον αλγόριθμο CUTOFF που έχει πολυπλοκότητα χρόνου $O(kn)$. Ο χρόνος εκτέλεσης για την προεπεξεργασία είναι $O(m|\Sigma|)$ και οι απαιτήσεις χώρου είναι $O(n + \frac{m|\Sigma|}{logn})$.

Τέλος, οι [73] και [127] πρότειναν ένα άλλο τρόπο για να μειώσουν τις απαιτήσεις χώρου. Αυτός ο τρόπος είναι μια προσαρμογή της εργασίας [10], η οποία προτάθηκε για την απόσταση Hamming. Η

ιδέα ήταν να κατασκευάσει το αυτόματο σε μορφή lazzy, δηλαδή να κατασκευάσει μόνο τις καταστάσεις και μεταπτώσεις που πραγματικά βρίσκονται στην επεξεργασία του κειμένου. Αυτός ο αλγόριθμος έχει χρόνο εκτέλεσης $O(n + mmin(s, n))$ όπου s είναι ο συνολικός αριθμός των μεταπτώσεων σε ένα πλήρες αυτόματο και απαιτεί $O(min(|\Sigma|, n)min(m, |\Sigma|))$ χώρο στην χειρότερη περίπτωση. Όμως, η πολυπλοκότητα του μέσου χρόνου για αυτό τον αλγόριθμο είναι $O(n + ms(1 - e^{-n/s}))$ όπου s είναι ο συνολικός αριθμός των μεταπτώσεων σε ένα πλήρες αυτόματο. Τέλος, στα πειράματα μας ο παραπάνω αλγόριθμος περιορίζεται σε $m \leq 10$, επειδή για μεγαλύτερα πρότυπα, ο αλγόριθμος απαιτεί μεγάλες ποσότητες μνήμης.

Στα πειράματα μας συμπεριλάβαμε δύο αλγόριθμους από αυτή την κατηγορία, τον αλγόριθμο Wu-Manber-Myers (για συντομία, WMM) και τον αλγόριθμο Partial DFA (PDFA).

Προσέγγιση Φιλτραρίσματος

Η μέθοδος φιλτραρίσματος είναι πολύ νεότερη και πολύ ενεργή. Η μέθοδος αυτή βασίζεται πάνω στην εύρευση γρήγορων αλγορίθμων για να απαλείψει μεγάλες περιοχές κειμένου που δεν μπορούν να ταυτιστούν και εφαρμόζουν έναν άλλο αλγόριθμο στην υπόλοιπη περιοχή κειμένου χρησιμοποιώντας την προσέγγιση απλού δυναμικού προγραμματισμού.

Πρώτα, οι Tarhio-Ukkonen [157] έχουν σχεδιάσει ένα αλγόριθμο προσεγγιστικής αναζήτησης αλφαριθμητικών (για συντομία, TUD) που προσπάθησαν να χρησιμοποιήσουν τις τεχνικές Boyer-Moore-Horspool [20, 58] για να φιλτράρουν το κείμενο. Η φάση της προεπεξεργασίας έχει $O(k+|\Sigma|m)$ χρόνο και ο χώρος που απαιτείται από αυτό τον αλγόριθμο είναι $O(|\Sigma|m)$. Η φάση της αναζήτησης του αλγορίθμου TUD έχει $O(\frac{mn}{k})$ χρόνο στη χειρότερη περίπτωση και $(\frac{|\Sigma|}{|\Sigma|-2k})kn(\frac{k}{|\Sigma|} + 2k^2 + \frac{1}{m})$ χρόνο στη μέση περίπτωση.

Ο Navarro [126] ανέπτυξε ένα αλγόριθμο COUNT των [64] και [50] ο οποίος είναι ένα φίλτρο βασισμένο στην απαρίθμηση θέσεων ταύτισης. Με άλλα λόγια, η κύρια ιδέα είναι να ψάξουμε για υποαλφαριθμητικά του κειμένου όπου η κατανομή των χαρακτήρων διαφέρει από την κατανομή των χαρακτήρων μέσα στο πρότυπο τόσο πολύ όσο στις k διαφορές. Η φάση της προεπεξεργασίας του αλγορίθμου COUNT έχει $O(|\Sigma| + m)$ χρόνο και η φάση της αναζήτησης έχει $O(n)$ χρόνο αν ο αριθμός των εξακριβώσεων είναι αμελητέος. Τέλος, αυτός ο αλγόριθμος χρησιμοποιεί $O(|\Sigma|)$ χώρο.

Οι Wu-Manber [167] πρότειναν ένα απλό φίλτρο το οποίο λέγεται προσέγγιση διαμερισμού προτύπου (pattern partition). Η προσέγγιση αυτή βασίζεται στην ακόλουθη ιδέα: μια εμφάνιση με k διαφορές ενός προτύπου μήκους m περιλαμβάνει τουλάχιστον ένα υποαλφαριθμητικό μήκους r μέσα στο πρότυπο που ταυτίζεται ένα υποαλφαριθμητικό του κειμένου ακριβώς, όπου $r = \lceil \frac{m}{(k+1)} \rceil$. Υπάρχουν πολλοί τρόποι για να χρησιμοποιήσουμε την παραπάνω ιδέα. Ίσως ο απλούστερος τρόπος ο οποίος χρησιμοποιήθηκε στην εργασία [167], είναι να ψάξουμε για τα πρώτα $k+1$ διαδοχικά τμήματα (block) μήκους r του προτύπου p. Αν οποιοδήποτε από τα τμήματα ταυτίζεται ακριβώς, τότε επεκτείνουμε την ταύτιση, ελέγχοντας αν υπάρχουν το πολύ k διαφορές. Η ιδέα αυτή χρησιμοποιήθηκε σε συνδυασμό με την επέκταση του αλγορίθμου SO [7] σε αναζήτηση αλφαριθμητικών με διαφορές. Ο συνδυασμός της προσέγγισης διαμερισμού προτύπου με τον αλγόριθμο SO λέγεται MULTIWM. Ο αλγόριθμος MULTIWM έχει πολυπλοκότητα χρόνου $O(\frac{mn}{w})$. Επίσης, στα πειράματα μας ο παραπάνω αλγόριθμος περιορίζεται σε $m \leq 31$.

Τέλος, οι Baeza et al [14] πρότειναν έναν αλγόριθμο (για συντομία, BYPEP) ο οποίος συνδυάζει την προσέγγιση διαμερισμού προτύπου με τους παραδοσιακούς αλγορίθμους πολλαπλής αναζήτησης αλφαριθμητικών. Ο απλούστερος αλγόριθμος είναι να κατασκευάζουμε μια μηχανή Aho-Corasick [1] (επέκταση του αλγορίθμου KMP [69, 100] για αναζήτηση πολλαπλών προτύπων) για τα $k+1$ τμήματα μήκους r. Για κάθε ταύτιση που βρεθεί επεκτείνουμε την ταύτιση ελέγχοντας αν υπάρχουν το πολύ k διαφορές, χρησιμοποιώντας τον κλασσικό αλγόριθμο δυναμικού προγραμματισμού που ελέγχει την απόσταση διόρθωσης ανάμεσα σε δύο αλφαριθμητικά. Αυτός ο αλγόριθμος με την μηχανή AC έχει $O(n)$ μέσο χρόνο αναζήτησης για $k \leq O(\frac{m}{logm})$ χρησιμοποιώντας $O(m^2)$ χώρο. Επίσης, η φάση της αναζήτησης του προηγούμενου αλγορίθμου μπορεί να βελτιωθεί χρησιμοποιώντας έναν άλλο αλγόριθμο πολλαπλής αναζήτησης αλφαριθμητικών βασισμένο στον αλγόριθμο Boyer-Moore [28].

Προσέγγιση Bit-Παραλληλισμού

Έχουμε δεί αυτή την τεχνική να εφαρμόζεται στο πρόβλημα με k αποτυχίες. Συνεπώς, η προσέγγιση αυτή μπορεί να εφαρμοστεί με τον ίδιο τρόπο στο πρόβλημα με k διαφορές. Υπάρχουν δύο κύριες κατηγορίες: η προσομοίωση του μη ντετερμινιστικού πεπερασμένου αυτομάτου NFA (Non-deterministic Finite Automaton) και η προσομοίωση του πίνακα δυναμικού προγραμματισμού.

Ο αλγόριθμος Wu-Manber [167] (για συντομία, WM) χρησιμοποιεί την προσέγγιση bit - παραλληλισμού για να προσομοιώνει το αυτόματο κατά γραμμές. Ο αλγόριθμος αυτός έχει μια φάση προεπεξεργασίας που απαιτεί $O(m|\Sigma| + k\lceil\frac{m}{w}\rceil)$ χρόνο. Η φάση της αναζήτησης έχει $O(kn\lceil\frac{m}{w}\rceil)$ χρόνο στη χειρότερη και $O(kn)$ χρόνο στη μέση περίπτωση για πρότυπα στην αναζήτηση κειμένου δηλαδή $m \leq w$. Επίσης, ο αλγόριθμος αυτός απαιτεί $O(m|\Sigma|)$ χώρο. Στα πειράματα μας, ο αλγόριθμος αυτός περιορίζεται σε $m \leq 31$.

Οι Baeza et al [11, 12, 13] πρότειναν έναν άλλο αλγόριθμο (για συντομία, BYN) ο οποίος προσομοιώνει το NFA κατά διαγώνια. Η φάση προεπεξεργασίας του αλγόριθμου BYN χρειάζεται $O(|\Sigma| + mmin(m, |\Sigma|))$ χρόνο και απαιτεί $O(|\Sigma|)$ χώρο. Η φάση της αναζήτησης χρειάζεται $O(n)$ χρόνο στην χειρότερη και μέση περίπτωση. Επίσης, στα πειράματα μας ο παραπάνω αλγόριθμος περιορίζεται σε $m \leq 9$ για $w = 32$ bits.

Τέλος, ο Myers [124, 125] σχεδίασε έναν αλγόριθμο (για συντομία, MYE) που βασίζεται στην bit παράλληλη προσομοίωση του πίνακα δυναμικού προγραμματισμού. Η παραλληλοποίηση έχει βέλτιστη επιτάχυνση και η πολυπλοκότητα χρόνου είναι $O(\frac{kn}{w})$ στην μέση περίπτωση και $O(\frac{mn}{w})$ στην χειρότερη περίπτωση. Η φάση της προεπεξεργασίας του αλγόριθμου MYE απαιτεί $O(m|\Sigma|)$ χρόνο και $O(|\Sigma|)$ χώρο. Στην πειραματική μας μελέτη, ο αλγόριθμος αυτός περιορίζεται σε $m \leq 31$.

Οι πολυπλοκότητες χρόνου και χώρου των διαφόρων αλγορίθμων για την επίλυση της απόστασης διόρθωσης παρουσιάζονται στον Πίνακα 3.2 για την χειρότερη και μέση περίπτωση.

Σημειώνουμε ότι οι αλγόριθμοι SEL, CUTOFF, PDFA, COUNT, MULTIWM και BYPEP αναπτύχθηκαν για το πρόβλημα αναζήτησης αλφαριθμητικών με k διαφορές. Όμως, οι αλγόριθμοι αυτοί μπορούν να εφαρμοστούν για το πρόβλημα αναζήτησης αλφαριθμητικών με k αποτυχίες με ελάχιστες τροποποιήσεις. Συνεπώς, αναπτύξαμε τους αλγορίθμους SELM, CUTOFFM, PDFAM, COUNTM, MULTIWMM και BYPEPM για το πρόβλημα με k αποτυχίες και τα συμπεριλάβαμε στην πειραματική μας μελέτη.

Πίνακας 3.2: Πολυπλοκότητες χρόνου και χώρου για αναζήτηση αλφαριθμητικών με k διαφορές

Αλγόριθμος	Χρόνος Προεπεξεργασίας	Χρόνος Αναζήτησης		Χώρος Μνήμης
		Χειρότερη Περίπτωση	Μέση Περίπτωση	
SEL	-	mn	mn	mn
CUTOFF	-	mn	kn	m
GP	m^2	kn	kn	m^2
CL	$m\|\Sigma\|$	mn	$\frac{kn}{\sqrt{\|\Sigma\|}}$	$m\|\Sigma\|$
WMM	$m\|\Sigma\|$	$\frac{mn}{logm}$	$\frac{kn}{logn}$	$n+\frac{m\|\Sigma\|}{logn}$
PDFA	-	$n+nmin(s,n)$	$n+ms(1-e^{n/s})$	$min(\|\Sigma\|,n)min(m,\|\Sigma\|)$
TUD	$(k+\|\Sigma\|)m$	$\frac{mn}{k}$	$(\frac{\|\Sigma\|}{\|\Sigma\|-2k})kn(\frac{k}{\|\Sigma\|}+2k^2+\frac{1}{m})$	$m\|\Sigma\|$
COUNT	$\|\Sigma\|+m$	mn	n	$\|\Sigma\|$
MULTIWM	$\|\Sigma\|+m$	$\frac{mn}{w}$	$\frac{mn}{w}$	
BYPEP	m		$n,k\leq\frac{m}{logn}$	m^2
WM	$m\|\Sigma\|+k\lceil\frac{m}{w}\rceil$	$kn\lceil\frac{m}{w}\rceil$	$kn\lceil\frac{m}{w}\rceil$	$m\|\Sigma\|$
BYN	$\|\Sigma\|+mmin(m,\|\Sigma\|)$	n	n	$\|\Sigma\|$
MYE	$m\|\Sigma\|$	$\frac{mn}{w}$	$\frac{kn}{w}$	$\|\Sigma\|$

3.3 Πειραματική Μεθοδολογία

Σε αυτή την παράγραφο παρουσιάζουμε την πειραματική μεθοδολογία η οποία χρησιμοποιήθηκε στα πειράματα μας προκειμένου να συγκρίνουμε την απόδοση των αλγορίθμων προσεγγιστικής αναζήτησης αλφαριθμητικών. Οι παράμετροι που περιγράφουν την απόδοση των αλγορίθμων είναι οι εξής:

1. Το μέγεθος του κειμένου,
2. Το μήκος του προτύπου,
3. Ο αριθμός των αποτυχιών ή διαφορών και
4. Το μέγεθος του αλφαβήτου.

Όπως γνωρίζουμε κανένας από τους αλγορίθμους δεν είναι βέλτιστος ή καλύτερος για όλες τις τέσσερις περιπτώσεις. Συνεπώς, ο κύριος στόχος της πειραματικής μας μελέτης είναι να συγκρίνουμε την απόδοση των αλγορίθμων σε σχέση με το μήκος του προτύπου (μικρά και μεγάλα πρότυπα) και τον αριθμό των αποτυχιών ή διαφορών (μικρές και μεγάλες τιμές του k) κάτω από διαφορετικά μεγέθη αλφαβήτου (ή τύποι κειμένων) όπως το δυαδικό αλφάβητο, αλφάβητο μεγέθους 8, αγγλικό αλφάβητο και αλφάβητο DNA, τα οποία έχουν διαφορετικά χαρακτηριστικά.

Όσον αφορά το υπολογιστικό περιβάλλον και τα δοκιμαστικά δεδομένα που χρησιμοποιήθηκαν σε αυτή την πειραματική μας μελέτη είναι τα ίδια ακριβώς όπως περιγράφονται στον προηγούμενο κεφάλαιο.

3.3.1 Μετρήσεις της Απόδοσης

Για την σύγκριση των αλγορίθμων προσεγγιστικής αναζήτησης χρησιμοποιήσαμε το χρόνο εκτέλεσης. Ο χρόνος εκτέλεσης είναι ο συνολικός χρόνος που απαιτείται από έναν αλγόριθμο για να αναζητήσει ένα πρότυπο μέσα στο κείμενο, περιλαμβάνοντας το χρόνο της προεπεξεργασίας. Ο χρόνος εκτέλεσης προκύπτει από την κλήση της συνάρτησης C clock() και μετριέται σε δευτερόλεπτα.

Προκειμένου να εξετάσουμε την επίδραση του μήκους του προτύπου και την επίδραση του αριθμού των σφαλμάτων, μετρήσαμε τον χρόνο εκτέλεσης για όλους τους αλγόριθμους που περιγράψαμε στην παράγραφο 3.2, για τις δύο ακόλουθες σειρές πειραμάτων:

- Μετρήσαμε την επίδραση του μήκους του προτύπου μεταβάλλοντας τις τιμές του m=3, 4, 8, 10, 20, 30, 40, 60, 80 και 100 για σταθερή τιμή k=3. Στην περίπτωση του αλφαβήτου DNA χρησιμοποιήσαμε μεγαλύτερα πρότυπα επειδή στις βιολογικές εφαρμογές συναντώνται μόνο μεγάλα πρότυπα. Για αυτό τον λόγο, μετρήσαμε την επίδραση του μήκους του προτύπου μεταβάλοντας τις τιμές του m=10, 20, 30, 40, 50 και 100 για σταθερή τιμή k=3, και

- Μετρήσαμε την επίδραση του αριθμού των σφαλμάτων σε τρεις υπό-σειρές πειραμάτων:

  – μεταβάλλοντας τις τιμές k=1, 2, 4, 6 και 8 για σταθερή τιμή m=8, εκτός του αλφαβήτου DNA.

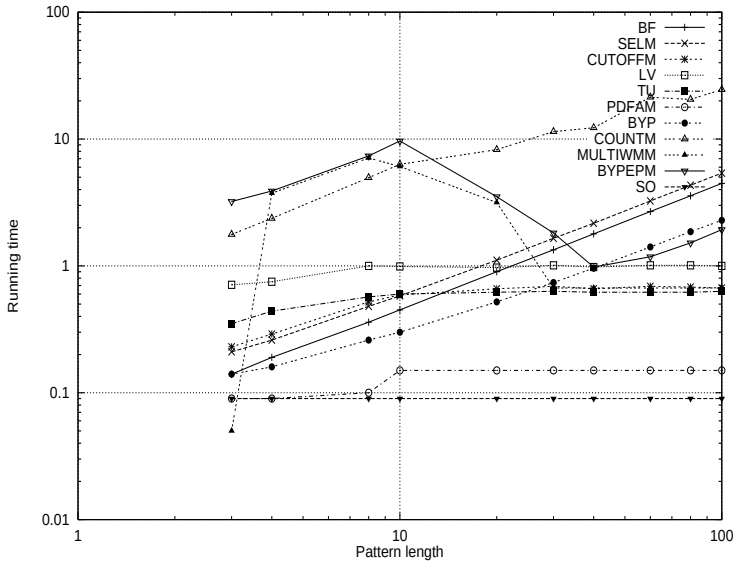
  – μεταβάλλοντας τις τιμές k=1, 8, 10, 15 και 19 για σταθερή τιμή m=20, και

  – μεταβάλλοντας τις τιμές k=1, 6, 13, 25 και 40 για σταθερή τιμή m=50.

Τέλος, τα πειραματικά αποτελέσματα των αλγορίθμων προκύπτουν από τον μέσο όρο δέκα εκτελέσεων με διαφορετικά πρότυπα για κάθε μήκος για να περιορίζουμε τις τυχαίες διακυμάνσεις.

3.4 Πειραματικά Αποτελέσματα

Στις προηγούμενες παραγράφους παρουσιάσαμε σύντομα τους γνωστούς αλγορίθμους προσεγγιστικής αναζήτησης αλφαριθμητικών και την πειραματική μας μεθοδολογία. Σε αυτή την παράγραφο, παρουσιάζουμε τα πειραματικά αποτελέσματα για τους αλγόριθμους αναζήτησης αλφαριθμητικών με k αποτυχίες και τους αλγόριθμους αναζήτησης αλφαριθμητικών με k διαφορές. Η απόδοση των αλγορίθμων για τα δύο παραπάνω προβλήματα προκύπτει από τον μέσο όρο δέκα επαναλήψεων πάνω σε τέσσερις τύπους κειμένων. Συγκεριμένα, η απόδοση του κάθε αλγορίθμου απεικονίζεται σε συνάρτηση με το μήκος του προτύπου και τον αριθμό των σφαλμάτων k για κάθε τύπο κειμένου.

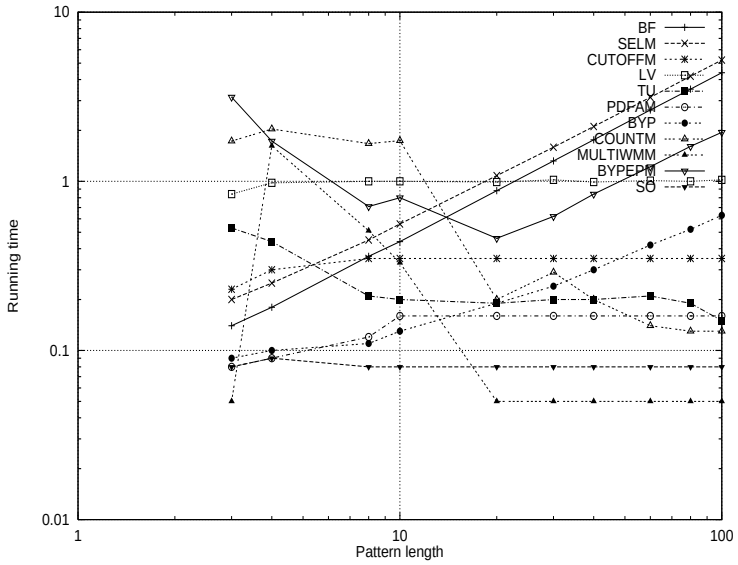


Σχήμα 3.1: $|\Sigma| = 2$ και $k = 3$

3.4.1 Αποτελέσματα για Αλγορίθμους Αναζήτησης Αλφαριθμητικών με k Αποτυχίες

Απόδοση σε σχέση με το Μήκος του Προτύπου

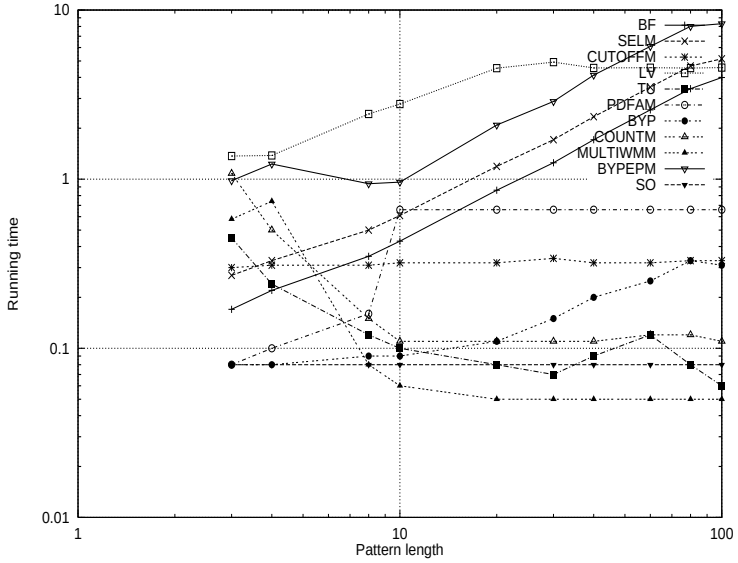
Εδώ αναφέρουμε τα πειραματικά αποτελέσματα για σταθερή τιμή k=3 και μεταβλητό το μήκος του προτύπου. Στα Σχήματα 3.1- 3.4 δείχνουν τον χρόνο εκτέλεσης για το δυαδικό αλφάβητο, αλφάβητο μεγέθους 8, αγγλικό αλφάβητο και αλφάβητο DNA αντίστοιχα. Από τα παραπάνω αποτελέσματα παρατηρούμε ότι υπάρχει μια γενική συμφωνία ανάμεσα στα πειραματικά και θεωρητικά αποτελέσματα των αλγορίθμων στις πιο πολλές περιπτώσεις. Όμως, τα πειραματικά αποτελέσματα των αλγορίθμων φιλτραρίσματος όπως COUNTM, MULTIWMM και BYPEPM δεν ακολουθούν την θεωρητική πολυπλοκότητα χρόνου. Πρέπει να σημειώσουμε εδώ ότι τα πειραματικά αποτελέσματα του αλγορίθμου COUNTM συμφωνούν με τους θεωρητικούς υπολογισμούς μόνο για μεγάλα αλφάβητα όπως το αγγλικό αλφάβητο και μεγάλα πρότυπα.



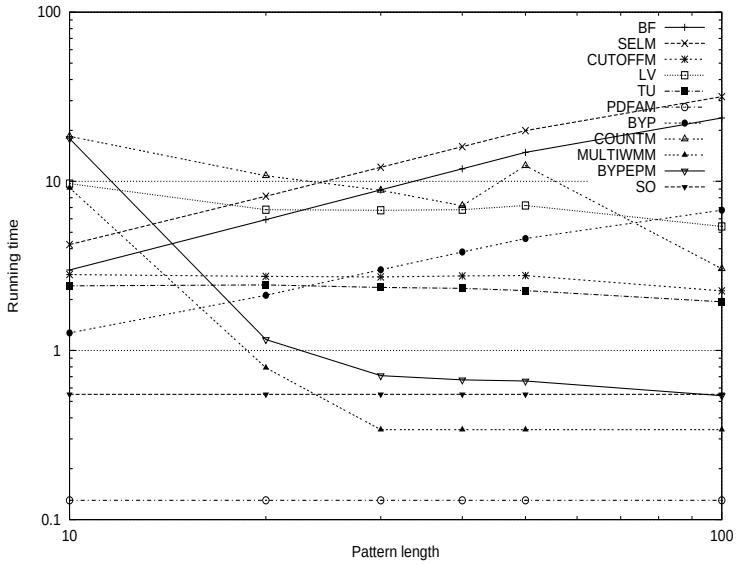
Σχήμα 3.2: $|\Sigma|$=8 και $k = 3$

Απόδοση σε σχέση με τις Τιμές του k

Στα Σχήματα 3.5- 3.8 δείχνουν τον χρόνο εκτέλεσης για το δυαδικό αλφάβητο, αλφάβητο μεγέθους 8, αγγλικό αλφάβητο και αλφάβητο DNA αντίστοιχα, με πρότυπα μήκους ($m = 8, 20, 50$) και διά-



Σχήμα 3.3: $|\Sigma|$=70 και $k = 3$



Σχήμα 3.4: $|\Sigma|$=4 και $k = 3$

φορες τιμές του k. Παρατηρούμε ότι η θεωρητική πολυπλοκότητα χρόνου της ομάδας αλγορίθμων φιλτραρίσματος δεν επιβεβαιώνεται στην πράξη για όλα τα μεγέθη αλφαβήτου και τις διάφορες τιμές του k.

3.4.2 Αποτελέσματα για Αλγορίθμους Αναζήτησης Αλφαριθμητικών με k Διαφορές

Απόδοση σε σχέση με το Μήκος του Προτύπου

Στα Σχήματα 3.9- 3.12 φαίνεται ο χρόνος εκτέλεσης για το δυαδικό αλφάβητο, αλφάβητο μεγέθους 8, αγγλικό αλφάβητο και αλφάβητο DNA αντίστοιχα. Παρατηρούμε ότι η ομάδα αλγορίθμων φιλτραρίσματος όπως οι αλγόριθμοι COUNT, MULTIWM και BYPEP σε πολλές περιπτώσεις δεν επιβεβαιώνει την θεωρητική πολυπλοκότητα χρόνου. Επίσης, στα πειράματα μας ο αλγόριθμος CL που βασίζεται στην προσέγγιση δυναμικού προγραμματισμού δεν συμφωνεί απόλυτα με τα θεωρητικά αποτελέσματα για όλα τα μεγέθη του αλφαβήτου. Τέλος, η υπολογιστική συμπεριφορά των υπολοίπων αλγορίθμων γενικά συμφωνεί με την θεωρία στις περισσότερες περιπτώσεις.

Απόδοση σε σχέση με τις Τιμές του k

Στα Σχήματα 3.13- 3.16 δείχνουν τον χρόνο εκτέλεσης για το δυαδικό αλφάβητο, αλφάβητο μεγέθους 8, αγγλικό αλφάβητο και αλφάβητο DNA αντίστοιχα, με πρότυπα μήκους ($m = 8, 20, 50$) και διάφορες τιμές του k. Παρατηρούμε ότι η θεωρητική ανάλυση των αλγορίθμων φιλτραρίσματος όπως οι αλγόριθμοι COUNT, MULTIWM και BYPEP δεν είναι σύμφωνη στην πράξη στις περισσότερες περιπτώσεις. Επίσης, οι αλγόριθμοι CL και MYE δεν παρουσιάζουν την αναμενόμενη συμπεριφορά. Τέλος, οι υπόλοιποι αλγόριθμοι παρουσιάζουν απόδοση σύμφωνα με τα θεωρητικά αποτελέσματα.

Γενικά Σχόλια

Από τα πειραματικά αποτελέσματα μπορούμε να συμπεράνουμε ότι οι αλγόριθμοι BF, SELM και SEL σε όλες τις περιπτώσεις είναι γραμμικοί στον χρόνο εκτέλεσης. Οι αλγόριθμοι αυτοί παρέχουν σχετικά καλούς χρόνους εκτέλεσης παρά την απλότητα τους. Συγκεριμένα, ο αλγόριθμος BF είναι η καλύτερη προσέγγιση με εξαίρεση τους αλγορίθμους SO και PDFAM όταν το μήκος του προτύπου είναι πολύ μικρό ή το k είναι κοντά στο m. Επίσης, ο αλγόριθμος SEL είναι ο ταχύτερος όταν το m είναι μικρό ή όταν το k είναι κοντά στο m. Αυτή η παρατήρηση ισχύει για όλα τα αλφάβητα. Αυτό που πρέπει να σημειωθεί είναι ότι οι αλγόριθμοι BF, SELM και SEL δεν απαιτούν μνήμη και πολύπλοκη κωδικοποίηση. Όμως, οι αλγόριθμοι αυτοί γίνονται αργοί για μεγάλα πρότυπα και για μεγάλες τιμές του k.

Παρόλο που οι αλγόριθμοι CUTOFFM, CUTOFF και GP είναι τετραγωνικοί στην θεωρητική χειρότερη περίπτωση, στα πειράματα μας επιβεβαιώνουν τον $O(kn)$ μέσο χρόνο εκτέλεσης. Επίσης, σημειώνουμε ότι στα πειράματα μας δεν επιβεβαιώνουν την πρόταση του Chang και Lampe [24] σύμφωνα με την οποία ο αλγόριθμος CL είναι πάντα ο ταχύτερος από τον αλγόριθμο CUTOFF. Ειδικά, για μεγάλα πρότυπα και μικρά αλφάβητα ο αλγόριθμος CL είναι πολύ πιο αργός από τον αλγόριθμο CUTOFF. Όμως, στα πειράματα μας έδειξαν ότι ο αλγόριθμος CL είναι καλύτερος από τον αλγόριθμο CUTOFF για μεγάλες τιμές του k. Τέλος, πρέπει να σημειώσουμε ότι οι αλγόριθμοι CUTOFFM και CUTOFF είναι οι καλύτερες προσεγγίσεις για όλα τα αλφάβητα όταν το k είναι σχετικά μεγάλο.

Έχουμε δείξει πειραματικά ότι οι αλγόριθμοι PDFAM και PDFA εκτελούνται καλύτερα από τους άλλους αλγορίθμους για μικρές τιμές του k και μικρά πρότυπα. Επίσης, ο αλγόριθμος WMM πραγματοποιεί υποτετραγωνικό χρόνο εκτέλεσης χειρότερης περίπτωσης και πολύ καλό μέσο χρόνο εκτέλεσης

για μεγάλες τιμές του k και μεγάλα πρότυπα. Ο αλγόριθμος αυτός έχει το πλεονέκτημα ότι μπορεί να δουλέψει για πρότυπα που περιέχουν κλάση χαρακτήρων, άρνηση ενός χαρακτήρα ή μιας κλάσης χαρακτήρων και αδιάφορους χαρακτήρες. Όμως, το κύριο μειονέκτημα των παραπάνω μεθόδων είναι ότι απαιτούν μεγάλες ποσότητες μνήμης.

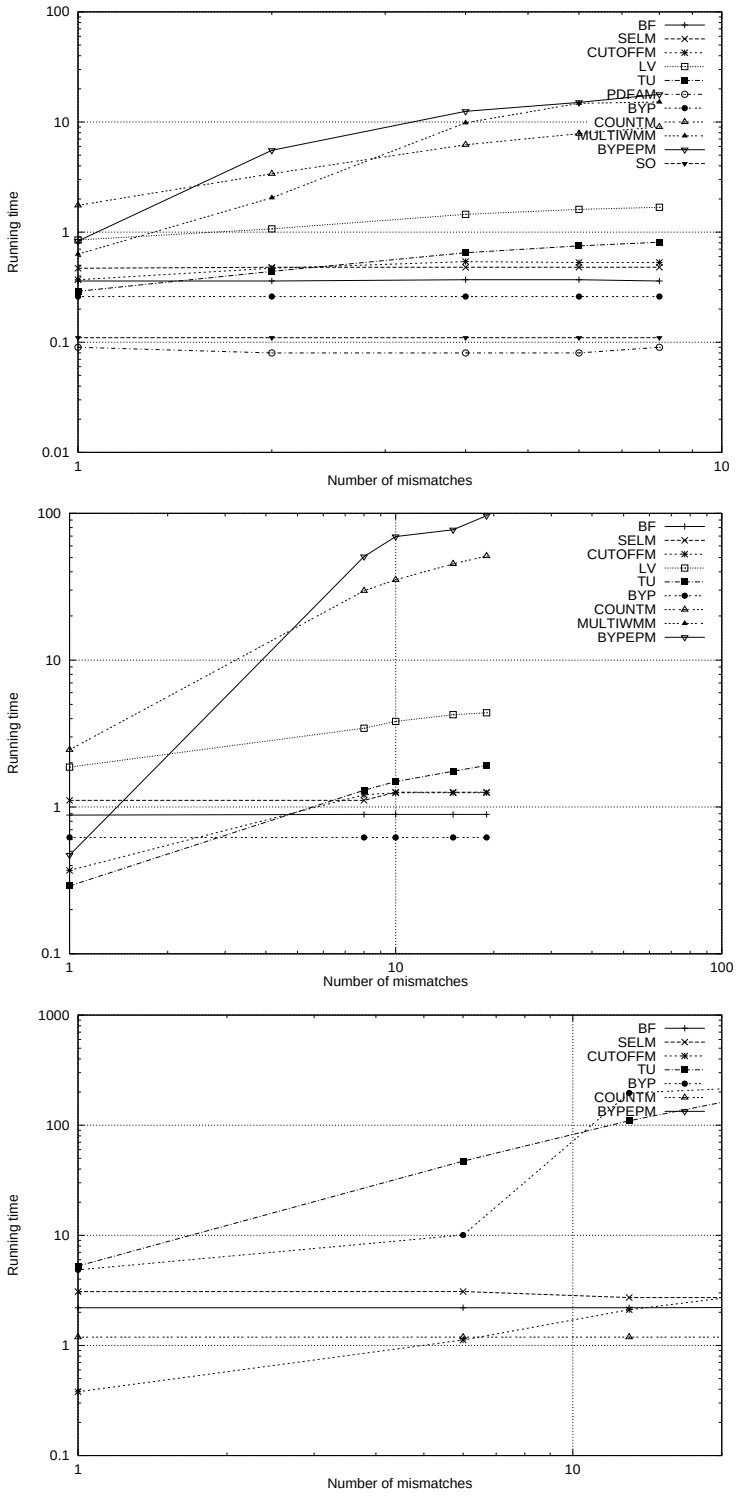
Η θεωρητική ανάλυση των αλγορίθμων φιλτραρίσματος δεν ισχύει στην πράξη στις περισσότερες περιπτώσεις. Οι αλγόριθμοι COUNTM, COUNT, MULTIWMM, MULTIWM, BYPEPM και BYPEP επιτρέπουν τη σάρωση του κειμένου σε γραμμικό χρόνο. Πρέπει να παρατηρήσουμε ότι οι αλγόριθμοι MULTIWMM, MULTIWM, BYPEPM και BYPEP έχουν ελαφρώς καλύτερη απόδοση από τους αλγορίθμους COUNTM και COUNT μόνο για μικρές τιμές του k. Από την άλλη μεριά, για μεγάλες τιμές του k οι αλγόριθμοι MULTIWMM, MULTIWM, BYPEPM και BYPEP έχουν χαμηλότερο χρόνο εκτέλεσης από τους υπόλοιπους αλγορίθμους. Οι αλγόριθμοι COUNTM και COUNT είναι απλοί και ταχύτεροι στην πράξη για μικρές τιμές του k. Επίσης, οι αλγόριθμοι αυτοί είναι ταχύτεροι για μεγάλα πρότυπα και αλφάβητα (δηλαδή, αλφάβητο μεγέθους 8 και αγγλικό αλφάβητο). Όμως, οι αλγόριθμοι COUNTM και COUNT δεν είναι οι ταχύτεροι από τα καλύτερα υπογραμμικά φίλτρα επειδή οι αλγόριθμοι αυτοί σαρώνουν όλους τους χαρακτήρες του κειμένου. Σύμφωνα με την πειραματική μας μελέτη, παρατηρούμε ότι ο αλγόριθμος BYP πραγματοποιεί $O(n)$ χρόνο στην χειρότερη περίπτωση ανεξάρτητα από τις τιμές του k και χωρίς περιορισμούς στο μήκος του προτύπου. Αυτό μπορεί εύκολα να προσαρμοστεί για να βρούμε την 'καλύτερη ταύτιση' (best match) (μικρότερο k). Αυτό είναι επιθυμητό σε πολλές περιπτώσεις όπου το όριο του k δεν είναι γνωστό εκ των προτέρων. Επίσης, ο αλγόριθμος BYP έχει χρησιμοποιηθεί για το πρόβλημα της δισδιάστατης αναζήτησης κειμένου (two dimensional text searching) [15]. Τέλος, ο χρόνος εκτέλεσης των αλγορίθμων TU και TUD μειώνεται καθώς το μήκος του προτύπου και το μέγεθος του αλφαβήτου αυξάνεται. Αυτό το γεγονός υποστηρίζει την θεωρητική απόδειξη ότι οι αλγόριθμοι TU και TUD είναι υπογραμμικοί στο μέσο χρόνο εκτέλεσης. Συνεπώς, οι αλγόριθμοι αυτοί εκτελούνται καλά κατά μέσο όρο για μικρές τιμές του k και μεγάλα πρότυπα και αλφάβητα.

Έχουμε δείξει πειραματικά ότι όλοι οι αλγόριθμοι του bit-παραλληλισμού με εξαίρεση τον αλγόριθμο WM είναι οι ταχύτεροι για τυπική αναζήτηση κειμένου. Συγκεριμένα, οι αλγόριθμοι SO, BYN και MYE για μικρά πρότυπα σαρώνουν το κείμενο σε γραμμικό χρόνο, ανεξάρτητα από την τιμή του k. Αυτό που μπορούμε επίσης να δούμε είναι ότι οι αλγόριθμοι αυτοί είναι ταχύτεροι για μικρά πρότυπα

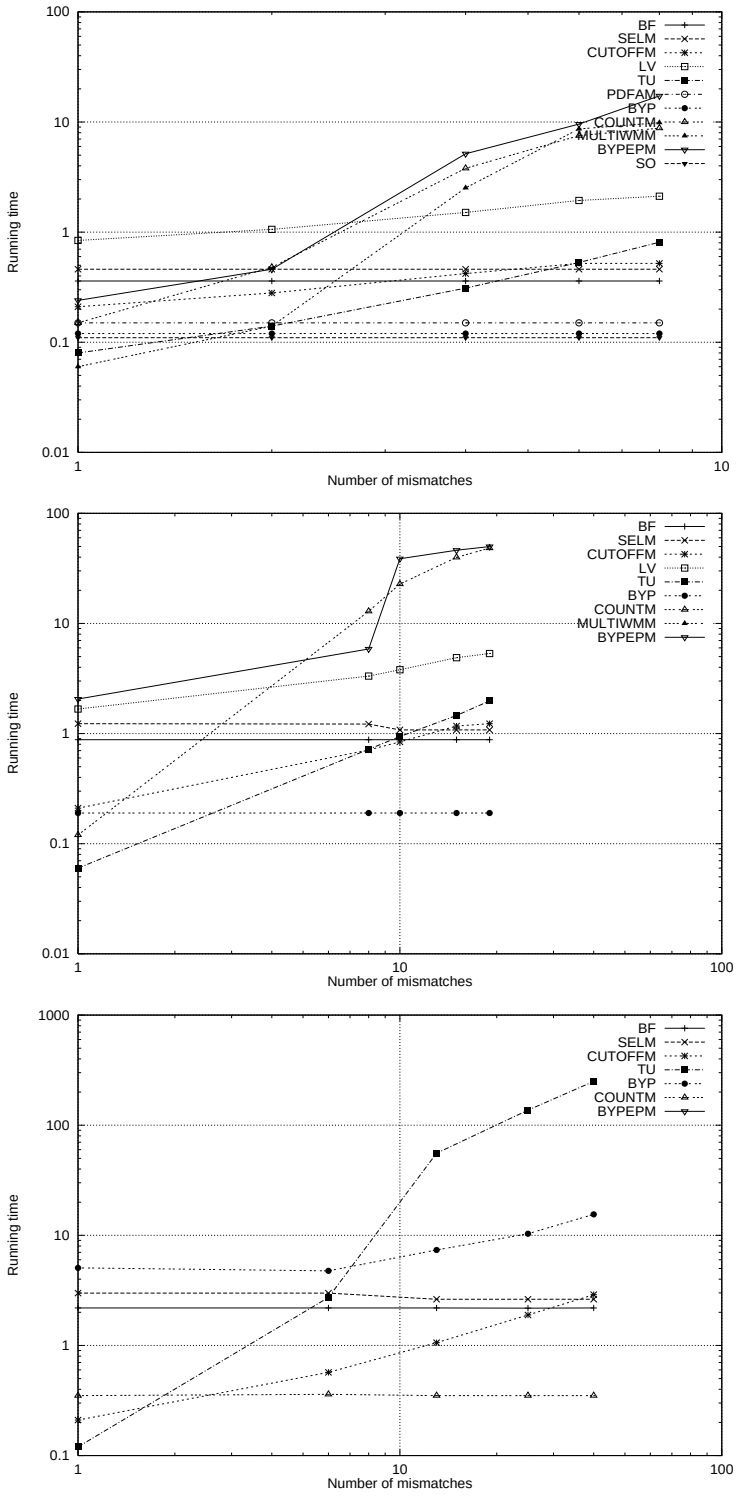
και μεσαίες τιμές του k για όλα τα αλφάβητα. Ο αλγόριθμος WM είναι επίσης γραμμικός σύμφωνα με τα πειράματα μας αλλά είναι το λιγότερο αποτελεσματικό σχήμα σήμερα. Οι αλγόριθμοι του bit-παραλληλισμού είναι απλοί να υλοποιηθούν και είναι επίσης πολύ ευέλικτοι. Δηλαδή, οι αλγόριθμοι του bit-παραλληλισμού μπορούν να εφαρμοστούν σε περιπτώσεις όπου το πρότυπο ίσως περιέχει μια κλάση χαρακτήρων και αδιάφορους χαρακτήρες. Τέλος, δεν έχουμε μελετήσει άλλες περιπτώσεις όπως πολύ μεγάλα πρότυπα επειδή όλοι οι αλγόριθμοι bit-παραλληλισμού δεν εκτελούνται τόσο καλά όσο οι άλλοι αλγόριθμοι.

3.5 Πειραματικός Χάρτης

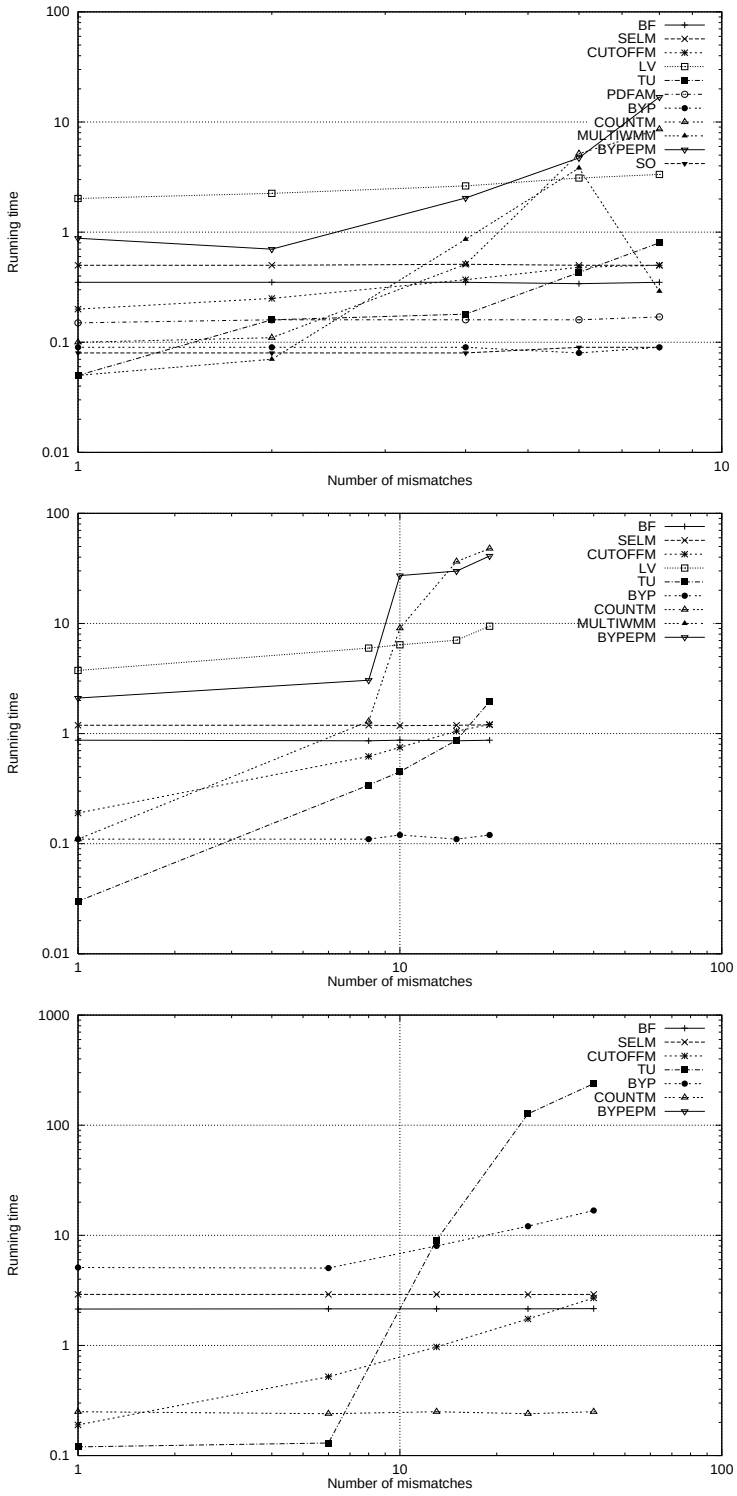
Παρουσιάζουμε ένα χάρτη των πιο αποτελεσματικών αλγορίθμων για τα προβλήματα προσεγγιστικής αναζήτησης με k αποτυχίες και k διαφορές. Στο Σχήμα 3.17 δείχνει περιοχές όπου κάθε αλγόριθμος υπερέχει για διαφορετικές τιμές του m και k στις περιπτώσεις αγγλικού και δυαδικού αλφαβήτου για το πρόβλημα με k αποτυχίες. Ομοίως, στο Σχήμα 3.18 παρουσιάζει περιοχές υπεροχής για το πρόβλημα με k διαφορές.



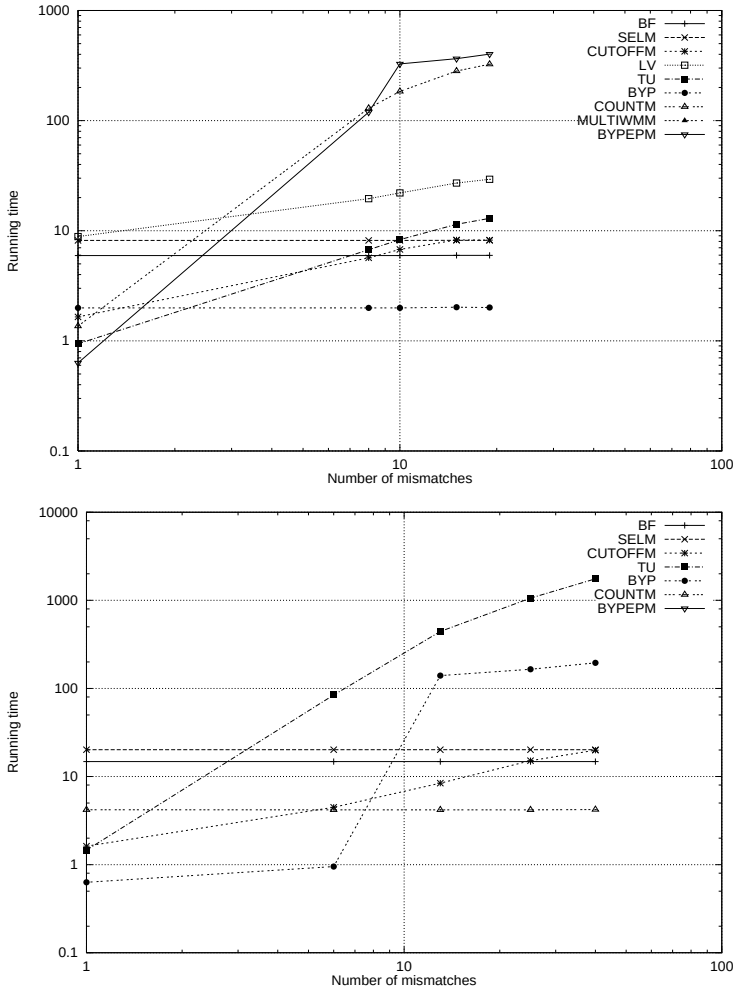
Σχήμα 3.5: $|\Sigma|$=2 και $m = 8, 20, 50$



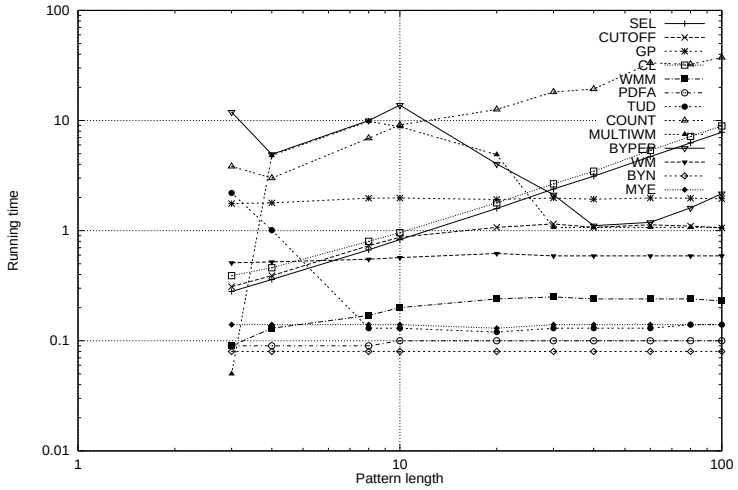
Σχήμα 3.6: $|\Sigma|$=8 και $m = 8, 20, 50$



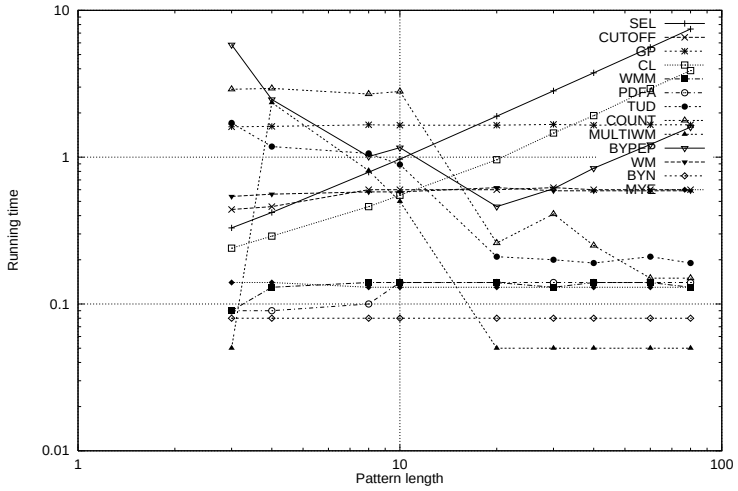
Σχήμα 3.7: $|\Sigma|$=70 και $m = 8, 20, 50$



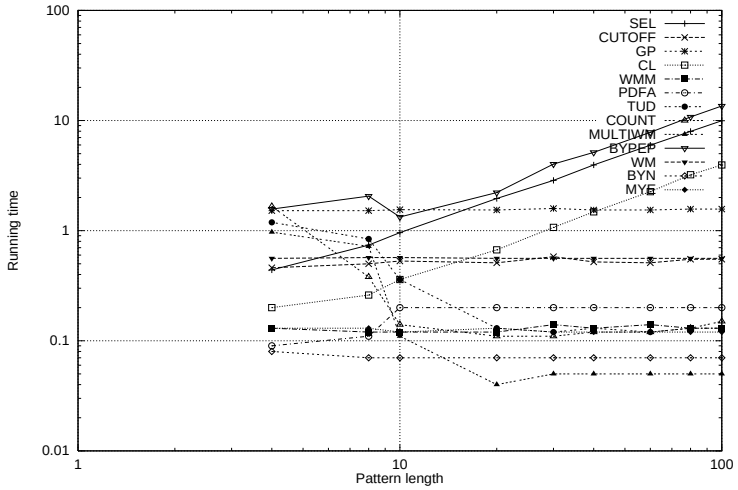
Σχήμα 3.8: $|\Sigma|$=4 και $m = 20, 50$



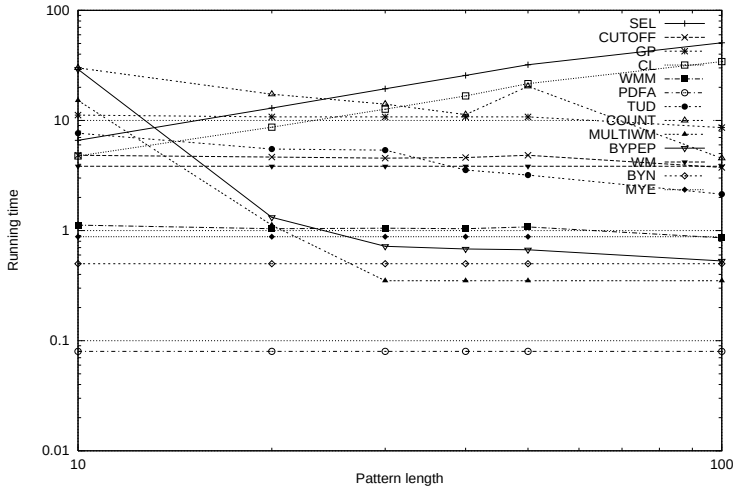
Σχήμα 3.9: $|\Sigma|$=2 και $k = 3$



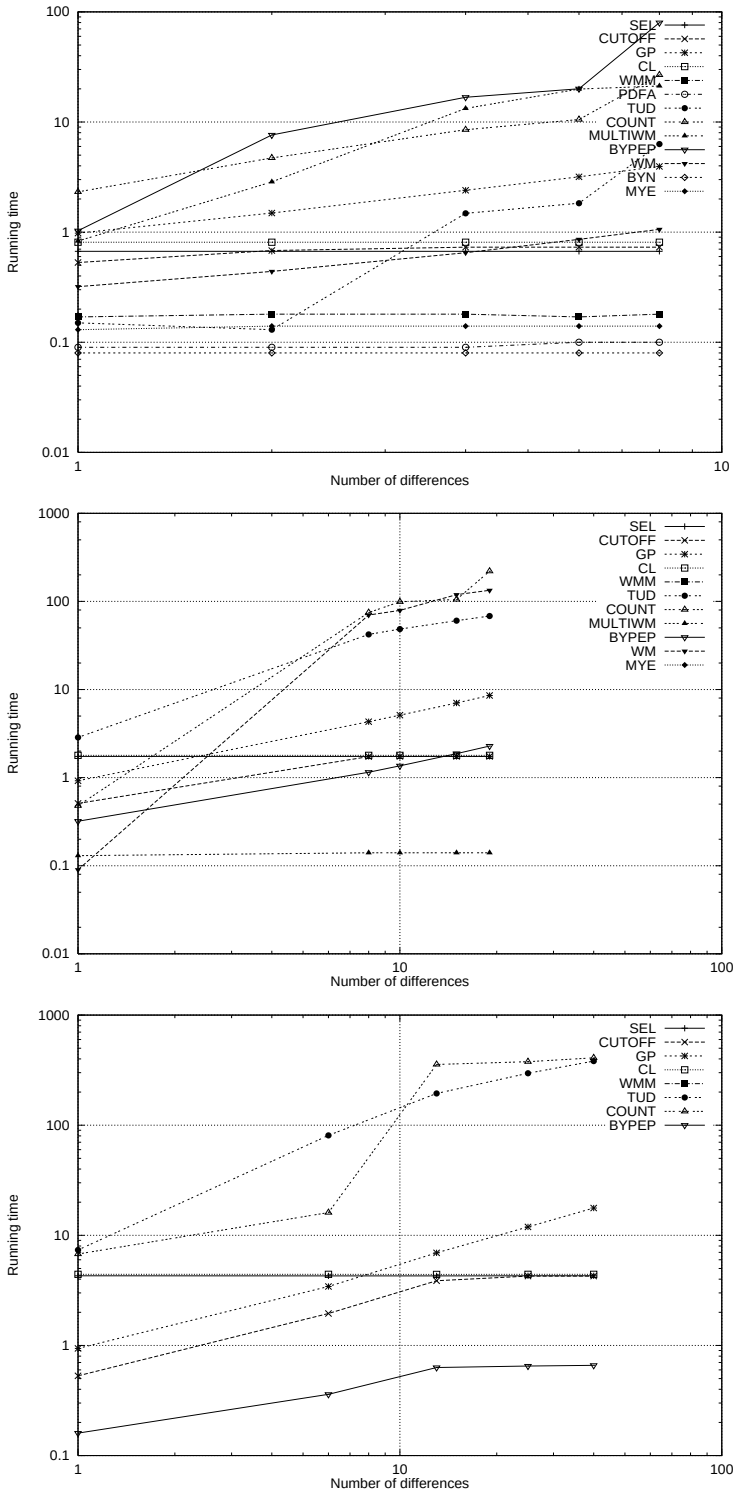
Σχήμα 3.10: $|\Sigma|$=8 και $k = 3$



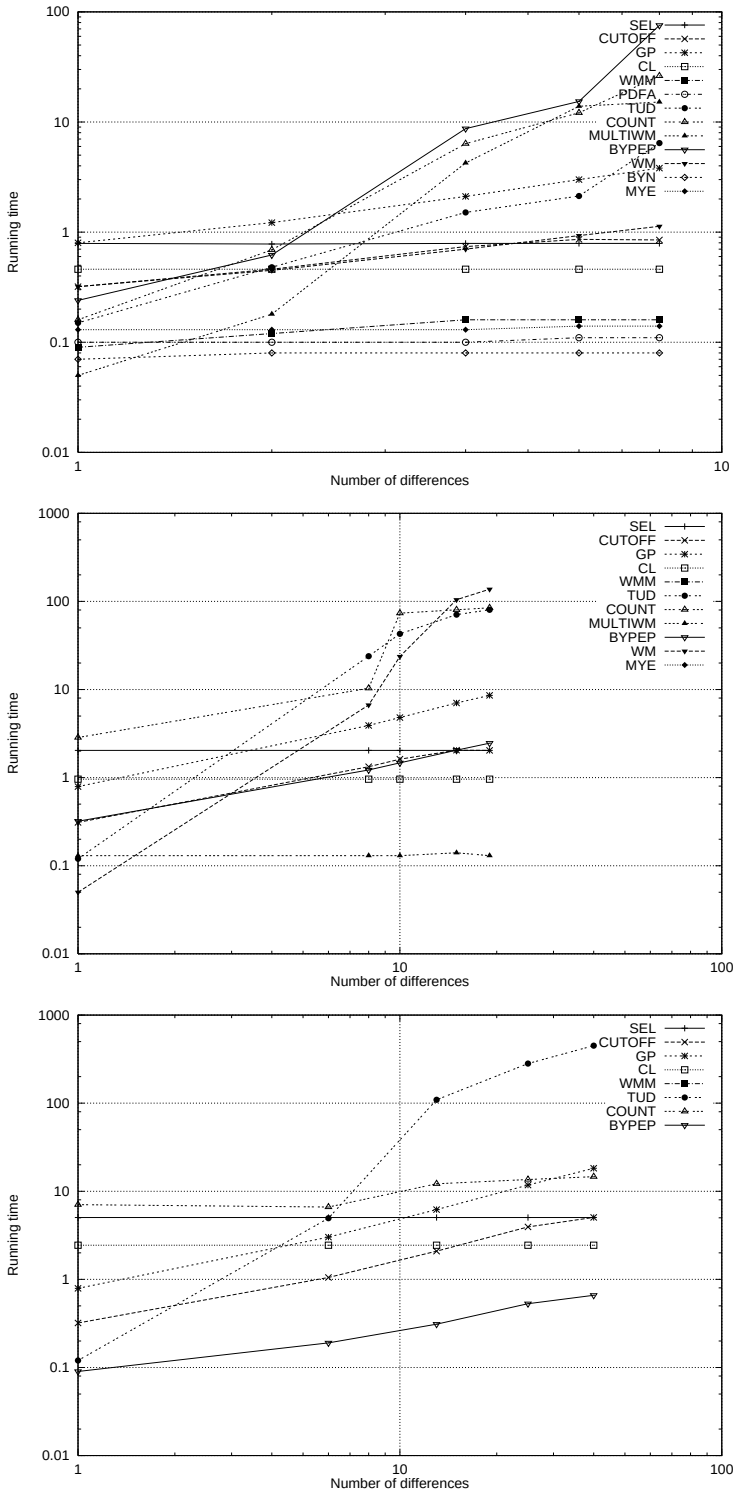
Σχήμα 3.11: $|\Sigma|$=70 και $k = 3$



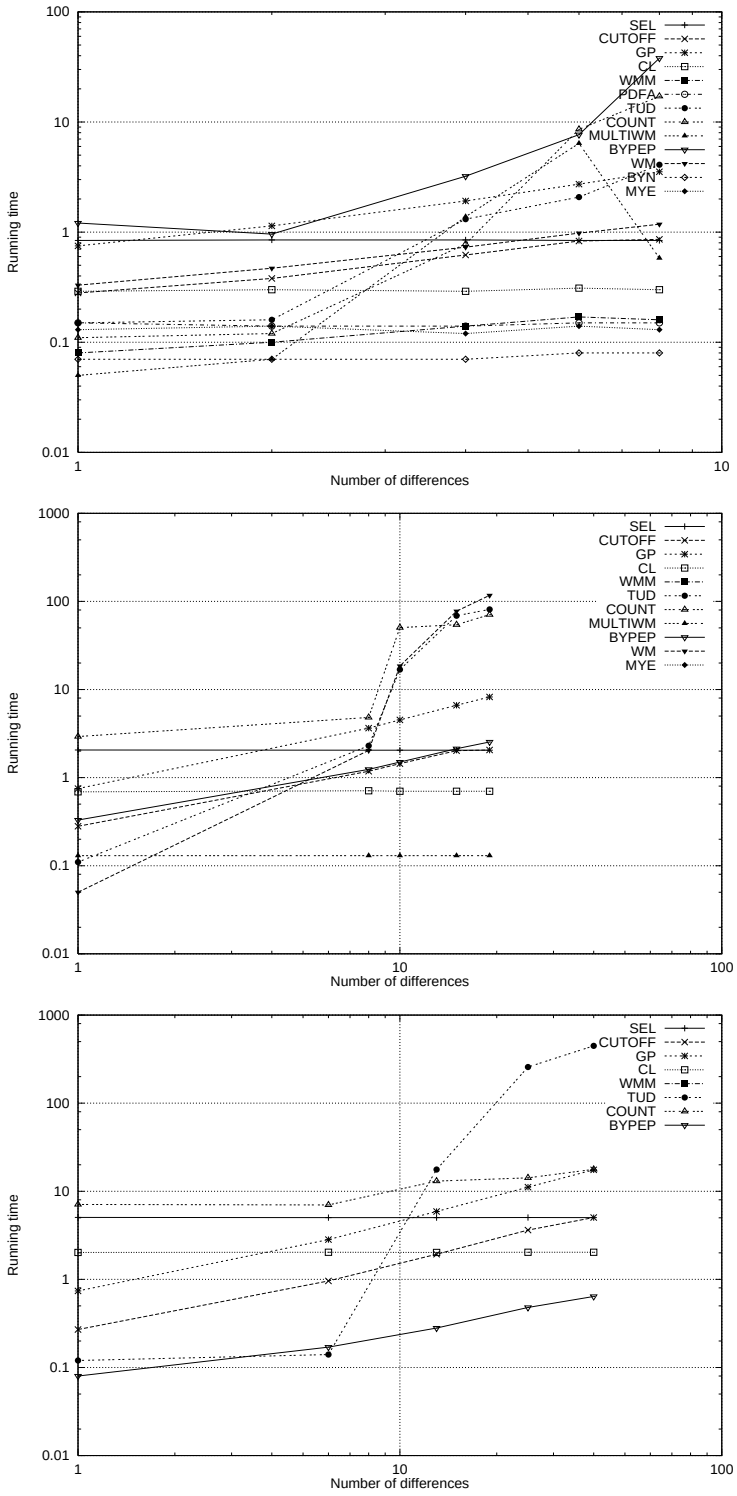
Σχήμα 3.12: $|\Sigma|$=4 και $k = 3$



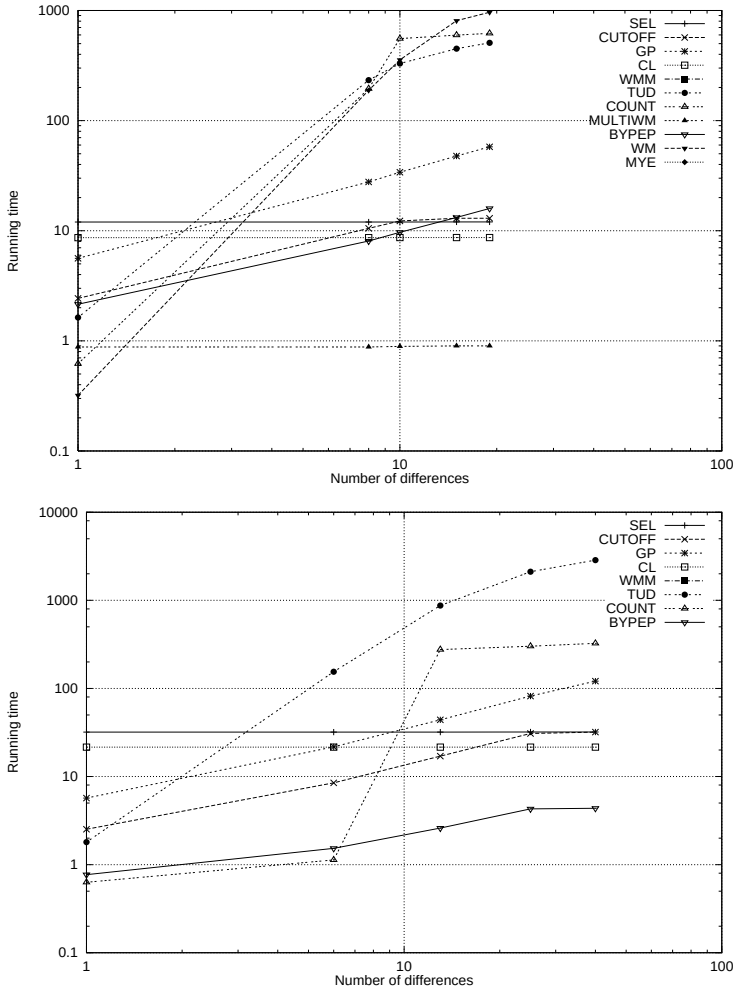
Σχήμα 3.13: $|\Sigma|$=2 και $m = 8, 20, 50$



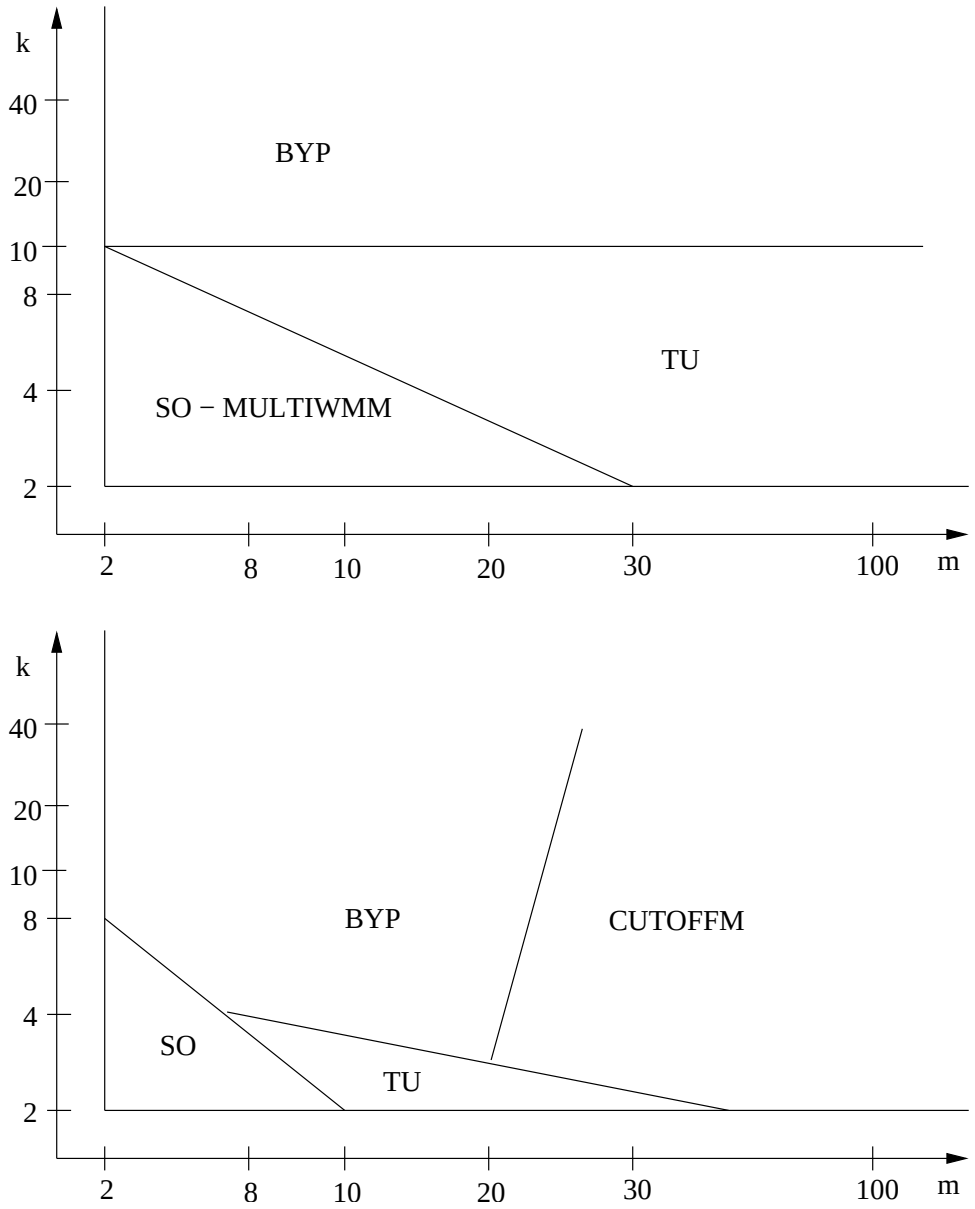
Σχήμα 3.14: $|\Sigma|$=8 και $m = 8, 20, 50$



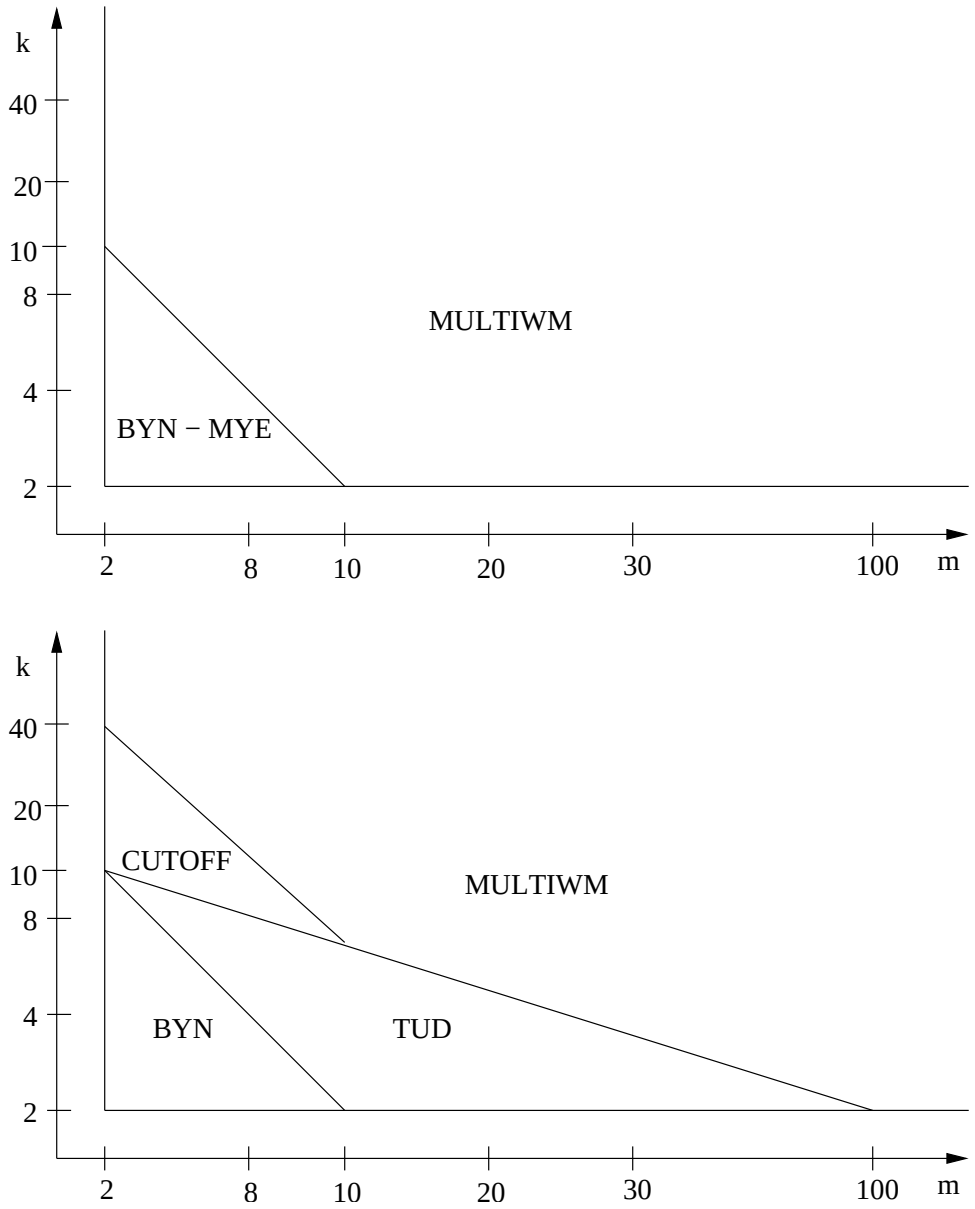
Σχήμα 3.15: $|\Sigma|$=70 και $m = 8, 20, 50$



Σχήμα 3.16: $|\Sigma|$=4 και $m = 20, 50$



Σχήμα 3.17: Περιοχές όπου κάθε αλγόριθμος για το πρόβλημα με k αποτυχίες είναι αποτελεσματικός. Πάνω είναι για το αγγλικό αλφάβητο και κάτω είναι για το δυαδικό αλφάβητο.



Σχήμα 3.18: Περιοχές όπου κάθε αλγόριθμος για το πρόβλημα με k διαφορές είναι αποτελεσματικός. Πάνω είναι για το αγγλικό αλφάβητο και κάτω είναι για το δυαδικό αλφάβητο.

Κεφάλαιο 4

Στοιχεία Παράλληλης και Κατανεμημένης Επεξεργασίας

4.1 Εισαγωγή

Οι υπολογιστικές εφαρμογές συχνά χρειάζονται περισσότερη υπολογιστική ισχύ από αυτή που προσφέρει ένας ακολουθιακός υπολογιστής. Ένας τρόπος για να ξεπεράσουμε αυτό τον περιορισμό είναι να βελτιώσουμε την ταχύτητα των επεξεργαστών έτσι ώστε να προσφέρουν την υπολογιστική ισχύ που χρειάζονται οι υπολογιστικά απαιτητικές εφαρμογές. Παρόλο που αυτό είναι λογικό μέχρι ένα σημείο, οι μελλοντικές βελτιώσεις περιορίζονται από ορισμένους παράγοντες όπως: η ταχύτητα του φωτός, οι νόμοι της θερμοδυναμικής και το υψηλό κόστος κατασκευής επεξεργαστών. Μια βιώσιμη και αποτελεσματική λύση είναι να συνδέσουμε πολλούς επεξεργαστές μαζί και να συναρμονίσουμε τις υπολογιστικές τους προσπάθειες. Αυτό έχει σαν αποτέλεσμα την δημιουργία συστημάτων που είναι γνωστά ως παράλληλοι και κατανεμημένοι υπολογιστές (parallel and distributed computers) και επιτρέπουν τον διαμοιρασμό μιας υπολογιστικής εργασίας μεταξύ των πολλαπλών επεξεργαστών. Ο Pfister [138] δείχνει ότι υπάρχουν τρεις τρόποι για να βελτιώσουμε την απόδοση:

- η σκληρή εργασία,
- η έξυπνη εργασία, και

- η λήψη βοήθειας.

Σε όρους τεχνολογίας υπολογιστών, η σκληρή εργασία αντιστοιχεί στο να χρησιμοποιείται ταχύτερο υλικό (όπως επεξεργαστές, μνήμη, ή περιφερειακές συσκευές υψηλής απόδοσης). Η έξυπνη εργασία αντιστοιχεί στο να εκτελούνται οι διάφορες εργασίες πιο αποτελεσματικά και σχετίζεται με τους αλγόριθμους και τεχνικές που χρησιμοποιούνται για να εκτελέσουν τις υπολογιστικές εργασίες. Τέλος, η λήψη βοήθειας αναφέρεται στο να συνεργάζονται οι υπολογιστές μεταξύ τους ώστε να λύσουν ένα υπολογιστικό πρόβλημα. Η συνεργασία μπορεί να είναι περισσότερη ή λιγότερη στενή, υποβοηθούμενη από ειδικό υλικό ή μόνο με τη χρήση κάποιου λογισμικού.

Το κεφάλαιο αυτό οργανώνεται ως εξής: στην επόμενη ενότητα παρουσιάζουμε σύντομα τις αρχιτεκτονικές των παράλληλων υπολογιστών. Στην ενότητα 4.3 αναφέρουμε τα κίνητρα για την μετακίνηση προς στην περιοχή παράλληλου υπολογισμού χαμηλού κόστους. Στην ενότητα 4.4 παρουσιάζουμε την αρχιτεκτονική μιας συστοιχίας υπολογιστών (cluster of computers). Στην ενότητα 4.5 περιγράφουμε την ταξινόμηση των συστοιχιών σε διάφορες κατηγορίες. Στην ενότητα 4.6 αναφέρουμε σύντομα τα περιβάλλοντα και τα εργαλεία που χρησιμοποιούνται για τον παράλληλο προγραμματισμό. Στην ενότητα 4.7 παρουσιάζουμε τις βασικές συναρτήσεις της βιβλιοθήκης MPI. Στην ενότητα 4.8 παραθέτουμε ορισμένα μοντέλα παράλληλου προγραμματισμού. Στην ενότητα 4.9 παρουσιάζουμε σύντομα την αρχιτεκτονική των διατάξεων επεξεργαστών και την μεθοδολογία απεικόνισης αλγορίθμων στις διατάξεις αυτές. Τέλος, στην ενότητα 4.11 παρουσιάζουμε μέτρα απόδοσης που χρησιμοποιούνται στην παράλληλη επεξεργασία.

4.2 Γενικές Αρχιτεκτονικές Υπολογιστών

Ο Flynn [39] ταξινόμησε τους υπολογιστές με βάση τον αριθμό των εντολών και των ρευμάτων δεδομένων που μπορούν να επεξεργαστούν ταυτόχρονα στις ακόλουθες τέσσερις κατηγορίες:

1. Μοναδική Εντολή Μοναδικό Δεδομένο (Single Instruction Single Data - SISD): Ένα υπολογιστής SISD είναι μια μηχανή ενός επεξεργαστή που εκτελεί ακολουθιακά τις εντολές μία προς μία πάνω σε ένα ρεύμα δεδομένων. Οι υπολογιστές που υιοθετούν αυτό το μοντέλο ονομάζονται ακολουθιακοί ή σειριακοί υπολογιστές. Για να μπορέσει ο υπολογιστής να επεξεργαστεί όλες

τις εντολές και τα δεδομένα πρέπει πρώτα να αποθηκεύονται στην κύρια μνήμη. Παραδείγματα συστημάτων SISD είναι IBM-PC, Macintosh, σταθμοί εργασίας (workstations) κλπ.

2. Μοναδική Εντολή Πολλαπλά Δεδομένα (Single Instruction Multiple Data - SIMD): Ένας υπολογιστής SIMD είναι μια μηχανή από επεξεργαστές που εκτελούν μια κοινή ακολουθία εντολών πάνω σε διαφορετικά ρεύματα δεδομένων. Οι υπολογιστές που βασίζονται στον μοντέλο SIMD είναι κατάλληλοι για επιστημονικούς υπολογισμούς που περιλαμβάνουν πολλές πράξεις σε διανύσματα και πίνακες. Παραδείγματα συστημάτων SIMD είναι ο διανυσματικός επεξεργαστής Cray, ο Fujitsu VP, ο Maspar MP-1 και MP-2, το Connection Machine CM-2, κλπ.

3. Πολλαπλές Εντολές Μοναδικό Δεδομένο (Multiple Instruction Single Data - MISD): Ένας υπολογιστής MISD είναι μια μηχανή από επεξεργαστές που εκτελούν διαφορετικές εντολές πάνω στο ίδιο ρεύμα δεδομένων. Οι υπολογιστές που βασίζονται στο μοντέλο MISD δεν είναι χρήσιμοι σε πολλές εφαρμογές.

4. Πολλαπλές Εντολές Πολλαπλά Δεδομένα (Multiple Instruction Multiple Data - MIMD): Ένας υπολογιστής MIMD είναι μια μηχανή από επεξεργαστές που εκτελούν πολλαπλές εντολές πάνω σε πολλαπλά ρεύματα δεδομένων. Οι επεξεργαστές του συστήματος αυτού λειτουργούν αυτόνομα αφού έχουν την δική τους μονάδα ελέγχου και τοπική μνήμη. Οι υπολογιστές της κατηγορίας αυτής είναι κατάλληλοι για την επίλυση πολλών εφαρμογών. Οι επεξεργαστές ενός συστήματος MIMD λειτουργούν ασύγχρονα σε αντίθεση με τους υπολογιστές SIMD. Όσο πιο ασύγχρονη και χαλαρή είναι η σύνδεση ενός MIMD συστήματος τόσο μπορούμε να μιλάμε για κατανεμημένη επεξεργασία.

Μια ειδική κατηγορία των υπολογιστών SIMD είναι οι διατάξεις επεξεργαστών (array processors). Οι διατάξεις επεξεργαστών εκτελούν σύγχρονα μια κοινή ακολουθία πράξεων πάνω σε διαφορετικά δεδομένα. Η ιδιαιτερότητα τους σε σχέση με τους άλλους υπολογιστές SIMD βρίσκεται στο γεγονός ότι σχεδιάζονται για να εκτελούν με βέλτιστο τρόπο μια συγκεριμένη ομάδα αλγορίθμων. Είναι δηλαδή συστήματα ειδικού και όχι γενικού σκοπού. Στην συνέχεια του κεφαλαίου αυτού θα παρουσιάζουμε την αρχιτεκτονική των διατάξεων επεξεργαστών.

Οι υπολογιστές MIMD μπορούν επίσης να ταξινομηθούν με βάση την οργάνωση και διαχείριση της μνήμης τους σε δύο μεγάλες κατηγορίες:

1. Υπολογιστές MIMD Διαμοιραζόμενης Μνήμης (Shared Memory MIMD Machine): Στους υπολογιστές διαμοιραζόμενης μνήμης, που συχνά ονομάζονται και πολυεπεξεργαστές (multiprocessors) όλοι οι επεξεργαστές έχουν πρόσβαση σε διαμοιραζόμενη μνήμη μέσω της οποίας γίνεται και η επικοινωνία μεταξύ τους. Τα δεδομένα αποθηκεύονται στην διαμοιραζόμενη μνήμη και η τροποποίηση των δεδομένων αυτών από ένα επεξεργαστή φαίνονται και στους υπόλοιπους επεξεργαστές. Παραδείγματα διαμοιραζόμενης μνήμης είναι οι διάφοροι τύποι υπολογιστών Sun, SGI, SMP (Symmetric Multi-Processing), Cray Y-MP, Cray-4, κλπ.

2. Υπολογιστές MIMD Κατανεμημένης Μνήμης (Distributed Memory MIMD Machine): Στους υπολογιστές κατανεμημένης μνήμης, που συχνά ονομάζονται και πολυυπολογιστές (multicomputers) όλοι οι επεξεργαστές έχουν την δική τους τοπική μνήμη και η επικοινωνία μεταξύ των επεξεργαστών γίνεται με την ανταλλαγή μηνυμάτων μέσω του δικτύου διασύνδεσης (interconnection network). Το δίκτυο διασύνδεσης των υπολογιστών μπορεί να οργανωθεί σε διάφορες τοπολογίες όπως δένδρο (tree), δακτύλιο (ring), πλέγμα (grid) και τόρος (torus). Παραδείγματα κατανεμημένης μνήμης είναι οι SP/2 της IBM, Paragon της Intel, Hypercube της Intel, Meiko i860, nCUBE Hypercube, κλπ. Όμως όπως θα δούμε παρακάτω το δίκτυο διασύνδεσης μπορεί να είναι και ένα απλό τοπικό δίκτυο.

Περισσότερες λεπτομέρειες για τα χαρακτηριστικά των παράλληλων υπολογιστών που αναφέραμε προηγουμένως παρουσιάζονται στα βιβλία [138, 165].

4.3 Υπολογισμοί Χαμηλού Κόστους και Κίνητρα

Η χρήση των παράλληλων και κατανεμημένων συστημάτων σαν μέσο για την παροχή υπολογισμών υψηλής απόδοσης για μεγάλα μεγέθη εφαρμογών έχουν ερευνηθεί εκτενώς στα τελευταία χρόνια. Όμως μέχρι πρόσφατα, τα οφέλη από αυτήν την έρευνα περιορίστηκαν στα άτομα τα οποία είχαν πρόσβαση σε μεγάλες και ακριβές υπολογιστικές πλατφόρμες. Σήμερα η τάση στον παράλληλο και κατανεμημένο υπολογισμό μετακινείται από τις ειδικές υπερυπολογιστικές πλατφόρμες σε φθηνά συστήματα γενικού σκοπού όπως η συστοιχία υπολογιστών που αποτελείται από επεξεργαστές ή σταθμούς εργασίας ή πολυεπεξεργαστές. Επιπλέον, ακόμη και στη περίπτωση των πολυεπεξεργαστών ή άλλων αρχιτεκτονικών βελτιώσεων της απόδοσης των υπολογιστών, η τάση βρίσκεται στο σχεδιασμό συνδεδεμένων

συν-επεξεργαστών (attached co-processor) που προστίθενται στα παραδοσιακά συστήματα με κάρτες επέκτασης. Αυτή η μετακίνηση οφείλεται κυρίως στις πρόσφατες εξελίξεις στα δίκτυα υψηλής ταχύτητας και την βελτιωμένη απόδοση των μικροεπεξεργαστών και των σταθμών εργασίας. Αυτές οι εξελίξεις σημαίνουν ότι η συστοιχία υπολογιστών γίνεται ένα αποτελεσματικό μέσο για παράλληλο και κατανεμημένο υπολογισμό. Ένας ακόμη σημαντικός παράγοντας είναι η προτυποποίηση πολλών εργαλείων που χρησιμοποιούνται από τις παράλληλες εφαρμογές. Παραδείγματα τέτοιων προτύπων εργαλείων είναι η βιβλιοθήκη περάσματος μηνυμάτων MPI (Message Passing Interface) [162, 152] και η γλώσσα παραλληλισμού δεδομένων (data-parallel) HPF (High Performance Fotran) [92]. Η προτυποποίηση καθιστά δυνατή την ανάπτυξη, δοκιμή και εκτέλεση των εφαρμογών σε μια συστοιχία υπολογιστών και στο τελευταίο στάδιο τη μεταφορά τους με λίγη τροποποίηση πάνω στις αποκλειστικές παράλληλες πλατφόρμες. Η ακόλουθη λίστα υπογραμμίζει μερικούς από τους λόγους που η συστοιχία υπολογιστών χρησιμοποιείται περισσότερο σε σχέση με τους ειδικούς παράλληλους υπολογιστές [21]:

- Οι σταθμοί εργασίας γίνονται ολοένα και ταχύτεροι. Η απόδοση του σταθμού εργασίας έχει αυξηθεί δραματικά στα τελευταία χρόνια και διπλασιάζεται κάθε 18 έως 24 μήνες. Αυτό φαίνεται από το γεγονός ότι στην αγορά κυκλοφορούν ταχύτεροι επεξεργαστές και αποτελεσματικοί πολυεπεξεργαστές (SMP).

- Το εύρος ζώνης των επικοινωνιών (communication bandwidth) ανάμεσα στους σταθμούς εργασίας αυξάνεται και η καθυστέρηση (latency) μειώνεται καθώς καινούργιες τεχνολογίες δικτύων και πρωτοκόλλων υλοποιούνται στα τοπικά δίκτυα.

- Οι συστοιχίες σταθμών εργασίας είναι πιο εύκολο να ολοκληρωθούν στα υπάρχοντα δίκτυα από ότι οι ειδικοί παράλληλοι υπολογιστές.

- Οι προσωπικοί σταθμοί εργασίας δεν χρησιμοποιούνται τόσο αποδοτικά από τους χρήστες, άρα υπάρχει διαθέσιμη λανθάνουσα ισχύς.

- Η ανάπτυξη εργαλείων για τους σταθμούς εργασίας είναι πιο ώριμη σε σχέση με τους ειδικούς παράλληλους υπολογιστές και αυτό οφείλεται κυρίως στην έλλειψη προτύπων που υπάρχουν στα περισσότερα παράλληλα συστήματα.

- Οι συστοιχίες σταθμών εργασίας είναι φθηνές και εύκολα διαθέσιμες σε σχέση με τις ειδικές παράλληλες υπολογιστικές πλατφόρμες.

- Οι συστοιχίες μπορούν εύκολα να επεκταθούν και η χωρητικότητα του κάθε κόμβου (node) μπορεί εύκολα να αυξηθεί, εγκαθιστώντας επιπλέον μνήμη ή επεξεργαστές.

Σε βασικό επίπεδο, μια συστοιχία είναι μια συλλογή από σταθμούς εργασίας ή υπολογιστές που είναι διασυνδεδεμένοι μέσω δικτύου. Η συστοιχία σταθμών εργασίας είναι κατάλληλη για εφαρμογές που δεν έχουν υψηλές απαιτήσεις επικοινωνίας αφού ένα τοπικό δίκτυο έχει υψηλές καθυστερήσεις και χαμηλό εύρος ζώνης επικοινωνίας. Αν όμως οι εφαρμογές έχουν υψηλές απαιτήσεις επικοινωνίας, τότε οι συστοιχίες πρέπει να αποτελούνται από σταθμούς εργασίας υψηλής απόδοσης και το δίκτυο να είναι με υψηλό εύρος ζώνης και να έχει χαμηλή καθυστέρηση. Μια τέτοια συστοιχία μπορεί να παρέχει γρήγορες και αξιόπιστες υπηρεσίες στις υπολογιστικά απαιτητικές εφαρμογές ακόμα και σε εφαρμογές με υψηλές απαιτήσεις επικοινωνίας.

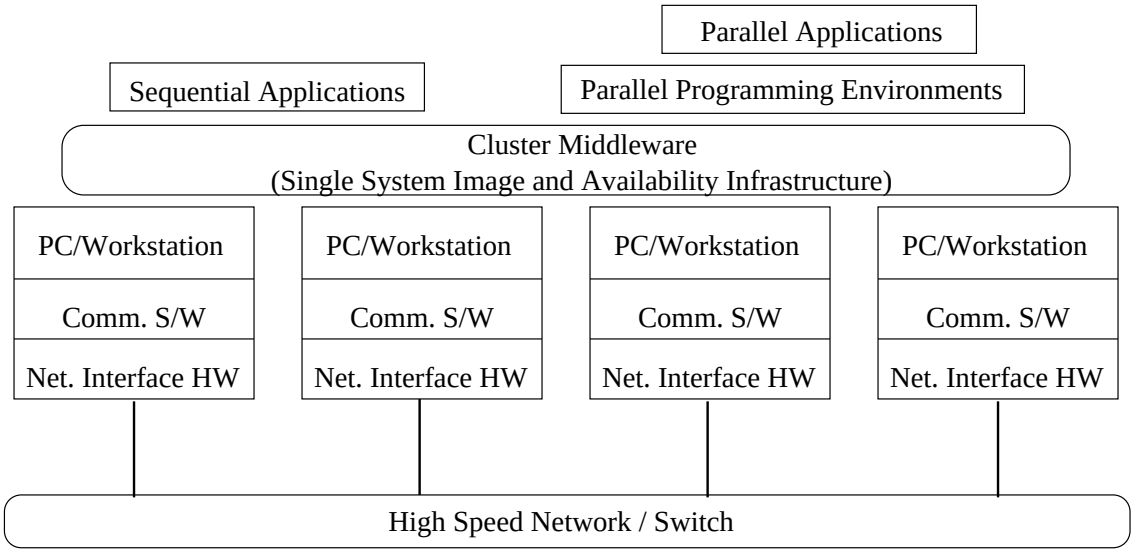
4.4 Αρχιτεκτονική μιας Συστοιχίας Σταθμών Εργασίας

Μια συστοιχία ή ένα δίκτυο σταθμών εργασίας είναι ένας είδος παράλληλου και κατανεμημένου συστήματος η οποία αποτελείται από μια συλλογή διασυνδεδεμένων κόμβων υπολογιστών που δουλεύουν μαζί σαν μια ολοκληρωμένη πηγή υπολογισμού.

Ένας κόμβος υπολογιστής (computer node) μπορεί να είναι μια μηχανή (δηλαδή, προσωπικός υπολογιστής ή σταθμός εργασίας) ή ένα σύστημα πολυεπεξεργασίας (δηλαδή, SMP) με μνήμη, με λειτουργίες Εισόδου/Εξόδου (Ε/Ε) και με ένα λειτουργικό σύστημα. Γενικά, ο όρος συστοιχία σημαίνει ότι συνδέονται δύο ή περισσότεροι υπολογιστές (ή κόμβοι) μαζί. Οι κόμβοι μπορεί να βρίσκονται σε ένα χώρο ή να είναι ξεχωριστά και να συνδέονται μέσω τοπικού δίκτυου (Local Area Network). Μια διασυνδεδεμένη συστοιχία υπολογιστών εμφανίζεται σαν ένα μοναδικό και ενιαίο σύστημα στους χρήστες και στις εφαρμογές. Η αρχιτεκτονική μιας τέτοιας συστοιχίας παρουσιάζεται στο Σχήμα 4.1.

Τα βασικά συστατικά μιας συστοιχία υπολογιστών είναι τα εξής:

1. Πολλαπλοί υπολογιστές υψηλής απόδοσης (όπως PCs, σταθμούς εργασίας ή SMPs)
2. Σύγχρονα λειτουργικά συστήματα (όπως πυρήνας (micro-kernel))
3. Δίκτυα ή Διακόπτες υψηλής απόδοσης (όπως Fast Ethernet, Gigabit Ethernet και Myrinet)



Σχήμα 4.1: Αρχιτεκτονική μιας συστοιχίας σταθμών εργασίας

4. Κάρτες δικτύων διεπαφής (Network Interface Cards - NICs)

5. Πρωτόκολλα επικοινωνίας και υπηρεσιών (όπως Active και Fast Messages)

6. Υποδομή λογισμικού της συστοιχίας (Cluster Middleware)

7. Περιβάλλοντα και εργαλεία παράλληλου προγραμματισμού (όπως μεταγλωτιστές, PVM (Parallel Virtual Machine) και MPI (Message Passing Interface)).

Η κάρτα δικτύου διεπαφής λειτουργεί σαν επεξεργαστής επικοινωνίας και είναι υπεύθυνη για τη μετάδοση και την λήψη πακέτων δεδομένων ανάμεσα στους κόμβους της συστοιχίας μέσω δικτύου.

Το λογισμικό επικοινωνίας προσφέρει μια γρήγορη και αξιόπιστη επικοινωνία μεταξύ των κόμβων της συστοιχίας. Συνήθως, η συστοιχία με ένα δίκτυο όπως το δίκτυο Myrinet χρησιμοποιεί πρωτόκολλα επικοινωνίας όπως τα active messages για ταχύτερη επικοινωνία μεταξύ των κόμβων της συστοιχίας.

Η υποδομή λογισμικού της συστοιχίας προσφέρει την δυνατότητα η ίδια συστοιχία να φαίνεται σαν ένα μοναδικό και ενιαίο παράλληλο σύστημα. Πρόκειται για ειδικό λογισμικό που επιτρέπει την ενιαία εγκατάσταση λογισμικού, διαχείριση υλικού και αποθηκευτικού χώρου, διαχείριση των διαφόρων πόρων κατά την εκτέλεση, όπως για παράδειγμα NPACI Rocks, Globus, κλπ.

Τέλος, τα περιβάλλοντα παράλληλου προγραμματισμού προσφέρουν μεταφέρσιμα και αποτελεσματικά εργαλεία για την ανάπτυξη των εφαρμογών. Τέτοια εργαλεία είναι συνήθως βιβλιοθήκες περάσματος μηνυμάτων (message passing libraries), αποσφαλματωτές (debuggers) και προφίλ απόδοσης παράλληλων προγραμμάτων (profilers).

4.5 Ταξινομήσεις Συστοιχιών

Οι συστοιχίες ταξινομούνται σε πολλές κατηγορίες σύμφωνα με τα παρακάτω κριτήρια [21]:

1. Ιδιοκτησία του κόμβου (node ownership). Με βάση το κριτήριο αυτό οι συστοιχίες διακρίνονται σε δύο κατηγορίες: αποκλειστική συστοιχία (dedicated cluster) και μη αποκλειστική συστοιχία (nondedicated cluster). Η αποκλειστική συστοιχία αναφέρεται στο γεγονός ότι κάθε σταθμός εργασίας εκτελεί αποκλειστικά εργασίες ενός παράλληλου υπολογισμού, ενώ η μη αποκλειστική συστοιχία αναφέρεται στο ότι κάθε σταθμός εργασίας εκτελεί τις κανονικές του ρουτίνες και χρησιμοποιεί μόνους τους αδρανείς κύκλους της CPU για να εκτελέσει παράλληλες εργασίες.

2. Υλικό του κόμβου (node hardware). Οι συστοιχίες μπορεί να αποτελούνται από προσωπικούς υπολογιστές ή από σταθμούς εργασίας ή από μηχανές πολυπεπεξεργασίας όπως για παράδειγμα SMPs ή ακόμη και από επιμέρους τοπικά δίκτυα (grid).

3. Είδος του κόμβου (node configuration). Το κριτήριο αυτό αφορά στην αρχιτεκτονική του κάθε κόμβου της συστοιχίας. Έτσι έχουμε τις ομοιογενείς συστοιχίες (homogeneous clusters) και τις ετερογενείς συστοιχίες (heterogeneous clusters). Οι ομοιογενείς συστοιχίες έχουν κόμβους με παρόμοιες αρχιτεκτονικές (δηλαδή, όλοι οι κόμβοι έχουν τις ίδιες ταχύτητες επεξεργασίας), ενώ οι ετερογενείς συστοιχίες έχουν κόμβους που έχουν διαφορετικές αρχιτεκτονικές (δηλαδή, όλοι οι κόμβοι έχουν διαφορετικές ταχύτητες επεξεργασίας).

4.6 Περιβάλλοντα και Εργαλεία Παράλληλου Προγραμματισμού

Λόγω της διαθεσιμότητας των προτύπων εργαλείων προγραμματισμού, οι συστοιχίες θεωρούνται σαν μια εναλλακτική πλατφόρμα για παράλληλη και κατανεμημένη επεξεργασία. Σε αυτή την ενότητα

περιγράφουμε μερικά από τα δημοφιλή εργαλεία.

4.6.1 Νήματα (Threads)

Τα νήματα είναι ένα δημοφιλές εργαλείο για ταυτόχρονο προγραμματισμό (concurrent programming) σε συμβατικό (ή ακολουθιακό) υπολογιστή και σε συστήματα πολυεπεξεργασίας. Στα συστήματα πολυεπεξεργασίας, τα νήματα χρησιμοποιούνται κυρίως για να αξιοποιήσουν ταυτόχρονα όλους τους επεξεργαστές. Από την άλλη μεριά στα συμβατικά συστήματα, τα νήματα χρησιμοποιούνται για να αξιοποιήσουν αποτελεσματικά τους πόρους του συστήματος. Ένα άλλο χαρακτηριστικό είναι ότι η δημιουργία ενός νήματος είναι φθηνότερη σε σχέση με την δημιουργία μιας διεργασίας. Τα νήματα επικοινωνούν μεταξύ τους με την χρήση των διαμοιραζόμενων μεταβλητών.

Τα νήματα είναι μεταφέρσιμα καθώς υπάρχει το πρότυπο της IEEE POSIX (Portable Operating System Interface) που λέγεται pthreads [134]. Το πρότυπο αυτό υποστηρίζει σε πολλές πλατφόρμες όπως προσωπικούς υπολογιστές, σταθμούς εργασίας, SMPs και συστοιχίες υπολογιστών. Τέλος, τα νήματα έχουν χρησιμοποιηθεί εκτενώς στην ανάπτυξη εφαρμογών και λογισμικού συστήματος.

4.6.2 Πέρασμα Μηνυμάτων - MPI και PVM

Οι βιβλιοθήκες περάσματος μηνυμάτων μας επιτρέπουν να γράφουμε παράλληλα προγράμματα για κατανεμημένα συστήματα. Επίσης, οι βιβλιοθήκες προσφέρουν ρουτίνες για την διαχείριση του περιβάλλοντος μηνυμάτων και ρουτίνες για την αποστολή και λήψη μηνυμάτων. Στην παρούσα φάση υπάρχουν δύο δημοφιλή συστήματα περάσματος μηνυμάτων για επιστημονικές και μηχανικές εφαρμογές που είναι το PVM (Parallel Virtual Machine) [46] από το Oak Ridge National Laboratory και το MPI (Message Passing Interface) [162, 152] που ορίστηκε από το MPI Forum.

Το MPI είναι η πιο διαδεδομένη πρότυπη βιβλιοθήκη για πέρασμα μηνυμάτων που μπορεί να χρησιμοποιηθεί για την ανάπτυξη μεταφέρσιμων προγραμμάτων περάσματος μηνυμάτων χρησιμοποιώντας τις γλώσσες προγραμματισμού C ή Fortran. Το πρότυπο MPI ορίζει συντακτικά και σημασιολογικά ένα σύνολο από συναρτήσεις βιβλιοθήκης που είναι χρήσιμες για την σύνταξη προγραμμάτων περάσματος μηνυμάτων. Το MPI αναπτύχθηκε από μια κοινότητα ερευνητών οι οποίοι προέρχονται από πανεπι-

Πίνακας 4.1: Ένα σύνολο από συναρτήσεις MPI

Συναρτήσεις MPI	Σημασία
`MPI_Init`	Εκκίνηση του MPI
`MPI_Finalize`	Τερματισμό του MPI
`MPI_Comm_size`	Προσδιορίζει τον αριθμό των διεργασιών
`MPI_Comm_rank`	Προσδιορίζει την σειρά της καλούσας διεργασίας
`MPI_Send`	Στέλνει ένα μήνυμα
`MPI_Recv`	Λαμβάνει ένα μήνυμα

στήμια και βιομηχανίες και υποστηρίζεται από όλους τους προμηθευτές υλικού. Επίσης, το MPI είναι διαθέσιμο σε περισσότερα παράλληλα υπολογιστικά συστήματα (από υπολογιστές διαμοιραζόμενης και κατανεμημένης μνήμης μέχρι και συστοιχίες υπολογιστών). Στην επόμενη ενότητα παρουσιάσουμε σύντομα την βιβλιοθήκη MPI την οποία χρησιμοποιούμε σε αυτήν την διατριβή. Περισσότερες πληροφορίες σχετικά με την βιβλιοθήκη MPI μπορούν να βρεθούν στα βιβλία [162, 136, 152, 165].

4.7 Βιβλιοθήκη MPI

Η βιβλιοθήκη MPI αποτελείται πάνω από 125 ρουτίνες αλλά ο αριθμός των βασικών εννοιών είναι πολύ μικρότερος. Μπορούν να αναπτυχθούν πλήρη προγράμματα περάσματος μηνυμάτων χρησιμοποιώντας μόνο έξι ρουτίνες όπως φαίνεται στον Πίνακα 4.1.

Στις επόμενες υποενότητες περιγράφουμε τις παραπάνω συναρτήσεις και τις βασικές έννοιες που είναι απαραίτητες για την σύνταξη σωστών και αποτελεσματικών προγραμμάτων περάσματος μηνυμάτων.

4.7.1 Εκκίνηση και Τερματισμός της βιβλιοθήκης MPI

Κάθε πρόγραμμα MPI πρέπει να περιέχει το αρχείο επικεφαλίδα `#include "mpi.h"` Το αρχείο `mpi.h` περιέχει ορισμούς και δηλώσεις που είναι απαραίτητες για την μεταγλώττιση ενός προγράμματος MPI.

Το MPI χρησιμοποιεί σταθερό σχήμα για την ονομασία των ρουτινών του MPI. Όλες οι ρουτίνες του MPI αρχίζουν με το πρόθεμα `"MPI_"` και τον επόμενο πρώτο χαρακτήρα κεφαλαίο γράμμα. Οι

υπόλοιποι χαρακτήρες των περισσότερων τύπων δεδομένων του MPI γράφονται με κεφαλαία γράμματα.

Η ρουτίνα `MPI_Init` καλείται πριν οποιαδήποτε κλήση ρουτινών MPI. Ο σκοπός της ρουτίνας αυτής είναι να εκκινήσει το περιβάλλον MPI. Η κλήση της ρουτίνας `MPI_Init` περισσότερο από μια φορά κατά την διάρκεια εκτέλεσης ενός προγραμμάτος καταλήγει σε σφάλμα. Η ρουτίνα `MPI_Finalize` καλείται στο τέλος του υπολογισμού και εκτελεί διάφορες εργασίες καθαρισμού για να τερματίσει το περιβάλλον MPI. Δεν εκτελούνται κλήσεις MPI πριν την κλήση της ρουτίνας `MPI_Init` και μετά την κλήση της ρουτίνας `MPI_Finalize`. Οι ρουτίνες `MPI_Init` και `MPI_Finalize` πρέπει να καλούνται από όλες τις διεργασίες διαφορετικά η συμπεριφορά του MPI θα είναι απροσδιόριστη. Η σύνταξη των δύο παραπάνω ρουτινών για την C είναι ως εξής:

```
int MPI_Init(int *argc, char ***argv)

int MPI_Finalize()
```

Τα ορίσματα `argc` και `argv` της ρουτίνας `MPI_Init` είναι ορίσματα γραμμής - διαταγής του προγράμματος C. Η επιτυχής εκτέλεση των ρουτινών `MPI_Init` και `MPI_Finalize` επιστρέφει την σταθερά `MPI_SUCCESS` διαφορετικά ένα προκαθορισμένο αριθμό σφάλματος. Συνεπώς, ένα πρόγραμμα MPI έχει την ακόλουθη τυπική δομή:

```
#include "mpi.h"

main(int argc, char *argv[]) {

        MPI_Init(&argc,&argv);

        ...

        MPI_Finalize();

}
```

4.7.2 Κανάλια Επικοινωνίας

Τα κανάλια επικοινωνίας (communicators) χρησιμοποιούνται στο MPI για επικοινωνίες από σημείο σε σημείο (point-to-point) και συλλογικές (collective). Ένα κανάλι επικοινωνίας χρησιμοποιείται για

να ορίζει ένα σύνολο από διεργασίες που μπορούν να επικοινωνούν μεταξύ τους. Το σύνολο των διεργασιών σχηματίζει μια περιοχή επικοινωνίας (communication domain). Πληροφορίες σχετικά με τις περιοχές επικοινωνίας αποθηκεύονται στις μεταβλητές του τύπου `MPI_Comm`. Στο MPI υπάρχει ένα εξ ορισμού κανάλι επικοινωνίας που ονομάζεται `MPI_COMM_WORLD` το οποίο περιλαμβάνει όλες τις διεργασίες που υπάρχουν σε μια παράλληλη εφαρμογή. Όμως, υπάρχουν περιπτώσεις που θέλουμε να πραγματοποιήσουμε επικοινωνία μόνο σε μια ορισμένη ομάδα διεργασιών, δηλαδή να ορίσουμε δικά μας κανάλια επικοινωνίας. Προς το παρόν, παρακάτω χρησιμοποιούμε το `MPI_COMM_WORLD` σαν όρισμα κανάλι επικοινωνίας σε όλες τις συναρτήσεις MPI που απαιτούν ένα κανάλι επικοινωνίας.

4.7.3 Πλήθος Διεργασιών και Σειρά Διεργασίας

Οι συναρτήσεις `MPI_Comm_size` και `MPI_Comm_rank` χρησιμοποιούνται για να προσδιορίζουν τον αριθμό των διεργασιών και την σειρά της καλούσας διεργασίας αντίστοιχα. Η σύνταξη των δύο παραπάνω συναρτήσεων είναι ως εξής:

```
int MPI_Comm_size(MPI_Comm comm, int *size)

int MPI_Comm_rank(MPI_Comm comm, int *rank)
```

Η συνάρτηση `MPI_Comm_size` επιστρέφει στην μεταβλητή `size` τον αριθμό των διεργασιών που ανήκουν στο κανάλι επικοινωνίας `comm`. Κάθε διεργασία που ανήκει σε ένα κανάλι επικοινωνίας αναγνωρίζεται μοναδικά από την σειρά της `rank`. Η σειρά μιας διεργασίας είναι ένας ακέραιος αριθμός που κυμαίνεται από 0 μέχρι $n-1$, όταν υπάρχουν n διεργασίες. Μια διεργασία μπορεί να προσδιορίζει την σειρά της σε ένα κανάλι επικοινωνίας από την κλήση της συνάρτησης `MPI_Comm_rank` όπου παίρνει δύο ορίσματα: το κανάλι επικοινωνίας και μια ακέραια μεταβλητή `rank`. Με την κλήση της συνάρτησης αυτής επιστρέφεται στη μεταβλητή `rank` τη σειρά της διεργασίας.

4.7.4 Επικοινωνία από Σημείο σε Σημείο

Η επικοινωνία από σημείο σε σημείο (point-to-point) περιλαμβάνει πάντα δύο διεργασίες. Η μια διεργασία στέλνει ένα μήνυμα στην άλλη διεργασία. Γνωρίζουμε ότι υπάρχουν διάφορες εκδόσεις για την αποστολή και λήψη μηνυμάτων. Έτσι, παρακάτω περιγράφουμε τις συγχρονισμένες (blocking) και μη

συγχρονισμένες (nonblocking) συναρτήσεις για αποστολή και λήψη μηνυμάτων.

Συγχρονισμένες Συναρτήσεις

Στο MPI οι συγχρονισμένες συναρτήσεις για αποστολή ή λήψη μηνημάτων επιστρέφουν τον έλεγχο τους όταν η επικοινωνία ολοκληρωθεί. Οι συγχρονισμένες συναρτήσεις για αποστολή και λήψη μηνυμάτων στο MPI είναι οι `MPI_Send` και `MPI_Recv`, αντίστοιχα. Η συγχρονισμένη συνάρτηση `MPI_Send` επιστρέφει όταν το μήνυμα έχει σταλεί στον παραλήπτη. Η σύνταξη της συνάρτησης `MPI_Send` έχει ως εξής:

```
int MPI_Send(void *buf, int count, MPI_Datatype datatype, int dest, int tag,
MPI_Comm comm)
```

όπου

- **buf** είναι η διεύθυνση του ενδιάμεσου χώρου αποθήκευσης που περιέχει δεδομένα που πρόκειται να σταλούν.
- **count** είναι ο αριθμός των στοιχείων του τύπου δεδομένων που ορίζεται από την παράμετρο **datatype** που περιέχονται στον ενδιάμεσο χώρο αποθήκευσης **buf**.
- **datatype** είναι ο τύπος δεδομένων του MPI για το κάθε στοιχείο του **buf**. Στον Πίνακα 4.2 παρουσιάζεται η αντιστοίχηση ανάμεσα στους τύπους δεδομένων που υποστηρίζει το MPI και τους τύπους δεδομένων που υποστηρίζει η γλώσσα προγραμματισμού C. Σημειώνουμε ότι στο MPI υπάρχουν δύο τύποι δεδομένων που δεν υποστηρίζει C. Οι τύποι αυτοί είναι οι `MPI_BYTE` και `MPI_PACKED`. Ο τύπος `MPI_BYTE` αντιστοιχεί σε ένα byte (8 bits). Ο τύπος `MPI_PACKED` αντιστοιχεί σε μια συλλογή δεδομένων που έχει δημιουργηθεί με συνένωση.
- **dest** είναι η διεργασία παραλήπτη που θα λάβει το μήνυμα. Η παράμετρος αυτή είναι η σειρά της διεργασίας παραλήπτη στην περιοχή επικοινωνίας που ορίζεται από το κανάλι επικοινωνίας **comm**.
- **tag** είναι ετικέτα που έχει μια ακέραια τιμή και χρησιμοποιείται για να διακρίνει τους διαφορετικούς τύπους μηνυμάτων. Η ετικέτα **tag** παίρνει τιμές από 0 μέχρι `MPI_TAG_UB` που είναι μια προκαθορισμένη σταθερά του MPI. Η τιμή της σταθεράς `MPI_TAG_UB` είναι τουλάχιστον 32,767.

Πίνακας 4.2: Αντιστοίχηση ανάμεσα στους τύπους δεδομένων που υποστηρίζει το MPI και τους τύπους δεδομένων που υποστηρίζει η C

Τύπος δεδομένων του MPI	Τύπος δεδομένων της C
MPI_CHAR	signed char
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	singed long int
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_BYTE	
MPI_PACKED	

- **comm** είναι το κανάλι επικοινωνίας που διαμοιράζεται από τις διεργασίες αποστολέα και παραλήπτη.

Η συγχρονισμένη συνάρτηση **MPI_Recv** επιστρέφει όταν έχει λάβει το μήνυμα και έχει αντιγραφεί το μήνυμα στο ενδιάμεσο χώρο αποθήκευσης του παραλήπτη. Η σύνταξη της συνάρτησης **MPI_Recv** έχει ως εξής:

```
int MPI_Recv(void *buf, int count, MPI_Datatype datatype, int source, int tag,
MPI_Comm comm, MPI_Status *status)
```

όπου

- **buf** είναι η διεύθυνση του ενδιάμεσου χώρου αποθήκευσης που περιέχει δεδομένα μετά την λήψη του μηνύματος.
- **count** είναι ο μέγιστος αριθμός των στοιχείων του τύπου δεδομένων που ορίζεται από την παράμετρο **datatype** που μπορεί να περιέχονται στον ενδιάμεσο χώρο αποθήκευσης **buf**. Ο αριθμός των δεδομένων που πραγματικά λαμβάνει πρέπει να είναι λιγότερο από τον αριθμό **count**. Άν όμως ο αριθμός των δεδομένων που λαμβάνει είναι μεγαλύτερος από το μέγεθος του ενδιάμεσου χώρου (**count**) τότε εμφανίζεται σφάλμα υπερχείλισης και η συνάρτηση επιστρέφει το σφάλμα MPI_ERR_TRUNCATE.

- **datatype** είναι ο τύπος δεδομένων του MPI για το μήνυμα. Ο τύπος αυτός πρέπει να ταυτίζεται με τον τύπο δεδομένων που ορίζεται στην συνάρτηση MPI_Send.
- **source** είναι η σειρά της διεργασίας αποστολέα του μηνύματος στην περιοχή επικοινωνίας που ορίζεται από το κανάλι επικοινωνίας comm. Στην περίπτωση που η παράμετρος source έχει οριστεί με τιμή MPI_ANY_SOURCE τότε λαμβάνει μηνύματα από οποιοδήποτε αποστολέα.
- **tag** είναι ετικέτα που δηλώνει τι είδους μηνύματα μπορεί να λάβει η διεργασία. Αν η ετικέτα αυτή έχει τιμή MPI_ANY_TAG τότε λαμβάνει μηνύματα με οποιαδήποτε τιμή ετικέτας.
- **comm** είναι το κανάλι επικοινωνίας που διαμοιράζεται από τις διεργασίες αποστολέα και παραλήπτη.
- **status** περιέχει πληροφορίες σχετικά με την λήψη του μηνύματος.

Μη Συγχρονισμένες Συναρτήσεις

Μια μη συγχρονισμένη συνάρτηση επιστρέφει αμέσως και συνεχίζει να εκτελεί την επόμενη εντολή του προγράμματος. Σε κάποια μεταγενέστερη χρονική στιγμή η συνάρτηση ελέγχει για την ολοκλήρωση της επικοινωνίας. Οι μη συγχρονισμένες συναρτήσεις για αποστολή και λήψη μηνυμάτων στο MPI είναι οι MPI_Isend και MPI_Irecv, αντίστοιχα. Η συνάρτηση MPI_Isend επιστρέφει ανεξάρτητα με την ολοκλήρωση της επικοινωνίας. Ενώ η συνάρτηση MPI_Irecv επιστρέφει ακόμα και αν δεν υπάρχει μήνυμα να λάβει. Η σύνταξη των δύο παραπάνω συναρτήσεων είναι ως εξής:

```
int MPI_Isend(void *buf, int count, MPI_Datatype datatype, int dest, int tag,
MPI_Comm comm, MPI_Request *request)
```

```
int MPI_Irecv(void *buf, int count, MPI_Datatype datatype, int source, int tag,
MPI_Comm comm, MPI_Request *request)
```

Παρατηρούμε από τις παραπάνω συναρτήσεις ότι έχουν παρόμοια ορίσματα με τις συγχρονισμένες συναρτήσεις εκτός από ένα όρισμα. Το όρισμα αυτό είναι το **request** που χρησιμοποιείται για τον έλεγχο ολοκλήρωσης της επικοινωνίας. Οι δύο παραπάνω συναρτήσεις μπορούν να ελέγξουν για το αν ολοκληρώθηκε η επικοινωνία με την χρήση των συναρτήσεων MPI_Wait και MPI_Test. Η συνάρτηση MPI_Wait περιμένει μέχρι να ολοκληρωθεί η επικοινωνία. Η συνάρτηση MPI_Test εξετάζει αν η επικοινωνία εκείνη τη στιγμή που καλείται η συνάρτηση έχει ολοκληρωθεί και επιστρέφει αμέσως με μια

λογική απάντηση TRUE ή FALSE. Δεν παρουσιάζουμε την σύνταξη των συναρτήσεων αυτών διότι δεν τις χρησιμοποιούμε συχνά σε αυτή την διατριβή.

4.7.5 Συλλογική Επικοινωνία

Η συλλογική (collective) επικοινωνία περιλαμβάνει ένα σύνολο από διεργασίες σε σχέση με την επικοινωνία από σημείο σε σημείο που περιλαμβάνει μόνο δύο διεργασίες. Το σύνολο των διεργασιών που ορίζει το κανάλι επικοινωνίας συμμετέχουν στην συλλογική επικοινωνία. Η συλλογική επικοινωνία μπορεί να υλοποιηθεί από τον προγραμματιστή χρησιμοποιώντας τις συναρτήσεις MPI_Send και MPI_Recv. Όμως, η βιβλιοθήκη MPI παρέχει ένα σύνολο από ειδικές συναρτήσεις συλλογικής επικοινωνίας. Οι συναρτήσεις είναι οι παρακάτω:

Φράγμα (Barrier)

Η συνάρτηση για το φράγμα είναι η MPI_Barrier. Η συνάρτηση αυτή συγχρονίζει την καλούσα διεργασία με όλες τις διεργασίες που πρέπει να καλέσουν την συνάρτηση αυτή. Με άλλα λόγια συγχρονίζει όλες τις διεργασίες. Η σύνταξη της είναι ως εξής:

```
int MPI_Barrier(MPI_Comm comm)
```

Εκπομπή (Brodacast)

Η συνάρτηση για την εκπομπή είναι η MPI_Bcast. Η συνάρτηση αυτή στέλνει δεδομένα από μια διεργασία ρίζα προς όλες τις διεργασίες που μετέχουν στο κανάλι επικοινωνίας. Η σύνταξη της παραπάνω συνάρτησης έχει ως εξής:

```
int MPI_Bcast(void *buf, int count, MPI_Datatype datatype, int root,
MPI_Comm comm)
```

όπου τα ορίσματα buf, count και datatype είναι παρόμοια με τα ορίσματα των συναρτήσεων MPI_Send και MPI_Recv. Το όρισμα root είναι η σειρά της διεργασίας ρίζας.

Συλλογή (Gather)

Η συνάρτηση για την συλλογή είναι η `MPI_Gather`. Με τη συνάρτηση αυτή συγκεντρώνονται δεδομένα από όλες τις διεργασίες που μετέχουν στο κανάλι επικοινωνίας στην διεργασία ρίζα. Η σύνταξη της συνάρτησης αυτής είναι ως εξής:

```
int MPI_Gather(void *sendbuf, int sendcount, MPI_Datatype sendtype,
void *recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm)
```

όπου

- **sendbuf** είναι η διεύθυνση του ενδιάμεσου χώρου αποθήκευσης των δεδομένων της κάθε διεργασίας.
- **sendcount** είναι ο αριθμός των στοιχείων του τύπου δεδομένων που ορίζεται από την παράμετρο **sendtype** που περιέχονται στο **sendbuf** της κάθε διεργασίας.
- **sendtype** είναι ο τύπος δεδομένων του MPI για το κάθε στοιχείο του **sendbuf** της κάθε διεργασίας.
- **recvbuf** είναι η διεύθυνση του ενδιάμεσου χώρου αποθήκευσης της διεργασίας ρίζας που θα περιέχει τα δεδομένα τα οποία συλλέγονται από όλες τις διεργασίες.
- **recvcount** είναι ο αριθμός των στοιχείων του τύπου δεδομένων **recvtype** τα οποία συλλέγονται από κάθε διεργασία.
- **recvtype** είναι ο τύπος δεδομένων για το κάθε στοιχείο του **recvbuf** που έλαβε από κάθε διεργασία.
- **root** είναι η σειρά της διεργασίας ρίζας.
- **comm** είναι το κανάλι επικοινωνίας που περιέχει όλες τις διεργασίες που μετέχουν στην συλλογική επικοινωνία.

Πίνακας 4.3: Προκαθορισμένοι τελεστές πράξης

Τελεστές MPI	Σημασία
MPI_MAX	Μεγαλύτερη τιμή
MPI_MIN	Μικρότερη τιμή
MPI_SUM	Άθροισμα
MPI_PROD	Γινόμενο
MPI_LAND	Λογικό AND
MPI_LOR	Λογικό OR

Αναγωγή (Reduction)

Η συνάρτηση για την αναγωγή είναι η MPI_Reduce. Η συνάρτηση αυτή συνδυάζει τα δεδομένα όλων των διεργασιών χρησιμοποιώντας έναν τελεστή πράξης σε μια τιμή που αποθηκεύεται στην διεργασία ρίζα. Οι πράξεις μπορεί να είναι πρόσθεση, πολλαπλασιασμός, μεγαλύτερη τιμή, μικρότερη τιμή, το λογικό AND, κλπ. Η σύνταξη της συνάρτησης είναι ως εξής:

```
int MPI_Reduce(void *sendbuf, void *recvbuf, int count, MPI_Datatype datatype,
MPI_Op op, int root, MPI_Comm comm)
```

όπου τα ορίσματα **sendbuf**, **recvbuf**, **count**, **datatype**, **root** και **comm** είναι παρόμοια με τα ορίσματα των συναρτήσεων που έχουμε παρουσιάσει μέχρι τώρα. Το όρισμα **op** είναι ο τελεστής πράξης που συνδυάζει τα δεδομένα που περιέχονται στο **sendbuf** από κάθε διεργασία σε μια τιμή που αποθηκεύεται στο **recvbuf** της διεργασίας ρίζας. Στο MPI υπάρχουν προκαθορισμένοι τελεστές πράξης μερικούς από τους οποίους παρουσιάζουμε στον Πίνακα 4.3.

4.8 Μοντέλα Παράλληλου Προγραμματισμού

Στη διεθνή βιβλιογραφία της παράλληλης επεξεργασίας υπάρχουν πολλά μοντέλα τα οποία χρησιμοποιούνται στον παράλληλο προγραμματισμό. Περισσότερες λεπτομέρειες για τα μοντέλα παραλληλισμού αναφέρονται διεξοδικά στο βιβλίο [22]. Επομένως, στην ενότητα αυτή παρουσιάζουμε ορισμένα κοινά παράλληλα μοντέλα τα οποία χρησιμοποιούνται συχνά.

4.8.1 Μοντέλο Παράλληλων Δεδομένων (Data-Parallel Model)

Το μοντέλο παράλληλων δεδομένων είναι ένα από τα πιο απλά παράλληλα μοντέλα. Σε αυτό το μοντέλο, οι εργασίες απεικονίζονται στατικά πάνω στις διεργασίες και κάθε εργασία εκτελεί ίδιες πράξεις πάνω σε διαφορετικά δεδομένα. Αυτό το είδος παραλληλισμού που έχει σαν αποτέλεσμα την εκτέλεση παρόμοιων πράξεων πάνω σε διαφορετικά δεδομένα ταυτόχρονα ονομάζεται παραλληλισμός δεδομένων (data parallelism). Ο υπολογισμός μπορεί να εκτελείται σε διάφορες φάσεις και τα δεδομένα που επεξεργάζεται σε κάθε διαφορετική φάση μπορεί να είναι διαφορετικά. Έτσι, οι υπολογιστικές φάσεις εναλλάσονται σποραδικά με συγχρονισμό των εργασιών ή με αναμονή νέων δεδομένων για επεξεργασία. Η διάσπαση του προβλήματος σε εργασίες βασίζεται συνήθως στο σχήμα της κατάτμησης δεδομένων (data partitioning) επειδή μια ισομερής κατάτμηση των δεδομένων συνεπάγεται μια στατική απεικόνιση πάνω στις διεργασίες και μας παρέχει καλή ισορροπία του φορτίου (load balance).

Το μοντέλο παράλληλων δεδομένων μπορεί να υλοποιηθεί σε συστήματα διαμοιραζόμενης μνήμης και σε συστήματα κατανεμημένης μνήμης. Η υλοποίηση του μοντέλου αυτού είναι κατάλληλη σε συστήματα κατανεμημένης μνήμης γιατί η υπολογιστική δραστηριότητα της κάθε διεργασίας επικεντρώνεται κυρίως στην δική της περιοχή τοπικών δεδομένων και η επικοινωνία των διεργασιών είναι σχετικά σπάνια. Επιπλέον, η υλοποίηση του μοντέλου παράλληλων δεδομένων είναι κατάλληλη ειδικά για συστήματα διαμοιραζόμενης μνήμης όταν η διανομή των δεδομένων είναι διαφορετική σε διαφορετικές φάσεις του μοντέλου. Γενικά το μοντέλο αυτό ονομάζεται και SPMD (Single Program Multiple Data).

4.8.2 Μοντέλο Γράφου Διεργασιών (Task Graph Model)

Ένας παράλληλος αλγόριθμος μπορεί να παρασταθεί σαν γράφος διεργασιών που σαφώς χρησιμοποιείται για την απεικόνιση των διεργασιών. Στο μοντέλο γράφος διεργασιών, χρησιμοποιούνται οι σχέσεις μεταξύ των διεργασιών προκειμένου να μειωθεί το κόστος της επικοινωνίας. Το μοντέλο αυτό χρησιμοποιείται κυρίως για να λύσει προβλήματα στα οποία η ποσότητα των δεδομένων που συσχετίζεται με τις διεργασίες είναι μεγάλη σε σχέση με την ποσότητα υπολογισμού που συσχετίζεται με τις διεργασίες. Οι διεργασίες απεικονίζονται στατικά έτσι ώστε το κόστος επικοινωνίας μεταξύ των διεργασιών να είναι ελάχιστο. Μερικές φορές χρησιμοποιείται η αποκεντρωτική (decentralized) δυναμική απεικόνιση που χρησιμοποιεί πληροφορίες σχετικά με την δομή του γράφου διεργασιών και την δομή επικοινωνίας

των διεργασιών ώστε να ελαχιστοποιεί το κόστος της επικοινωνίας.

Υπάρχουν διάφορες τεχνικές για την μείωση της επικοινωνίας που εφαρμόζονται στο μοντέλο γράφου διεργασιών όπως η μείωση του όγκου και της συχνότητας της επικοινωνίας με την αύξηση της τοπικότητας των δεδομένων σε κάθε διεργασία ενώ η απεικόνιση των διεργασιών βασίζεται στην δομή επικοινωνίας μεταξύ των διεργασιών.

4.8.3 Μοντέλο Δεξαμενής Εργασίας (Work Pool Model)

Το μοντέλο δεξαμενής εργασίας χρησιμοποιείται όταν οι εργασίες δεν είναι γνωστές εκ των προτέρων, αλλά δημιουργούνται δυναμικά καθώς εκτελείται η εφαρμογή. Για να πετύχουμε εξισορρόπηση φορτίου (load balancing) οι εργασίες πρέπει να απεικονίζονται σε διεργασίες δυναμικά καθώς δημιουργούνται κατά την διάρκεια εκτέλεσης της εφαρμογής. Έτσι, σε αυτό το μοντέλο δεν υπάρχει προ-απεικόνιση των εργασιών στις διεργασίες. Η απεικόνιση των εργασιών μπορεί να είναι συγκεντρωτική (centralized) ή αποκεντρωτική (decentralized). Οι εργασίες μπορεί να αποθηκεύονται σε μια διαμοιραζόμενη λίστα όπως ουρά προτεραιότητας, πίνακας κατακερματισμού και δένδρο ή μπορούν να αποθηκεύονται σε μια κατανεμημένη δομή δεδομένων. Σε αυτό το μοντέλο, οι εργασίες αποθηκεύονται σε μια δεξαμενή εργασίας και κάθε διεργασία λαμβάνει μια εργασία από την δεξαμενή και στην συνέχεια πραγματοποιεί τον απαιτούμενο υπολογισμό. Κατά την διάρκεια της εκτέλεσης μιας εργασίας, οι διεργασίες μπορεί να δημιουργήσουν νέες εργασίες, οι οποίες προστίθενται στην δεξαμενή εργασίας. Τέλος, αν εφαρμόζεται στο μοντέλο δεξαμενής εργασίας αποκεντρωτική απεικόνιση, τότε απαιτείται ένας αλγόριθμος αχνίνευσης τερματισμού. Ο αλγόριθμος αυτός έχει σαν σκοπό όλες οι διεργασίες να ανιχνεύσουν την ολοκλήρωση της εφαρμογής (όταν είναι άδεια η δεξαμενή εργασίας) και να τερματίσουν τη λειτουργία τους.

4.8.4 Μοντέλο Συντονιστής - Εργαζόμενος (Master-Worker Model)

Το μοντέλο συντονιστής - εργαζόμενος αποτελείται από δύο οντότητες: τον συντονιστή και τους πολλαπλούς εργαζόμενους. Ο συντονιστής είναι υπεύθυνος για την διάσπαση του προβλήματος σε μικρές εργασίες, για την διανομή των εργασιών αυτών στους εργαζόμενους και για την συλλογή αποτελεσμάτων από τους εργαζόμενους για να παράγει το τελικό αποτέλεσμα του προβλήματος. Οι

εργαζόμενοι εκτελούν την ακόλουθη διαδικασία σε ένα απλό κύκλο χρόνου: λαμβάνουν μια εργασία, επεξεργάζονται την εργασία και αποστέλουν το αποτέλεσμα της επεξεργασίας στο συντονιστή. Έτσι, η επικοινωνία λαμβάνει χώρα μόνο ανάμεσα στον συντονιστή και στους εργαζόμενους.

Το μοντέλο συντονιστής - εργαζόμενος μπορεί να χρησιμοποιήσει είτε την στατική διανομή (static load balancing) ή την δυναμική διανομή (dynamic load balancing). Στην πρώτη περίπτωση, η διανομή των εργασιών γίνεται στην αρχή του υπολογισμού. Απο την άλλη μεριά, η δυναμική διανομή χρησιμοποιείται όταν ο αριθμός των εργασιών είναι μεγαλύτερος από τον αριθμό των επεξεργαστών ή όταν δεν γνωρίζουμε από την αρχή της εφαρμογής τον αριθμό των εργασιών.

Το μοντέλο συντονιστής - εργαζόμενος μπορεί να γενικευτεί σε ένα ιεραρχικό μοντέλο συντονιστής - εργαζόμενος (hierachical master-worker model). Στο ανώτερο επίπεδο του ιεραρχικού μοντέλου ο συντονιστής διανέμει μεγάλα κομμάτια εργασιών στους συντονιστές του δευτέρου επιπέδου οι οποίοι διαμερίζουν και διανέμουν τις εργασίες στους αντίστοιχους εργαζομένους τους και επίσης κάθε συντονιστής του δευτέρου επιπέδου μπορεί να εκτελέσει κάποιες εργασίες. Το ιεραρχικό μοντέλο είναι κατάλληλο σε συστήματα διαμοιραζόμενης μνήμης ή κατανεμημένης μνήμης εφόσον η αλληλεπίδραση είναι δύο επιπέδων (two-way), δηλαδή ο συντονιστής γνωρίζει ότι πρέπει να διανέμει εργασίες και οι εργαζόμενοι γνωρίζουν να λαμβάνουν εργασίες από τον συντονιστή.

Όταν χρησιμοποιούμε το μοντέλο συντονιστής - εργαζόμενος πρέπει να λαβούμε υπόψη ότι ο συντονιστής δεν πρέπει να υποστεί συμφόρηση (bottleneck) η οποία μπορεί να εμφανιστεί όταν οι εργασίες είναι πολύ μικρές ή όταν οι εργαζόμενοι είναι σχετικά γρήγοροι. Σε αυτό το μοντέλο, η διασπορά των εργασιών (granularity) πρέπει να είναι τέτοια ώστε το κόστος του υπολογισμού να κυριαρχείται από το κόστος της διανομής των εργασιών και το κόστος του συγχρονισμού.

4.8.5 Μοντέλο Διασωλήνωσης (Pipeline Model)

Στο μοντέλο διασωλήνωσης οι επεξεργαστές οργανώνονται σε μερικές κανονικές δομές όπως αυτή του δακτυλίου ή του διδιάστατου πλέγματος. Διαμέσου αυτής της κανονικής δομής τα δεδομένα μετακινούνται μέσα στη δομή, με κάθε επεξεργαστή να εκτελεί ένα συγκεριμένο τμήμα της συνολικής υπολογιστικής διαδικασίας. Γενικά σε αυτό το μοντέλο μια σειρά από επεξεργαστές σχηματίζουν μια γραμμική ή πλεγματική διασωλήνωση όπου κάθε επεξεργαστής λαμβάνει μια σειρά από δεδομένα από

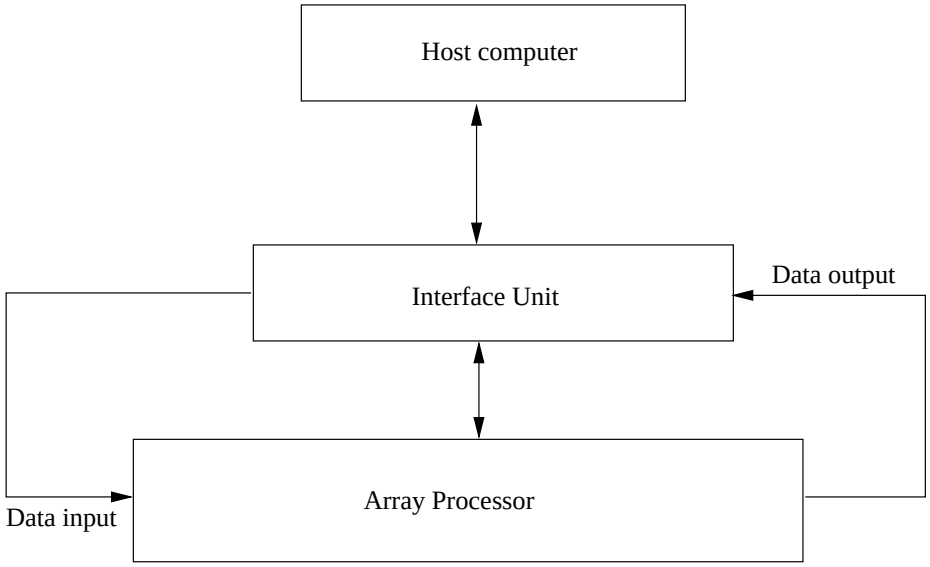
τον αριστερό επεξεργαστή, τα μετατρέπει κατόπιν επεξεργασίας και μετά στέλνει τα αποτελέσματα στον δεξιό επεξεργαστή. Κάθε αρχική τιμή που εισέρχεται στη διασωλήνωση υφίσταται μια σειρά από μετασχηματισμούς και εξέρχεται ως αποτέλεσμα από τη διασωλήνωση. Ο παραλληλισμός προέρχεται από το γεγονός ότι κάθε επεξεργαστής της διασωλήνωσης εκτελεί μετασχηματισμούς την ίδια ακριβώς στιγμή αλλά με διαφορετικά δεδομένα. Στο μοντέλο διασωλήνωσης η απεικόνιση των εργασιών πάνω στους επεξεργαστές είναι στατική.

Η εξισορρόπηση του φορτίου είναι συνάρτηση της διασποράς των εργασιών. Έτσι, όσο μεγαλύτερη είναι η διασπορά τόσο αυξάνεται ο παραλληλισμός στην διασωλήνωση. Αντίθετα, αν η διασπορά είναι πολύ μικρή τότε αυξάνεται η επιβάρυνση της επικοινωνίας εφόσον οι επεξεργαστές χρειάζονται να επικοινωνήσουν για να λάβουν νέα δεδομένα μετά από μια μικρή διάρκεια υπολογισμού.

4.9 Διατάξεις Επεξεργαστών

Οι διατάξεις επεξεργαστών (array processors) αναφέρθηκαν πρώτα από τους Kung και Leiserson σε άρθρο τους το 1980 [72]. Μια διάταξη επεξεργαστών είναι ένα δίκτυο από όμοια στοιχεία επεξεργασίας (processing elements) ή κελιά (cells) διασυνδεδεμένων ομοιόμορφα μεταξύ τους. Το δίκτυο μπορεί να είναι οργανωμένο σε μονοδιάστατες διατάξεις (όπως γραμμικές διατάξεις ή διατάξεις δακτυλίου) ή σε διδιάστατες διατάξεις (όπως τετραγωνικές, τριγωνικές διατάξεις ή διατάξεις δένδρου). Τα κελιά των μονοδιάστατων διατάξεων συνδέονται με το πολύ μέχρι δύο θύρες με τα γειτονικά τους κελιά μέσω των οποίων ανταλλάσουν δεδομένα, ενώ τα κελιά των διδιάστατων διατάξεων συνδέονται με το πολύ μέχρι τέσσερις θύρες με τα γειτονικά κελιά. Τα κελιά είναι όλα ίδια και αποτελούνται από απλά κυκλωματικά στοιχεία όπως καταχωρητές, αθροιστές, χωρίς αυτό όμως να αποκλείει τη δυνατότητα ύπαρξης ολόκληρων μικροεπεξεργαστών ως κελιών σε μια διάταξη επεξεργαστών.

Η διάταξη επεξεργαστών αποτελεί μια παράλληλη αρχιτεκτονική που όλα τα κελιά εκτελούν την ίδια λειτουργία με διαφορετικά κάθε φορά δεδομένα. Συνεπώς, τα κελιά μιας διάταξης επεξεργαστών έχουν όλα τον ίδιο χρονισμό και λειτουργούν ταυτόχρονα έτσι ώστε σε κάθε υπολογιστικό βήμα να υπάρχουν δύο φάσεις. Η μια φάση είναι η φάση της τοπικής μεταφοράς δεδομένων (ή επικοινωνία) από κελί σε κελί και η άλλη είναι η φάση της εκτέλεσης των υπολογισμών μέσα στα κελιά. Επομένως, όλα τα κελιά δουλεύουν δηλαδή ρυθμικά, επαναληπτικά και παράλληλα, τόσο ως προς τους υπολογισμούς, όσο



Σχήμα 4.2: Λειτουργία της διάταξης επεξεργαστών

και ως προς τη μεταφορά δεδομένων. Η ταυτόχρονη (ή παράλληλη) λειτουργία όλων των κελιών της διάταξης οδηγεί σε μεγάλους βαθμούς παραλληλισμού και συνεπώς σε μεγάλες ταχύτητες εκτέλεσης των υπολογισμών. Σαφή περιορισμό στην ταχύτητα λειτουργίας τους, θέτει η ταχύτητα επικοινωνίας μεταξύ δύο γειτονικών κελιών.

Ο συνολικός χρόνος εκτέλεσης υπολογισμού σε μια διάταξη επεξεργαστών είναι το άθροισμα του χρόνου υπολογισμού δεδομένων και του χρόνου μεταφοράς ανάμεσα στα γειτονικά κελιά.

Οι διατάξεις επεξεργαστών με βάση τις απαιτήσεις μιας εφαρμογής χωρίζονται σε δύο κατηγορίες: σε διατάξεις ειδικού σκοπού (algorithm-specific arrays) που είναι σχεδιασμένες για την επίλυση ενός συγκεριμένου αλγορίθμου και σε διατάξεις γενικού σκοπού (class-specific arrays) που μπορούν να προσαρμοστούν (ή να προγραμματιστούν) σε μια κλάση αλγορίθμων.

Στο Σχήμα 4.2 φαίνεται η βασική αρχή λειτουργίας μιας διάταξης επεξεργαστών.

Μια διάταξη επεξεργαστών λειτουργεί κάτω από την εποπτεία μιας μονάδας διεπαφής (interface unit), η οποία παράγει σήματα ελέγχου και τα δεδομένα εισόδου (data input), ενώ συλλέγει τα αποτελέσματα (data output). Με άλλα λόγια, αποτελεί τη γέφυρα επικοινωνίας της διάταξης επεξεργαστών με τον κεντρικό υπολογιστή (host computer).

Σε μια διάταξη επεξεργαστών τα δεδομένα εισόδου εισάγονται, υπόκεινται επεξεργασία και στο τέλος τα αποτελέσματα μεταφέρονται στην μονάδα διεπαφής. Η επικοινωνία αυτή γίνεται μέσω των ακραίων κελιών της διάταξης. Σύμφωνα με το μοντέλο αυτό, η απόδοση μιας τέτοιας διάταξης είναι άμεσα εξαρτώμενη τόσο από τον όγκο όσο και από τον ρυθμό εισόδου των δεδομένων. Γι' αυτό, τέτοιου είδους διατάξεις είναι κατάλληλες κυρίως για την υλοποίηση αλγορίθμων με μεγάλο υπολογιστικό φορτίο και όχι με συνεχείς διαδικασίες Εισόδου/Εξόδου (Ε/Ε). Αλλιώς, η απόδοση μειώνεται καθώς τα κελιά θα παραμένουν ανενεργά μέχρι να εισαχθούν τα νέα δεδομένα.

4.10 Μεθοδολογία Απεικόνισης Αλγορίθμων σε Διατάξεις Επεξεργαστών

Σε αυτή την ενότητα, παρουσιάζουμε την μεθοδολογία απεικόνισης κανονικών (regular) και επαναληπτικών (repetitive) αλγορίθμων σε διατάξεις επεξεργαστών. Σημειώνουμε ότι οι αλγόριθμοι που υλοποιούνται ιδανικά από μια διάταξη επεξεργαστών είναι όσοι περιέχουν μεγάλο αριθμό επαναληπτικών διαδικασιών. Άλλωστε, η λειτουργία της διάταξης επεξεργαστών βασίζεται στην επαναληπτική εφαρμογή απλών πράξεων σε μεγάλες δομές δεδομένων. Αντιπροσωπευτικοί αλγόριθμοι είναι αυτοί που αποτελούνται από φωλιασμένους for βρόχους (for loops).

Σκοπός της διαδικασίας απεικόνισης είναι η ανάθεση των διαφόρων επαναλήψεων ενός επαναληπτικού αλγορίθμου σε διαφορετικά κελιά, έτσι ώστε να υπάρχει όσο το δυνατόν μεγαλύτερη ταυτόχρονη λειτουργία των κελιών και μικρότερος χρόνος εκτέλεσης του βρόχου. Με άλλα λόγια, να απεικονιστούν οι επιμέρους επαναλήψεις σε κελιά, έτσι ώστε τα γειτονικά κελιά να εκτελούν υπολογισμούς που ανταλλάσουν μεταξύ τους δεδομένα και ποτέ δύο υπολογισμοί που εκτελούνται στο ίδιο κελί να μη χρειάζεται να εκτελεστούν ταυτόχρονα.

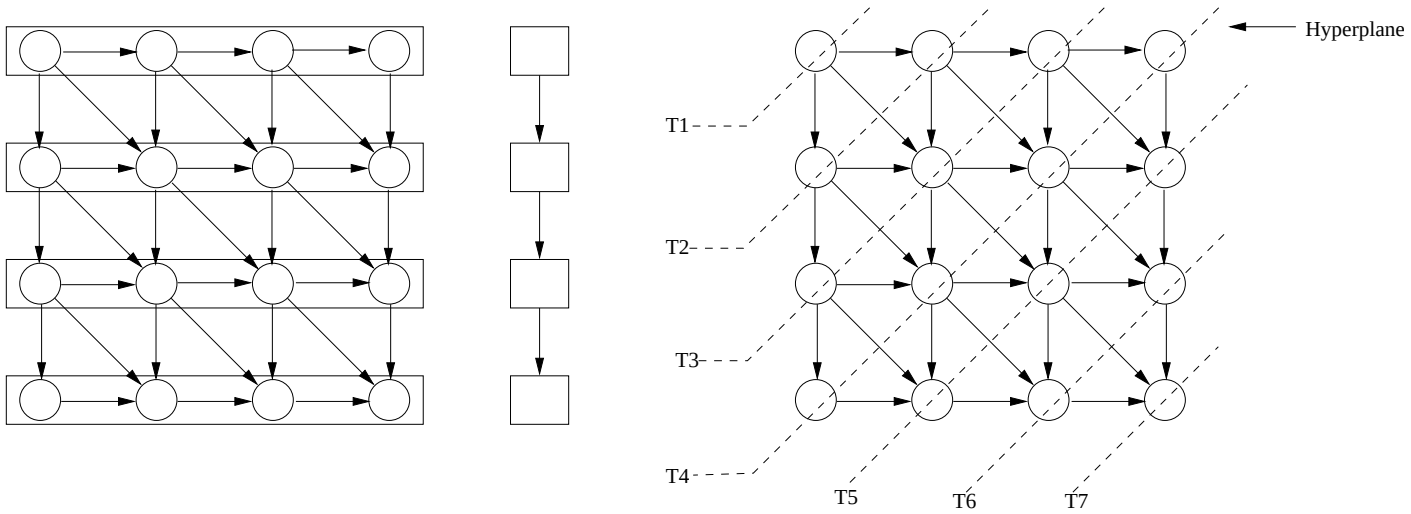
Συνεπώς, η μεθοδολογία μιας απεικόνισης αποτελείται από δύο στάδια: την εξαγωγή του γράφου εξάρτησης δεδομένων (data dependence graph) από τον αλγόριθμο και η απεικόνιση του γράφου εξάρτησης δεδομένων πάνω σε μια διάταξη επεξεργαστών. Παρόμοιοι μέθοδοι μπορούν να χρησιμοποιηθούν για την απεικόνιση αλγορίθμων σε διατάξεις επεξεργαστών [72, 121].

4.10.1 Εξαγωγή Γράφου Εξάρτησης από ένα Αλγόριθμο

Ένας γράφος εξάρτησης δεδομένων είναι ένας κατευθυνόμενος γράφος (directed graph) που παριστάνει τις εξαρτήσεις δεδομένων ενός αλγορίθμου. Ένας γράφος εξάρτησης αποτελείται από κόμβους (nodes) και ακμές (edges), όπου κάθε κόμβος παριστάνει μια πράξη ή ένα υπολογισμό και κάθε ακμή παριστάνει την εξάρτηση δεδομένων ανάμεσα στους υπολογισμούς. Για κανονικούς και επαναληπτικούς αλγορίθμους, ο γράφος εξάρτησης θα είναι επίσης κανονικός και μπορεί να παρασταθεί από ένα διδιάστατο πλέγμα. Το διδιάστατο πλέγμα ίσως περιέχει εξαρτήσεις εκπομπής (broadcast) όπως ένας κόμβος να μεταδίδει μια τιμή προς όλους τους κόμβους. Τέτοιο πλέγμα που περιέχει εξαρτήσεις εκπομπής πρέπει να μετασχηματίζεται σε τοπικό γράφο εξάρτησης (local dependence graph) ώστε να περιέχει μόνο τοπικές εξαρτήσεις ανάμεσα στους υπολογισμούς, αφού η διάταξη επεξεργαστών υποστηρίζει μόνο την επικοινωνία μεταξύ των γειτονικών κελιών. Έτσι, ο μετασχηματισμός του αρχικού γράφου σε τοπικό γράφο πραγματοποιείται αντικαθιστώντας τις εξαρτήσεις εκπομπής με τοπικές εξαρτήσεις.

4.10.2 Απεικόνιση του Γράφου Εξάρτησης σε Διατάξεις Επεξεργαστών

Για την απεικόνιση του τοπικού γράφου εξάρτησης ενός επαναληπτικού αλγορίθμου σε διατάξεις επεξεργαστών απαιτούνται δύο βήματα. Το πρώτο βήμα είναι η ανάθεση επεξεργαστών (processor assignment) και το δεύτερο βήμα είναι η ανάθεση χρονοδρομολόγησης (schedule assignment). Με άλλα λόγια, το πρώτο βήμα εξετάζει σε ποιον επεξεργαστή θα εκτελεστούν οι υπολογισμοί και το δεύτερο βήμα εξετάζει ποιοι υπολογισμοί μπορούν να εκτελεστούν ταυτόχρονα. Έτσι, για την ανάθεση επεξεργαστών χρησιμοποιούμε τη γραμμική προβολή στην οποία οι κόμβοι ενός γράφου που βρίσκονται στην ίδια γραμμή προβάλλονται ή ανατίθενται σε ένα επεξεργαστή της διάταξης επεξεργαστών, όπως φαίνεται στο Σχήμα 4.3 αριστερά. Για την ανάθεση χρονοδρομολόγησης χρησιμοποιούμε τη γραμμική χρονοδρομολόγηση στην οποία οι κόμβοι που βρίσκονται στο ίδιο υπερεπίπεδο (hyperplane) στο γράφο δρομολογούνται και επεξεργάζονται στην ίδια χρονική στιγμή, όπως φαίνεται στο Σχήμα 4.3 δεξιά.



Σχήμα 4.3: Αριστερά φαίνεται η γραμμική προβολή και δεξιά φαίνεται η γραμμική χρονοδρομολόγηση

4.11 Μέτρα Απόδοσης

Σε αυτή την ενότητα ορίζουμε ορισμένα κοινά μέτρα απόδοσης που αναφέρονται σε συστοιχίες από ομοιογενείς σταθμούς εργασίας.

Ο χρόνος T ενός παράλληλου αλγορίθμου ορίζεται να είναι η μέγιστη χρονική διάρκεια που μπορεί να μεσολαβήσει από την εκκίνηση του αλγορίθμου στον πρώτο σταθμό εργασίας που θα αρχίσει ως τον τερματισμό του αλγορίθμου στον τελευταίο επεξεργαστή που θα ολοκληρώσει την εκτέλεση του.

Η επιτάχυνση (speedup) S_p ενός παράλληλου αλγορίθμου που εκτελείται σε p σταθμούς εργασίας ορίζεται να είναι ο λόγος του χρόνου του ταχύτερου ακολουθιακού αλγορίθμου (όταν εκτελείται σε έναν από τους σταθμούς εργασίας) προς το χρόνο του παράλληλου αλγορίθμου στους p σταθμούς εργασίας και έχει ιδανική τιμή p. Έτσι, η επιτάχυνση ορίζεται από τον τύπο:

$$S_p = \frac{T_1}{T_p} \tag{4.1}$$

όπου T_1 και T_p είναι χρόνοι εκτέλεσης του ίδιου αλγορίθμου (που υλοποιείται για ακολουθιακή και παράλληλη εκτέλεση) σε 1 και p σταθμούς εργασίας αντίστοιχα.

Η αποδοτικότητα (efficiency) E_p ενός παράλληλου αλγορίθμου που εκτελείται σε p σταθμούς ερ-

γασίας ορίζεται να είναι ο λόγος της επιτάχυνσης S_p προς το πλήθος των σταθμών εργασίας p και έχει ιδανική τιμή 1, δηλαδή:

$$E_p = \frac{S_p}{p} \tag{4.2}$$

Η αποδοτικότητα E_p ορίζεται εναλλακτικά και ως το μέτρο του γινομένου επιφάνειας επί τον χρόνο,

$$AT_p = \frac{1T_1}{pT_p} = E_p \tag{4.3}$$

Επιπλέον το γινόμενο S_pE_p ορίζεται και ως AT_p^2, δηλαδή,

$$AT_p^2 = \frac{1T_1^2}{pT_p^2} = S_pE_p \tag{4.4}$$

Τέλος, η συνάρτηση κλιμάκωσης (scalability) δίνει το χρόνο εκτέλεσης του αλγορίθμου για διαφορετικά μεγέθη προβλήματος, καθώς μεταβάλλεται ο αριθμός των σταθμών εργασίας p ανάλογα με τη μεταβολή του μεγέθους του προβλήματος. Η συνάρτηση αυτή μετρά τη δυνατότητα του αλγορίθμου να κάνει χρήση περισσότερων σταθμών εργασίας σε περίπτωση που αυξηθεί το μέγεθος του προβλήματος.

Κεφάλαιο 5

Παράλληλες Υλοποιήσεις για Αναζήτηση Αλφαριθμητικών σε Ομοιογενές Σύστημα

5.1 Εισαγωγή

Σε αυτό τον κεφάλαιο παρουσιάζουμε τρεις παράλληλες υλοποιήσεις αλγορίθμων οι οποίοι λύνουν το πρόβλημα αναζήτησης αλφαριθμητικών χρησιμοποιώντας τη στατική και δυναμική διανομή των κειμένων. Επίσης, προτείνουμε μια καινούργια υβριδική παράλληλη μέθοδο αναζήτησης που συνδυάζει τα πλεονεκτήματα των μεθόδων στατικής και δυναμικής διανομής προκειμένου να μειώσει την ανισορροπία του φορτίου (load imbalance) και την επιβάρυνση επικοινωνίας (communication overhead). Οι παραπάνω μέθοδοι υλοποιούνται στην γλώσσα προγραμματισμού C σε συνδυασμό με την βιβλιοθήκη MPI (Message Passing Interface - Διεπαφή Περάσματος Μηνύματος) [162, 136, 152, 165] και εκτελούνται πάνω σε μια κατανεμημένη συστοιχία από ομοιογενείς σταθμούς εργασίας (cluster of homogeneous workstations). Τέλος, παρουσιάζουμε για πρώτη φορά ένα αναλυτικό μοντέλο πρόβλεψης απόδοσης που μπορεί να χρησιμοποιηθεί για να προβλέψουμε το χρόνο εκτέλεσης (execution time), την επιτάχυνση (speedup) και παρόμοια μέτρα απόδοσης για τις παράλληλες υλοποιήσεις αναζήτησης αλφαριθμητικών σε μια συστοιχία από ομοιογενείς σταθμούς εργασίας. Πρέπει να σημειωθεί εδώ ότι οι

παράλληλες υλοποιήσεις και το αναλυτικό μοντέλο πρόβλεψης απόδοσης που παρουσιάζονται παρακάτω αναφέρονται στο πρόβλημα προσεγγιστικής αναζήτησης αλφαριθμητικών, αφού το πρόβλημα αυτό είναι πιο γενικό από το πρόβλημα της απλής αναζήτησης αλφαριθμητικών όπως εξηγήσαμε στο πρώτο κεφάλαιο.

Το κεφάλαιο αυτό είναι οργανωμένο ως εξής: στην ενότητα 5.2 παρουσιάζεται μια σύντομη επισκόπηση σχετικά με υλοποιήσεις αναζήτησης αλφαριθμητικών σε διάφορες παράλληλες αρχιτεκτονικές. Στην ενότητα 5.3 παρουσιάζονται τέσσερις παράλληλες υλοποιήσεις αναζήτησης αλφαριθμητικών σε μια κατανεμημένη συστοιχία από ομοιογενείς σταθμούς εργασίας που βασίζονται στο προγραμματιστικό μοντέλο συντονιστής - εργαζόμενος (master-worker). Στην ενότητα 5.4 παρουσιάζονται πειραματικά αποτελέσματα των τεσσάρων παράλληλων υλοποιήσεων για τα προβλήματα απλής και προσεγγιστικής αναζήτησης αλφαριθμητικών. Στην ενότητα 5.5 προτείνουμε ένα αναλυτικό μοντέλο πρόβλεψης απόδοσης για τις προτεινόμενες παράλληλες υλοποιήσεις. Τέλος, στην ενότητα 5.6 παρουσιάζονται αναλυτικά αποτελέσματα από το θεωρητικό μοντέλο και τα συγκρίνουμε με τα πειραματικά αποτελέσματα.

5.2 Επισκόπηση

Παρόλο που δεν υπάρχει προηγούμενη δουλειά στην ανάπτυξη παράλληλων και κατανεμημένων υλοποιήσεων αναζήτησης αλφαριθμητικών κάτω από το περιβάλλον ομοιογενών σταθμών εργασίας, μερική από τη σχετική έρευνα στην περιοχή παράλληλης αναζήτησης αλφαριθμητικών είναι η ακόλουθη. Ο Cringean και άλλοι [63, 29, 62] παρουσίασαν μια παράλληλη υλοποίηση για το πρόβλημα απλής αναζήτησης αλφαριθμητικών που υλοποιήθηκε πάνω σε αρχιτεκτονική transputer. Η παράλληλη αυτή υλοποίηση βασίζεται στη δυναμική διανομή κειμένων με την βοήθεια του μοντέλου συντονιστής - εργαζόμενος. Επίσης, σε κάθε εργαζόμενο η υλοποίηση αυτή χρησιμοποιεί τον ακολουθιακό αλγόριθμο Boyer-Moore (BM) για την αναζήτηση.

Η Julich [65] έχει υλοποιήσει αλγορίθμους σύγκρισης ακολουθίας (sequence comparison) που είναι παραλλαγή του προβλήματος προσεγγιστικής αναζήτησης αλφαριθμητικών πάνω σε μια ποικιλία παράλληλων υπολογιστών, όπως η διαμοιραζόμενη αρχιτεκτονική Cray Y-MP 8/864 και οι δύο κατανεμημένες αρχιτεκτονικές Intel iPSC/860 και nCUBE.

Τέλος, ο Sittig και άλλοι [148] παρουσίασαν μια παραλληλοποίηση ενός αλγορίθμου ανάλυσης βιολογικής ακολουθίας (biological sequence analysis) πάνω στους υπολογιστές Sequent Symmetry και Intel Hypercube. Επίσης, ο Yap και άλλοι [171, 172] και [79] υλοποίησαν έναν αλγόριθμο σύγκρισης ακολουθίας πάνω σε ένα παράλληλο υπολογιστή Intel iPSC/860 και σε ομοιογενές κατανεμημένο σύστημα αντίστοιχα. Οι παραπάνω παράλληλες υλοποιήσεις βασίστηκαν στο δυναμικό μοντέλο συντονιστής - εργαζόμενος και σε κάθε εργαζόμενο χρησιμοποιούν τον αλγόριθμο δυναμικού προγραμματισμού των Smith και Waterman [151] όπως τροποποιήθηκε από τον Gotoh [49] για να συγκρίνουν δύο βιολογικές ακολουθίες.

5.3 Παράλληλες Υλοποιήσεις Αναζήτησης Αλφαριθμητικών

Ακολουθούμε το προγραμματιστικό μοντέλο συντονιστής - εργαζόμενος για να αναπτύξουμε παράλληλες και κατανεμημένες υλοποιήσεις αναζήτησης αλφαριθμητικών κάτω από την βιβλιοθήκη MPI [162, 136, 152, 165]. Αυτό το μοντέλο αποτελείται από ένα σταθμό εργασίας συντονιστή και μια συλλογή από σταθμούς εργασίας εργαζόμενους. Ο συντονιστής χρησιμοποιείται για να διαμερίσει μια συλλογή κειμένου σε ένα σύνολο από διάφορα μικρότερα υπο-κείμενα (subtext collections), διανέμει τα υπο-κείμενα σε όλους τους εργαζόμενους και τέλος συλλέγει τα τοπικά αποτελέσματα από τους εργαζόμενους. Οι εργαζόμενοι κυρίως εκτελούν ένα ακολουθιακό αλγόριθμο απλής ή προσεγγιστικής αναζήτησης αλφαριθμητικών πάνω στις αντίστοιχες συλλογές υπο-κειμένων τους [129, 100, 101, 110, 104]. Στις επόμενες ενότητες παρουσιάζονται οι στρατηγικές που βασίζονται στο στατικό και δυναμικό μοντέλο συντονιστής - εργαζόμενος.

5.3.1 MPI Στατικό Μοντέλο Συντονιστής - Εργαζόμενος

Υλοποίηση της Στατικής Διανομής Υπο-κειμένων

Προκειμένου να παρουσιάσουμε τη στατική υλοποίηση συντονιστής - εργαζόμενος, κάνουμε τις ακόλουθες παραδοχές: Πρώτον, οι σταθμοί εργασίας έχουν ένα αναγνωριστικό *myid* και αριθμούνται από 1 έως p, δεύτερον τα υπο-κείμενα της συλλογής μας είναι κατανεμημένα μεταξύ των διαφόρων σταθμών εργασίας και αποθηκεύονται στους τοπικούς δίσκους τους και τέλος το πρότυπο και ο αριθμός

των αποτυχιών ή διαφορών k αποθηκεύονται στην κύρια μνήμη σε όλους τους σταθμούς εργασίας. Η στρατηγική διαμερισμού αυτής της προσέγγισης είναι να διαμερίζει ολόκληρη την συλλογή κειμένου σε ένα αριθμό από υπο-κείμενα με βάση τον αριθμό των σταθμών εργασίας. Το μέγεθος του κάθε υπο-κειμένου είναι $\lceil \frac{n}{p} \rceil + m - 1$ διαδοχικοί χαρακτήρες της πλήρους συλλογής κειμένου. Υπάρχει μια επικάλυψη $m - 1$ χαρακτήρων προτύπου ανάμεσα σε διαδοχικά υπο-κείμενα δηλαδή, αποθηκεύονται επιπλέον $p(m - 1)$ χαρακτήρες. Συνεπώς, η στατική υλοποίηση συντονιστής - εργαζόμενος που ονομάζεται P1, αποτελείται από τέσσερις φάσεις. Στην πρώτη φάση, ο συντονιστής εκπέμπει (broadcast) το πρότυπο αλφαριθμητικό και τον αριθμό των αποτυχιών ή διαφορών k σε όλους τους εργαζόμενους. Στην δεύτερη φάση, κάθε εργαζόμενος διαβάζει την δικιά του συλλογή υπο-κειμένου από τον τοπικό δίσκο στην κύρια μνήμη. Στην τρίτη φάση, κάθε εργαζόμενος εκτελεί συγκρίσεις χαρακτήρων χρησιμοποιώντας ένα ακολουθιακό αλγόριθμο προσεγγιστικής αναζήτησης αλφαριθμητικών για να παράγει τον αριθμό των εμφανίσεων. Στην τέταρτη φάση, ο συντονιστής συλλέγει τον αριθμό των εμφανίσεων από κάθε εργαζόμενο. Η παραπάνω υλοποίηση κατασκευάστηκε έτσι ώστε να αντικαθιστούμε εύκολα εναλλακτικούς ακολουθιακούς αλγορίθμους προσεγγιστικής ή απλής αναζήτησης αλφαριθμητικών. Στο Σχήμα 5.1 παρουσιάζεται το πρόγραμμα για την παράλληλη υλοποίηση P1 με την βοήθεια των συναρτήσεων MPI. Σημειώνουμε εδώ ότι η κλήση της διαδικασίας **sel_search(pattern,text,m,n,k,&count)** καλεί τον αλγόριθμο της προσεγγιστικής αναζήτησης SEL. Με παρόμοια διαδικασία καλούμε τους υπόλοιπους αλγορίθμους απλά αλλάζοντας μόνο το πρόθεμα που αφορά στο όνομα του αλγορίθμου. Για παράδειγμα, αν θέλουμε να καλέσουμε τον αλγόριθμο WM τότε καλούμε την διαδικασία **wm_search(pattern,text,m,n,k,&count)**.

Σχήμα 5.1: Κώδικας της παράλληλης υλοποίησης P1

```
#include <stdio.h>
#include "mpi.h"
#define MAXLEN_PAT 100
#define MAXLEN_FILENAME 256
#define MASTER 0
#define MAXLEN_TEXT 1000000
main(int argc, char *argv[])
{
       int my_rank, /* rank of process */
           num_procs, /* number of processes */
           k, /* number of errors */
           n, /* length of local text */
```

```
    m, /* length of pattern */
    count, /* local number of occurrences */
    total_matches; /* global number of occurrences */
char *algorithm /* name of algorithm */
      pattern[MAXLEN_PAT], /* pattern */
      *filename_text, /* name of text file */
      tmp[MAXLEN_FILENAME],
      *text; /* local text */
FILE *text_file_ptr; /* pointer for local text file */

/* Start up MPI */
MPI_Init(&argc,&argv);

/* Find out number of process */
MPI_Comm_size(MPI_COMM_WORLD,&num_procs);

/* Find out process rank */
MPI_Comm_rank(MPI_COMM_WORLD,&my_rank);

/* arguments of command line */
strcpy(algorithm,argv[1]);
strcpy(pattern,argv[2]);
strcpy(filename_text,argv[3]);
k=atoi(argv[4]);

if (my_rank==MASTER) { /* master code */
      m=strlen(pattern);
      /* broadcast the pattern and the number of errors to workers */
      MPI_Bcast(&pattern,MAXLEN_PAT,MPI_CHAR,MASTER,MPI_COMM_WORLD);
      MPI_Bcast(&k,1,MPI_INT,MASTER,MPI_COMM_WORLD);

      /* gather the local results from workers */
      total_matches=0;count=0;
    MPI_Reduce(&count,&total_matches,1,MPI_INT,MPI_SUM,MASTER,MPI_COMM_WORLD);
      printf("total=%d \n",total_matches);
}
else { /* worker code */
      /* receive the pattern and the number of errors from master */
      MPI_Bcast(&pattern,MAXLEN_PAT,MPI_CHAR,MASTER,MPI_COMM_WORLD);
      MPI_Bcast(&k,1,MPI_INT,MASTER,MPI_COMM_WORLD);

      /* reading local text file */
      sprintf(tmp,"%s.%d",filename_text,my_rank);
      text_file_ptr=fopen(tmp,"r");
      text=(char *) malloc(MAXLEN_TEXT);
      n=0;
      n+=fread(text,sizeof(char),MAXLEN_TEXT,text_file_ptr);
      fclose(text_file_ptr);
```

```
            /* searching of pattern in local text file */
            n=strlen(text);m=strlen(pattern)-1;
            sel_search(pattern,text,m,n,k,&count);
            free(text);

            /* gather the local resuls to master */
            MPI_Reduce(&count,&total_matches,1,MPI_INT,MPI_SUM,MASTER,MPI_COMM_WORLD);
        }
        /* shut down MPI */
        MPI_Finalize();
}
```

Το πλεονέκτημα της στατικής υλοποίησης είναι η χαμηλή επιβάρυνση επικοινωνίας, διότι εκ των προτέρων αναθέτουμε σε κάθε εργαζόμενο να αναζητήσει το δικό του υπο-κείμενο χωρίς να επικοινωνεί με τους άλλους εργαζόμενους. Όμως, το βασικό μειονέκτημα είναι η πιθανή ανισορροπία φορτίου εξαιτίας της φτωχής τεχνικής διαμερισμού κειμένου.

5.3.2 MPI Δυναμικό Μοντέλο Συντονιστής - Εργαζόμενος

Σε αυτή την υποενότητα, υλοποιούμε δύο εκδόσεις του δυναμικού μοντέλου συντονιστής - εργαζόμενος. Η πρώτη έκδοση βασίζεται στην δυναμική διανομή υπο-κειμένων και την δεύτερη έκδοση βασίζεται στην δυναμική διανομή δεικτών κειμένου.

Υλοποίηση της Δυναμικής Διανομής Υπο-κειμένων

Η δυναμική στρατηγική συντονιστής - εργαζόμενος που υιοθετήσαμε είναι μια γνωστή στρατηγική παραλληλοποίησης και είναι δημοφιλής ως "φάρμα σταθμών εργασίας ". Κάνουμε την ακόλουθη παραδοχή: ολόκληρη συλλογή κειμένου αποθηκεύεται στον τοπικό δίσκο του συντονιστή. Η δυναμική υλοποίηση συντονιστής - εργαζόμενος που ονομάζεται P2, αποτελείται από έξι φάσεις. Στην πρώτη φάση, ο συντονιστής εκπέμπει το πρότυπο αλφαριθμητικό και τον αριθμό των αποτυχιών ή διαφορών k σε όλους τους εγαζόμενους. Στην δεύτερη φάση, ο συντονιστής διαβάζει από τον τοπικό δίσκο του μικρά τμήματα (block) από την συλλογή κειμένου. Το μέγεθος του κάθε τμήματος είναι $sb + m - 1$ διαδοχικοί χαρακτήρες όπου sb είναι το βέλτιστο μέγεθος τμήματος. Το μέγεθος του τμήματος είναι

μια σημαντική παράμετρος που σχετίζεται άμεσα με τους παραμέτρους Ε/Ε (Εισόδου/Εξόδου) και επικοινωνίας. Επιλέξαμε διάφορα μεγέθη τμήματος προκειμένου να βρούμε την καλύτερη απόδοση όπως παρουσιάζεται στην ενότητα 5.4.2. Στην τρίτη φάση, ο συντονιστής διανέμει τα πρώτα τμήματα της συλλογής κειμένου στους αντίστοιχους εργαζόμενους. Στην τέταρτη φάση, κάθε εργαζόμενος εκτελεί ένα ακολουθιακό αλγόριθμο απλής ή προσεγγιστικής αναζήτησης αλφαριθμητικών ανάμεσα στο αντίστοιχο τμήμα κειμένου και πρότυπο για να παράγει τον αριθμό των εμφανίσεων. Στην πέμπτη φάση, κάθε εργαζόμενος αποστέλει τον αριθμό των εμφανίσεων πίσω στο συντονιστή. Στην έκτη φάση, αν υπάρχουν ακόμη τμήματα από την συλλογή κειμένου, ο συντονιστής διαβάζει και διανέμει τα επόμενα τμήματα της συλλογής κειμένου στους εργαζόμενους και επαναλαμβάνει ξανά την τέταρτη έως και την έκτη φάση. Στο Σχήμα 5.2 παρουσιάζεται το πρόγραμμα για την παράλληλη υλοποίηση P2.

Σχήμα 5.2: Κώδικας της παράλληλης υλοποίησης P2

```
#include <stdio.h>
#include "mpi.h"
#define MAXLEN_PAT 100
#define MAXLEN_FILENAME 256
#define MASTER 0
#define EOS '\0'
#define WORKTAG 1
#define DIETAG 2
#define TRUE 1
#define MAXLEN_TEXT 1000000
main(int argc, char *argv[])
{
       int my_rank, /* rank of process */
           num_procs, /* number of processes */
           k, /* number of errors */
           n, /* length of local text */
           m, /* length of pattern */
           i,
           count, /* local number of occurrences */
           total_matches, /* global number of occurrences */
           siz, nb,
           sender, /* sender */
           active; /* number of active processes */
       long block_size, /* size of block */
             offset; /* current text file pointer */
       char *algorithm /* name of algorithm */
             pattern[MAXLEN_PAT], /* pattern */
```

```
      *filename_text, /* name of text file */
      tmp[MAXLEN_FILENAME],
      *text; /* local text */
FILE *text_file_ptr; /* pointer for local text file */
MPI_Status status;

/* Start up MPI */
MPI_Init(&argc,&argv);

/* Find out number of process */
MPI_Comm_size(MPI_COMM_WORLD,&num_procs);

/* Find out process rank */
MPI_Comm_rank(MPI_COMM_WORLD,&my_rank);

/* arguments of command line */
strcpy(algorithm,argv[1]);
strcpy(pattern,argv[2]);
strcpy(filename_text,argv[3]);
k=atoi(argv[4]);
block_size=atoi(argv[5]);

if (my_rank==MASTER) { /* master code */
      /* broadcast the pattern and the number of errors to workers */
      MPI_Bcast(&pattern,MAXLEN_PAT,MPI_CHAR,MASTER,MPI_COMM_WORLD);
      MPI_Bcast(&k,1,MPI_INT,MASTER,MPI_COMM_WORLD);

      /* Initialize the number of matches and active processes */
      total_matches=active=0;
      offset=0L; /* Initialize the current text file pointer */
      m=strlen(pattern)-1;
      text_file_ptr=fopen(filename_text,"r");
      /* Send the text to all workers */
      for(i=1;i<=num_procs-1;i++) {
          nb=0;
          fseek(text_file_ptr,offset,SEEK_SET);
          siz=fread(text+nb,1,block_size-nb,text_file_ptr);
          nb+=siz;
          text[nb]=EOS;
          MPI_Send(text,strlen(text),MPI_CHAR,i,WORKTAG,MPI_COMM_WORLD);
          active++; /* Increment the number of active processes */
          /* calculation of next text file pointer with overlap */
          offset+=block_size-m+1;
     }
```

```
            /* Receive the number of local matches from any worker and sends
            the next text */
            do {
                MPI_Recv(&count,1,MPI_INT,MPI_ANY_SOURCE,MPI_ANY_TAG,MPI_COMM_WORLD,
&status);
                active--; /* Decrement the number of active processes */
                /* Determine who sent the result */
                sender=(status.MPI_SOURCE);
                /* Update the total number of matches */
                total_matches+=count;
                /* Check if there are other chunks of text */
                if (offset<(13*MAXLEN_TEXT)) {
                    nb=0;
                    fseek(text_file_ptr,offset,SEEK_SET);
                    siz=fread(text+nb,1,block_size-nb,text_file_ptr);
                    nb+=siz;
                    text[nb]=EOS;
                    MPI_Send(text,strlen(text),MPI_CHAR,sender,WORKTAG,MPI_COMM_WORLD);
                    active++; /* Increment the mumber of active processes */
                    /* calculation of next text file pointer with overlap */
                    offset+=block_size-m+1;
                }
                else
                    /* Tell the worker to exit or die */
                    MPI_Send(0,0,MPI_LONG,sender,DIETAG,MPI_COMM_WORLD);
            }
            while (active>0);
            fclose(text_file_ptr);
            printf("Total matches=%d \n",total_matches);
        }
        else { /* worker code */
            /* Receives the pattern and the number of errors from master */
            MPI_Bcast(&pattern,MAXLEN_PAT,MPI_CHAR,MASTER,MPI_COMM_WORLD);
            MPI_Bcast(&k,1,MPI_INT,MASTER,MPI_COMM_WORLD);

            while(TRUE) {
                /* Receive the current text */
                MPI_Recv(text,MAXLEN_TEXT,MPI_CHAR,MASTER,MPI_ANY_TAG,MPI_COMM_WORLD,&status);
                /* Check the tag of the received message */
                if (status.MPI_TAG==DIETAG) break;

                /* Searching of the pattern in the own portion text */
                n=strlen(text);m=strlen(pattern)-1;
                sel_search(pattern,text,m,n,k,&count);
```

```
                /* Send the number of matches to the master */
                MPI_Send(&count,1,MPI_INT,MASTER,0,MPI_COMM_WORLD);
            }
        }
        /* Shut down MPI */
        MPI_Finalize();
}
```

Το πλεονέκτημα της δυναμικής προσέγγισης είναι η χαμηλή ανισορροπία φορτίου, ενώ το μειονέκτημα είναι η υψηλή επιβάρυνση επικοινωνίας.

Υλοποίηση της Δυναμικής Διανομής Δεικτών Κειμένου

Για την δυναμική υλοποίηση με δείκτες κειμένου, κάνουμε τις ακόλουθες παραδοχές: Πρώτον, ολόκληρη συλλογή κειμένου αποθηκεύεται στους τοπικούς δίσκους όλων των σταθμών εργασίας και δεύτερον, ο συντονιστής έχει ένα δείκτη κειμένου που δείχνει την τρέχουσα θέση μέσα στην συλλογή κειμένου. Η δυναμική διανομή δεικτών κειμένου που ονομάζεται P3 αποτελείται από έξι φάσεις. Στην πρώτη φάση, ο συντονιστής εκπέμπει το πρότυπο αλφαριθμητικό και τον αριθμό των αποτυχιών ή διαφορών k σε όλους τους εργαζόμενους. Στην δεύτερη φάση, ο συντονιστής διανέμει τους πρώτους δείκτες κειμένου στους αντίστοιχους εργαζόμενους. Στην τρίτη φάση, κάθε εργαζόμενος διαβάζει από το τοπικό δίσκο του $sb+m-1$ χαρακτήρες από την συλλογή κειμένου αρχίζοντας από το δείκτη που έλαβε. Στην τέταρτη φάση, κάθε εργαζόμενος εκτελεί έναν ακολουθιακό αλγόριθμο απλής ή προσεγγιστικής αναζήτησης αλφαριθμητικών ανάμεσα στο αντίστοιχο τμήμα κειμένου και το πρότυπο για να παράγει τον αριθμό των εμφανίσεων. Στην πέμπτη φάση, κάθε εργαζόμενος αποστέλει το αποτέλεσμα δηλαδή τον αριθμό των εμφανίσεων πίσω στο συντονιστή. Στην έκτη φάση, αν ο δείκτης κειμένου δεν έχει φτάσει στο τέλος της συλλογής κειμένου, τότε ο συντονιστής ενημερώνει τους δείκτες κειμένου για τη θέση των επόμενων τμημάτων κειμένου, διανέμει τους δείκτες στους εργαζόμενους και επαναλαμβάνει ξανά την τρίτη έως και την έκτη φάση. Στο Σχήμα 5.3 παρουσιάζεται το πρόγραμμα για την παράλληλη υλοποίηση P3.

Σχήμα 5.3: Κώδικας της παράλληλης υλοποίησης P3

```
#include <stdio.h>
#include "mpi.h"
#define MAXLEN_PAT 100
#define MAXLEN_FILENAME 256
#define MASTER 0
#define EOS '\0'
#define WORKTAG 1
#define DIETAG 2
#define TRUE 1
#define MAXLEN_TEXT 1000000
main(int argc, char *argv[])
{
       int my_rank, /* rank of process */
           num_procs, /* number of processes */
           k, /* number of errors */
           n, /* length of local text */
           m, /* length of pattern */
           i,
           count, /* local number of occurrences */
           total_matches, /* global number of occurrences */
           siz, nb,
           sender, /* sender */
           active; /* number of active processes */
       long block_size, /* size of block */
             offset; /* current text file pointer */
       char *algorithm /* name of algorithm */
             pattern[MAXLEN_PAT], /* pattern */
             *filename_text, /* name of text file */
             tmp[MAXLEN_FILENAME],
             *text; /* local text */
       FILE *text_file_ptr; /* pointer for local text file */
       MPI_Status status;

       /* Start up MPI */
       MPI_Init(&argc,&argv);

       /* Find out number of process */
       MPI_Comm_size(MPI_COMM_WORLD,&num_procs);

       /* Find out process rank */
       MPI_Comm_rank(MPI_COMM_WORLD,&my_rank);

       /* arguments of command line */
       strcpy(algorithm,argv[1]);
       strcpy(pattern,argv[2]);
```

```
        strcpy(filename_text,argv[3]);
        k=atoi(argv[4]);
        block_size=atoi(argv[5]);

        if (my_rank==MASTER) { /* master code */
              /* broadcast the pattern and the number of errors to workers */
              MPI_Bcast(&pattern,MAXLEN_PAT,MPI_CHAR,MASTER,MPI_COMM_WORLD);
              MPI_Bcast(&k,1,MPI_INT,MASTER,MPI_COMM_WORLD);

              /* Initialize the number of matches and active processes */
              total_matches=active=0;
              offset=0L; /* Initialize the current text file pointer */
              m=strlen(pattern)-1;
              /* Send the text to all workers */
              for(i=1;i<=num_procs-1;i++) {
                  MPI_Send(&offset,1,MPI_LONG,i,WORKTAG,MPI_COMM_WORLD);
                  active++; /* Increment the number of active processes */
                  /* calculation of next text file pointer with overlap */
                  offset+=block_size-m+1;
             }
              /* Receive the number of local matches from any worker and sends
              the next text file pointer*/
              do {
                  MPI_Recv(&count,1,MPI_INT,MPI_ANY_SOURCE,MPI_ANY_TAG,MPI_COMM_WORLD,
&status);
                  active--; /* Decrement the number of active processes */
                  /* Determine who sent the result */
                  sender=(status.MPI_SOURCE);
                  /* Update the total number of matches */
                  total_matches+=count;
                  /* Check if there are other chunks of text */
                  if (offset<(13*MAXLEN_TEXT)) {
                      MPI_Send(&offset,1,MPI_LONG,sender,WORKTAG,MPI_COMM_WORLD);
                      active++; /* Increment the mumber of active processes */
                      /* calculation of next text file pointer with overlap */
                      offset+=block_size-m+1;
                 }
                  else
                      /* Tell the worker to exit or die */
                      MPI_Send(0,0,MPI_LONG,sender,DIETAG,MPI_COMM_WORLD);
             }
              while (active>0);
              printf("Total matches=%d \n",total_matches);
        }
        else { /* worker code */
```

```
            /* Receives the pattern and the number of errors from master */
            MPI_Bcast(&pattern,MAXLEN_PAT,MPI_CHAR,MASTER,MPI_COMM_WORLD);
            MPI_Bcast(&k,1,MPI_INT,MASTER,MPI_COMM_WORLD);

            while(TRUE) {
                /* Receive the current text file pointer */
                MPI_Recv(&offset,1,MPI_LONG,MASTER,MPI_ANY_TAG,MPI_COMM_WORLD,&status);
                /* Check the tag of the received message */
                if (status.MPI_TAG==DIETAG) break;

                /* Reading of block text file */
                nb=0;
                text_file_ptr=fopen(filename_text,"r");
                fseek(text_file_ptr,offset,SEEK_SET);
                siz=fread(text+nb,1,block_size-nb,text_file_ptr);
                nb+=siz;
                text[nb]=EOS;
                fclose(text_file_ptr);

                /* Searching of the pattern in the own portion text */
                n=strlen(text);m=strlen(pattern)-1;
                sel_search(pattern,text,m,n,k,&count);

                /* Send the number of matches to the master */
                MPI_Send(&count,1,MPI_INT,MASTER,0,MPI_COMM_WORLD);
            }
        }
        /* Shut down MPI */
        MPI_Finalize();
}
```

Το πλεονέκτημα της παραπάνω υλοποίησης είναι ότι μειώνει την επιβάρυνση της επικοινωνίας αφού σε κάθε σταθμό εργασίας υπάρχει ένα όλοκληρο αντίγραφο της συλλογής κειμένου στον τοπικό δίσκο. Η υλοποίηση απαιτεί επιπλέον τοπικό χώρο (ή δίσκο) αλλά το μέγεθος του τοπικού δίσκου στις παράλληλες και κατανεμημένες αρχιτεκτονικές είναι αρκετά μεγάλο.

5.3.3 MPI Υβριδικό Μοντέλο Συντονιστής - Εργαζόμενος

Γνωρίζουμε ότι η στατική υλοποίηση έχει το μειονέκτημα της υψηλής ανισορροπίας φορτίου από την άνιση διανομή της συλλογής κειμένου και η δυναμική υλοποίηση έχει το μειονέκτημα της υψηλής επιβά-

ρυνσης επικοινωνίας. Τα προβλήματα αυτά μπορούν να εξαλειφθούν από μια μέθοδο προεπεξεργασίας διανομής κειμένου, όπου τα υπο-κείμενα τοποθετούνται σε ένα από τους p σταθμούς εργασίας έτσι ώστε η διαφορά ανάμεσα στο μέγεθος του υπο-κειμένου στο μικρότερο και μεγαλύτερο σταθμό εργασίας να ελαχιστοποιείται. Στην επόμενη υποενότητα, περιγράφουμε τη μέθοδο προεπεξεργασίας διανομής κειμένου.

Μέθοδος Προεπεξεργασίας Διανομής Κειμένου

Η διαδικασία της τοποθέτησης των υπο-κειμένων στους σταθμούς εργασίας περιγράφεται ως εξής: Πρώτον, ολόκληρη η συλλογή κειμένου διαμερίζεται σε πολλά μικρότερα υπο-κείμενα όπως στο δυναμικό μοντέλο συντονιστής - εργαζόμενος. Δεύτερον, τα υπο-κείμενα ταξινομούνται σε φθίνουσα σειρά μεγέθους. Τότε αρχίζοντας από το μεγαλύτερο μέγεθος, κάθε υπο-κείμενο τοποθετείται στο σταθμό εργασίας που έχει το τρέχον μικρότερο άθροισμα μεγέθους των υπο-κειμένων. Σε περίπτωση ισοψηφίας, επιλέγεται ο μικρότερος σταθμός εργασίας. Μετά την εκτέλεση της φάσης προεπεξεργασίας διανομής κειμένου ακολουθεί η στατική παράλληλη υλοποίηση P1. Συνεπώς, έχουμε μια υβριδική παράλληλη υλοποίηση που εξασφαλίζει το χαμηλό κόστος επικοινωνίας και την υψηλή εξισορρόπηση φορτίου. Αυτή η υλοποίηση ονομάζεται P4. Ο κώδικας της υλοποίησης P4 είναι παρόμοιος με το κώδικα του Σχήματος 5.1.

Στον Πίνακα 5.1 παρουσιάζεται περιληπτικά η σύγκριση των τεσσάρων υλοποιήσεων.

Πίνακας 5.1: Σύγκριση των παράλληλων υλοποιήσεων

	P1	P2	P3	P4
Επικοινωνία Συντονιστής - Εργαζόμενος	Χαμηλή	Υψηλή	Μεσαία	Χαμηλή
Επικοινωνία Εργαζόμενος - Εργαζόμενος	Όχι	Όχι	Όχι	Όχι
Ισορροπία φορτίου	Πολύ Χαμηλή	Υψηλή	Υψηλή	Υψηλή
Πολυπλοκότητα υλοποίησης	Χαμηλή	Χαμηλή	Χαμηλή	Χαμηλή
Μεταφερσιμότητα	Π. Καλή	Π. Καλή	Π. Καλή	Π. Καλή
Απαιτήσεις δίσκου	Μεσαίες	Χαμηλές	Υψηλές	Μεσαίες

5.4 Πειραματικά Αποτελέσματα

Σε αυτή την ενότητα παρουσιάζουμε τα πειραματικά αποτελέσματα για την απόδοση των τεσσάρων παράλληλων υλοποιήσεων για τα προβλήματα απλής και προσεγγιστικής αναζήτησης αλφαριθμητικών. Οι παραπάνω παράλληλοι μέθοδοι υλοποιήθηκαν σε γλώσσα προγραμματισμού ANSI C [67] χρησιμοποιώντας την βιβλιοθήκη MPI [162, 136, 152, 165] για τις πράξεις επικοινωνίας από σημείο σε σημείο (point-to-point) και συλλογικής (collective).

5.4.1 Πειραματικό Περιβάλλον

Η πλατφόρμα για την πειραματική μας μελέτη είναι μια συστοιχία υπολογιστών συνδεδεμένη σε δίκτυο 100 Mb/s Fast Ethernet. Συγκεριμένα, η συστοιχία αποτελείται από εννέα ομοιογενείς σταθμούς εργασίας των 100 MHz Intel Pentium με 64MB μνήμη. Η υλοποίηση MPI που χρησιμοποιήθηκε στο δίκτυο είναι η MPICH έκδοση 1.2. Κατά την διάρκεια των πειραμάτων, το ομοιογενές σύστημα υπολογιστών ήταν αποκλειστικό (dedicated). Τέλος, για να πάρουμε αξιόπιστα αποτελέσματα απόδοσης τρέξαμε το κάθε πείραμα δέκα φορές και αναφέρουμε στα αποτελέσματα τον μέσο όρο των δέκα εκτελέσεων. Η δοκιμαστική αδόμητη βάση κειμένου που χρησιμοποιήσαμε προήλθε από διάφορες ιστοσελίδες του διαδικτύου.

5.4.2 Πειραματικά Αποτελέσματα

Πριν παρουσιάσουμε τα αποτελέσματα απόδοσης των τεσσάρων παράλληλων υλοποιήσεων για τα προβλήματα της απλής ή προσεγγιστικής αναζήτησης αλφαριθμητικών, παρουσιάζουμε τα αποτελέσματα της εκτέλεσης ενός πειράματος για την επίδραση του μεγέθους τμήματος (block size) για τις δύο δυναμικές μεθόδους συντονιστής - εργαζόμενος. Επιλέξαμε διάφορες τιμές τμήματος για τις δύο δυναμικές υλοποιήσεις P2 και P3 και προσδιορίσαμε ότι ένα μέγεθος τμήματος κοντά στους $sb = 100,000$ χαρακτήρες παρέχει βέλτιστη απόδοση. Τα παρακάτω πειράματα εκτελέστηκαν χρησιμοποιώντας την παραπάνω βέλτιστη τιμή τμήματος για τις υλοποιήσεις P2 και P3. Από το παραπάνω πείραμα παρατηρήσαμε ότι η χειρότερη απόδοση προκύπτει για τις πολύ μικρές και μεγάλες τιμές του τμήματος. Αυτό συμβαίνει επειδή οι μικρές τιμές του τμήματος αυξάνουν τη δια-επεξεργαστική επικοινωνία (inter-

Πίνακας 5.2: Πειραματικοί χρόνοι εκτέλεσης (σε δευτερόλεπτα) για μεγέθος κειμένου 3MB και για διαφορετικά μήκη προτύπων χρησιμοποιώντας τον αλγόριθμο BF

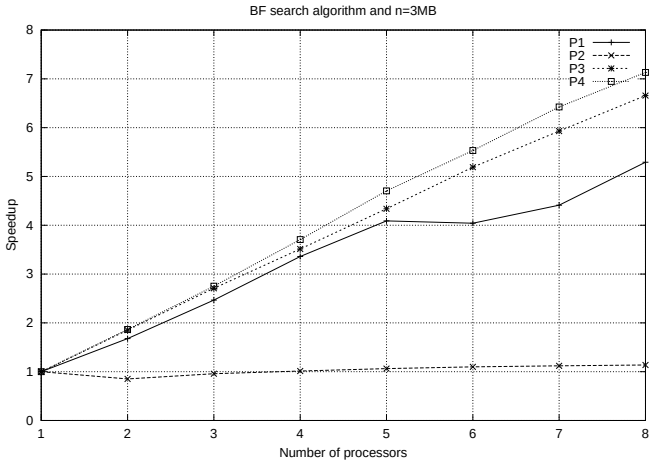
m/p	Παρ. Υλοπ.	1	2	3	4	5	6	7	8
5	P1	1.155	0.707	0.487	0.356	0.276	0.298	0.264	0.230
	P2	1.155	1.355	1.207	1.142	1.090	1.055	1.035	1.020
	P3	1.155	0.623	0.426	0.326	0.266	0.218	0.196	0.175
	P4	1.155	0.622	0.419	0.312	0.252	0.203	0.181	0.160
10	P1	1.111	0.659	0.445	0.326	0.274	0.271	0.246	0.202
	P2	1.112	1.318	1.173	1.108	1.057	1.023	1.003	0.989
	P3	1.112	0.602	0.411	0.319	0.253	0.216	0.191	0.169
	P4	1.111	0.596	0.405	0.301	0.247	0.205	0.183	0.160
30	P1	1.087	0.640	0.434	0.319	0.269	0.266	0.226	0.203
	P2	1.087	1.307	1.165	1.102	1.053	1.019	1.001	0.986
	P3	1.087	0.590	0.403	0.309	0.257	0.217	0.186	0.167
	P4	1.087	0.584	0.396	0.296	0.242	0.197	0.173	0.155
60	P1	1.182	0.698	0.473	0.349	0.290	0.287	0.292	0.222
	P2	1.183	1.353	1.195	1.125	1.070	1.033	1.013	0.996
	P3	1.183	0.637	0.435	0.328	0.270	0.223	0.193	0.171
	P4	1.182	0.631	0.428	0.318	0.266	0.212	0.179	0.161

processor communication), ενώ οι μεγάλες τιμές του τμήματος παρέχουν ανισορροπία του φορτίου.

Σύγκριση Τεσσάρων Παράλληλων Υλοποιήσεων

Ο Πίνακας 5.2 δείχνει τους χρόνους εκτέλεσης για τέσσερα διαφορετικά μήκη προτύπων, για διαφορετικό αριθμό σταθμών εργασίας και για τον αλγόριθμο απλής αναζήτησης αλφαριθμητικών BF. Επίσης, το Σχήμα 5.4 παρουσιάζει επιταχύνσεις σε σχέση με το αριθμό των σταθμών εργασίας. Είναι σημαντικό να σημειωθεί ότι οι επιταχύνσεις που απεικονίζονται στο Σχήμα 5.4 είναι αποτέλεσμα του μέσου όρου για τέσσερα διαφορετικά μήκη προτύπων.

Όπως αναμέναμε, τα αποτελέσματα απόδοσης δείχνουν ότι η υλοποίηση P2 χρησιμοποιώντας τη δυναμική διανομή υπο-κειμένων είναι λιγότερο αποτελεσματική από τις άλλες τρεις υλοποιήσεις. Η υλοποίηση P2 δίνει μεγαλύτερο χρόνο εκτέλεσης και χαμηλότερη επιτάχυνση. Όταν ο αριθμός των σταθμών εργασίας αυξάνει από ένα σε δύο, ο χρόνος εκτέλεσης για $m = 10$ με τον αλγόριθμο BF αυξάνει από 1.112 σε 1.318 δευτερόλεπτα και η επιτάχυνση μειώνεται σε 80%-90%.



Σχήμα 5.4: Επιτάχυνση των παράλληλων υλοποιήσεων αναζήτησης αλφαριθμητικών σε συνάρτηση με τον αριθμό των σταθμών εργασίας για μέγεθος κειμένου 3MB χρησιμοποιώντας τον αλγόριθμο BF

Επίσης, η υλοποίηση P1 χρησιμοποιώντας τη στατική διανομή υπο-κειμένων παρέχει καλύτερα αποτελέσματα από την υλοποίηση P2. Όμως, όταν ο αριθμός των σταθμών εργασίας αυξάνεται από ένα σε δύο, ο χρόνος εκτέλεσης δεν μειώνεται αντιστρόφως ανάλογα. Επίσης, η επιτάχυνση αυξάνεται όταν ο αριθμός των σταθμών εργασίας αυξάνεται μέχρι πέντε σταθμούς εργασίας και αυξάνεται ελαφρώς μετά από αυτό το σημείο. Το φαινόμενο αυτό οφείλεται την ανισορροπία του φορτίου εξαιτίας της φτωχής στρατηγικής διαμερισμού του κειμένου. Μερικοί σταθμοί εργασίας που έχουν λιγότερα υπο-κείμενα για να εκτελέσουν αναζήτηση αλφαριθμητικών θα τελειώσουν τους υπολογισμούς τους πιο γρήγορα από τους άλλους και στην συνέχεια θα παραμείνουν αδρανείς (idle).

Τέλος, τα πειραματικά αποτελέσματα δείχνουν ότι οι υλοποιήσεις P3 και P4 φαίνονται να έχουν την καλύτερη απόδοση σε σύγκριση με τις άλλες υλοποιήσεις. Συγκεριμένα, η προσέγγιση P4 παρέχει σχεδόν βέλτιστη απόδοση. Από τα αποτελέσματα, μπορούμε να δούμε μια σαφή μείωση του χρόνου εκτέλεσης του αλγορίθμου όταν χρησιμοποιούμε τις παράλληλες υλοποιήσεις P3 και P4. Για παράδειγμα, για μέγεθος κειμένου 3MB, για $m = 10$ και για τον αλγόριθμο BF, μειώνεται ο χρόνος υπολογισμού από 1.112 δευτερόλεπτα στην ακολουθιακή έκδοση σε 0.169 και σε 0.160 δευτερόλεπτα στις παράλληλες εκδόσεις P3 και P4 αντίστοιχα, χρησιμοποιώντας τους οκτώ σταθμούς εργασίας. Με άλλα λόγια, παρατηρούμε ότι για σταθερό μέγεθος κειμένου υπάρχει μια αναμενόμενη αντίστροφη σχέση ανάμεσα στον παράλληλο χρόνο εκτέλεσης και στον αριθμό των σταθμών εργασίας. Επιπλέον,

Πίνακας 5.3: Πειραματικοί χρόνοι εκτέλεσης (σε δευτερόλεπτα) για μεγέθος κειμένου 24MB και για διαφορετικά μήκη προτύπων χρησιμοποιώντας τον αλγόριθμο BF

m/p	Παρ. Υλοπ.	1	2	3	4	5	6	7	8
5	P3	9.237	4.785	3.253	2.480	2.017	1.701	1.486	1.321
	P4	9.237	4.642	3.095	2.334	1.875	1.558	1.331	1.165
10	P3	8.724	4.613	3.138	2.398	1.944	1.648	1.438	1.277
	P4	8.724	4.460	2.969	2.260	1.801	1.492	1.281	1.130
30	P3	8.513	4.516	3.073	2.356	1.914	1.621	1.409	1.254
	P4	8.513	4.375	2.926	2.225	1.785	1.472	1.263	1.095
60	P3	9.284	4.892	3.324	2.530	2.059	1.736	1.510	1.340
	P4	9.284	4.769	3.201	2.421	1.962	1.631	1.370	1.198

οι μέθοδοι P3 και P4 πραγματοποιούν σχεδόν τέλειες καμπύλες επιταχύνσεις για όλα τα μήκη προτύπων και για όλους τους σταθμούς εργασίας. Συνεπώς, περισσότερος χρόνος καταναλώνεται στην αναζήτηση αλφαριθμητικών παρά στην επικοινωνία με τον σταθμό εργασίας συντονιστής.

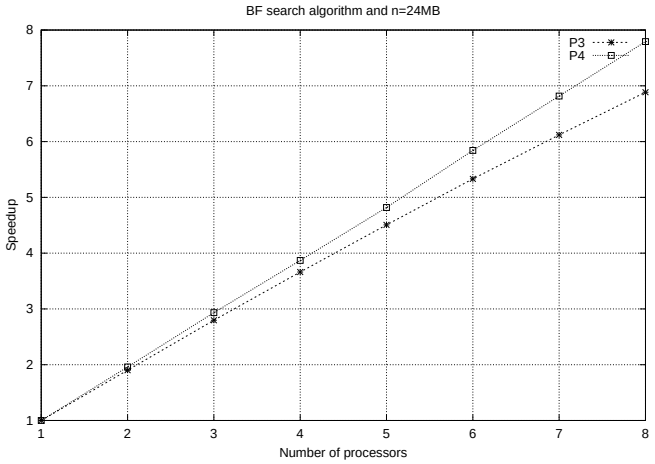
Ζήτημα Κλιμάκωσης

Για να μελετήσουμε την κλιμάκωση των δύο προτεινόμενων παράλληλων υλοποιήσεων P3 και P4, εκτελούμε τα πειράματα με το ακόλουθο τρόπο. Πολλαπλασιάζουμε οκτώ φορές την παλιά συλλογή κειμένου. Η καινούργια συλλογή κειμένου είναι γύρω στα 24MB. Τα αποτελέσματα για την καινούργια συλλογή κειμένου απεικονίζονται στον Πίνακα 5.3 και στο Σχήμα 5.5.

Τα αποτελέσματα δείχνουν ότι οι δύο παράλληλες υλοποιήσεις κλιμακώνουν καλά παρόλο που το μέγεθος του κειμένου αυξήθηκε οκτώ φορές. Ο χρόνος εκτέλεσης για $m = 10$ και για τον αλγόριθμο BF μειώνεται σε 1.277 και σε 1.130 δευτερόλεπτα για τις υλοποιήσεις P3 και P4 αντίστοιχα, όταν ο αριθμός των σταθμών εργασίας είναι οκτώ. Επίσης, οι επιταχύνσεις των δύο μεθόδων αυξάνουν γραμμικά όταν ο αριθμός των σταθμών εργασίας αυξάνεται.

5.4.3 Αποτελέσματα Άλλων Αλγορίθμων Αναζήτησης Αλφαριθμητικών

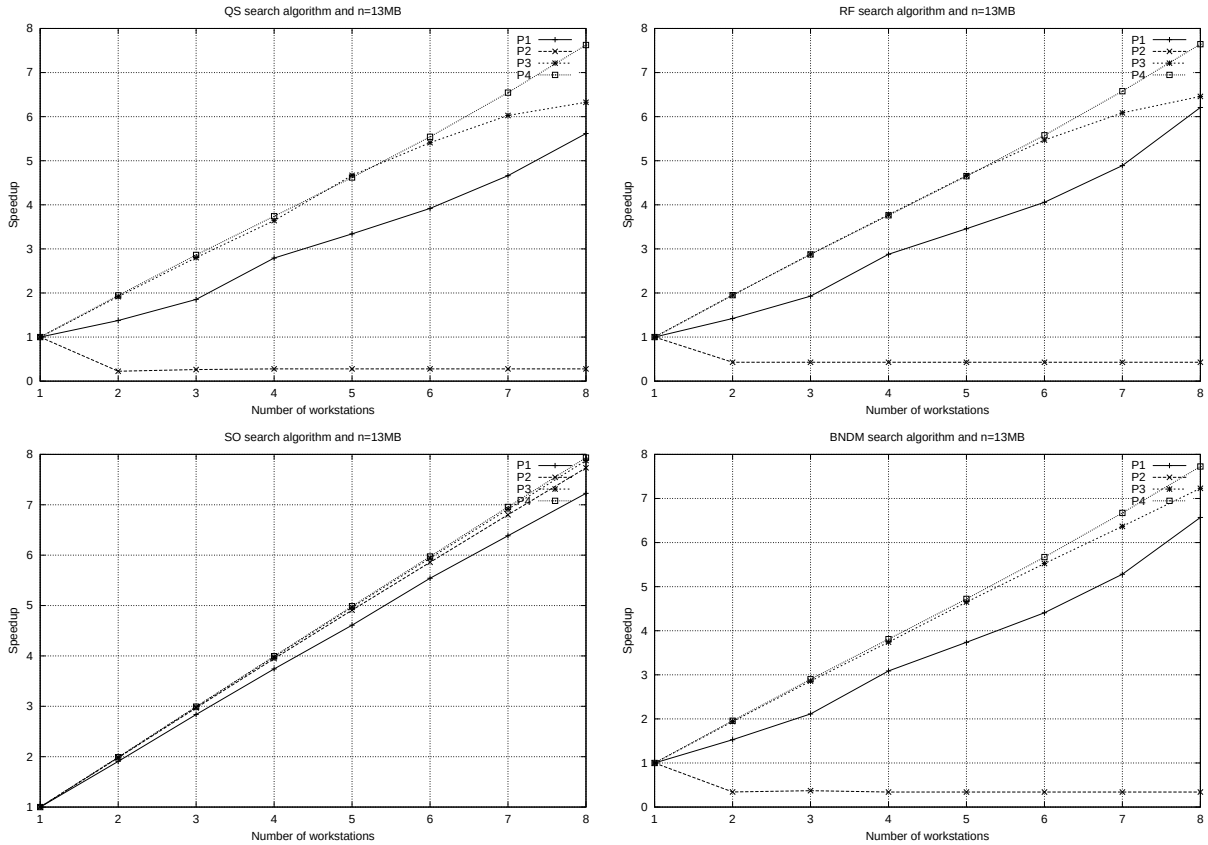
Σε αυτή την ενότητα παρουσιάζουμε τα πειραματικά αποτελέσματα των παράλληλων υλοποιήσεων χρησιμοποιώντας ορισμένους ακολουθιακούς αλγορίθμους απλής και προσεγγιστικής αναζήτησης αλφαριθ-



Σχήμα 5.5: Επιτάχυνση των παράλληλων υλοποιήσεων αναζήτησης αλφαριθμητικών σε συνάρτηση με τον αριθμό των σταθμών για μέγεθος κειμένου 24MB χρησιμοποιώντας τον αλγόριθμο BF

μητικών. Χρησιμοποιήσαμε τους αλγορίθμους από τους πειραματικούς χάρτες που έχουμε κατασκευάσει στα κεφάλαια 2 και 3 για τα προβλήματα απλής και προσεγγιστικής αναζήτησης αλφαριθμητικών. Το Σχήμα 5.6 παρουσιάζει επιταχύνσεις των τεσσάρων παράλληλων υλοποιήσεων σε σχέση με τον αριθμό των σταθμών εργασίας χρησιμοποιώντας τους αλγορίθμους της ακριβούς αναζήτησης όπως QS, RF, SO και BNDM. Παρόμοια, το Σχήμα 5.7 παρουσιάζει επιταχύνσεις για μέγεθος κειμένου 27MB. Επίσης, τα Σχήματα 5.8 και 5.9 παρουσιάζουν τις επιταχύνσεις των παράλληλων υλοποιήσεων για μεγέθη κειμένου 13MB και 27MB αντίστοιχα για τους αλγορίθμους αναζήτησης αλφαριθμητικών με k αποτυχίες όπως TU, BYP και SO. Τέλος, τα Σχήματα 5.10 και 5.11 παρουσιάζουν τις επιταχύνσεις των υλοποιήσεων για μεγέθη κειμένου 13MB και 27MB αντίστοιχα για τους αλγορίθμους αναζήτησης αλφαριθμητικών με k διαφορές όπως TUD, WM, MULTIWM και MYE.

Πρέπει να σημειωθεί ότι στους αλγορίθμους SO, WM και MYE παρατηρείται μια διαφορετική συμπεριφορά απόδοσης των υλοποιήσεων P1 και P2 σε σχέση με τους άλλους αλγορίθμους. Συγκεριμένα, η υλοποίηση P2 είναι καλύτερη από την υλοποίηση P1 σε όλους τους σταθμούς εργασίας. Αυτό το γεγονός οφείλεται στο ότι η φάση της αναζήτησης των αλγορίθμων SO, WM και MYE απαιτεί μεγάλη ποσότητα χρόνου σε σχέση με το χρόνο επικοινωνίας. Με άλλα λόγια, ο χρόνος εκτέλεσης των υλοποιήσεων κυριαρχείται από τον χρόνο αναζήτησης παρά από την επικοινωνία. Οσον αφορά στις υλοποιήσεις P3 και P4 παρουσιάζουν καλύτερη απόδοση όπως είδαμε προηγουμένως στον αλγόριθμο

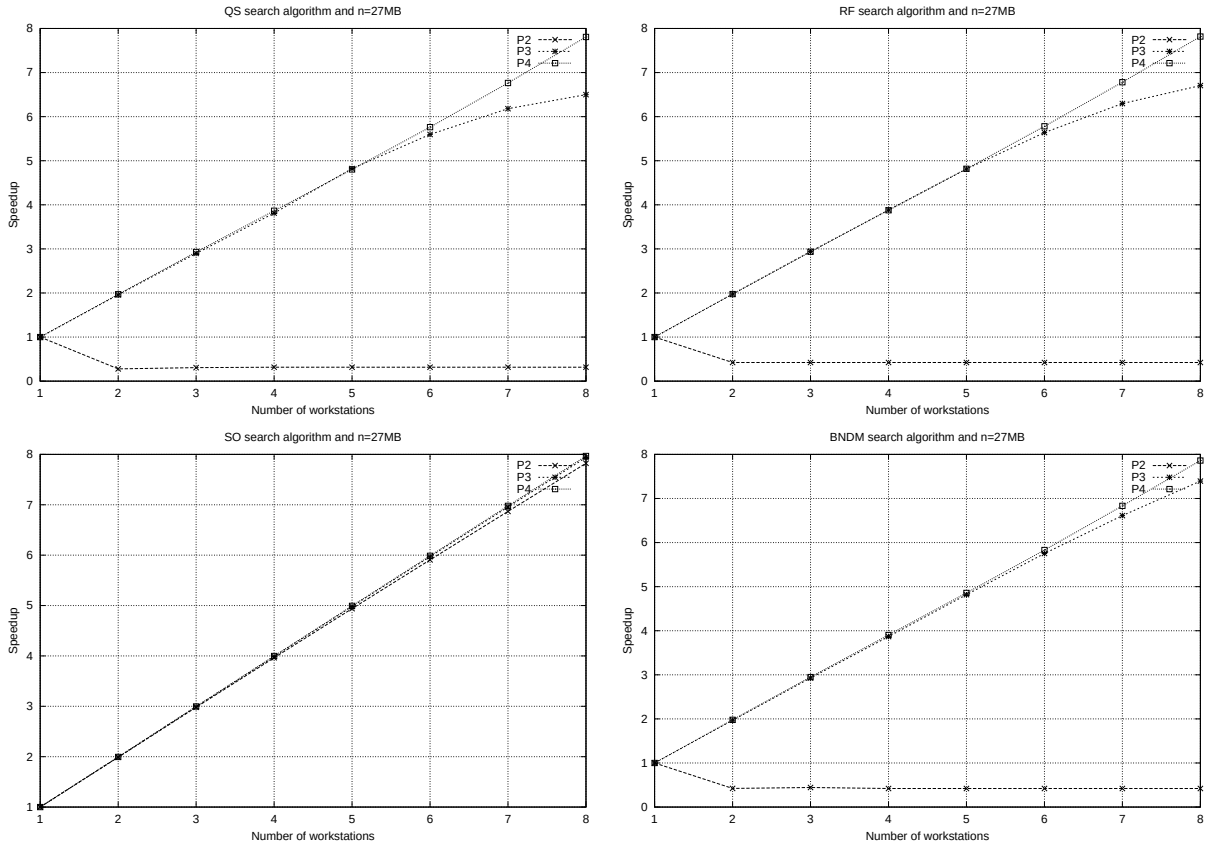


Σχήμα 5.6: Επιτάχυνση των παράλληλων υλοποιήσεων αναζήτησης αλφαριθμητικών σε συνάρτηση με τον αριθμό των σταθμών εργασίας για μέγεθος κειμένου 13MB χρησιμοποιώντας τους αλγορίθμους QS, RF, SO και BNDM

BF.

5.5 Αναλυτικό Μοντέλο Πρόβλεψης Απόδοσης

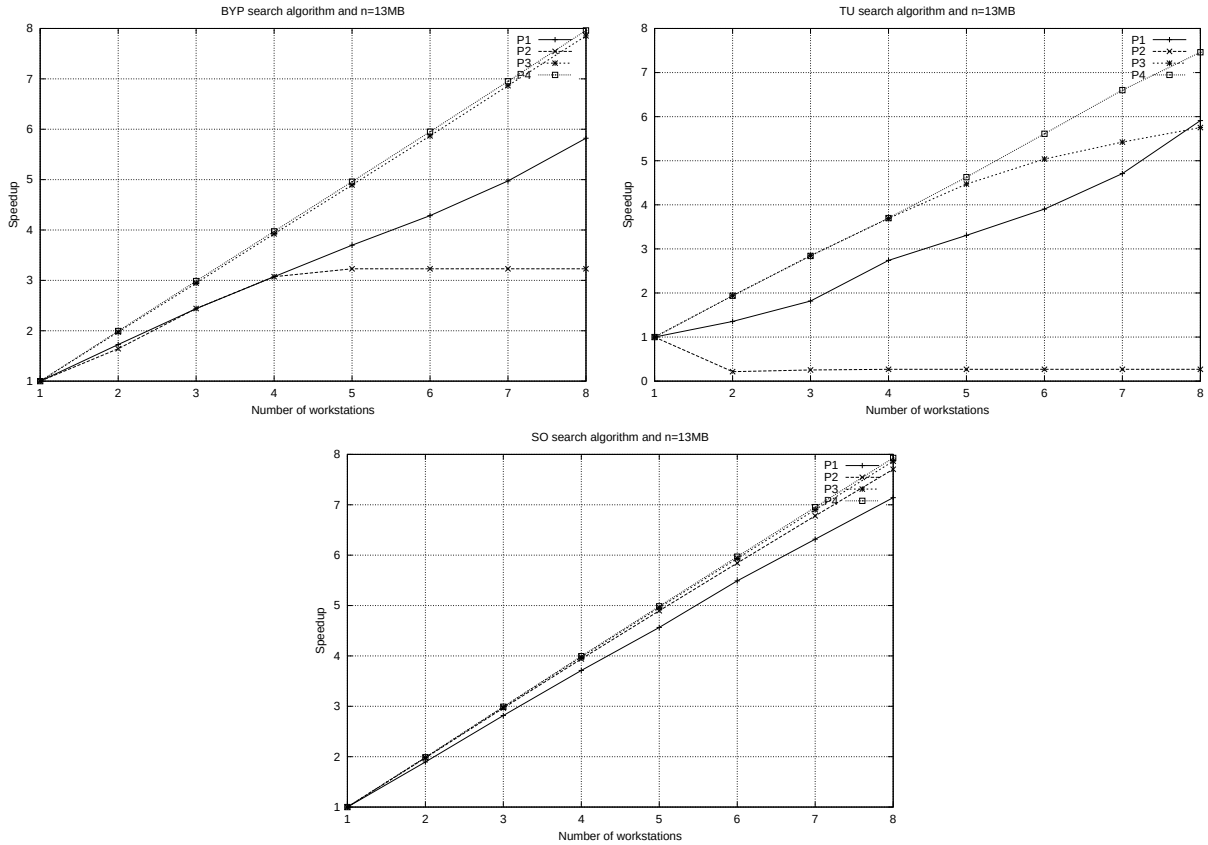
Στις επόμενες υποενότητες θα αναπτύξουμε ένα αναλυτικό μοντέλο απόδοσης για τις παράλληλες υλοποιήσεις P2, P3 και P4. Αξίζει να σημειωθεί εδώ ότι δεν μοντελοποιούμε την απόδοση της παράλληλης υλοποίησης P1 αφού αυτή η προσέγγιση είναι παρόμοια με την υλοποίηση P4 χρησιμοποιώντας την βελτιωμένη μέθοδο προεπεξεργασίας διανομής κειμένου.



Σχήμα 5.7: Επιτάχυνση των παράλληλων υλοποιήσεων αναζήτησης αλφαριθμητικών σε συνάρτηση με τον αριθμό των σταθμών εργασίας για μέγεθος κειμένου 27MB χρησιμοποιώντας τους αλγορίθμους QS, RF, SO και BNDM

5.5.1 Χρόνοι Ε/Ε, Προεπεξεργασίας, Αναζήτησης και Επικοινωνίας

Για να μαθηματικοποιήσουμε τις εξισώσεις που μας δίνουν τις αναμενόμενες καμπύλες επιτάχυνσης των παράλληλων υλοποιήσεων, πρέπει πρώτα να μαθηματικοποιήσουμε τις εξισώσεις για το χρόνο Ε/Ε (Εισόδου/Εξόδου), για το χρόνο προεπεξεργασίας, για το χρόνο αναζήτησης και για το χρόνο επικοινωνίας, οι οποίες δίνουν το χρόνο εκτέλεσης ολόκληρης της υλοποίησης. Ο χρόνος Ε/Ε είναι ανάλογος με το μέγεθος του κειμένου. Ας υποθέσουμε ότι $\Theta_{E/E}(n)$ είναι η συνάρτηση πολυπλοκότητας Ε/Ε η οποία δίνει τον αριθμό των βημάτων για να διαβαστεί μια συλλογή κειμένου. Έτσι, η συνάρτηση πολυπλοκότητας Ε/Ε για να διαβαστεί η συλλογή κειμένου απαιτεί n προσπελάσεις στο δίσκο ανεξάρτητα από τον αλγόριθμο αναζήτησης αλφαριθμητικών που χρησιμοποιείται κάθε φορά. Αν γ είναι ο μέσος



Σχήμα 5.8: Επιτάχυνση των παράλληλων υλοποιήσεων αναζήτησης αλφαριθμητικών σε συνάρτηση με τον αριθμό των σταθμών εργασίας για μέγεθος κειμένου 13MB και $k = 3$ χρησιμοποιώντας διαφορετικά μήκη προτύπων χρησιμοποιώντας τους αλγορίθμους BYP, TU και SO

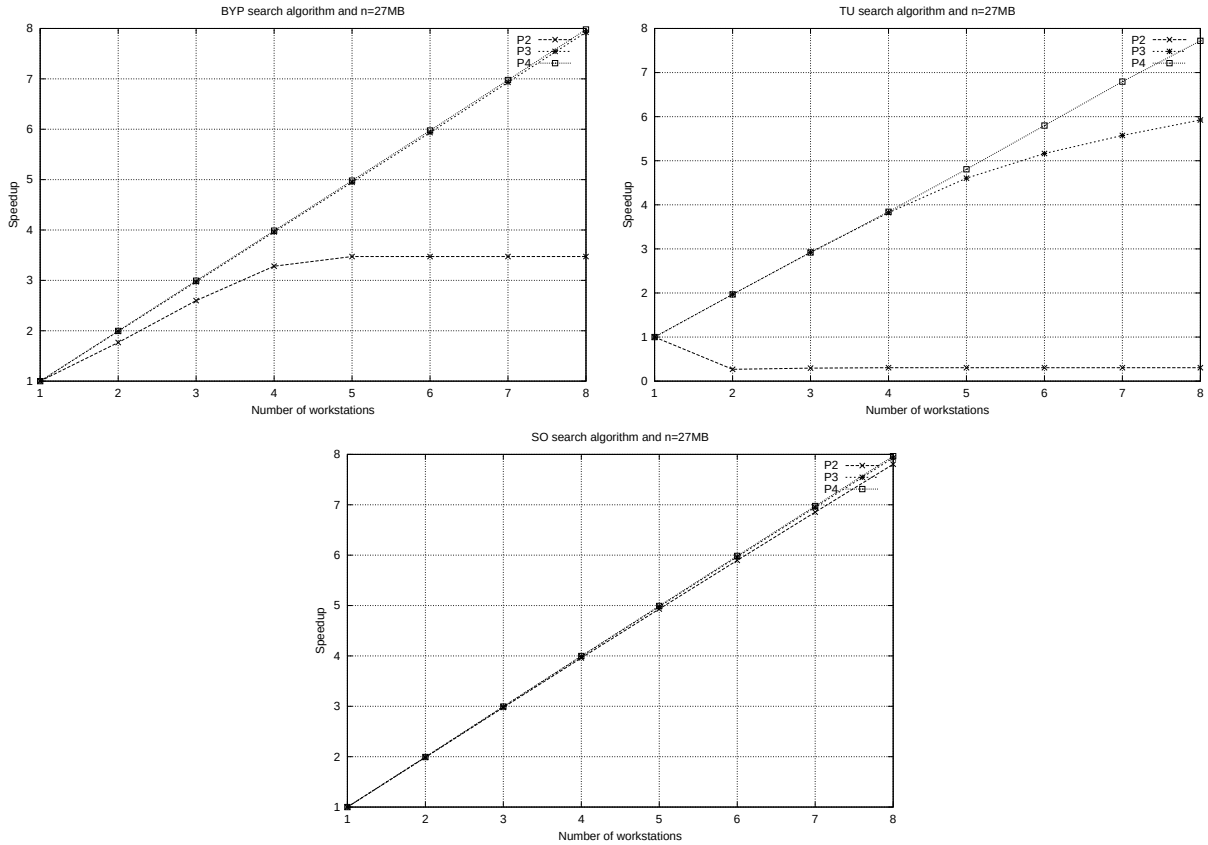
χρόνος για να εκτελέσει ένα βήμα E/E, τότε ο χρόνος E/E, $T_{E/E}$, δίνεται από:

$$T_{E/E} = \Theta_{E/E}(n)\gamma = n\gamma \tag{5.1}$$

Αν ο φόρτος εργασίας (workload) διαιρείται ακριβώς με p σταθμούς εργασίας, τότε ο χρόνος E/E του κάθε σταθμού εργασίας δίνεται από:

$$T_{E/E} = (\lceil \frac{\Theta_{E/E}(n)}{p} \rceil + m - 1)\gamma \tag{5.2}$$

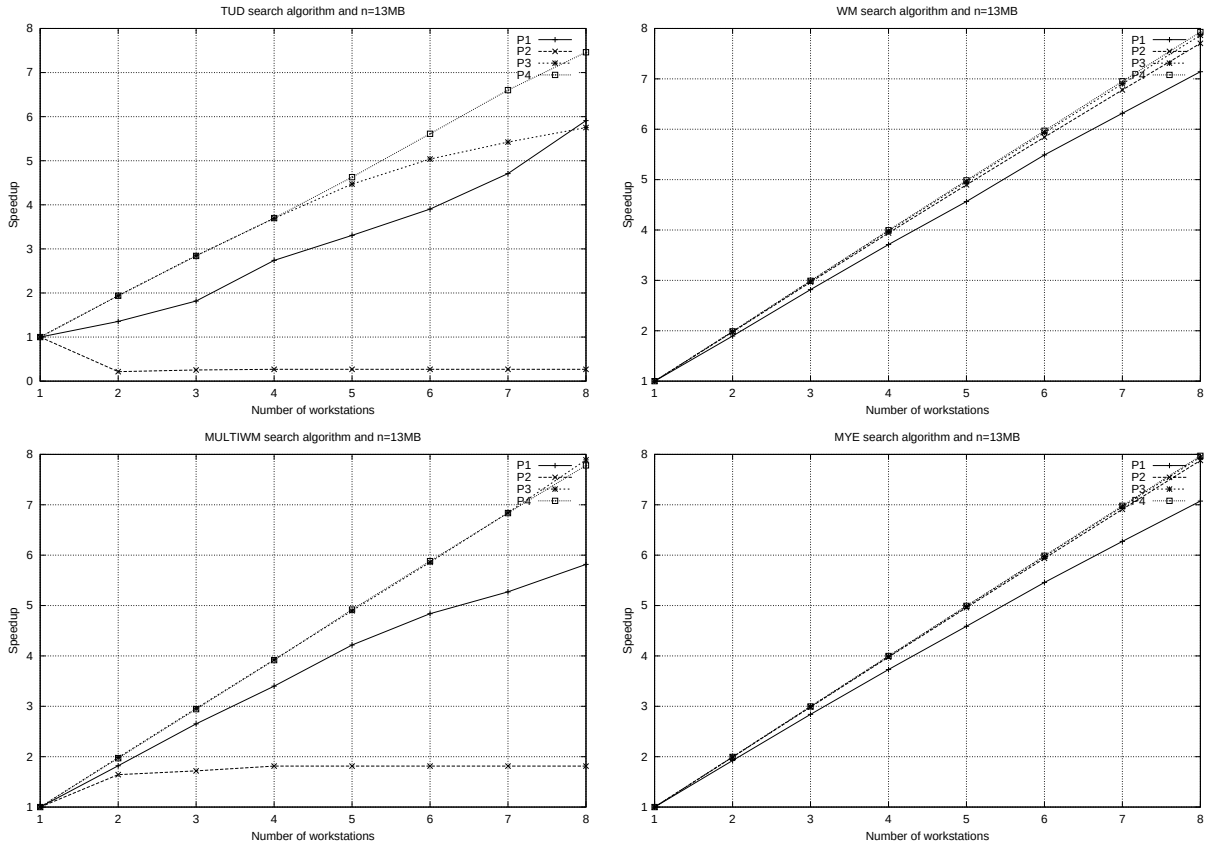
Γνωρίζουμε ότι σε μερικούς αλγορίθμους αναζήτησης αλφαριθμητικών αρχίζει από την φάση της προεπεξεργασίας του προτύπου και στην συνέχεια ακολουθεί η φάση της αναζήτησης. Ο χρόνος



Σχήμα 5.9: Επιτάχυνση των παράλληλων υλοποιήσεων αναζήτησης αλφαριθμητικών σε συνάρτηση με τον αριθμό των σταθμών εργασίας για μέγεθος κειμένου 27MB και $k = 3$ χρησιμοποιώντας διαφορετικά μήκη προτύπων χρησιμοποιώντας τους αλγορίθμους BYP, TU και SO

προεπεξεργασίας εξαρτάται από τον αλγόριθμο απλής ή προσεγγιστικής αναζήτησης αλφαριθμητικών που χρησιμοποιούμε κάθε φορά σε κάθε εργαζόμενο των παράλληλων υλοποιήσεων. Συμβολίζουμε $\Theta_{prep}(m, k)$ την συνάρτηση πολυπλοκότητας προεπεξεργασίας του προτύπου η οποία δίνει τον αριθμό των υπολογιστικών βημάτων που απαιτείται για την ολοκλήρωση της φάσης της προεπεξεργασίας. Έστω ότι δ είναι ο μέσος χρόνος για να εκτελέσει ένα βήμα προεπεξεργασίας. Τότε ο χρόνος προεπεξεργασίας T_{prep}, δίνεται από:

$$T_{prep} = \Theta_{prep}(m, k)\delta \tag{5.3}$$

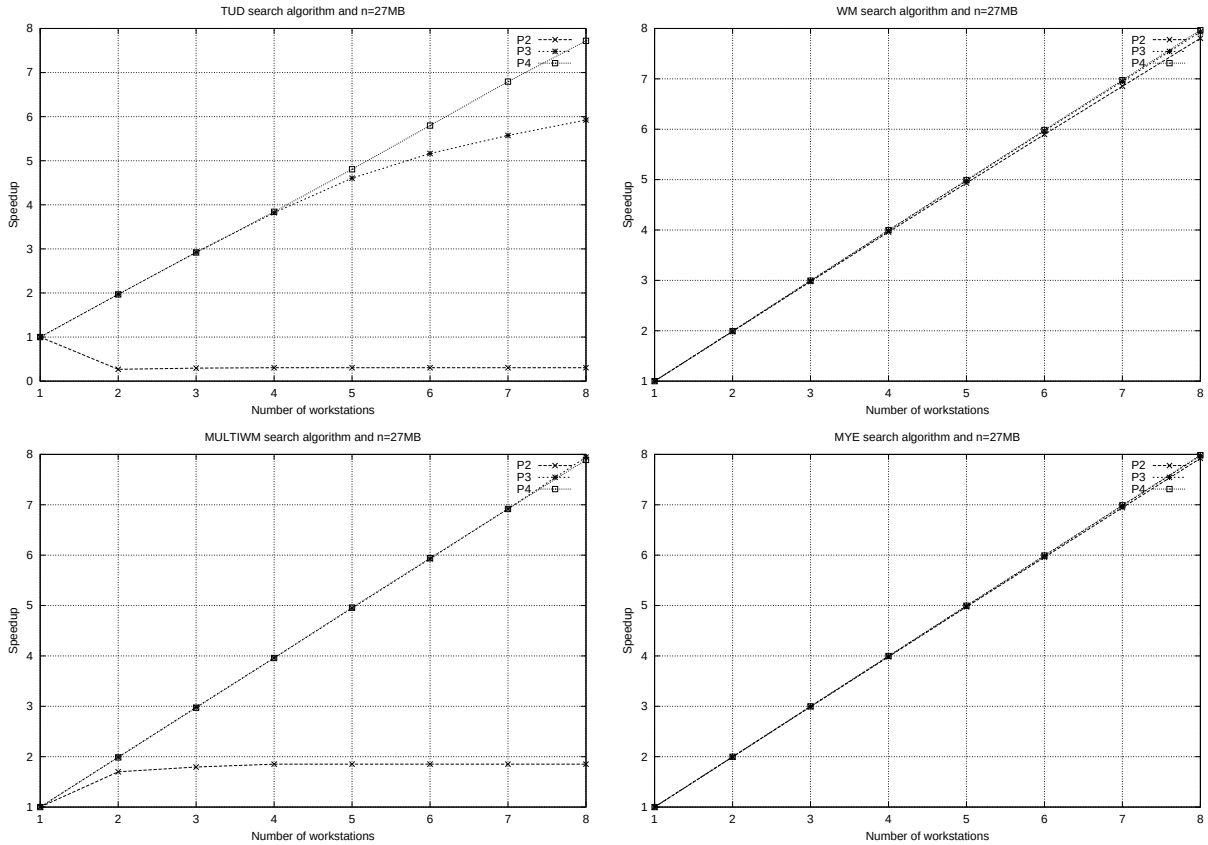


Σχήμα 5.10: Επιτάχυνση των παράλληλων υλοποιήσεων αναζήτησης αλφαριθμητικών σε συνάρτηση με τον αριθμό των σταθμών εργασίας για μέγεθος κειμένου 13MB και $k = 3$ χρησιμοποιώντας διαφορετικά μήκη προτύπων χρησιμοποιώντας τους αλγορίθμους TUD, WM, MULTIWM και MYE

Ο χρόνος προεπεξεργασίας σε κάθε σταθμό εργασίας είναι ίδιος με την προηγούμενη εξίσωση αφού δεν διαμερίζουμε το πρότυπο αλφαριθμητικό.

Ο χρόνος αναζήτησης εξαρτάται από την πολυπλοκότητα του αλγορίθμου απλής ή προσεγγιστικής αναζήτησης που χρησιμοποιούμε κάθε φορά σε κάθε εργαζόμενο. Συμβολίζουμε $\Theta_{search}(n, m, k)$ την συνάρτηση πολυπλοκότητας αναζήτησης η οποία δίνει τον αριθμό των υπολογιστικών βημάτων που απαιτείται από ένα αλγόριθμο απλής ή προσεγγιστικής αναζήτησης αλφαριθμητικών. Ας ϵ είναι ο μέσος χρόνος για να εκτελέσει ένα υπολογιστικό βήμα αναζήτησης. Τότε, ο χρόνος αναζήτησης, T_{search}, δίνεται από:

$$T_{search} = \Theta_{search}(n, m, k)\epsilon \tag{5.4}$$



Σχήμα 5.11: Επιτάχυνση των παράλληλων υλοποιήσεων αναζήτησης αλφαριθμητικών σε συνάρτηση με τον αριθμό των σταθμών εργασίας για μέγεθος κειμένου 27MB και $k = 3$ χρησιμοποιώντας διαφορετικά μήκη προτύπων χρησιμοποιώντας τους αλγορίθμους TUD, WM, MULTIWM και MYE

Επιπλέον, ο χρόνος αναζήτησης του κάθε σταθμού εργασίας δίνεται από:

$$T_{search} = (\lceil \frac{\Theta_{search}(n, m, k)}{p} \rceil + m - 1)\epsilon \tag{5.5}$$

Ο συνολικός χρόνος επικοινωνίας μιας παράλληλης υλοποίησης είναι το άθροισμα δύο συστατικών: του χρόνου καθυστέρησης (latency time) και του χρόνου μετάδοσης (transmission time). Ο χρόνος καθυστέρησης, α, είναι ένας σταθερός χρόνος επιβάρυνσης αρχικοποίησης που χρειάζεται για να προετοιμαστεί η αποστολή ενός μηνύματος από έναν σταθμό εργασίας σε έναν άλλο. Ο χρόνος μετάδοσης είναι ανάλογος με το μέγεθος του προτύπου. Έστω ότι β είναι ο αυξητικός χρόνος μετάδοσης ανά byte. Σημειώνουμε ότι αυτό είναι συνήθως στην περίπτωση που το $\alpha >> \beta$. Τότε ο χρόνος επικοινωνίας,

T_{comm}, για να αποσταλούν R bytes δεδομένα (μηνύματα) ορίζεται ως εξής:

$$T_{comm} = \alpha + R\beta \tag{5.6}$$

5.5.2 Αναλυτική Μοντελοποίηση για την Δυναμική Διανομή Υπο-κειμένων

Αν οι συναρτήσεις $\Theta_{E/E}(n), \Theta_{prep}(m,k)$ και $\Theta_{search}(n,m,k)$ και οι τιμές $\alpha, \beta, \gamma, \delta$ και ϵ είναι γνωστές, τότε μπορεί εύκολα να εκτιμηθεί ο χρόνος εκτέλεσης της παράλληλης υλοποίησης P2 όπως θα δείξουμε παρακάτω. Επίσης, υποθέτουμε ότι T_a, T_b, T_c, T_d και T_e είναι οι χρόνοι που καταναλώνονται σε κάθε μια από τις πέντε φάσεις της υλοποίησης P2 αντίστοιχα. Τα αναλυτικά κόστη (σε χρόνο) είναι ως εξής:

Η πρώτη φάση περιλαμβάνει το χρόνο επικοινωνίας για να γίνει η εκπομπή του προτύπου αλφαριθμητικού και του αριθμού των διαφορών k σε όλους τους σταθμούς εργασίας. Χρησιμοποιήσαμε τη συνάρτηση MPI_Bcast για την εκπομπή των πληροφοριών που ολοκληρώνεται σε $log_2 p$ βήματα. Σε κάθε βήμα, μία ή δύο παράλληλες πράξεις αποστολής εκτελούνται ανά σταθμό εργασίας. Το μέγεθος ενός m προτύπου αλφαριθμητικού είναι m bytes και ο αριθμός των διαφορών είναι 1 byte. Συνεπώς, η εκπομπή μεταφέρει $m+1$ bytes στους άλλους $p-1$ σταθμούς εργασίας. Ο χρόνος T_a για αυτή την πρώτη φάση δίνεται από:

$$T_a = log_2 p(\alpha + (m+1)\beta) \tag{5.7}$$

Η δεύτερη φάση είναι ο συνολικός χρόνος Ε/Ε για να διαβαστεί ολόκληρη η συλλογή κειμένου από τον τοπικό δίσκο του συντονιστή. Είναι γνωστό ότι ο συντονιστής διαβάζει τη συλλογή κειμένου από τον τοπικό δίσκο σε διάφορα τμήματα μεγέθους $sb + m - 1$ χαρακτήρες (ή bytes). Συνεπώς, ο συντονιστής διαβάζει n bytes συνολικά από την συλλογή κειμένου. Τότε, ο συνολικός χρόνος T_b για τη δεύτερη φάση δίνεται από:

$$T_b = n\gamma \tag{5.8}$$

Η τρίτη φάση περιλαμβάνει τον χρόνο επικοινωνίας για να αποσταλούν όλα τα τμήματα από την συλλογή κειμένου σε όλους τους εργαζόμενους. Ο συντονιστής αποστέλει ένα τμήμα μεγέθους $sb +$

$m-1$ bytes σε έναν εργαζόμενο. Συνεπώς, ο χρόνος επικοινωνίας T_c για να αποστείλει όλα τα τμήματα στους p σταθμούς εργασίας δίνεται από:

$$T_c = \frac{n}{sb+m-1}(\alpha + (sb+m-1)\beta) \tag{5.9}$$

Η τέταρτη φάση είναι ο μέσος χρόνος αναζήτησης αλφαριθμητικών σε ένα σταθμό εργασίας συμπεριλαβανομένου και του χρόνου προεπεξεργασίας. Κάθε σταθμός εργασίας πρέπει πρώτα να εκτελέσει την προεπεξεργασία του προτύπου και στην συνέχεια να αναζητήσει το πρότυπο αλφαριθμητικό μέσα σε ένα τμήμα κειμένου, όπου το κάθε τμήμα έχει μέγεθος $sb+m-1$ χαρακτήρες. Η πολυπλοκότητα προεπεξεργασίας που εκτελείται σε κάθε σταθμό εργασίας για το πρότυπο εκφράζεται από την συνάρτηση $\Theta_{prep}(m,k)$ και η πολυπλοκότητα αναζήτησης που εκτελείται σε κάθε σταθμό εργασίας για ένα τμήμα κειμένου εκφράζεται από την συνάρτηση $\Theta_{search}(sb+m-1,m,k)$ για οποιοδήποτε ακολουθιακό αλγόριθμο αναζήτησης αλφαριθμητικών. Συνεπώς, ο χρόνος αναζήτησης μαζί με το χρόνο προεπεξεργασίας T_d σε ένα σταθμό εργασίας δίνεται από:

$$T_d = \frac{\frac{n}{sb+m-1}}{p}(\Theta_{prep}(m,k)\delta + \Theta_{search}(sb+m-1,m,k)\epsilon) \tag{5.10}$$

Η πέμπτη φάση περιλαμβάνει το χρόνο επικοινωνίας για τη λήψη των αποτελεσμάτων της αναζήτησης αλφαριθμητικών από ένα σταθμό εργασίας. Γνωρίζουμε ότι ο συντονιστής πρέπει να λάβει $\frac{n}{sb+m-1}$ αποτελέσματα. Κάθε εργαζόμενος αποστέλει πίσω μια τιμή (δηλαδή τον αριθμό των εμφανίσεων) στο συντονιστή. Συνεπώς ο χρόνος επικοινωνίας T_e για την λήψη των αποτελεσμάτων από ένα σταθμό εργασίας δίνεται από:

$$T_e = \frac{n}{sb+m-1}(\alpha+\beta) \tag{5.11}$$

Επίσης, σημειώνουμε ότι στην δυναμική υλοποίηση P2 στην πράξη υπάρχει παράλληλη επικοινωνία και υπολογισμός και για αυτό τον λόγο παίρνουμε την μέγιστη τιμή ανάμεσα στον χρόνο επικοινωνίας δηλαδή T_c+T_e και στον χρόνο υπολογισμού δηλαδή T_d. Συνεπώς, ο συνολικός χρόνος εκτέλεσης της δυναμικής υλοποίησης αναζήτησης αλφαριθμητικών T_p, χρησιμοποιώντας τους p σταθμούς εργασίας, δίνεται από:

$$T_p = T_a + T_b + max\{T_c+T_e, T_d\} \tag{5.12}$$

5.5.3 Αναλυτική Μοντελοποίηση για την Δυναμική Διανομή Δεικτών Κειμένου

Υποθέτουμε ότι T_a, T_b, T_c, T_d και T_e είναι οι χρόνοι που καταναλώνονται σε κάθε μια από τις πέντε φάσεις της υλοποίησης P3 αντίστοιχα. Τα αναλυτικά κόστη (σε χρόνο) είναι ως εξής:

Η πρώτη φάση περιλαμβάνει το χρόνο επικοινωνίας για την εκπομπή του προτύπου αλφαριθμητικού και του αριθμού των διαφορών k σε όλους τους σταθμούς εργασίας. Ο χρόνος T_a είναι ίδιος με το T_a της προηγούμενης υλοποίησης και ορίζεται ως εξής:

$$T_a = log_2 p(\alpha + (m+1)\beta) \tag{5.13}$$

Η δεύτερη φάση περιλαμβάνει τον χρόνο επικοινωνίας για να αποσταλούν δείκτες κειμένου αντί τμημάτων από την συλλογή κειμένου σε όλους τους εργαζόμενους. Συνεπώς, ο χρόνος επικοινωνίας T_b για να αποσταλούν όλοι οι δείκτες κειμένου στους p σταθμούς εργασίας δίνεται από:

$$T_b = \frac{n}{sb + m - 1}(\alpha + \beta) \tag{5.14}$$

Η τρίτη φάση είναι ο μέσος χρόνος Ε/Ε για να διαβαστούν όλα τα τμήματα κειμένου από τον τοπικό δίσκο ενός εργαζομένου. Ο κάθε σταθμός εργασίας διαβάζει $\lceil \frac{n}{p(sb+m-1)} \rceil$ κομμάτια κειμένου από το τοπικό δίσκο στην κύρια μνήμη, όπου κάθε τμήμα κειμένου έχει $sb+m-1$ χαρακτήρες. Τότε ο χρόνος Ε/Ε T_c σε ένα σταθμό εργασίας δίνεται από:

$$T_c = \lceil \frac{n}{p} \rceil \gamma \tag{5.15}$$

Η τέταρτη φάση είναι ο μέσος χρόνος αναζήτησης αλφαριθμητικών σε ένα σταθμό εργασίας. Η ποσότητα χρόνου της φάσης αυτής είναι ίδια με το χρόνο T_d της προηγούμενης υλοποίησης και ορίζεται ως εξής:

$$T_d = \frac{\frac{n}{sb+m-1}}{p}(\Theta_{prep}(m,k)\delta + \Theta_{search}(sb+m-1,m,k)\epsilon) \tag{5.16}$$

Η πέμπτη φάση περιλαμβάνει τον χρόνο επικοινωνίας για την λήψη των αποτελεσμάτων της αναζήτησης αλφαριθμητικών από ένα σταθμό εργασίας. Ο χρόνος T_e είναι ίδιος με το T_e της προηγούμενης

υλοποίησης και ορίζεται ως εξής:

$$T_e = \frac{n}{sb + m - 1}(\alpha + \beta) \tag{5.17}$$

Γνωρίζουμε ότι στην υλοποίηση P3 υπάρχει επικάλυψη (overlap) μεταξύ επικοινωνίας και υπολογισμού και για αυτό τον λόγο παίρνουμε τη μέγιστη τιμή ανάμεσα στον χρόνο επικοινωνίας δηλαδή $T_b + T_e$ και στον χρόνο υπολογισμού δηλαδή $T_c + T_d$. Συνεπώς, ο συνολικός χρόνος εκτέλεσης της δυναμικής υλοποίησης αναζήτησης αλφαριθμητικών T_p, χρησιμοποιώντας τους p σταθμούς εργασίας, δίνεται από:

$$T_p = T_a + max\{T_b + T_e, T_c + T_d\} \tag{5.18}$$

5.5.4 Αναλυτική Μοντελοποίηση για την Υβριδική Διανομή Υπο-κειμένων

Υποθέτουμε ότι T_a, T_b, T_c και T_d είναι οι χρόνοι που καταναλώνονται σε κάθε μια από τις τέσσερις φάσεις της υλοποίησης P4 αντίστοιχα. Τα αναλυτικά κόστη (σε χρόνο) είναι ως εξής:

Η πρώτη φάση περιλαμβάνει τον χρόνο επικοινωνίας για την εκπομπή του προτύπου αλφαριθμητικού και του αριθμού των διαφορών k σε όλους τους σταθμούς εργασίας. Ο χρόνος T_a είναι ίδιος με τον T_a της υλοποίησης P2 και ορίζεται ως εξής:

$$T_a = log_2 p(\alpha + (m + 1)\beta) \tag{5.19}$$

Η δεύτερη φάση είναι ο μέσος χρόνος Ε/Ε για την ανάγνωση του υπο-κειμένου από τον τοπικό δίσκο ενός σταθμού εργασίας. Γνωρίζουμε ότι ο κάθε σταθμός εργασίας πρέπει να διαβάζει από το τοπικό δίσκο στην κύρια μνήμη $\lceil \frac{n}{p} \rceil + m - 1$ χαρακτήρες. Συνεπώς, ο χρόνος Ε/Ε T_b σε ένα σταθμό εργασίας δίνεται από:

$$T_b = (\lceil \frac{n}{p} \rceil + m - 1)\gamma \tag{5.20}$$

Η τρίτη φάση περιλαμβάνει το μέσο χρόνο αναζήτησης αλφαριθμητικών σε ένα σταθμό εργασίας. Ο κάθε σταθμός εργασίας πρέπει να αναζητήσει το πρότυπο αλφαριθμητικό μέσα στο υπο-κείμενο που

έχει μέγεθος $\lceil \frac{n}{p} \rceil + m - 1$ χαρακτήρες. Η πολυπλοκότητα προεπεξεργασίας που εκτελείται σε κάθε σταθμό εργασίας για το πρότυπο εκφράζεται από την συνάρτηση $\Theta_{prep}(m, k)$ και η πολυπλοκότητα αναζήτησης που εκτελείται σε κάθε σταθμό εργασίας για ένα τμήμα κειμένου εκφράζεται από την συνάρτηση $\Theta_{search}(\lceil \frac{n}{p} \rceil + m - 1, m, k)$ για οποιοδήποτε αλγόριθμο αναζήτησης αλφαριθμητικών. Τότε ο χρόνος αναζήτησης μαζί με το χρόνο προεπεξεργασίας T_c σε ένα σταθμό εργασίας δίνεται από:

$$T_c = \Theta_{prep}(m, k)\delta + \Theta_{search}(\lceil \frac{n}{p} \rceil + m - 1, m, k)\epsilon \tag{5.21}$$

Η τέταρτη φάση περιλαμβάνει το χρόνο επικοινωνίας για να γίνει η συλλογή p αποτελεσμάτων από την αναζήτηση αλφαριθμητικών από τους p σταθμούς εργασίας ταυτόχρονα. Κάθε σταθμός εργασίας αποστέλει πίσω μια τιμή (στην περίπτωση μας, τον αριθμό των εμφανίσεων). Χρησιμοποιήσαμε τη συνάρτηση MPI_Reduce για να συλλέξουμε τα αποτελέσματα που ολοκληρώνεται σε $log_2 p$ βήματα. Συνεπώς, ο χρόνος επικοινωνίας T_d για να συλλεγούν τα αποτελέσματα από ένα σταθμό εργασίας δίνεται από:

$$T_d = log_2 p(\alpha + \beta) \tag{5.22}$$

Ο συνολικός χρόνος εκτέλεσης της στατικής υλοποίησης αναζήτησης αλφαριθμητικών T_p, χρησιμοποιώντας τους p σταθμούς εργασίας είναι το άθροισμα των τεσσάρων φάσεων και δίνεται από:

$$T_p = T_a + T_b + T_c + T_d \tag{5.23}$$

5.5.5 Πολυπλοκότητες Προεπεξεργασίας και Αναζήτησης

Για την εκτέλεση των τεσσάρων παράλληλων υλοποιήσεων χρησιμοποιήσαμε τους ακολουθιακούς αλγορίθμους BF, QS, RF, SO και BNDM για το πρόβλημα απλής αναζήτησης αλφαριθμητικών σε κάθε εργαζόμενο. Επίσης, για το πρόβλημα αναζήτησης αλφαριθμητικών με k αποτυχίες χρησιμοποιήσαμε τους αλγορίθμους BYP, TU και SO. Για το πρόβλημα αναζήτησης αλφαριθμητικών με k διαφορές χρησιμοποιήσαμε τους αλγορίθμους TUD, WM, MULTIWM και MYE. Οι πολυπλοκότητες προεπεξεργασίας και αναζήτησης, $\Theta_{prep}(m, k)$ και $\Theta_{search}(n, m, k)$, των παραπάνω ακολουθιακών αλγορίθμων φαίνονται στον Πίνακα 5.4 και μπορούν να χρησιμοποιηθούν στο αναλυτικό μοντέλο πρόβλεψης που παρουσιάσαμε προηγουμένως.

Πίνακας 5.4: Πολυπλοκότητες Προεπεξεργασίας και Αναζήτησης

Αλγόριθμος	$\Theta_{prep}(m,k)$	$\Theta_{search}(n,m,k)$
BF	-	mn
QS	$m+\|\Sigma\|$	mn
RF	m	mn
SO	$m+\|\Sigma\|$	mn
BNDM	$m+\|\Sigma\|$	mn
BYP	$2m+\|\Sigma\|$	n
TU	$m+k\|\Sigma\|$	mn
SO	$(m+\|\Sigma\|)log\frac{k}{w}$	$mnlog\frac{k}{w}$
TUD	$(k+\|\Sigma\|)m$	$\frac{mn}{k}$
WM	$m\|\Sigma\|+k$	mnk
MULTIWM	$\|\Sigma\|+m$	$\frac{mn}{w}$
MYE	$m\|\Sigma\|$	mn

5.6 Αναλυτικά Αποτελέσματα

Σε αυτή την ενότητα παρουσιάζουμε αναλυτικά αποτελέσματα για την εκτίμηση της υπολογιστικής συμπεριφοράς των τεσσάρων υλοποιήσεων αναζήτησης αλφαριθμητικών για τα προβλήματα απλής και προσεγγιστικής αναζήτησης αλφαριθμητικών χρησιμοποιώντας το θεωρητικό μοντέλο απόδοσης.

5.6.1 Προσδιορισμός των $\alpha, \beta, \gamma, \delta$ και ϵ

Σε αυτήν ενότητα δείχνουμε πως υπολογίζουμε τις τιμές $\alpha, \beta, \gamma, \delta$ και ϵ, που χρησιμοποιούνται στο μοντέλο μας πρόβλεψης απόδοσης. Οι τιμές γ, δ και ϵ μπορούν εύκολα να εκτιμηθούν με τον ακόλουθο τρόπο:

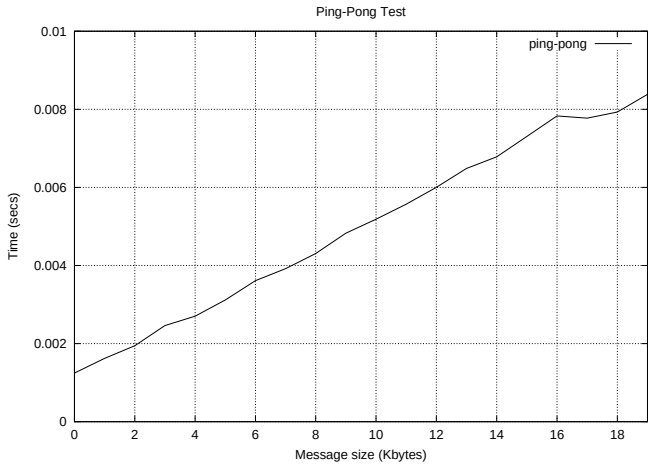
$$\gamma = \frac{T_{i/o}}{n} \qquad \delta = \frac{T_{prep}}{\Theta_{prep}(m,k)} \qquad \epsilon = \frac{T_{search}}{\Theta_{search}(n,m,k)} \tag{5.24}$$

Στον Πίνακα 5.5 παρουσιάζονται οι τιμές των γ, δ και ϵ (σε δευτερόλεπτα) για τον αλγόριθμο απλής αναζήτησης αλφαριθμητικών BF και για διαφορετικά μήκη προτύπων.

Προκειμένου να βρούμε τις τιμές α και β, τρέξαμε μερικές απλές δοκιμές ping-pong που στέλνουν και λαμβάνουν ένα αριθμό μηνυμάτων αναμέσα σε δύο σταθμούς εργασίας. Το Σχήμα 5.12 δείχνει τους αντίστοιχους χρόνους επικοινωνίας για αποστολή και λήψη μηνυμάτων χρησιμοποιώντας τη δοκιμή

Πίνακας 5.5: Τιμές των γ, δ και ϵ για τον αλγόριθμο BF

m	γ	δ	ϵ
5	8.87E-08	-	2.77E-07
10	7.91E-08	-	2.73E-07
30	7.95E-08	-	2.65E-07
60	7.90E-08	-	2.95E-07



Σχήμα 5.12: Ο χρόνος επικοινωνίας του ping-pong

ping-pong που αποτελείται από δύο διεργασίες όπου κάθε διεργασία εκτελείται σε ένα σταθμό εργασίας. Και οι δύο διεργασίες δεν κάνουν τίποτα άλλο από το να στέλνουν και να λαμβάνουν μηνύματα. Τα αποτελέσματα προκύπτουν από το μέσο όρο των 100 ξεχωριστών επαναλήψεων. Χρησιμοποιώντας τη γραμμική μεθόδου παλινδρόμησης (linear regression) για να προσεγγίσουμε με μια ευθεία γραμμή την καμπύλη της επικοινωνίας, βρήκαμε τις τιμές α και β οι οποίες είναι ο χρόνος καθυστέρησης και χρόνος μετάδοσης ανά byte αντίστοιχα. Οι τιμές που βρέθηκαν είναι 0.00062371 δευτερόλεπτα και 0.000000194885 δευτερόλεπτα αντίστοιχα στο υπολογιστικό μας περιβάλλον.

5.6.2 Αναλυτικά Αποτελέσματα

Οι Πίνακες 5.6 και 5.7 παρουσιάζουν τους χρόνους εκτέλεσης για διάφορες τιμές m, n και p χρησιμοποιώντας τον αλγόριθμο BF που προκύπτουν από τις εξισώσεις 5.12, 5.18 και 5.23 για τις υλοποιήσεις P2, P3 και P4 αντίστοιχα. Τα Σχήματα 5.13, 5.14 και 5.15 παρουσιάζουν τις επιταχύνσεις για διά-

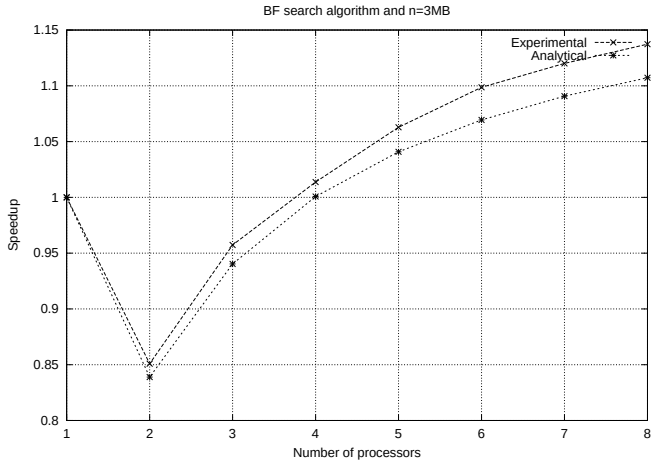
Πίνακας 5.6: Αναλυτικοί χρόνοι εκτέλεσης (σε δευτερόλεπτα) για μεγέθος κειμένου 3MB και για διαφορετικά μήκη προτύπων χρησιμοποιώντας τον αλγόριθμο BF

m/p	Παρ. Υλοπ.	1	2	3	4	5	6	7	8
5	P2	1.156	1.375	1.229	1.157	1.113	1.084	1.063	1.048
	P3	1.156	0.598	0.406	0.310	0.252	0.213	0.186	0.165
	P4	1.156	0.580	0.389	0.294	0.237	0.199	0.170	0.151
10	P2	1.113	1.338	1.195	1.123	1.080	1.051	1.031	1.016
	P3	1.113	0.577	0.391	0.299	0.243	0.206	0.180	0.160
	P4	1.113	0.559	0.375	0.283	0.228	0.192	0.172	0.152
30	P2	1.089	1.327	1.187	1.118	1.076	1.048	1.029	1.014
	P3	1.089	0.565	0.383	0.293	0.238	0.202	0.176	0.157
	P4	1.088	0.547	0.367	0.277	0.223	0.187	0.162	0.145
60	P2	1.183	1.373	1.218	1.140	1.094	1.063	1.041	1.024
	P3	1.183	0.612	0.415	0.316	0.257	0.218	0.190	0.169
	P4	1.183	0.594	0.398	0.300	0.242	0.203	0.175	0.155

φορες τιμές n και p χρησιμοποιώντας τον αλγόριθμο BF που προκύπτουν από τα πειράματα και από τις εξισώσεις 5.12, 5.18 και 5.23 για τις υλοποιήσεις P2, P3 και P4 αντίστοιχα. Είναι σημαντικό να σημειώσουμε ότι οι επιταχύνσεις που απεικονίζονται στα Σχήματα είναι αποτέλεσμα του μέσου όρου για τέσσερα διαφορετικά μήκη προτύπων. Από τους Πίνακες και τα Σχήματα παρατηρούμε ότι υπάρχουν μικρές διαφορές ανάμεσα στις πειραματικές και αναλυτικές τιμές αλλά η γενικότερη συμπεριφορά είναι όμοια.

Πίνακας 5.7: Αναλυτικοί χρόνοι εκτέλεσης (σε δευτερόλεπτα) για μεγέθος κειμένου 24MB και για διαφορετικά μήκη προτύπων χρησιμοποιώντας τον αλγόριθμο BF

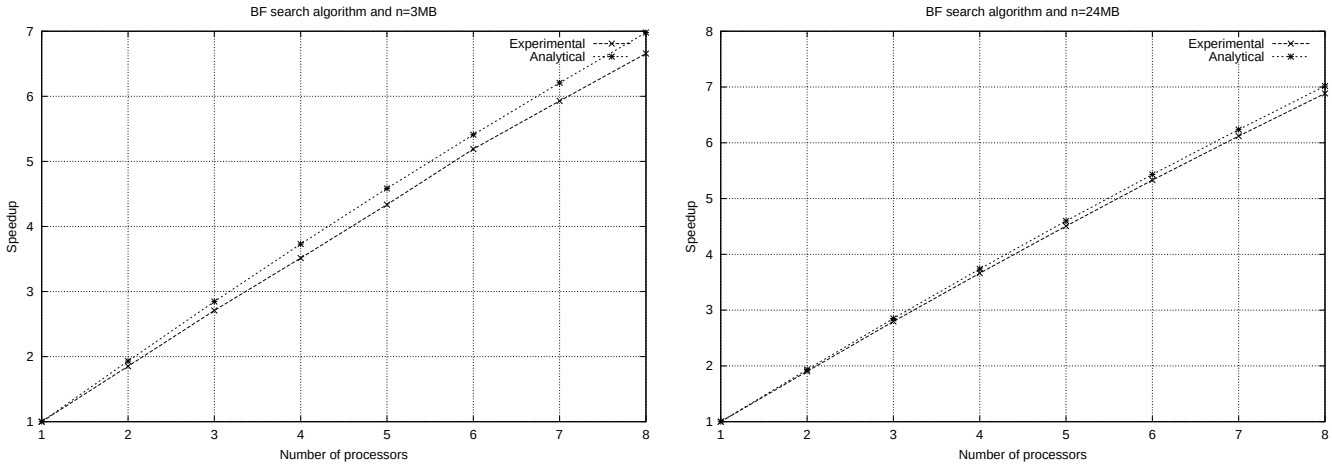
m/p	Παρ. Υλοπ.	1	2	3	4	5	6	7	8
5	P3	9.225	4.770	3.233	2.464	2.003	1.696	1.476	1.311
	P4	9.225	4.615	3.079	2.311	1.850	1.544	1.322	1.157
10	P3	8.881	4.598	3.118	2.378	1.934	1.638	1.427	1.268
	P4	8.881	4.443	2.964	2.225	1.782	1.486	1.275	1.115
30	P3	8.686	4.501	3.053	2.330	1.895	1.606	1.399	1.244
	P4	8.686	4.345	2.899	2.176	1.743	1.454	1.249	1.089
60	P3	9.439	4.877	3.304	2.518	2.046	1.731	1.507	1.338
	P4	9.437	4.722	3.150	2.364	1.893	1.579	1.355	1.186



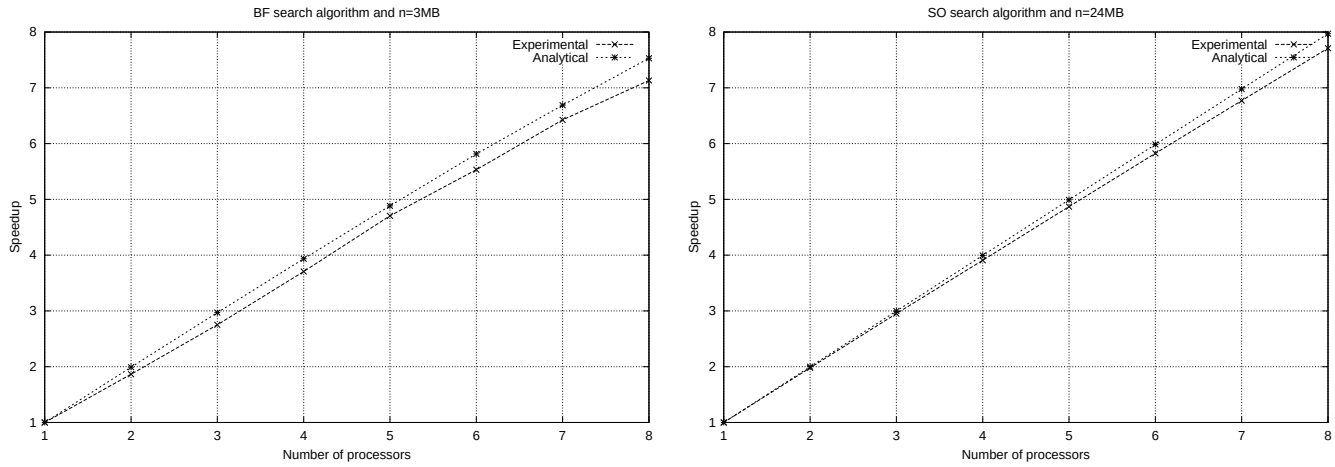
Σχήμα 5.13: Αναλυτική και πειραματική επιτάχυνση σε συνάρτηση με το αριθμό των σταθμών εργασίας για την υλοποίηση P2 χρησιμοποιώντας τον αλγόριθμο BF

5.6.3 Αποτελέσματα Άλλων Αλγορίθμων Αναζήτησης Αλφαριθμητικών

Τα Σχήματα 5.16, 5.17 και 5.18 παρουσιάζουν τις επιταχύνσεις χρησιμοποιώντας τους αλγορίθμους της ακριβούς αναζήτησης αλφαριθμητικών όπως QS, RF, SO και BNDM που προκύπτουν από τα πειράματα και από τις εξισώσεις 5.12, 5.18 και 5.23 για τις υλοποιήσεις P2, P3 και P4 αντίστοιχα. Παρόμοια,



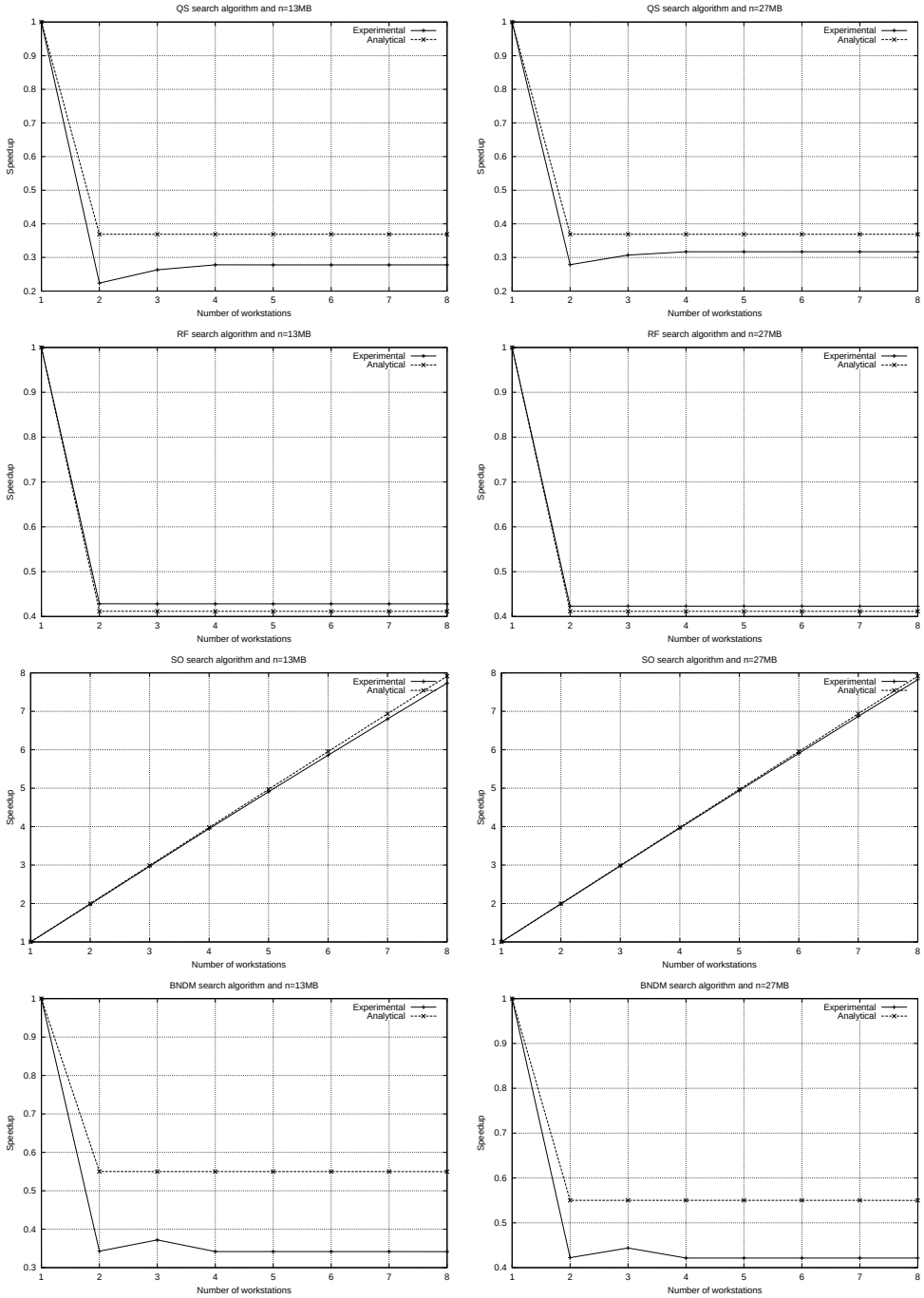
Σχήμα 5.14: Αναλυτική και πειραματική επιτάχυνση σε συνάρτηση με το αριθμό των σταθμών εργασίας για την υλοποίηση P3 χρησιμοποιώντας τον αλγόριθμο BF



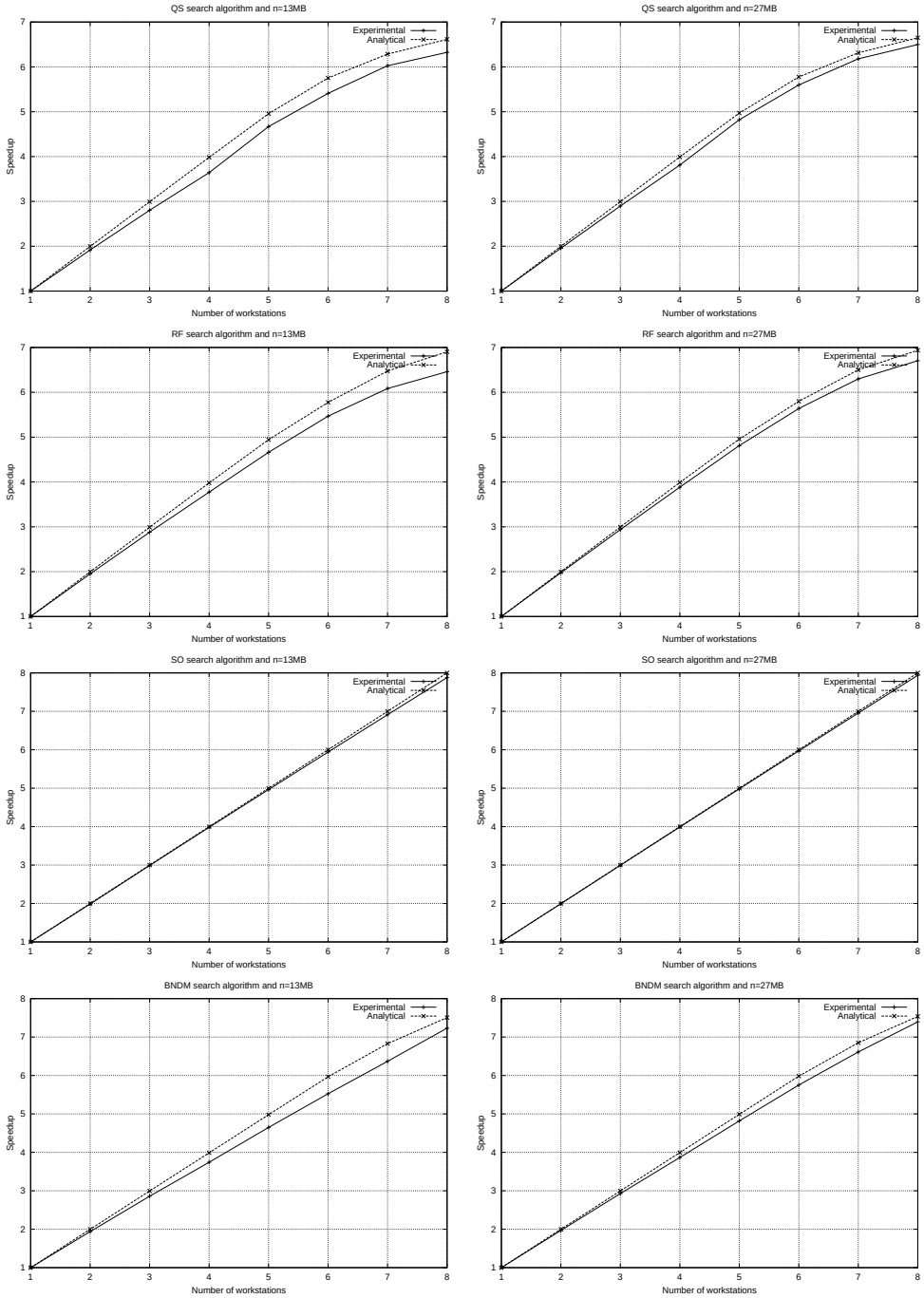
Σχήμα 5.15: Αναλυτική και πειραματική επιτάχυνση σε συνάρτηση με το αριθμό των σταθμών εργασίας για την υλοποίηση P4 χρησιμοποιώντας τον αλγόριθμο BF

τα Σχήματα 5.19, 5.20 και 5.21 παρουσιάζουν τις επιταχύνσεις χρησιμοποιώντας τους αλγορίθμους της αναζήτησης αλφαριθμητικών με k αποτυχίες όπως BYP, TU και SO. Τέλος, τα Σχήματα 5.22, 5.23 και 5.24 παρουσιάζουν τις επιταχύνσεις χρησιμοποιώντας τους αλγορίθμους της αναζήτησης αλφαριθμητικών με k διαφορές όπως TUD, WM, MULTIWM και MYE.

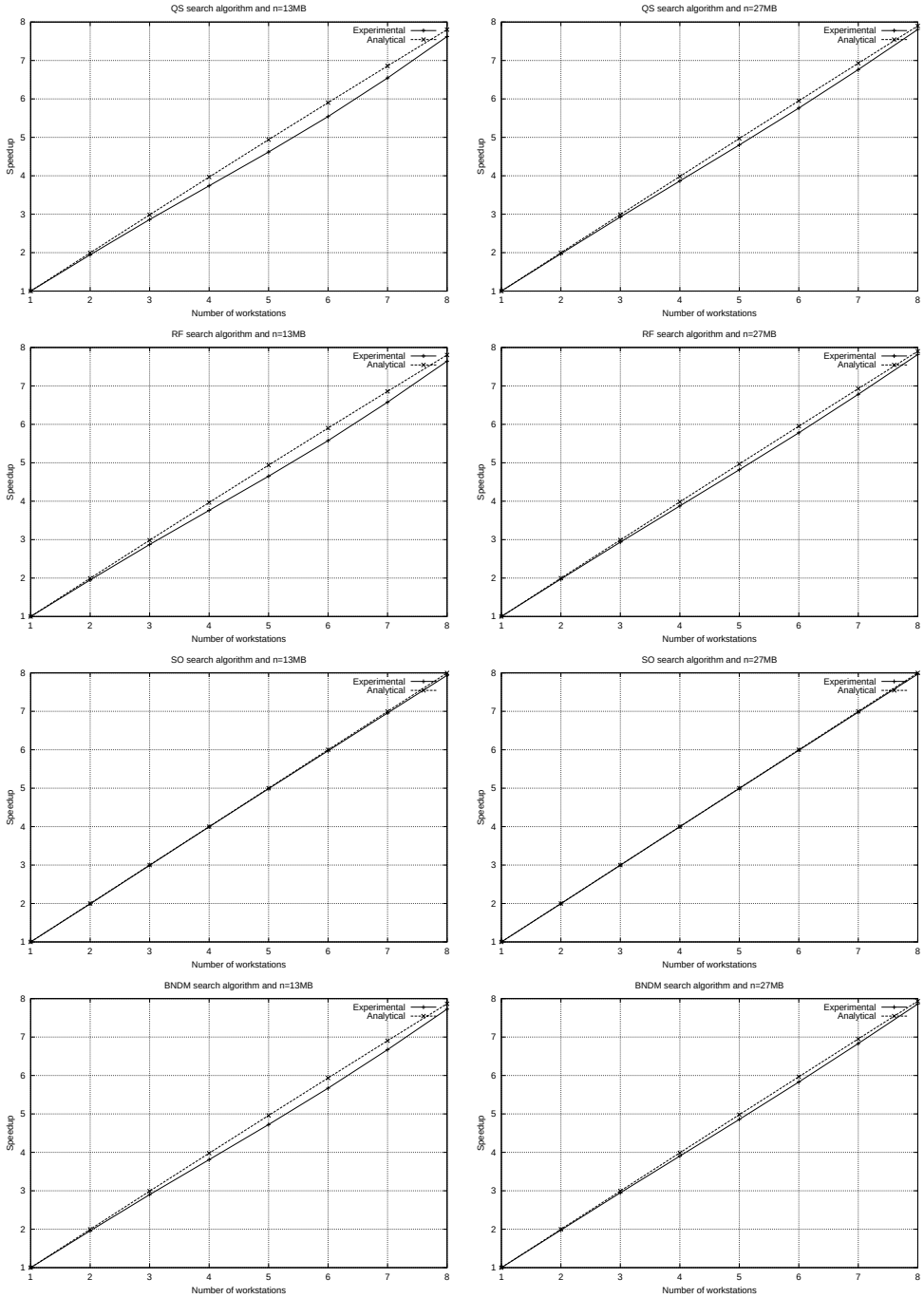
Από τα παραπάνω παρατηρούμε ότι τα πειραματικά αποτελέσματα των παράλληλων υλοποιήσεων για τους αλγορίθμους που έχουμε χρησιμοποιήσει επιβεβαιώνονται από τα θεωρητικά αποτελέσματα του μοντέλου μας πρόβλεψης απόδοσης που παρουσιάσαμε προηγουμένως.



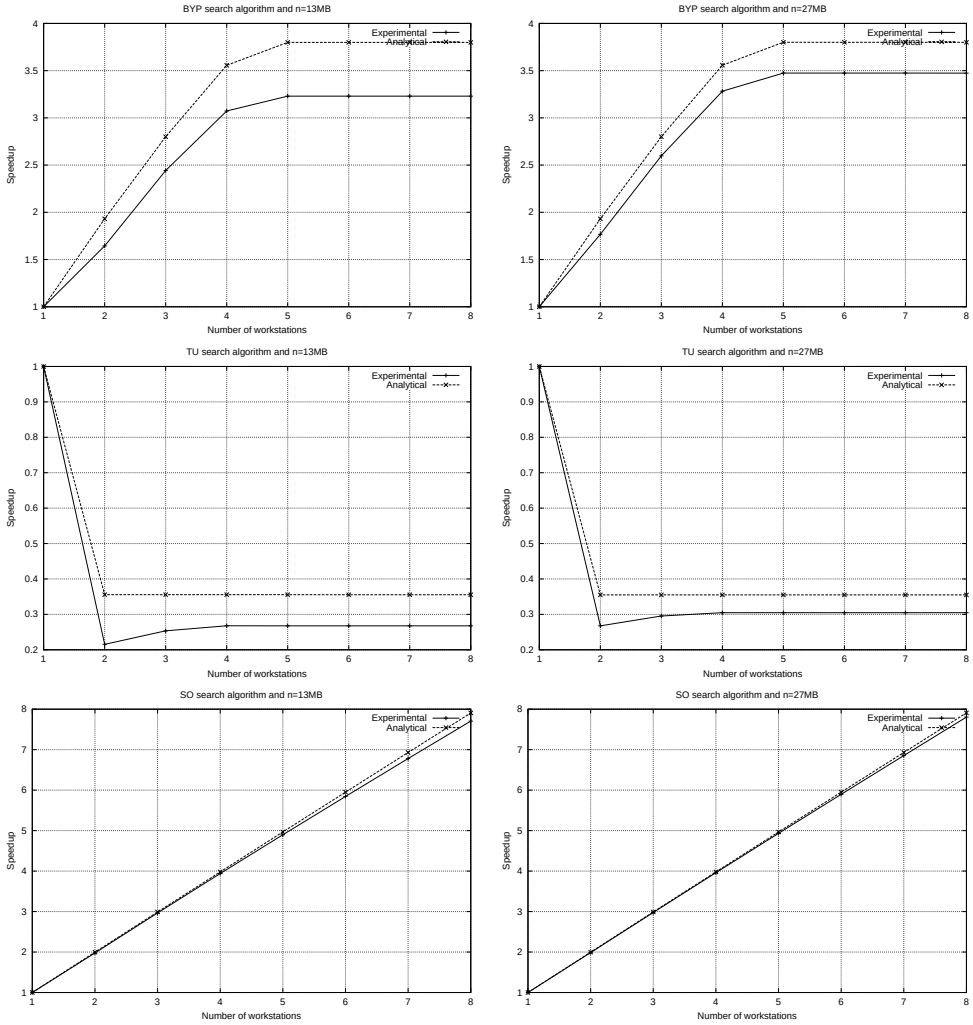
Σχήμα 5.16: Αναλυτική και πειραματική επιτάχυνση σε συνάρτηση με τον αριθμό των σταθμών εργασίας για την υλοποίηση P2 χρησιμοποιώντας τους αλγορίθμους QS, RF, SO και BNDM



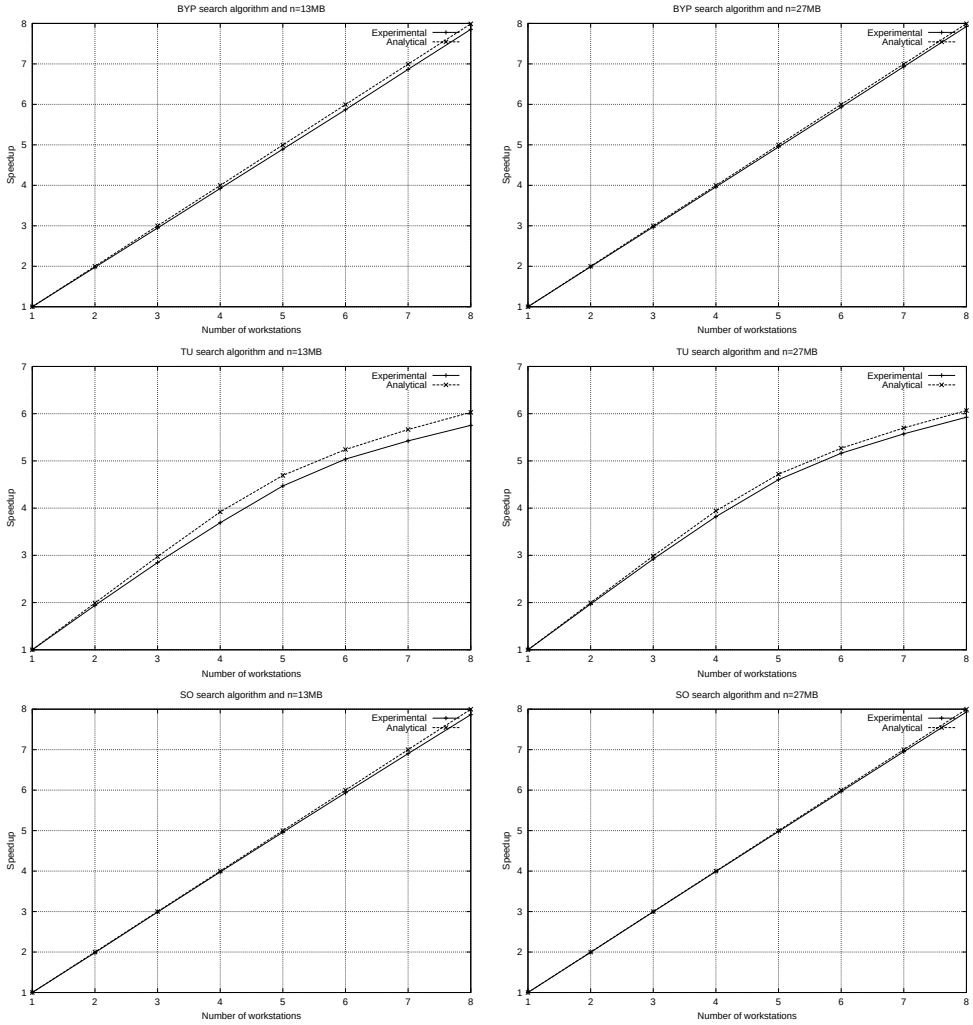
Σχήμα 5.17: Αναλυτική και πειραματική επιτάχυνση σε συνάρτηση με τον αριθμό των σταθμών εργασίας για την υλοποίηση P3 χρησιμοποιώντας τους αλγορίθμους QS, RF, SO και BNDM



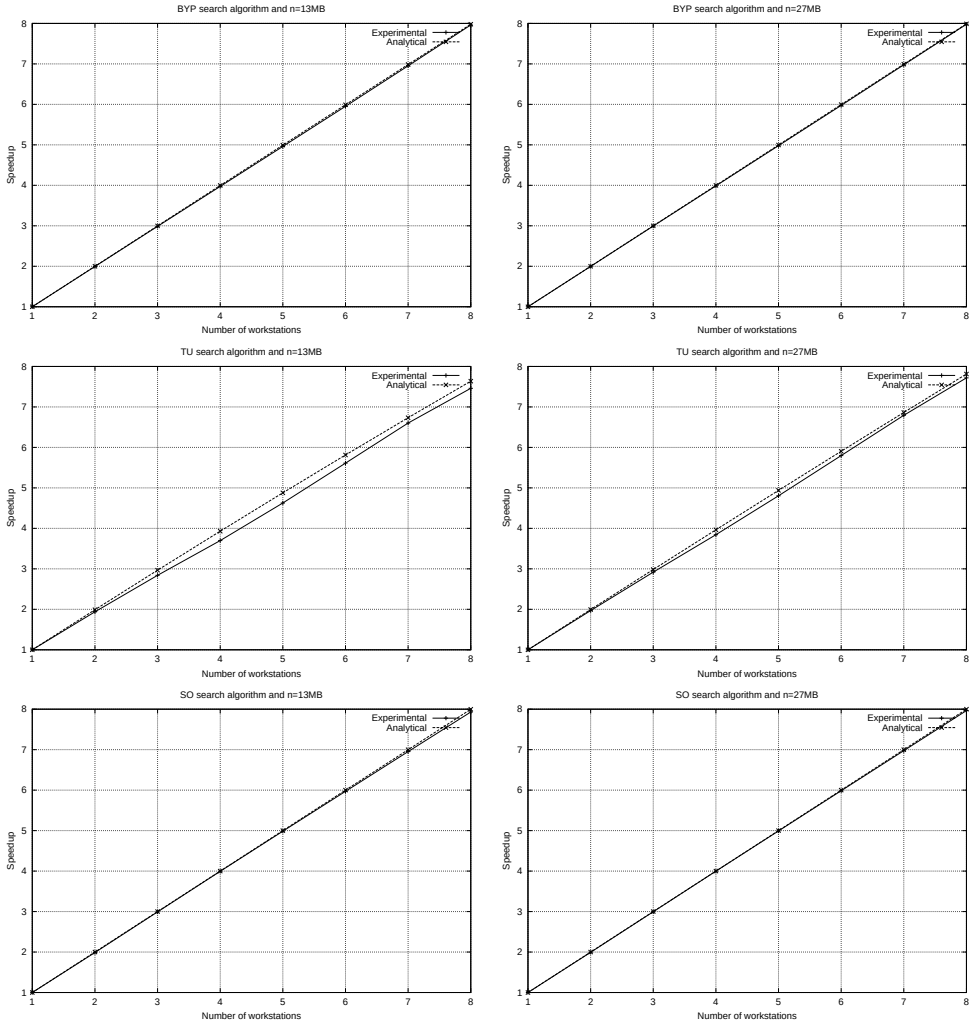
Σχήμα 5.18: Αναλυτική και πειραματική επιτάχυνση σε συνάρτηση με τον αριθμό των σταθμών εργασίας για την υλοποίηση P4 χρησιμοποιώντας τους αλγορίθμους QS, RF, SO και BNDM



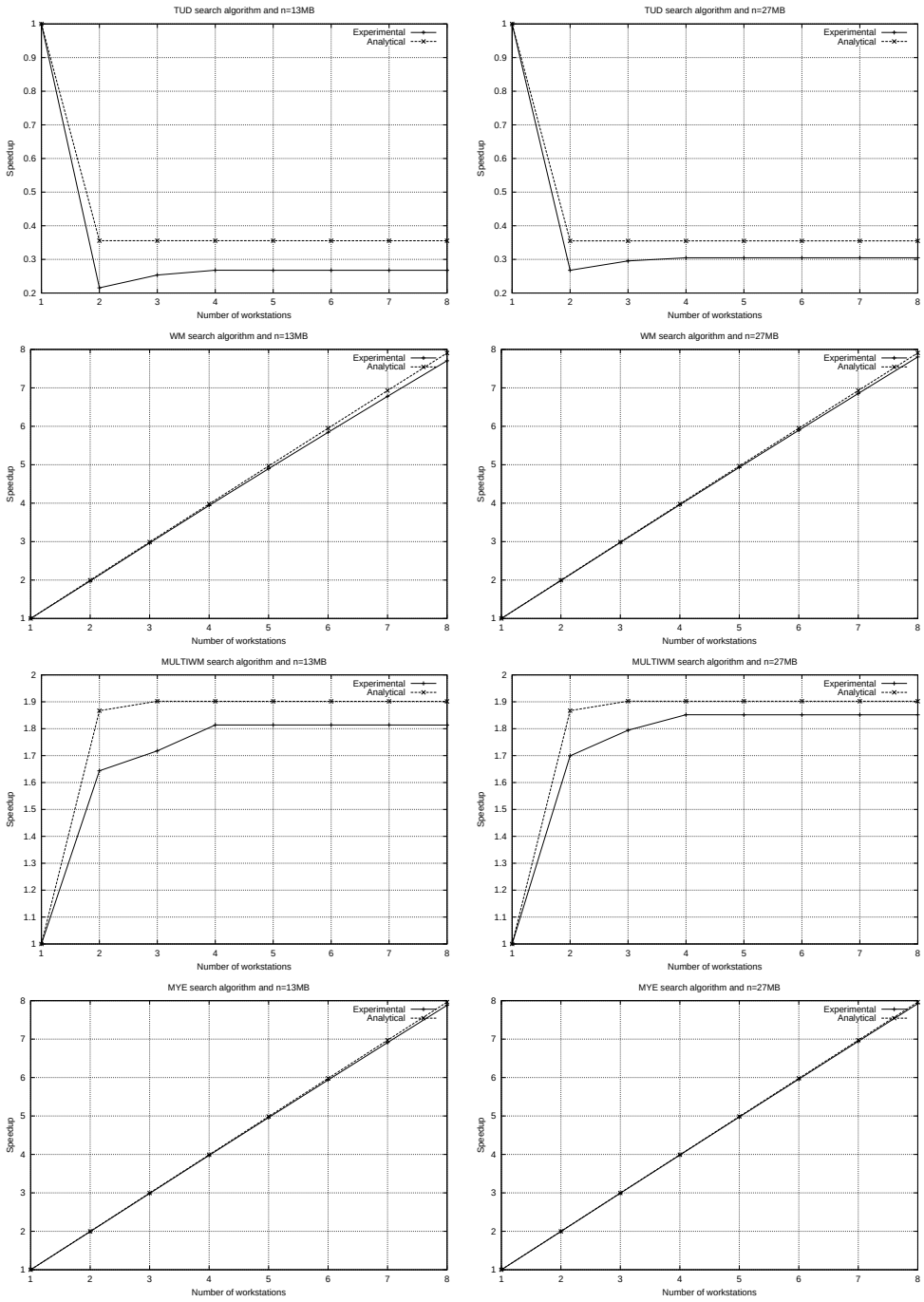
Σχήμα 5.19: Αναλυτική και πειραματική επιτάχυνση σε συνάρτηση με τον αριθμό των σταθμών εργασίας για την υλοποίηση P2 χρησιμοποιώντας τους αλγορίθμους BYP, TU και SO



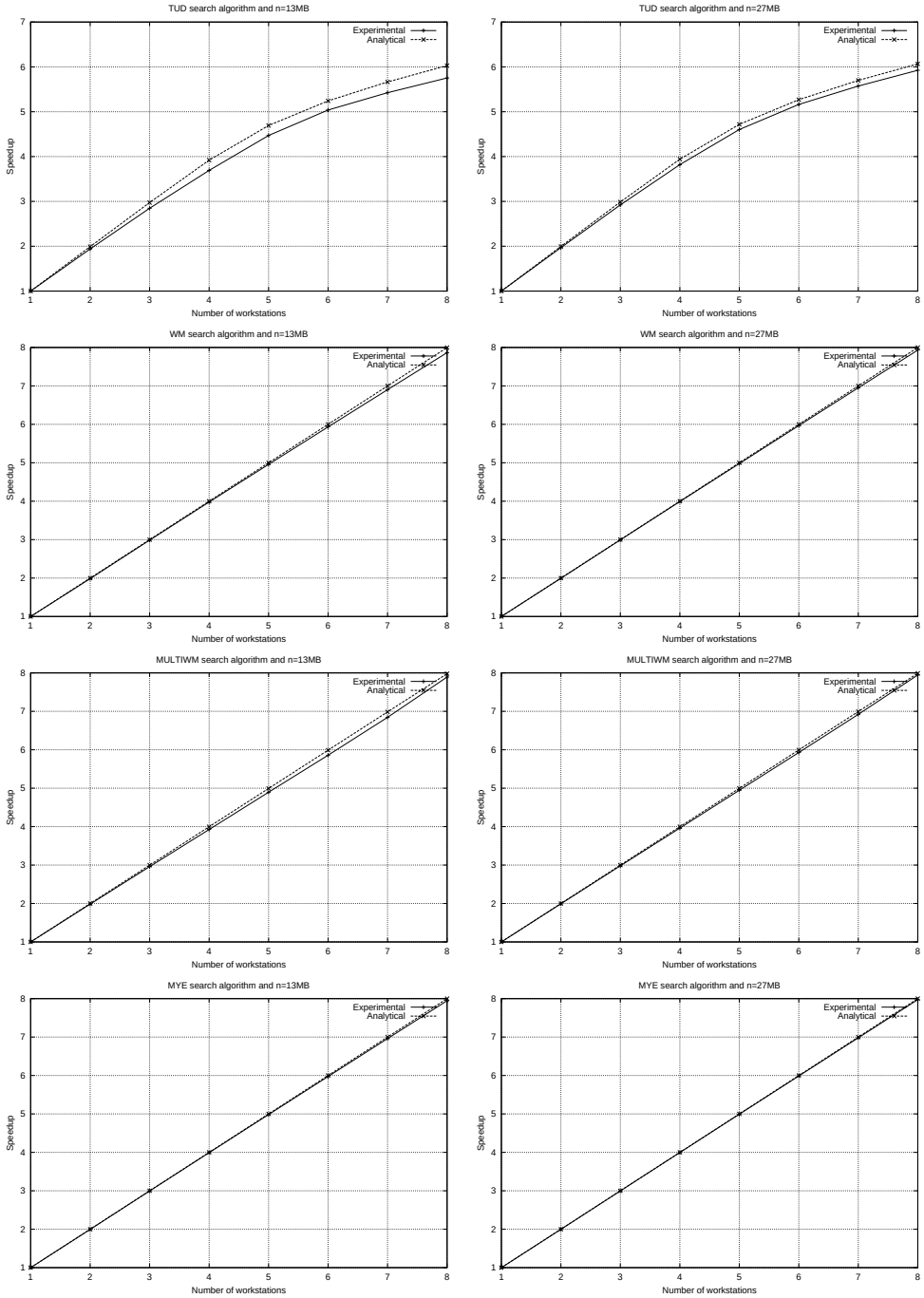
Σχήμα 5.20: Αναλυτική και πειραματική επιτάχυνση σε συνάρτηση με τον αριθμό των σταθμών εργασίας για την υλοποίηση P3 χρησιμοποιώντας τους αλγορίθμους BYP, TU και SO



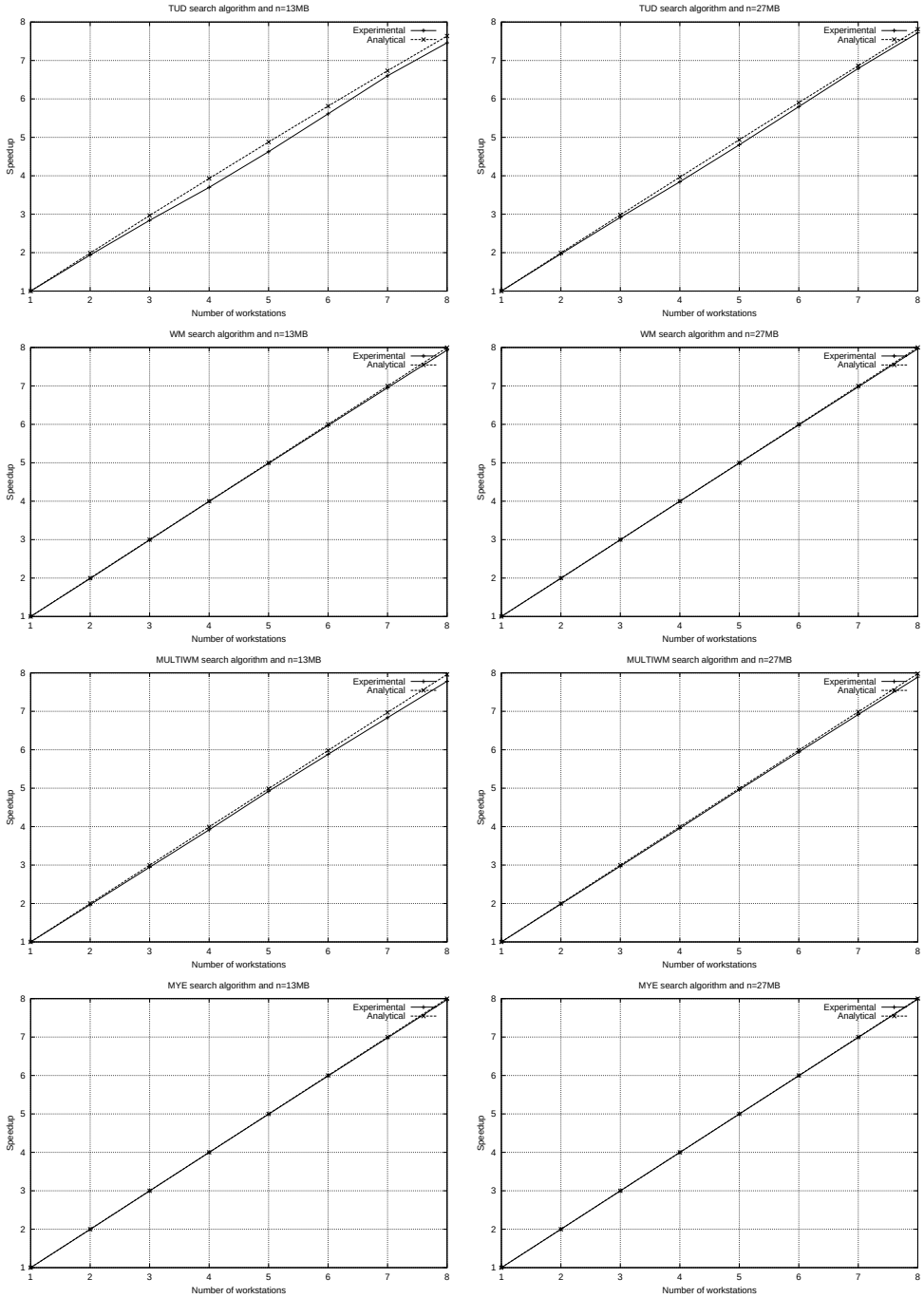
Σχήμα 5.21: Αναλυτική και πειραματική επιτάχυνση σε συνάρτηση με τον αριθμό των σταθμών εργασίας για την υλοποίηση P4 χρησιμοποιώντας τους αλγορίθμους BYP, TU και SO



Σχήμα 5.22: Αναλυτική και πειραματική επιτάχυνση σε συνάρτηση με τον αριθμό των σταθμών εργασίας για την υλοποίηση P2 χρησιμοποιώντας τους αλγορίθμους TUD, WM, MULTIWM και MYE



Σχήμα 5.23: Αναλυτική και πειραματική επιτάχυνση σε συνάρτηση με τον αριθμό των σταθμών εργασίας για την υλοποίηση P3 χρησιμοποιώντας τους αλγορίθμους TUD, WM, MULTIWM και MYE



Σχήμα 5.24: Αναλυτική και πειραματική επιτάχυνση σε συνάρτηση με τον αριθμό των σταθμών εργασίας για την υλοποίηση P4 χρησιμοποιώντας τους αλγορίθμους TUD, WM, MULTIWM και MYE

Κεφάλαιο 6

Παράλληλες Υλοποιήσεις για Αναζήτηση Αλφαριθμητικών σε Ετερογενές Σύστημα

6.1 Εισαγωγή

Γνωρίζουμε ότι στην υπάρχουσα βιβλιογραφία δεν υπάρχει προηγούμενη έρευνα σχετικά με τις παράλληλες υλοποιήσεις αναζήτησης αλφαριθμητικών σε καινούργια παράλληλη αρχιτεκτονική όπως μια κατανεμημένη συστοιχία από ετερογενείς σταθμούς εργασίας (cluster of heterogeneous workstations). Έτσι, σε αυτό το κεφάλαιο παρουσιάζουμε τέσσερις υλοποιήσεις αναζήτησης αλφαριθμητικών σε παράλληλη αρχιτεκτονική από ετερογενείς σταθμούς εργασίας με σκοπό να κερδίσουμε χρόνο σε χαμηλό κόστος. Οι τρεις πρώτες υλοποιήσεις είναι ίδιες με αυτές που παρουσιάστηκαν στο προηγούμενο κεφάλαιο που βασίζονται στο μοντέλο συντονιστής - εργαζόμενος χρησιμοποιώντας την στατική και δυναμική διανομή κειμένου. Όμως, προτείνουμε μια τέταρτη αναθεωρημένη υβριδική παράλληλη υλοποίηση όπου λαμβάνουμε υπόψη τους ετερογενείς σταθμούς εργασίας. Συνεπώς, η υβριδική υλοποίηση βασίζεται στην ακόλουθη βέλτιστη στρατηγική διανομής: η συλλογή κειμένου διανέμεται ανάλογα με την ταχύτητα του κάθε σταθμού εργασίας. Οι τέσσερις προσεγγίσεις υλοποιούνται χρησιμοποιώντας τη βιβλιοθήκη MPI [162, 136, 152, 165] πάνω σε μια συστοιχία από ετερογενείς σταθμούς εργασίας.

Τέλος, προτείνουμε ένα θεωρητικό μοντέλο πρόβλεψης απόδοσης για τις τέσσερις υλοποιήσεις αναζήτησης αλφαριθμητικών σε μια συστοιχία από ετερογενείς σταθμούς εργασίας που συνδέονται με δίκτυο Fast Ethernet. Όπως γνωρίζουμε σε αυτό το κεφάλαιο γίνεται μια πρώτη προσπάθεια για υλοποίηση και μοντελοποίηση της απόδοσης της εφαρμογής αναζήτησης αλφαριθμητικών χρησιμοποιώντας την στατική και δυναμική διανομή σε μια κατανεμημένη συστοιχία από ετερογενείς σταθμούς εργασίας.

Το κεφάλαιο αυτό είναι διαθρωμένο ως εξής: στην ενότητα 6.2 παρουσιάζεται το ετερογενές υπολογιστικό μοντέλο και τα μέτρα απόδοσης που είναι απαραίτητα για να συγκρίνουμε παράλληλες υλοποιήσεις σε ετερογενείς σταθμούς εργασίας. Στην ενότητα 6.3 παρουσιάζεται μόνο η τέταρτη αναθεωρημένη υβριδική παράλληλη υλοποίηση σε μια συστοιχία από ετερογενείς σταθμούς εργασίας που βασίζεται στο προγραμματιστικό μοντέλο συντονιστής - εργαζόμενος. Στην ενότητα 6.4 παρουσιάζονται πειραματικά αποτελέσματα των τεσσάρων παράλληλων υλοποιήσεων για τα προβλήματα απλής και προσεγγιστικής αναζήτησης αλφαριθμητικών. Στην ενότητα 6.5 προτείνουμε ένα θεωρητικό μοντέλο πρόβλεψης απόδοσης για τις τέσσερις παράλληλες υλοποιήσεις. Τέλος, στην ενότητα 6.6 παρουσιάζονται αναλυτικά αποτελέσματα από το θεωρητικό μοντέλο και τα συγκρίνουμε με τα πειραματικά αποτελέσματα.

6.2 Ετερογενές Υπολογιστικό Μοντέλο

Ένα ετερογενές δίκτυο (heterogeneous network - HN) μπορεί να απεικονιστεί σαν ένας συνεκτικός γράφος HN(M,C) όπου

- $M = \{M_1, M_2, ..., M_p\}$ είναι ένα σύνολο από ετερογενείς σταθμούς εργασίας (όπου p είναι ο αριθμός των σταθμών εργασίας). Η χωρητικότητα υπολογισμού (computation capacity) του κάθε σταθμού εργασίας προσδιορίζεται από την ισχύ της CPU, της Ε/Ε και της ταχύτητας προσπέλασης μνήμης (memory access speed).
- C είναι ένα δίκτυο διασύνδεσης (interconnection network) για τους σταθμούς εργασίας, όπως τα δίκτυα Fast Ethernet ή ATM, όπου οι σύνδεσμοι επικοινωνίας ανάμεσα σε οποιοδήποτε ζεύγος σταθμών εργασίας έχουν την ίδια ταχύτητα (bandwidth).

Σύμφωνα με τον προηγούμενο ορισμό, αν το δίκτυο αποτελείται από ένα σύνολο ομοίων σταθμών εργασίας, τότε το σύστημα είναι ομοιογενές. Επίσης, ένα ετερογενές δίκτυο μπορεί να χωριστεί σε

δύο κατηγορίες: ένα αποκλειστικό σύστημα (dedicated system) όπου κάθε σταθμός εργασίας εκτελεί αποκλειστικά εργασίες ενός παράλληλου υπολογισμού και μη αποκλειστικό σύστημα (non-dedicated system) όπου κάθε σταθμός εργασίας εκτελεί τις κανονικές του ρουτίνες και χρησιμοποιεί μόνο τους αδρανείς κύκλους της CPU για να εκτελέσει παράλληλες εργασίες. Σε αυτό το κεφάλαιο χρησιμοποιούμε ένα αποκλειστικό δίκτυο από ετερογενείς σταθμούς εργασίας.

6.2.1 Μέτρα

Τα μέτρα (metrics) βοηθάνε για να συγκρίνουμε και να χαρακτηρίζουμε παράλληλα υπολογιστικά συστήματα. Τα μέτρα που παρουσιάζουμε σε αυτή την ενότητα ορίστηκαν και δημοσιεύτηκαν σε προηγούμενη εργασία [169]. Τα μέτρα διακρίνονται σε μέτρα χαρακτηρισμού (characterization metrics) και μέτρα απόδοσης (performance metrics).

Μέτρα Χαρακτηρισμού

Για να υπολογίσουμε το βάρος δύναμης (power weight) μεταξύ των σταθμών εργασίας ορίζουμε το παρακάτω απλό μέτρο ως εξής:

$$W_i(A) = \frac{min_{j=1}^{p}\{T(A, M_j)\}}{T(A, M_i)} \tag{6.1}$$

όπου A είναι μια εφαρμογή και $T(A, M_i)$ είναι ο χρόνος εκτέλεσης για τον υπολογισμό της εφαρμογής A στο σταθμό εργασίας M_i. Ο τύπος 6.1 δείχνει ότι το βάρος δύναμης ενός σταθμού εργασίας αναφέρει την υπολογιστική του ταχύτητα σε σχέση με το ταχύτερο σταθμό εργασίας στο δίκτυο. Η τιμή του βάρους δύναμη είναι λιγότερη από ή ίση με την 1.

Για να υπολογίσουμε το χρόνο εκτέλεσης ενός υπολογιστικού κομματιού (computational segment) δηλαδή, χρειάζεται να υπολογίσουμε τη ταχύτητα (speed), που συμβολίζεται με S_f ενός ταχύτερου σταθμού εργασίας εκτελώντας τις βασικές πράξεις μιας εφαρμογής που ορίζεται από την ακόλουθη εξίσωση:

$$S_f = \frac{\Theta(c)}{t_c} \tag{6.2}$$

όπου c είναι ένα υπολογιστικό κομμάτι, $\Theta(c)$ είναι μία συνάρτηση πολυπλοκότητας (complexity function) που δίνει τον αριθμό των βασικών πράξεων σε ένα υπολογιστικό κομμάτι και t_c είναι ο χρόνος εκτέλεσης του c στον ταχύτερο σταθμό εργασίας στο δίκτυο.

Χρησιμοποιώντας την ταχύτητα του ταχύτερου σταθμού εργασίας, S_f, μπορούμε να υπολογίζουμε τις ταχύτητες των άλλων σταθμών εργασίας του συστήματος, που συμβολίζονται με S_i $(i = 1, ..., p)$, χρησιμοποιώντας το βάρος δύναμης ως εξής:

$$S_i = S_f * W_i, i = 1, ..., p, \text{ και } i \neq f \tag{6.3}$$

όπου W_i είναι το βάρος δύναμης του M_i. Έτσι, από την εξίσωση 6.3, ο χρόνος εκτέλεσης του κομματιού c σε οποιοδήποτε αποκλειστικό σταθμό εργασίας M_i $(1 \leq i \leq p)$ που συμβολίζεται με $T_{cpu}(c,M_i)$, μπορεί να υπολογιστεί ως εξής:

$$T_{cpu}(c, M_i) = \frac{\Theta(c)}{S_f * W_i} = \frac{\Theta(c)}{S_i} \tag{6.4}$$

Το T_{cpu} θεωρείται ο απαιτούμενος χρόνος CPU για το κομμάτι.

Συνεπώς, ο παράλληλος χρόνος εκτέλεσης ενός κομματιού c μέσω του ετερογενούς δικτύου HN, που συμβολίζεται με T_{cpu}(c,HN), μπορεί να υπολογιστεί ως εξής:

$$T_{cpu}(c, HN) = \frac{\Theta(c)}{\sum_{i=1}^{p} S_i} \tag{6.5}$$

όπου $\sum_{i=1}^{p} S_i$ είναι η υπολογιστική χωρητικότητα που προκύπτει από τον άθροισμα των ταχυτήτων των ξεχωριστών σταθμών εργασίας που χρησιμοποιούνται στο δίκτυο.

Μέτρα Απόδοσης

Η επιτάχυνση (speedup) χρησιμοποιείται για να ποσοτικοποιήσει το κέρδος απόδοσης από ένα παράλληλο υπολογισμό μιας εφαρμογής A σε σχέση με τον υπολογισμό του σε μια μηχανή του ετερογενούς δικτύου. Η επιτάχυνση ενός ετερογενούς υπολογισμού δίνεται από:

$$SP(A) = \frac{min_{j=1}^{p}\{T(A, M_j)\}}{T(A, HN)} \tag{6.6}$$

όπου T(A,HN) είναι ο συνολικός παράλληλος χρόνος εκτέλεσης για την εφαρμογή A σε HN, και T(A,M_j) είναι ο χρόνος εκτέλεσης της εφαρμογής A σε σταθμό εργασίας M_j, j=1,...,p.

Απόδοση (efficiency) ή βαθμός χρήσης (utilization) είναι μια μέτρηση του ποσοστού χρόνου για την οποία η μηχανή είναι απασχολημένη στον παράλληλο υπολογισμό. Συνεπώς, ο βαθμός χρήσης του παράλληλου υπολογισμού της εφαρμογής A σε αποκλειστικό ετερογενές δίκτυο ορίζεται ως εξής:

$$E = \frac{SP(A)}{\sum_{j=1}^{p} W_j} \tag{6.7}$$

Ο προηγούμενος τύπος δείχνει ότι αν η επιτάχυνση είναι μεγαλύτερη από την δύναμη του συστήματος (system computing power) $\sum_{j=1}^{p} W_j$, τότε ο υπολογισμός παρουσιάζει μια υπερ-γραμμική επιτάχυνση (superlinear speedup) σε ένα αποκλειστικό ετερογενές δίκτυο.

6.3 Παράλληλες Υλοποιήσεις για Αναζήτηση Αλφαριθμητικών

Σε αυτή την ενότητα παρουσιάζουμε μόνο την τέταρτη υβριδική υλοποίηση για αναζήτηση αλφαριθμητικών σε ετερογενές σύστημα. Οι άλλες τρεις παράλληλες υλοποιήσεις είναι ακριβώς ίδιες όπως περιγράφηκαν στο προηγούμενο κεφάλαιο της διατριβής μας. Πρέπει να σημειώσουμε εδώ ότι η τέταρτη υβριδική υλοποίηση σε ομοιογενές σύστημα δεν μπορεί να εφαρμοστεί αποτελεσματικά στην καινούργια αρχιτεκτονική του ετερογενούς συστήματος και για αυτό το λόγο προτείνουμε μια νέα υβριδική παράλληλη υλοποίηση.

6.3.1 MPI Υβριδικό Μοντέλο Συντονιστής - Εργαζόμενος

Η νέα αυτή υβριδική υλοποίηση συντονιστής - εργαζόμενος συνδυάζει τα πλεονεκτήματα των τριών παράλληλων υλοποιήσεων (δηλαδή, τις τρεις υλοποιήσεις σε ομοιογενές σύστημα) προκειμένου να μειώσει την ανισορροπία του φορτίου και την επιβάρυνση επικοινωνίας. Η νέα αυτή υλοποίηση βασίζεται σε μια βέλτιστη στρατηγική διανομής της συλλογής κειμένου η οποία εκτελείται στατικά. Στην επόμενη υποενότητα περιγράφουμε τη βέλτιστη στρατηγική διανομής κειμένου.

Βέλτιστη Στρατηγική Διανομής Κειμένου

Για να αποφύγουμε την περίπτωση οι αργοί σταθμοί εργασίας να προσδιορίζουν τον παράλληλο χρόνο αναζήτησης αλφαριθμητικών, το φορτίο πρέπει να διανέμεται ανάλογα με την χωρητικότητα του κάθε σταθμού εργασίας. Ο στόχος μας είναι να αναθέσουμε σε κάθε σταθμό εργασίας την ποσότητα κειμένου η οποία θα οδηγήσει σε ίσο χρόνο εκτέλεσης.

Επομένως, για να πραγματοποιήσουμε μια καλή ισορροπημένη διανομή μεταξύ των ετερογενών σταθμών εργασίας, η ποσότητα του κειμένου που διανέμεται σε κάθε σταθμό εργασίας πρέπει να είναι ανάλογη με την υπολογιστική χωρητικότητα σε σχέση με ολόκληρο το δίκτυο:

$$l_i = \frac{S_i}{\sum_{j=1}^{p} S_j} \tag{6.8}$$

Συνεπώς, η ποσότητα της συλλογής κειμένου που διανέμεται σε κάθε σταθμό εργασίας M_i $(1 \leq i \leq p)$ είναι $l_i * (n + m - 1)$ διαδοχικοί χαρακτήρες. Υπάρχει μια επικάλυψη των $m - 1$ χαρακτήρων προτύπου ανάμεσα στα διαδοχικά υπο-κείμενα.

Η υβριδική υλοποίηση που ονομάζεται P4 είναι ίδια με την υλοποίηση P1 του ομοιογενούς συστήματος αλλά χρησιμοποιούμε τη βέλτιστη μέθοδο διανομής κειμένου αντί την ίση διανομή.

6.4 Πειραματικά Αποτελέσματα

Σε αυτή την ενότητα παρουσιάζουμε τα πειραματικά αποτελέσματα για την απόδοση των τεσσάρων κατανεμημένων υλοποιήσεων για τα προβλήματα απλής και προσεγγιστικής αναζήτησης αλφαριθμητικών. Οι παράλληλοι μέθοδοι υλοποιήθηκαν σε γλώσσα προγραμματισμού ANSI C χρησιμοποιώντας τη βιβλιοθήκη MPI [162, 136, 152, 165].

6.4.1 Πειραματικό Περιβάλλον

Η πλατφόρμα για την πειραματική μας μελέτη είναι μια συστοιχία από ετερογενείς σταθμούς εργασίας συνδεδεμένη με δίκτυο 100 Mb/s Fast Ethernet. Συγκεριμένα, η συστοιχία αποτελείται από τέσσερις επεξεργαστές Pentium MMX 166 MHz με 32 MB μνήμη και πέντε Pentium 100 MHz με 64 MB μνήμη.

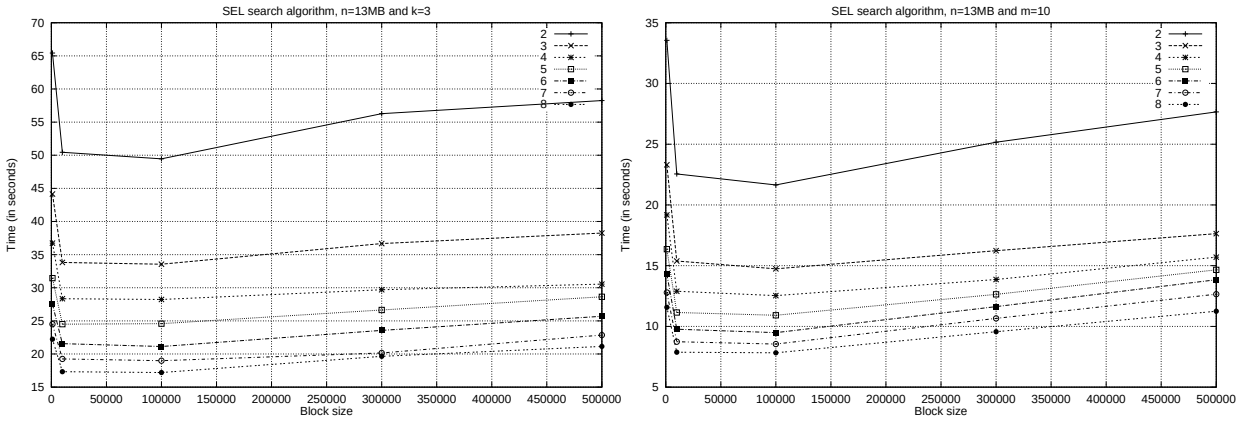
Το ένα Pentium MMX χρησιμοποιείται ως συντονιστής. Η υλοποίηση MPI που χρησιμοποιήθηκε στο δίκτυο είναι η MPICH έκδοση 1.2. Κατά τη διάρκεια των πειραμάτων, το ετερογενές σύστημα σταθμών εργασίας ήταν αποκλειστικό. Τέλος, για να πάρουμε αξιόπιστα αποτελέσματα απόδοσης τρέξαμε το κάθε πείραμα δέκα φορές και αναφέρουμε αποτελέσματα από τον μέσο όρο των δέκα εκτελέσεων. Η συλλογή κειμένου που χρησιμοποιήσαμε προερχόταν από διάφορες ιστοσελίδες του διαδικτύου.

6.4.2 Πειραματικά Αποτελέσματα

Σε αυτή την υποενότητα παρουσιάζουμε πειραματικά αποτελέσματα από τρία σύνολα πειραμάτων. Για το πρώτο πειραματικό σύνολο εξετάζουμε την απόδοση των δύο δυναμικών υλοποιήσεων P2 και P3 σε συνάρτηση με το μέγεθος του τμήματος (block). Για το δεύτερο πειραματικό σύνολο συγκρίνουμε την απόδοση των τεσσάρων υλοποιήσεων P1, P2, P3 και P4. Τέλος, για το τρίτο πειραματικό σύνολο εξετάζουμε το ζήτημα της κλιμάκωσης των υλοποιήσεων μας διπλασιάζοντας τη συλλογή κειμένου.

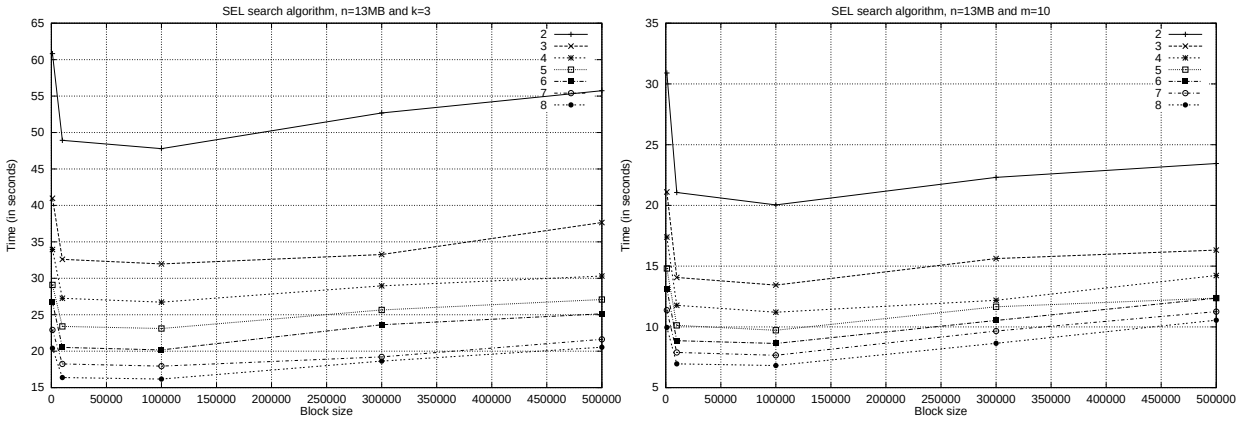
Επίδραση του Μεγέθους Τμήματος

Τα Σχήματα 6.1 και 6.2 δείχνουν την απόδοση των δύο δυναμικών υλοποιήσεων P2 και P3 σε συνάρτηση με το μέγεθος του τμήματος για διαφορετικό αριθμό σταθμών εργασίας και για μέγεθος κειμένου 13MB. Παρομοίως, τα Σχήματα 6.3 και 6.4 δείχνουν τους πειραματικούς χρόνους εκτέλεσης των δύο υλοποιήσεων P2 και P3 για μέγεθος κειμένου 27MB. Σημειώνουμε ότι οι χρόνοι εκτέλεσης που απεικονίζονται στα Σχήματα 6.1, 6.2, 6.3 και 6.4 αριστερά είναι αποτέλεσμα του μέσου όρου για πέντε διαφορετικά μήκη προτύπων ($m = 5, 10, 20, 30$ και 60) για σταθερή τιμή $k = 3$ και στα Σχήματα 6.1, 6.2, 6.3 και 6.4 δεξιά είναι αποτέλεσμα του μέσου όρου για τέσσερις διαφορετικές τιμές του k ($k = 1, 3, 6$ και 9) για σταθερό μήκος προτύπου $m = 10$. Από τα πειραματικά αποτελέσματα παρατηρούμε ότι χρησιμοποιώντας διαφορετικές τιμές τμήματος καταλήγουμε σε διαφορετικές αποδόσεις. Η χειρότερη απόδοση προκύπτει για πολύ μικρές και μεγάλες τιμές του τμήματος. Αυτό συμβαίνει επειδή οι μικρές τιμές του τμήματος αυξάνουν την επιβάρυνση της διαεπεξεργαστικής επικοινωνίας, ενώ οι μεγάλες τιμές του τμήματος παρέχουν χαμηλή ισορροπία φορτίου. Όμως, από τα Σχήματα προκύπτει ότι υπάρχει μια βέλτιστη τιμή τμήματος κοντά στους 100,000 χαρακτήρες για τις υλοποιήσεις P2 και P3, που δίνει την καλύτερη απόδοση.

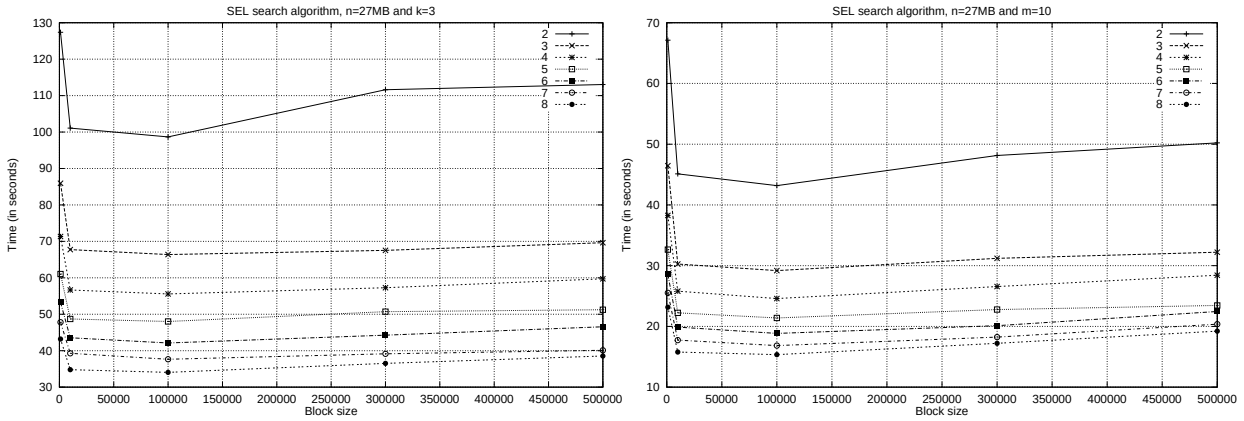


Σχήμα 6.1: Πειραματικοί χρόνοι εκτέλεσης για την υλοποίηση P2 σε σχέση με το μέγεθος του τμήματος για διαφορετικό αριθμό σταθμών εργασίας, για μέγεθος κειμένου 13MB και $k = 3$ (αριστερά) και για μέγεθος κειμένου 13MB και $m = 10$ (δεξιά)

Έχοντας προσδιορίσει ότι το μέγεθος του τμήματος κοντά στους 100,000 χαρακτήρες είναι αποτελεσματικό για τις υλοποιήσεις P2 και P3, εκτελέσαμε τα παρακάτω πειράματα χρησιμοποιώντας την τιμή αυτή για τις δύο δυναμικές υλοποιήσεις.



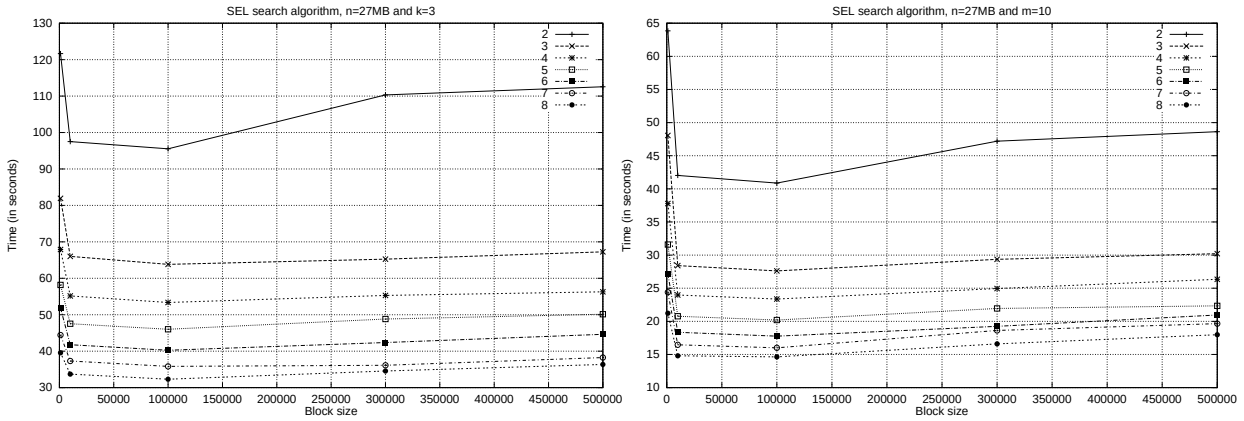
Σχήμα 6.2: Πειραματικοί χρόνοι εκτέλεσης για την υλοποίηση P3 σε σχέση με το μέγεθος του τμήματος για διαφορετικό αριθμό σταθμών εργασίας, για μέγεθος κειμένου 13MB και $k = 3$ (αριστερά) και για μέγεθος κειμένου 13MB και $m = 10$ (δεξιά)



Σχήμα 6.3: Πειραματικοί χρόνοι εκτέλεσης για την υλοποίηση P2 σε σχέση με το μέγεθος του τμήματος για διαφορετικό αριθμό σταθμών εργασίας, για μέγεθος κειμένου 27MB και $k = 3$ (αριστερά) και για μέγεθος κειμένου 27MB και $m = 10$ (δεξιά)

Σύγκριση Τεσσάρων Παράλληλων Υλοποιήσεων

Ο Πίνακας 6.1 δείχνει τους χρόνους εκτέλεσης για σταθερή τιμή $k = 3$ και διαφορετικά μήκη προτύπων, για διαφορετικό αριθμό σταθμών εργασίας και για τον αλγόριθμο προσεγγιστικής αναζήτησης αλφαριθμητικών SEL. Ο Πίνακας 6.2 δείχνει τους χρόνους εκτέλεσης για σταθερό μήκος προτύπου $m = 10$ και διαφορετικές τιμές του k για τον αλγόριθμο SEL. Επίσης, το Σχήμα 6.5 παρουσιάζει επιταχύνσεις



Σχήμα 6.4: Πειραματικοί χρόνοι εκτέλεσης για την υλοποίηση P3 σε σχέση με το μέγεθος του τμήματος για διαφορετικό αριθμό σταθμών εργασίας, για μέγεθος κειμένου 27MB και $k = 3$ (αριστερά) και για μέγεθος κειμένου 27MB και $m = 10$ (δεξιά)

Πίνακας 6.1: Πειραματικοί χρόνοι εκτέλεσης (σε δευτερόλεπτα) για μεγέθος κειμένου 13MB και $k = 3$ για διαφορετικά μήκη προτύπων χρησιμοποιώντας τον αλγόριθμο SEL

m/p	Παρ. Υλοπ.	1	2	3	4	5	6	7	8
5	P1	21.025	10.546	7.088	9.193	7.389	6.265	5.398	4.748
	P2	20.850	12.321	8.393	7.149	6.254	5.439	4.928	4.537
	P3	20.801	10.727	7.194	6.021	5.216	4.645	4.089	3.720
	P4	21.025	10.546	7.088	6.108	5.260	4.638	4.205	3.765
10	P1	39.406	19.696	13.176	16.849	13.833	11.413	9.791	8.587
	P2	39.405	21.608	14.734	12.468	10.896	9.463	8.533	7.828
	P3	39.285	19.969	13.379	11.172	9.673	8.468	7.602	6.814
	P4	39.406	19.696	13.176	11.241	9.661	8.456	7.603	6.850
20	P1	76.337	38.168	25.487	32.434	25.926	21.812	18.705	16.387
	P2	76.492	40.152	27.274	22.968	19.941	17.309	15.563	14.236
	P3	76.248	38.491	25.744	21.524	18.608	16.265	14.483	13.098
	P4	76.337	38.168	25.487	21.551	18.531	16.214	14.523	13.066
30	P1	113.614	56.815	37.923	48.000	38.393	32.149	27.595	24.164
	P2	113.903	58.871	39.927	33.532	29.237	25.264	22.702	20.673
	P3	113.550	57.199	38.239	31.971	27.651	24.144	21.548	19.449
	P4	113.614	56.815	37.923	31.875	27.366	23.962	21.429	19.293
60	P1	224.077	112.060	74.753	94.560	75.644	63.500	54.874	48.114
	P2	224.773	114.330	77.415	65.056	56.569	48.197	43.238	38.754
	P3	224.123	112.567	75.308	62.923	54.392	47.314	42.024	37.802
	P4	224.077	112.060	74.753	62.770	53.829	47.396	42.348	38.420

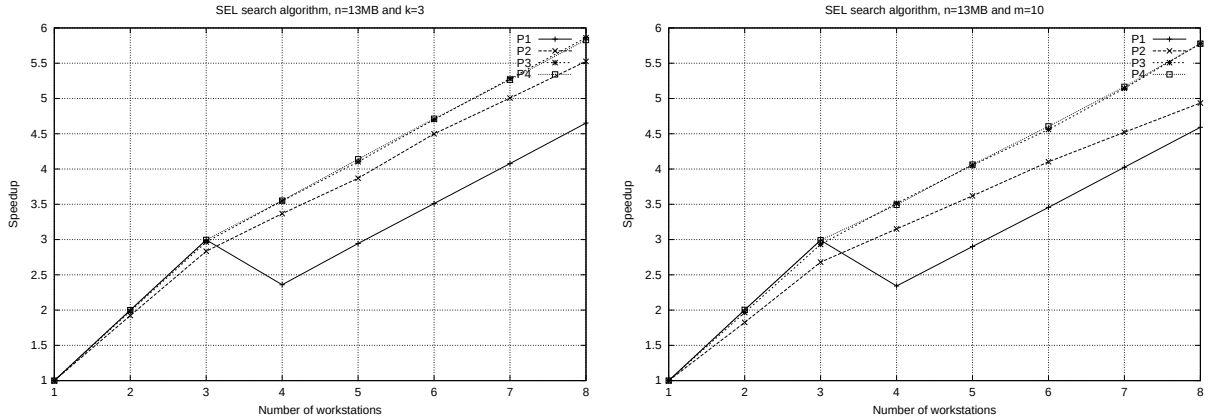
σε συνάρτηση με τον αριθμό των σταθμών εργασίας χρησιμοποιώντας τον αλγόριθμο SEL. Είναι σημαντικό να σημειωθεί ότι οι επιταχύνσεις που απεικονίζονται στο Σχήμα 6.5 αριστερά είναι αποτέλεσμα του μέσου όρου για πέντε διαφορετικά μήκη προτύπων και στο Σχήμα 6.5 δεξιά είναι αποτέλεσμα του μέσου όρου για τέσσερις διαφορετικές τιμές του k.

Όπως αναμέναμε, τα αποτελέσματα απόδοσης δείχνουν ότι η υλοποίηση P1 χρησιμοποιώντας την στατική στρατηγική είναι λιγότερο αποτελεσματική απο τις άλλες τρεις υλοποιήσεις στην περίπτωση του ετερογενούς συστήματος. Επίσης από το Σχήμα 6.5 παρατηρούμε ότι υπάρχει μια απότομη πτώση στις καμπύλες των επιταχύνσεων της υλοποίησης P1 στους τέσσερις σταθμούς εργασίας. Το φαινόμενο αυτό οφείλεται στο γεγονός ότι υπάρχει μια ανισορροπία του φορτίου εξαιτίας της φτωχής τεχνικής διαμερισμού κειμένου που χρησιμοποιήθηκε στην υλοποίηση P1. Με άλλα λόγια, ο αργός σταθμός εργασίας τελειώνει πάντα τελευταίος στην αναζήτηση αλφαριθμητικών και υπάρχει σημαντικός χρόνος αδράνειας για τους ταχύτερους σταθμούς εργασίας σε ένα ετερογενές περιβάλλον. Επίσης, η υλοποίηση

Πίνακας 6.2: Πειραματικοί χρόνοι εκτέλεσης (σε δευτερόλεπτα) για μεγέθος κειμένου 13MB και $m = 10$ για διαφορετικές τιμές του k χρησιμοποιώντας τον αλγόριθμο SEL

k/p	Παρ. Υλοπ.	1	2	3	4	5	6	7	8
1	P1	39.485	19.696	13.196	16.848	13.562	11.437	9.820	8.609
	P2	39.445	21.627	14.712	12.529	10.908	9.488	8.532	7.834
	P3	39.344	20.028	13.429	11.194	9.713	8.673	7.659	6.821
	P4	39.485	19.696	13.196	11.316	9.738	8.599	7.661	6.825
3	P1	39.406	19.696	13.176	16.849	13.833	11.413	9.791	8.587
	P2	39.405	21.608	14.734	12.468	10.896	9.463	8.533	7.828
	P3	39.285	19.969	13.379	11.172	9.673	8.468	7.602	6.814
	P4	39.406	19.696	13.176	11.241	9.661	8.456	7.603	6.850
6	P1	39.467	19.703	13.201	16.860	13.496	11.416	9.816	8.605
	P2	39.451	21.602	14.713	12.534	10.908	9.468	8.541	7.818
	P3	39.358	20.028	13.448	11.201	9.710	8.677	7.662	6.818
	P4	39.467	19.703	13.201	11.316	9.727	8.615	7.646	6.834
9	P1	39.726	19.822	13.281	16.952	13.591	11.478	9.843	8.624
	P2	39.696	21.754	14.799	12.602	10.945	9.536	8.583	7.833
	P3	39.593	20.141	13.512	11.269	9.774	8.728	7.713	6.823
	P4	39.726	19.822	13.281	11.381	9.783	8.656	7.691	6.850

P2 χρησιμοποιώντας την δυναμική διανομή υπο-κειμένων παρέχει καλύτερα αποτελέσματα από την υλοποίηση P1 στην περίπτωση του ετερογενές συστήματος. Τέλος, τα πειραματικά αποτελέσματα οι



Σχήμα 6.5: Επιτάχυνση των παράλληλων υλοποιήσεων αναζήτησης αλφαριθμητικών σε συνάρτηση με τον αριθμό των σταθμών εργασίας για μέγεθος κειμένου 13MB και $k = 3$ χρησιμοποιώντας διαφορετικά μήκη προτύπων (αριστερά) και $m = 10$ χρησιμοποιώντας διαφορετικές τιμές του k (δεξιά)

υλοποιήσεις P3 και P4 φαίνονται να έχουν την καλύτερη απόδοση σε σύγκριση με τις άλλες υλοποιήσεις. Επίσης, οι υλοποιήσεις P3 και P4 δίνουν τους μικρότερους χρόνους εκτέλεσης και τις υψηλότερες επιταχύνσεις από τις υλοποιήσεις P1 και P2 όταν προσθέτουμε στη συστοιχία ετερογενείς σταθμούς εργασίας δηλαδή μετά τον τρίτο σταθμό εργασίας.

Τώρα εξετάζουμε την απόδοση των παράλληλων υλοποιήσεων P2, P3 και P4. Από τα αποτελέσματα παρατηρούμε μια σαφή μείωση στο χρόνο εκτέλεσης του αλγορίθμου SEL όταν χρησιμοποιούμε τις τρεις παράλληλες υλοποιήσεις. Για παράδειγμα, για $k = 3$ και $m = 10$, μειώνεται ο χρόνος εκτέλεσης από 39.406 δευτερόλεπτα στην ακολουθιακή έκδοση σε 7.828, 6.814 και 6.850 δευτερόλεπτα στις κατανεμημένες υλοποιήσεις P2, P3 και P4 αντίστοιχα, χρησιμοποιώντας τους οκτώ σταθμούς εργασίας. Με άλλα λόγια, από τους Πίνακες παρατηρούμε ότι για σταθερό μέγεθος συλλογής κειμένου υπάρχει μια αναμενόμενη αντίστροφη σχέση ανάμεσα στον παράλληλο χρόνο εκτέλεσης και στον αριθμό των σταθμών εργασίας. Όσον αφορά το μήκος του προτύπου παρατηρούμε ότι υπάρχει μια σημαντική επίδραση στην απόδοση των τριών υλοποιήσεων. Συγκεριμένα, βλέπουμε ότι για σταθερό αριθμό σταθμών εργασίας ο παράλληλος χρόνος εκτέλεσης των υλοποιήσεων αυξάνεται καθώς αυξάνεται το μήκος του προτύπου. Όμως, ο χρόνος επικοινωνίας είναι περίπου ο ίδιος σε όλες τις περιπτώσεις για ένα δεδομένο μέγεθος κειμένου αφού η επίδραση του μήκους του προτύπου στην επικοινωνία δεν είναι τόσο σημαντική. Συνεπώς, ο λόγος του χρόνου υπολογισμού προς το χρόνο επικοινωνίας εξαρτάται κυρίως από το μήκος του προτύπου και αυξάνεται όταν το μήκος του προτύπου αυξάνεται.

Επίσης, από τους Πίνακες παρατηρούμε ότι υπάρχει μια σταθερή σχέση ανάμεσα στο χρόνο εκτέλεσης και τον αριθμό των διαφορών k. Συνεπώς, δεν υπάρχει διαφορά στην απόδοση των παράλληλων υλοποιήσεων για διαφορετικές τιμές του k και οι μετρήσεις μας επιβεβαιώνουν την θεωρητική πολυπλοκότητα του αλγορίθμου δυναμικού προγραμματισμού SEL [145]. Το φαινόμενο αυτό σε πολλές περιπτώσεις είναι επιθυμητό όπου ένα όριο στην τιμή του k δεν είναι γνωστό εκ των προτέρων. Επιπλέον, οι τρεις υλοποιήσεις πραγματοποιούν λογικές επιταχύνσεις για όλους τους σταθμούς εργασίας. Για παράδειγμα, για $k = 3$ και διαφορετικά μήκη προτύπων έχουμε αυξανόμενες καμπύλες επιταχύνσεις μέχρι 5.52, 5.86 και 5.90 στις κατανεμημένες μεθόδους P2, P3 και P4 αντίστοιχα στους οκτώ σταθμούς εργασίας που είχαν το θεωρητικό βάρος δύναμης των 5.35, 5.90 και 5.97.

Ζήτημα Κλιμάκωσης

Για να μελετήσουμε την κλιμάκωση των τριών προτεινόμενων παράλληλων υλοποιήσεων P2, P3 και P4, κατασκευάσαμε τα πειράματα με το ακόλουθο τρόπο. Διπλασιάζουμε την παλιά συλλογή κειμένου. Η καινούργια συλλογή κειμένου είναι γύρω στα 27MB. Τα αποτελέσματα από τα πειράματα απεικονίζονται στους Πίνακες 6.3 και 6.4 και στο Σχήμα 6.6. Τα αποτελέσματα δείχνουν ότι οι τρεις παράλληλες υλοποιήσεις κλιμακώνουν καλά παρόλο που το μέγεθος του κειμένου αυξήθηκε δύο φορές (δηλαδή, διπλασιάστηκε η συλλογή κειμένου). Οι χρόνοι εκτέλεσης για $k = 3$ και $m = 10$ μειώνονται σε 15.343, 13.605 και 13.634 δευτερόλεπτα για τις υλοποιήσεις P2, P3 και P4 αντίστοιχα όταν ο αριθμός των σταθμών εργασίας είναι οκτώ. Επίσης, οι επιταχύνσεις των τριών μεθόδων αυξάνουν γραμμικά καθώς ο αριθμός των σταθμών εργασίας αυξάνεται. Όμως, οι καμπύλες επιτάχυνσης αυξάνουν ελαφρώς καθώς αυξάνεται το μέγεθος του κειμένου. Σημειώνουμε ότι οι επιταχύνεις των τριών μεθόδων είναι πολύ κοντά στο ιδανικό βάρος δύναμης του συστήματος, $\sum_{j=1}^{p} W_j$, για όλους τους σταθμούς εργασίας. Το φαινόμενο αυτό εμφανίζεται όταν το μέγεθος του κειμένου είναι αρκετά μεγάλο με αποτέλεσμα να εκτελούνται παράλληλα παρά πολύ υπολογισμοί αναζήτησης σε κάθε σταθμό εργασίας παρά την επικοινωνία με το συντονιστή. Όμως, η επιβάρυνση επικοινωνίας ανάμεσα στο συντονιστή και τους εργαζόμενους αυξάνεται ελαφρώς καθώς αυξάνεται το μέγεθος του κειμένου αλλά η επιβάρυνση αυτή δεν φαίνεται να επηρεάζει στην συνολική απόδοση των παράλληλων υλοποιήσεων. Συνεπώς, ο λόγος του χρόνου υπολογισμού προς τον χρόνο επικοινωνίας είναι αρκετά υψηλός. Τέλος, σε αυτή την μεγάλη συλλογή κειμένου τα καλύτερα αποτελέσματα απόδοσης προκύπτουν με τις μεθόδους P3 και P4.

6.4.3 Αποτελέσματα Άλλων Αλγορίθμων Αναζήτησης Αλφαριθμητικών

Το Σχήμα 6.7 παρουσιάζει επιταχύνσεις σε σχέση με τον αριθμό των σταθμών εργασίας χρησιμοποιώντας τους αλγορίθμους της ακριβούς αναζήτησης όπως QS, RF, SO και BNDM. Παρόμοια, το Σχήμα 6.8 παρουσιάζει επιταχύνσεις για μέγεθος κειμένου 27MB. Επίσης, τα Σχήματα 6.9 και 6.10 παρουσιάζουν τις επιταχύνσεις για μεγέθη κειμένου 13MB και 27MB αντίστοιχα για τους αλγορίθμους αναζήτησης αλφαριθμητικών με k αποτυχίες όπως TU, BYP και SO. Τέλος, τα Σχήματα 6.11 και 6.12 παρουσιάζουν τις επιταχύνσεις για μεγέθη κειμένου 13MB και 27MB αντίστοιχα για τους αλγορίθμους αναζήτησης αλφαριθμητικών με k διαφορές όπως TUD, WM, MULTIWM και MYE.

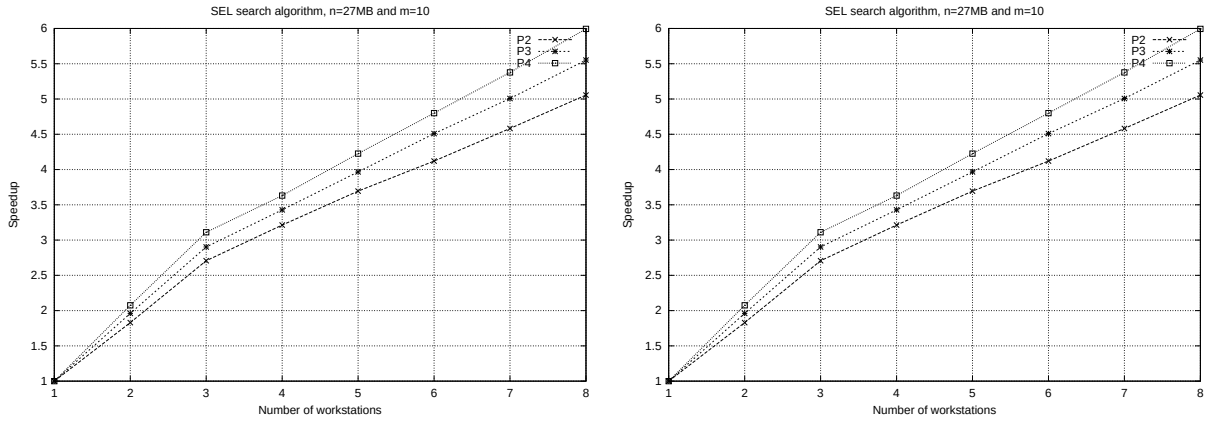
Πίνακας 6.3: Πειραματικοί χρόνοι εκτέλεσης (σε δευτερόλεπτα) για μεγέθος κειμένου 27MB και $k = 3$ για διαφορετικά μήκη προτύπων χρησιμοποιώντας τον αλγόριθμο SEL

m/p	Παρ. Υλοπ.	1	2	3	4	5	6	7	8
5	P2	41.768	24.439	16.620	13.999	12.257	10.832	9.814	8.944
	P3	43.152	21.429	14.366	12.043	10.410	9.214	8.170	7.410
	P4	46.123	23.858	14.517	12.513	10.742	9.393	8.483	7.563
10	P2	78.935	43.067	29.121	24.548	21.238	18.772	16.849	15.343
	P3	80.141	39.917	26.715	22.341	19.284	16.907	15.126	13.605
	P4	81.866	42.653	26.351	22.503	19.312	16.891	15.200	13.634
20	P2	153.260	80.071	53.999	45.156	39.140	34.508	30.594	27.765
	P3	154.228	76.942	51.415	42.996	37.066	32.455	28.903	26.099
	P4	155.599	79.787	50.998	43.100	37.004	32.403	29.014	26.053
30	P2	228.309	117.473	79.018	66.094	57.021	50.203	44.611	40.358
	P3	228.995	114.321	76.357	63.880	55.096	48.223	42.999	38.791
	P4	230.121	117.163	75.835	63.753	54.703	47.887	42.825	38.466
60	P2	450.178	228.247	153.179	128.051	110.479	96.287	86.336	77.904
	P3	450.598	225.033	150.323	125.576	108.186	94.447	83.967	75.586
	P4	451.510	227.677	149.502	125.466	107.626	94.810	84.638	76.534

Από τα παραπάνω αποτελέσματα σημειώνουμε ότι η απόδοση της υλοποίησης P2 χρησιμοποιώντας τους αλγορίθμους QS, RF, BNDM, TU, TUD και MULTIWM δεν παρουσιάζει την καλύτερη συμπεριφορά και είναι σαφώς χειρότερη και από την υλοποίηση P1. Το γεγονός αυτό οφείλεται στο ότι ο

Πίνακας 6.4: Πειραματικοί χρόνοι εκτέλεσης (σε δευτερόλεπτα) για μεγέθος κειμένου 27MB και $m = 10$ για διαφορετικές τιμές του k χρησιμοποιώντας τον αλγόριθμο SEL

k/p	Παρ. Υλοπ.	1	2	3	4	5	6	7	8
1	P2	78.844	43.132	29.158	24.554	21.383	18.836	16.792	15.353
	P3	79.710	40.741	27.530	23.314	20.150	17.687	15.964	14.605
	P4	82.224	39.587	26.420	22.712	19.521	17.179	15.284	13.716
3	P2	78.935	43.067	29.121	24.548	21.238	18.772	16.849	15.343
	P3	80.141	40.855	27.616	23.348	20.190	17.884	16.009	14.597
	P4	81.866	39.494	26.351	22.503	19.312	16.891	15.200	13.718
6	P2	78.881	43.153	29.151	24.550	21.400	18.830	16.808	15.354
	P3	79.964	40.798	27.550	23.335	20.176	17.701	15.975	14.620
	P4	82.289	39.604	26.402	22.617	19.448	17.181	15.292	13.723
9	P2	79.376	43.376	29.314	24.698	21.531	18.976	16.894	15.361
	P3	80.742	41.113	27.770	23.499	20.307	17.818	16.072	14.748
	P4	82.573	39.822	26.565	22.754	19.548	17.285	15.393	13.717

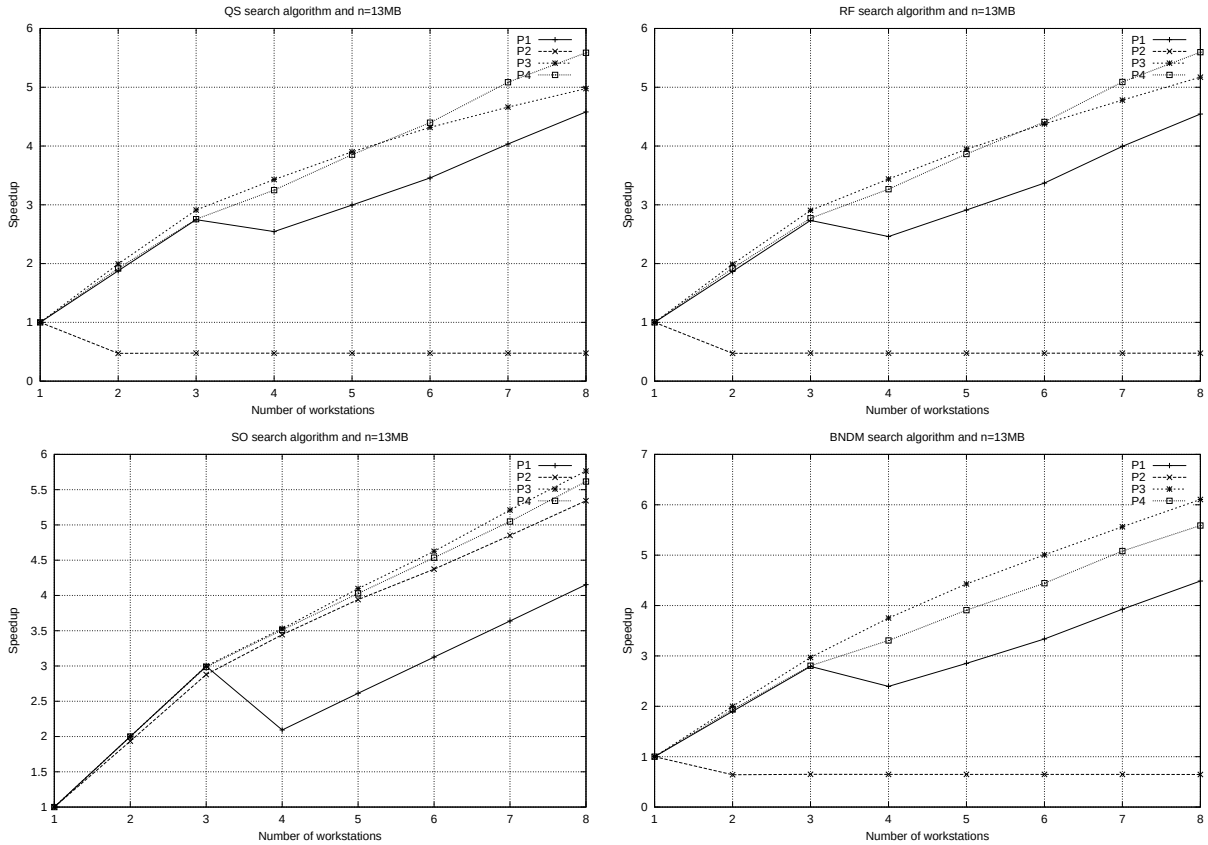


Σχήμα 6.6: Επιτάχυνση των παράλληλων υλοποιήσεων αναζήτησης αλφαριθμητικών σε συνάρτηση με τον αριθμό των σταθμών εργασίας για μέγεθος κειμένου 27MB και $k = 3$ χρησιμοποιώντας διαφορετικά μήκη προτύπων (αριστερά) και $m = 10$ χρησιμοποιώντας διαφορετικές τιμές του k (δεξιά)

χρόνος αναζήτησης των αλγορίθμων αυτών είναι παρά πολύ μικρός σε σχέση με τον χρόνο επικοινωνίας. Αφού οι παραπάνω αλγόριθμοι είναι ταχύτεροι από τους απλούς ακολουθιακούς αλγορίθμους όπως SEL, SO, WM και MYE. Συνεπώς, ο λόγος του χρόνου υπολογισμού προς τον χρόνο επικοινωνίας της υλοποίησης P2 είναι πολύ χαμηλός. Τέλος, όσον αφορά την συμπεριφορά των υπολοίπων υλοποιήσεων P1, P3 και P4 είναι ίδια όπως έχουμε παρουσιάσει για τον αλγόριθμο SEL.

6.5 Αναλυτικό Μοντέλο Πρόβλεψης Απόδοσης

Σε αυτή την ενότητα αναπτύσσουμε ένα αναλυτικό μοντέλο πρόβλεψης απόδοσης για να περιγράψουμε την υπολογιστική συμπεριφορά των τεσσάρων παράλληλων υλοποιήσεων αναζήτησης αλφαριθμητικών. Το μοντέλο που περιγράφεται εδώ είναι λίγο διαφορετικό από το μοντέλο σε ομοιογενές σύστημα αφού λαμβάνουμε υπόψη μια νέα αρχιτεκτονική όπως το ετερογενές σύστημα.

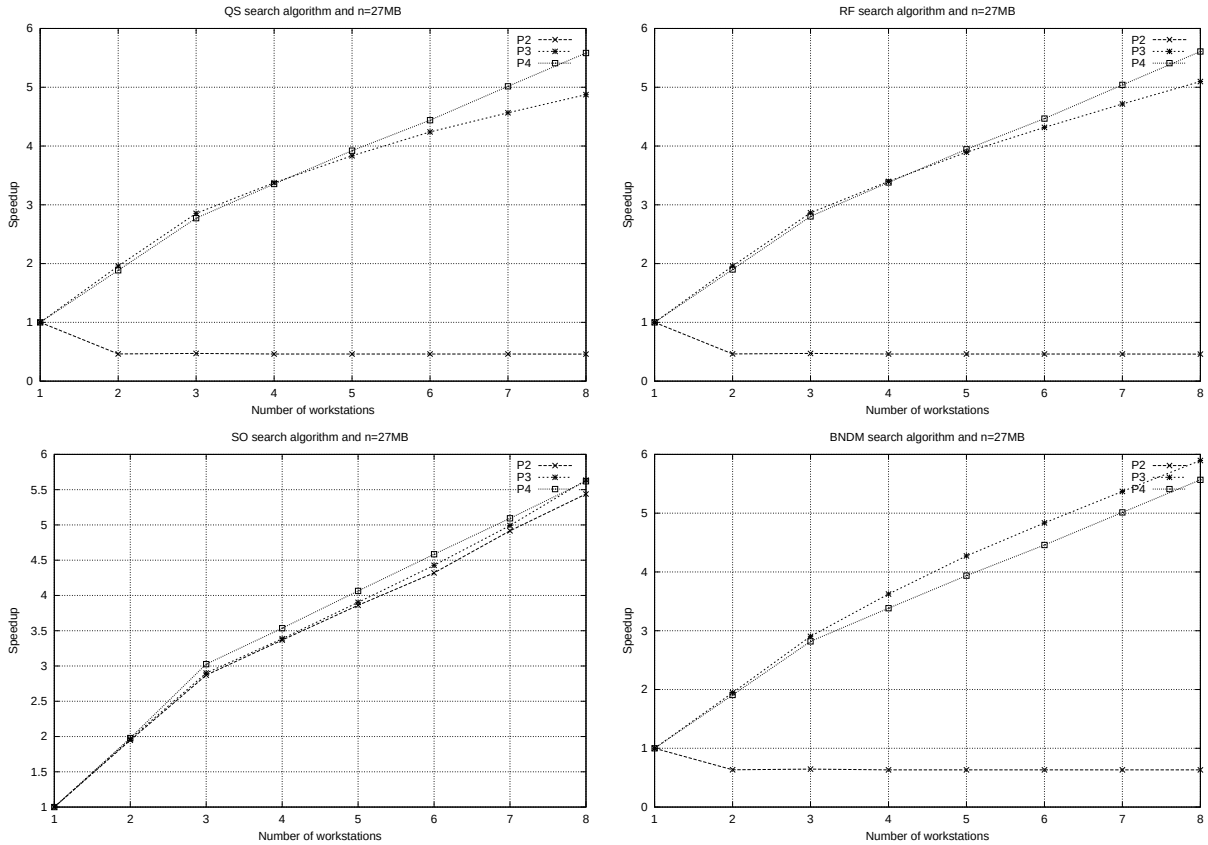


Σχήμα 6.7: Επιτάχυνση των παράλληλων υλοποιήσεων αναζήτησης αλφαριθμητικών σε συνάρτηση με τον αριθμό των σταθμών εργασίας για μέγεθος κειμένου 13MB χρησιμοποιώντας τους αλγορίθμους QS, RF, SO και BNDM

6.5.1 Αναλυτική Μοντελοποίηση για την Στατική Υλοποίηση Συντονιστής - Εργαζόμενος

Υποθέτουμε ότι T_a, T_b, T_c και T_d είναι οι χρόνοι που καταναλώνονται σε κάθε από τις τέσσερις φάσεις της υλοποίησης P1 αντίστοιχα. Τα αναλυτικά κόστη (σε χρόνο) είναι ως εξής:

Η πρώτη φάση περιλαμβάνει το χρόνο επικοινωνίας για να γίνει η εκπομπή του πρότυπου αλφαριθμητικού και του αριθμού των αποτυχιών ή διαφορών k σε όλους τους σταθμούς εργασίας που περιλαμβάνονται στην επεξεργασία αναζήτησης αλφαριθμητικών. Χρησιμοποιήσαμε την συνάρτηση MPI_Bcast για την εκπομπή των πληροφοριών που ολοκληρώνεται σε $log_2 p$ βήματα. Το μέγεθος ενός



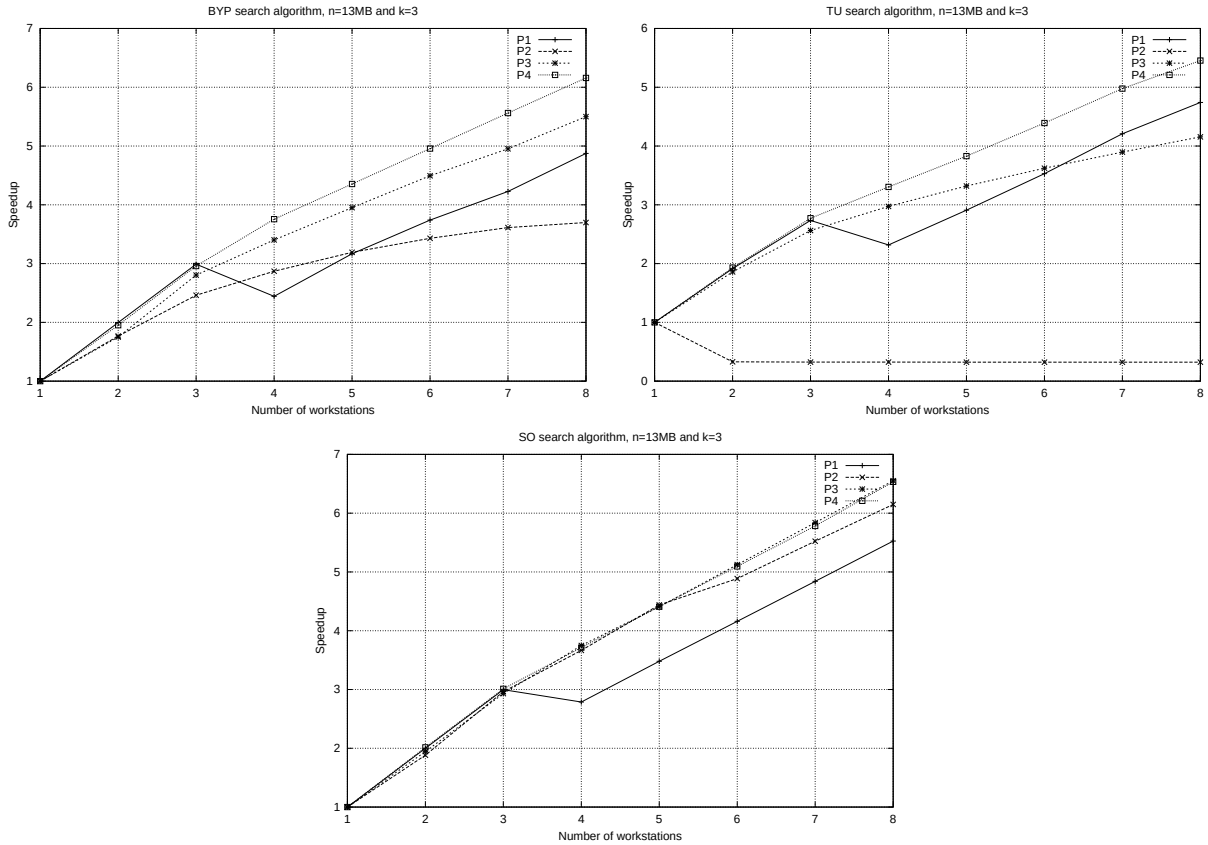
Σχήμα 6.8: Επιτάχυνση των παράλληλων υλοποιήσεων αναζήτησης αλφαριθμητικών σε συνάρτηση με τον αριθμό των σταθμών εργασίας για μέγεθος κειμένου 27MB χρησιμοποιώντας τους αλγορίθμους QS, RF, SO και BNDM

m προτύπου αλφαριθμητικού είναι m bytes και ο αριθμός των αποτυχιών ή διαφορών k είναι 1 byte. Συνεπώς, η εκπομπή μεταφέρει $m+1$ bytes στους άλλους $p-1$ σταθμούς εργασίας. Ο χρόνος T_a για την πρώτη φάση δίνεται από:

$$T_a = log_2 p(\alpha + (m+1)\beta) \tag{6.9}$$

Η δεύτερη φάση είναι ο μέσος χρόνος Ε/Ε για να διαβαστούν τα αντίστοιχα υπο-κείμενα μεγέθους $\lceil n/p \rceil + m - 1$ χαρακτήρες από τους τοπικούς δίσκους των διαφόρων σταθμών εργασίας. Τότε, ο χρόνος T_b για την δεύτερη φάση είναι ως εξής:

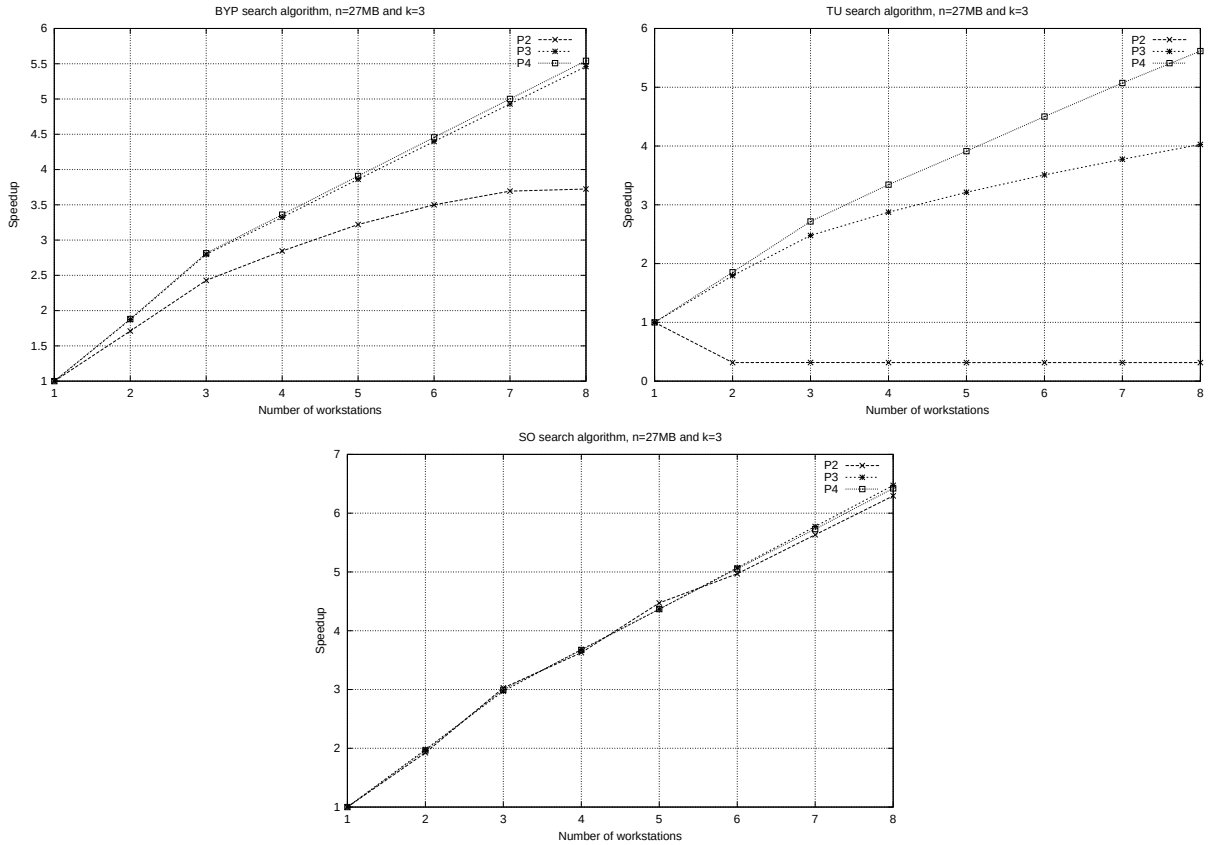
$$T_b = max_{j=1}^{p}\{\frac{\lceil n/p \rceil + m - 1}{(S_{E/E})_j}\} \tag{6.10}$$



Σχήμα 6.9: Επιτάχυνση των παράλληλων υλοποιήσεων αναζήτησης αλφαριθμητικών σε συνάρτηση με τον αριθμό των σταθμών εργασίας για μέγεθος κειμένου 13MB και $k = 3$ χρησιμοποιώντας διαφορετικά μήκη προτύπων χρησιμοποιώντας τους αλγορίθμους BYP, TU και SO

όπου $(S_{E/E})_j$ είναι χωρητικότητα E/E του ετερογενούς συστήματος όταν χρησιμοποιείται ο σταθμός εργασίας j.

Η τρίτη φάση είναι ο μέσος χρόνος αναζήτησης αλφαριθμητικών συμπεριλαμβανομένου και του χρόνου προεπεξεργασίας διαμέσου του ετερογενούς συστήματος. Κάθε σταθμός εργασίας πρέπει να εκτελέσει την προεπεξεργασία του προτύπου και στην συνέχεια να αναζητήσει ένα m πρότυπο αλφαριθμητικό σε ένα υπο-κείμενο μεγέθους $\lceil n/p \rceil + m - 1$ χαρακτήρων. Η πολυπλοκότητα προεπεξεργασίας που εκτελείται σε κάθε σταθμό εργασίας για το m πρότυπο εκφράζεται από την συνάρτηση $\Theta_{prep}(m, k)$ και η πολυπλοκότητα αναζήτησης που εκτελείται σε κάθε εργαζόμενο σε ένα υπο-κείμενο εκφράζεται από την συνάρτηση $\Theta_{search}(\lceil \frac{n}{p} \rceil + m - 1, m, k)$ για οποιοδήποτε ακολουθιακό αλγόριθμο αναζήτησης



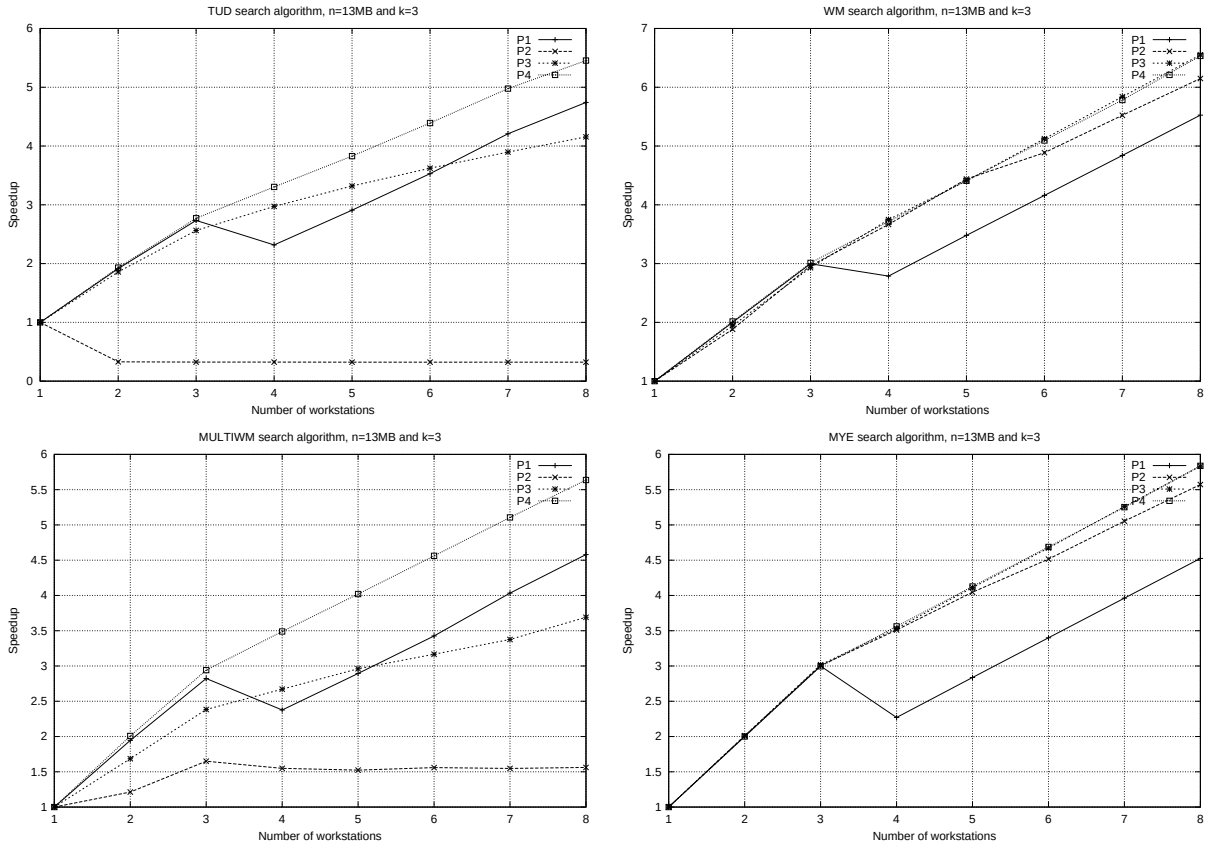
Σχήμα 6.10: Επιτάχυνση των παράλληλων υλοποιήσεων αναζήτησης αλφαριθμητικών σε συνάρτηση με τον αριθμό των σταθμών εργασίας για μέγεθος κειμένου 27MB και $k = 3$ χρησιμοποιώντας διαφορετικά μήκη προτύπων χρησιμοποιώντας τους αλγορίθμους BYP, TU και SO

αλφαριθμητικών. Τότε, ο χρόνος T_c για την φάση αναζήτησης αλφαριθμητικών δίνεται από:

$$T_c = max_{j=1}^{p}\{\frac{\Theta_{prep}(m,k) + \Theta_{search}(\lceil n/p \rceil + m - 1, m, k)}{(S_{search})_j}\} \tag{6.11}$$

όπου $(S_{search})_j$ είναι η χωρητικότητα αναζήτησης αλφαριθμητικών του ετερογενούς συστήματος όταν χρησιμοποιείται ο σταθμός εργασίας j.

Τέλος, η τέταρτη φάση περιλαμβάνει το χρόνο επικοινωνίας για να συλλεγούν p αποτελέσματα από την αναζήτηση αλφαριθμητικών που εκτελέστηκαν στα αντίστοιχα υπο-κείμενα από τους p σταθμούς εργασίας παράλληλα. Κάθε σταθμός εργασίας στέλνει πίσω μια τιμή (στην περίπτωση μας τον αριθμό των εμφανίσεων). Χρησιμοποιήσαμε την συνάρτηση MPI_Reduce για να συλλέξουμε τα αποτελέσματα



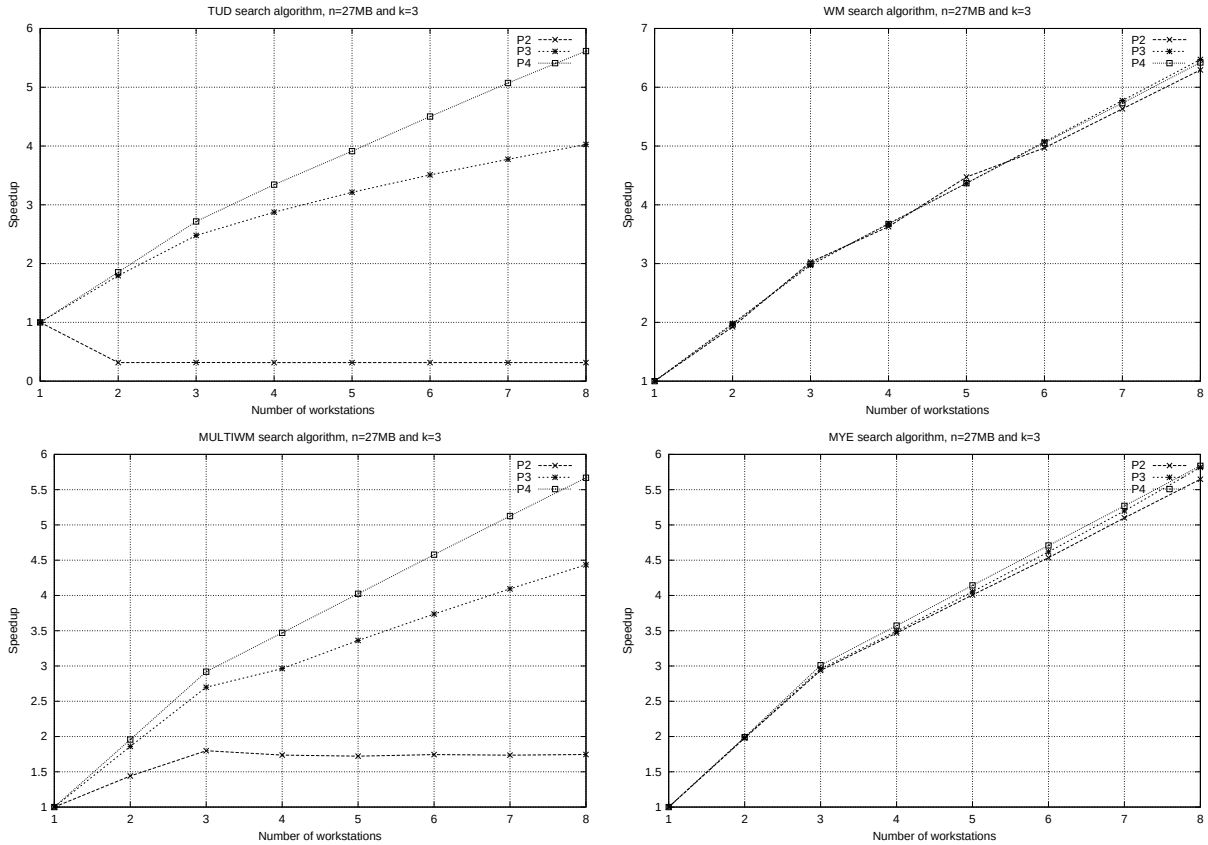
Σχήμα 6.11: Επιτάχυνση των παράλληλων υλοποιήσεων αναζήτησης αλφαριθμητικών σε συνάρτηση με τον αριθμό των σταθμών εργασίας για μέγεθος κειμένου 13MB και $k = 3$ χρησιμοποιώντας διαφορετικά μήκη προτύπων χρησιμοποιώντας τους αλγορίθμους TUD, WM, MULTIWM και MYE

που ολοκληρώνεται σε $log_2 p$ βήματα. Συνεπώς, ο χρόνος T_d για αυτή την φάση είναι ως εξής:

$$T_d = log_2 p(\alpha + \beta) \tag{6.12}$$

Ο συνολικός χρόνος εκτέλεσης της στατικής υλοποίησης αναζήτησης αλφαριθμητικών, T_p, χρησιμοποιώντας τους p σταθμούς εργασίας, είναι το άθροισμα των τεσσάρων φάσεων και δίνεται από:

$$T_p = T_a + T_b + T_c + T_d \tag{6.13}$$



Σχήμα 6.12: Επιτάχυνση των παράλληλων υλοποιήσεων αναζήτησης αλφαριθμητικών σε συνάρτηση με τον αριθμό των σταθμών εργασίας για μέγεθος κειμένου 27MB και $k = 3$ χρησιμοποιώντας διαφορετικά μήκη προτύπων χρησιμοποιώντας τους αλγορίθμους TUD, WM, MULTIWM και MYE

6.5.2 Αναλυτική Μοντελοποίηση για Δυναμικές Υλοποιήσεις Συντονιστής - Εργαζόμενος

Σε αυτή την υποενότητα μοντελοποιούμε τις δύο εκδόσεις του δυναμικού μοντέλου συντονιστής - εργαζόμενος. Η πρώτη έκδοση βασίζεται στην δυναμική διανομή υπο-κειμένων και την δεύτερη έκδοση βασίζεται στην δυναμική διανομή δεικτών κειμένου.

Δυναμική Διανομή Υπο-κειμένων

Υποθέτουμε ότι T_a, T_b, T_c, T_d και T_e είναι οι χρόνοι που καταναλώνονται σε κάθε από τις πέντε φάσεις της υλοποίησης P2 αντίστοιχα. Τα αναλυτικά κόστη (σε χρόνο) είναι ως εξής:

Η πρώτη φάση περιλαμβάνει τον χρόνο επικοινωνίας για την εκπομπή του προτύπου αλφαριθμητικού και τον αριθμό των αποτυχιών ή διαφορών k σε όλους τους σταθμούς εργασίας. Ο χρόνος T_a για την πρώτη φάση είναι ίδιος με το T_a της προηγούμενης υλοποίησης και ορίζεται ως εξής:

$$T_a = log_2 p(\alpha + (m+1)\beta) \tag{6.14}$$

Η δεύτερη φάση είναι ο μέσος χρόνος Ε/Ε για να διαβαστεί η συλλογή κειμένου σε διάφορα τμήματα μεγέθους $sb + m - 1$ bytes από τον τοπικό δίσκο του συντονιστή. Συνεπώς, ο συντονιστής διαβάζει n bytes συνολικά από την συλλογή κειμένου. Τότε, ο χρόνος T_b για την δεύτερη φάση είναι ως εξής:

$$T_b = \frac{n}{(S_{E/E})_{master}} \tag{6.15}$$

όπου $(S_{E/E})_{master}$ είναι η χωρητικότητα Ε/Ε του σταθμού εργασίας συντονιστής.

Η τρίτη φάση περιλαμβάνει το χρόνο επικοινωνίας για να σταλούν όλα τα τμήματα από την συλλογή κειμένου σε όλους τους εργαζόμενους. Το μέγεθος του κάθε τμήματος είναι $sb + m - 1$ bytes. Τότε ο συνολικός χρόνος επικοινωνίας T_c για την τρίτη φάση είναι ως εξής:

$$T_c = \frac{n}{sb + m - 1}(\alpha + (sb + m - 1)\beta) \tag{6.16}$$

Η τέταρτη φάση είναι ο μέσος χρόνος αναζήτησης αλφαριθμητικών συμπεριλαμβανομένου και του χρόνου προεπεξεργασίας διαμέσου του ετερογενούς συστήματος. Κάθε εργαζόμενος εκτελεί πρώτα την προεπεξεργασία του προτύπου και στην συνέχεια εκτελεί αναζήτηση ενός m προτύπου αλφαριθμητικού σε ένα τμήμα κειμένου μεγέθους $sb + m - 1$ χαρακτήρων. Η πολυπλοκότητα προεπεξεργασίας που εκτελείται σε κάθε σταθμό εργασίας για το πρότυπο εκφράζεται από την συνάρτηση $\Theta_{prep}(m, k)$ και η πολυπλοκότητα αναζήτησης που εκτελείται σε κάθε σταθμό εργασίας για ένα τμήμα κειμένου εκφράζεται από την συνάρτηση $\Theta_{search}(sb + m - 1, m, k)$ για οποιοδήποτε αλγόριθμο αναζήτησης

αλφαριθμητικών. Τότε, ο χρόνος T_d για την φάση αναζήτηση αλφαριθμητικών δίνεται από:

$$T_d = \frac{(\frac{n}{sb+m-1} - p)(\Theta_{prep}(m,k) + \Theta_{search}(sb+m-1,m,k))}{\sum_{j=1}^{p}(S_{search})_j} + \\ max_{j=1}^{p}\{\frac{\Theta_{prep}(m,k) + \Theta_{search}(sb+m-1,m,k))}{(S_{search})_j}\} \quad (6.17)$$

όπου $\sum_{j=1}^{p}(S_{search})_j$ είναι η χωρητικότητα αναζήτησης αλφαριθμητικών του ετερογενούς συστήματος όταν χρησιμοποιούνται p σταθμοί εργασίας. Στην εξίσωση 6.17 συμπεριλάμβαμε ένα δεύτερο όρο max που ορίζει την ανισορροπία του φορτίου στην χειρότερη περίπτωση στο τέλος της εκτέλεσης όταν δεν υπάρχουν αρκετά τμήματα κειμένου για να κρατήσουν όλους τους σταθμούς εργασίας απασχολημένους.

Η πέμπτη φάση είναι ο χρόνος επικοινωνίας για να λάβει ο συντονιστής $\frac{n}{sb+m-1}$ αποτελέσματα από όλους τους εργαζόμενους. Κάθε εργαζόμενος στέλνει πίσω μια τιμή δηλαδή τον αριθμό των εμφανίσεων. Συνεπώς, ο χρόνος επικοινωνίας T_e για να λάβει όλα τα αποτελέσματα ο συντονιστής δίνεται από:

$$T_e = \frac{n}{sb+m-1}(\alpha + \beta) \quad (6.18)$$

Επίσης, σημειώνουμε ότι αυτή η δυναμική υλοποίηση στην πράξη υπάρχει παράλληλη επικοινωνία και υπολογισμός και για αυτό τον λόγο παίρνουμε την μέγιστη τιμή ανάμεσα στον χρόνο επικοινωνίας δηλαδή $T_c + T_e$ και στον χρόνο υπολογισμού δηλαδή T_d. Συνεπώς, ο συνολικός χρόνος εκτέλεσης της δυναμικής υλοποίησης αναζήτησης αλφαριθμητικών, T_p, χρησιμοποιώντας τους p σταθμούς εργασίας, δίνεται από:

$$T_p = T_a + T_b + max\{T_c + T_e, T_d\} \quad (6.19)$$

Δυναμική Διανομή Δεικτών Κειμένου

Υποθέτουμε ότι T_a, T_b, T_c, T_d και T_e είναι οι χρόνοι που καταναλώνονται σε κάθε από τις πέντε φάσεις της υλοποίησης P3 αντίστοιχα. Τα αναλυτικά κόστη (σε χρόνο) είναι ως εξής:

Η πρώτη φάση είναι ο χρόνος επικοινωνίας για την εκπομπή του προτύπου αλφαριθμητικού και τον αριθμό των αποτυχιών ή διαφορών k σε όλους τους σταθμούς εργασίας. Ο χρόνος T_a για την πρώτη

φάση είναι ίδιος με το T_a της προηγούμενης δυναμικής υλοποίησης και ορίζεται ως εξής:

$$T_a = log_2 p(\alpha + (m+1)\beta) \tag{6.20}$$

Η δεύτερη φάση είναι ίδια με την τρίτη φάση της προηγούμενης υλοποίησης P2 αλλά περιλαμβάνει τον χρόνο επικοινωνίας για να σταλούν δείκτες κειμένου αντί τμήματα κειμένου σε όλους τους σταθμούς εργασίας. Συνεπώς, ο χρόνος επικοινωνίας T_b για αυτή την φάση είναι ως εξής:

$$T_b = \frac{n}{sb + m - 1}(\alpha + \beta) \tag{6.21}$$

Η τρίτη φάση είναι ο μέσος χρόνος Ε/Ε για να διαβαστεί η συλλογή κειμένου σε διάφορα τμήματα μεγέθους $sb + m - 1$ bytes από τους τοπικούς δίσκους των σταθμών εργασίας. Συνεπώς, ο μέσος χρόνος Ε/Ε T_c για την τρίτη φάση είναι ως εξής:

$$T_c = \frac{(\frac{n}{sb+m-1} - p)(sb + m - 1)}{\sum_{j=1}^{p}(S_{E/E})_j} + max_{j=1}^{p}\{\frac{sb + m - 1}{(S_{E/E})_j}\} \tag{6.22}$$

όπου $\sum_{j=1}^{p}(S_{E/E})_j$ είναι η χωρητικότητα Ε/Ε του ετερογενούς συστήματος όταν χρησιμοποιούνται p σταθμοί εργασίας. Στην εξίσωση 6.22 συμπεριλάμβαμε ένα δεύτερο όρο max που ορίζει την ανισορροπία του φορτίου στην χειρότερη περίπτωση στο τέλος της εκτέλεσης όταν δεν υπάρχουν αρκετά τμήματα από την συλλογή κειμένου για να κρατήσουν όλους τους σταθμούς εργασίας απασχολημένους. Αυτός ο όρος επίσης αντιστοιχεί στην έκτη φάση της υλοποίησης P3.

Η τέταρτη φάση περιλαμβάνει το μέσο χρόνο αναζήτησης αλφαριθμητικών διάμεσου του ετερογενούς συστήματος. Ο χρόνος T_d είναι ίδιος με τον χρόνο T_d της υλοποίησης P2 και ορίζεται ως εξής:

$$\begin{aligned} T_d = & \frac{(\frac{n}{sb+m-1} - p)(\Theta_{prep}(m,k) + \Theta_{search}(sb + m - 1, m, k))}{\sum_{j=1}^{p}(S_{search})_j} + \\ & max_{j=1}^{p}\{\frac{\Theta_{prep}(m,k) + \Theta_{search}(sb + m - 1, m, k))}{(S_{search})_j}\} \end{aligned} \tag{6.23}$$

όπου $\sum_{j=1}^{p}(S_{search})_j$ είναι η χωρητικότητα αναζήτησης αλφαριθμητικών του ετερογενούς συστήματος όταν χρησιμοποιούνται p σταθμοί εργασίας.

Η πέμπτη φάση περιλαμβάνει το χρόνο επικοινωνίας για να ληφθούν τα αποτελέσματα της αναζήτησης αλφαριθμητικών από όλους τους σταθμούς εργασίας. Συνεπώς, ο χρόνος T_e είναι ίδιος με το χρόνο T_e της υλοποίησης P2 και ορίζεται ως εξής:

$$T_e = \frac{n}{sb + m - 1}(\alpha + \beta) \tag{6.24}$$

Γνωρίζουμε ότι στην υλοποίηση P3 υπάρχει επικάλυψη μεταξύ επικοινωνίας και υπολογισμού και για αυτό τον λόγο παίρνουμε την μέγιστη τιμή ανάμεσα στον χρόνο επικοινωνίας δηλαδή $T_b + T_e$ και στον χρόνο υπολογισμού δηλαδή $T_c + T_d$. Συνεπώς, ο συνολικός χρόνος εκτέλεσης της δυναμικής υλοποίησης αναζήτησης αλφαριθμητικών, T_p, χρησιμοποιώντας τους p σταθμούς εργασίας, δίνεται από:

$$T_p = T_a + max\{T_b + T_e, T_c + T_d\} \tag{6.25}$$

6.5.3 Αναλυτική Μοντελοποίηση για Υβριδική Υλοποίηση Συντονιστής - Εργαζόμενος

Υποθέτουμε ότι T_a, T_b, T_c και T_d είναι οι χρόνοι που καταναλώνονται σε κάθε από τις τέσσερις φάσεις της υλοποίησης P4 αντίστοιχα. Τα αναλυτικά κόστη (σε χρόνο) είναι ως εξής:

Η πρώτη φάση περιλαμβάνει τον χρόνο επικοινωνίας για την εκπομπή του προτύπου αλφαριθμητικού και τον αριθμό των αποτυχιών ή διαφορών k σε όλους τους σταθμούς εργασίας. Ο χρόνος T_a είναι ίδιος με το T_a της υλοποίησης P1 και ορίζεται ως εξής:

$$T_a = log_2 p(\alpha + (m + 1)\beta) \tag{6.26}$$

Η δεύτερη φάση είναι ο χρόνος E/E για να διαβάζει κάθε εργαζόμενος το αντίστοιχο υπο-κείμενο μεγέθους $l_i * (n + m - 1)$ $(1 \leq i \leq p)$ διαδοχικών χαρακτήρων από το τοπικό δίσκο. Τότε, ο χρόνος T_b για την δεύτερη φάση δίνεται από:

$$T_b = max_{j=1}^{p}\{\frac{l_i * (n + m - 1)}{(S_{E/E})_j}\} \tag{6.27}$$

όπου $(S_{E/E})_j$ είναι η χωρητικότητα E/E του ετερογενούς συστήματος όταν χρησιμοποιείται ο σταθμός εργασίας j.

Η τρίτη φάση είναι ο μέσος χρόνος αναζήτησης αλφαριθμητικών συμπεριλαμβανομένου και του χρόνου προεπεξεργασίας διαμέσου του ετερογενούς συστήματος. Κάθε εργαζόμενος εκτελεί πρώτα την προεπεξεργασία του προτύπου και στην συνέχεια εκτελεί αναζήτηση ενός m προτύπου αλφαριθμητικού σε ένα υπο-κείμενο μεγέθους $l_i * (n + m - 1)$ χαρακτήρων. Η πολυπλοκότητα προεπεξεργασίας που εκτελείται σε κάθε σταθμό εργασίας για το πρότυπο εκφράζεται από την συνάρτηση $\Theta_{prep}(m, k)$ και η πολυπλοκότητα αναζήτησης που εκτελείται σε κάθε σταθμό εργασίας για ένα υπο-κείμενο εκφράζεται από την συνάρτηση $\Theta_{search}(l_i * (n + m - 1), m, k)$ για οποιοδήποτε αλγόριθμο αναζήτησης αλφαριθμητικών. Τότε, ο χρόνος T_c για την φάση αναζήτηση αλφαριθμητικών δίνεται από:

$$T_c = max_{j=1}^{p}\{\frac{\Theta_{prep}(m, k) + \Theta_{search}(l_i * (n + m - 1), m, k)}{(S_{search})_j}\} \tag{6.28}$$

όπου $(S_{search})_j$ είναι η χωρητικότητα αναζήτησης αλφαριθμητικών του ετερογενούς συστήματος όταν χρησιμοποιείται ο σταθμός εργασίας j.

Τέλος, η τέταρτη φάση περιλαμβάνει το χρόνο επικοινωνίας για να συλλεγούν τα αποτελέσματα της αναζήτησης αλφαριθμητικών από τους p σταθμούς εργασίας. Τότε, ο χρόνος T_d είναι ίδιος με το T_d της υλοποίησης P1 και ορίζεται ως εξής:

$$T_d = log_2 p(\alpha + \beta) \tag{6.29}$$

Ο συνολικός χρόνος εκτέλεσης της υβριδικής υλοποίησης αναζήτησης αλφαριθμητικών, T_p, χρησιμοποιώντας τους p σταθμούς εργασίας, είναι το άθροισμα των τεσσάρων φάσεων και δίνεται από:

$$T_p = T_a + T_b + T_c + T_d \tag{6.30}$$

Οι πολυπλοκότητες προεπεξεργασίας και αναζήτησης, $\Theta_{prep}(m, k)$ και $\Theta_{search}(n, m, k)$, για τους αλγορίθμους που χρησιμοποιήσαμε στα πειράματα και χρησιμοποιούνται στο μοντέλο πρόβλεψης απόδοσης είναι ίδιες με τον Πίνακα 5.4 που παρουσιάστηκαν στο προηγούμενο κεφάλαιο. Η πολυπλοκότητα αναζήτησης για τον αλγόριθμο SEL είναι mn.

6.6 Αναλυτικά Αποτελέσματα

Σε αυτή την ενότητα παρουσιάζουμε τα αναλυτικά αποτελέσματα των τεσσάρων παράλληλων υλοποιήσεων που προκύπτουν από το προτεινόμενο μοντέλο πρόβλεψης που παρουσιάσαμε προηγουμένως.

6.6.1 Προσδιορισμός των α και β

Οι παράμετροι α και β που χρησιμοποιούνται στην ανάλυση απόδοση είναι $4.11e-04$ δευτερόλεπτα και $9.27e-08$ δευτερόλεπτα αντίστοιχα στον υπολογιστικό μας περιβάλλον. Μετρήσαμε και συλλέξαμε τις τιμές αυτές χρησιμοποιώντας το πείραμα ping-pong για ένα ορισμένο αριθμό επαναλήψεων (χρησιμοποιήσαμε 100) για διαφορετικά μεγέθη μηνυμάτων.

6.6.2 Βάρος Δύναμης και Ταχύτητα

Το υπολογιστικό βάρος δύναμης παρέχει μια μέση αναφορά απόδοσης περιλαμβάνοντας την ετερογένεια των συστημάτων. Το βάρος δύναμης είναι μια συνδυαστική μέτρηση του προγράμματος και της συστοιχίας υπολογιστών. Θεωρούμε ότι το μέγεθος της μνήμης επηρεάζει την εκτέλεση της εφαρμογής αναζήτησης αλφαριθμητικών και είναι ένας κύριος παράγοντας απόδοσης αρχιτεκτονικής.

Στα πειράματα μας, τα αποτελέσματα χρόνου της αναζήτησης αλφαριθμητικών στο δίκτυο μετρήθηκαν χρησιμοποιώντας διαφορετικά μεγέθη κειμένου. Η απόδοση σχετίζεται άμεσα με το μέγεθος της μνήμης σε κάθε σταθμό εργασίας. Με βάση τον τύπο 6.1, μετρήθηκαν και παρουσιάζονται στον Πίνακα 6.5 τα μέση βάρη δύναμης. Όλα τα πειράματα επαναλήφθηκαν δέκα φορές σε ένα αποκλειστικό σύστημα. Τα αποτελέσματα δείχνουν ότι τα βάρη δύναμης παρέμειναν σταθερά αν το μέγεθος των δεδομένων της εφαρμογής μας ήταν στα όρια της μνήμης.

Τέλος, οι ταχύτητες $S_{E/E}$ και S_{search} του ταχύτερου σταθμού εργασίας για όλα τα μήκη προτύπων και διαφορετικές τιμές του k, εκτελώντας την εφαρμογή της προσεγγιστικής αναζήτησης αλφαριθμητικών είναι κρίσιμοι παράμετροι για να προβλέψουμε τους χρόνους CPU των υπολογιστικών κομματιών σε άλλους σταθμούς εργασίας. Οι ταχύτητες μετρήθηκαν για διαφορετικά μεγέθη κειμένου και πήραμε τον μέσο όρο από το τύπο 6.2 όπως παρουσιάζονται στους Πίνακες 6.6, 6.7 και 6.8 για τις υλοποιήσεις P1/P4, P2 και P3 αντίστοιχα. Πρέπει να σημειώσουμε εδώ ότι μετρήσαμε τις ταχύτητες $S_{E/E}$ και

Πίνακας 6.5: Βάρη δύναμης των δύο τύπων σταθμών εργασίας για προσεγγιστική αναζήτηση αλφαριθμητικών

Υλοποίηση	Pentium MMX	Pentium
P1	1	0.595
P2	1	0.585
P3	1	0.588
P4	1	0.595

S_{search} για κάθε παράληλη υλοποίηση ξεχωριστά για να προβλέψουμε με ακρίβεια τον χρόνο εκτέλεσης για διάφορα υπολογιστικά κομμάτια.

6.6.3 Αναλυτικά Αποτελέσματα

Οι Πίνακες 6.9, 6.10, 6.11 και 6.12 δείχνουν τους χρόνους εκτέλεσης για διάφορες τιμές n, m, k και p χρησιμοποιώντας τον αλγόριθμο SEL που προκύπτουν από τις εξισώσεις 6.13, 6.19, 6.25 και 6.30 για τις υλοποιήσεις P1, P2, P3 και P4 αντίστοιχα. Το Σχήμα 6.13 παρουσιάζει τις επιταχύνσεις για διάφορες τιμές n, m, k και p χρησιμοποιώντας τον αλγόριθμο SEL που προκύπτουν από τα πειράματα και από τις εξισώσεις 6.13 και 6.6 για την υλοποίηση P1. Παρόμοια τα Σχήματα 6.14, 6.15 και 6.16 δείχνουν τις επιταχύνσεις που προκύπτουν από τα πειράματα και από τις εξισώσεις 6.19, 6.25, 6.30 και 6.6 για τις υλοποιήσεις P2, P3 και P4 αντίστοιχα. Είναι σημαντικό να σημειώσουμε ότι οι επιταχύνσεις που απεικονίζονται στα Σχήματα είναι αποτέλεσμα του μέσου όρου για πέντε διαφορετικά μήκη προτύπων και για τέσσερις διαφορετικές τιμές του k. Επίσης, σημειώνουμε εδώ ότι υπάρχουν μικρές διαφορές ανάμεσα στις πειραματικές και αναλυτικές τιμές.

6.6.4 Αποτελέσματα Άλλων Αλγορίθμων Αναζήτησης Αλφαριθμητικών

Τα Σχήματα 6.17, 6.18, 6.19 και 6.20 παρουσιάζουν τις επιταχύνσεις χρησιμοποιώντας τους αλγορίθμους της ακριβούς αναζήτησης αλφαριθμητικών όπως QS, RF, SO και BNDM που προκύπτουν από τα πειράματα και από τις εξισώσεις 6.13, 6.19, 6.25 και 6.30 για τις υλοποιήσεις P1, P2, P3 και P4 αντίστοιχα. Παρόμοια, τα Σχήματα 6.21, 6.22, 6.23 και 6.24 παρουσιάζουν τις επιταχύνσεις χρησιμοποιώντας τους αλγορίθμους της αναζήτησης αλφαριθμητικών με k αποτυχίες όπως BYP, TU και SO.

Πίνακας 6.6: Ταχύτητες (σε χαρακτήρες ανα δευτερόλεπτο) για Ε/Ε και αναζήτηση αλφαριθμητικών του ταχύτερου σταθμού εργασίας για τις υλοποιήσεις P1 και P4

m	$S_{E/E}$	S_{search}
5	17633124.89	3214035.78
10	21875604.98	3431670.343
20	22127447.78	3548906.721
30	22636217.71	3578006.568
60	20656662.11	3629517.722

k	$S_{E/E}$	S_{search}
1	21632627.9	3421117.265
3	21875605	3431670.343
6	21850527.5	3420130.829
9	21592938.1	3403312.67

Πίνακας 6.7: Ταχύτητες (σε χαρακτήρες ανα δευτερόλεπτο) για Ε/Ε και αναζήτηση αλφαριθμητικών του ταχύτερου σταθμού εργασίας για την υλοποίηση P2

m	$S_{E/E}$	S_{search}
5	39536183.29	3304400.483
10	29468421.17	3480067.399
20	29274861.22	3564879.621
30	29184588.62	3583178.115
60	28582450.83	3625836.6

k	$S_{E/E}$	S_{search}
1	29463661.61	3480346.546
3	29468421.17	3480067.399
6	29467958.37	3479227.666
9	29464990.6	3457393.887

Πίνακας 6.8: Ταχύτητες (σε χαρακτήρες ανα δευτερόλεπτο) για Ε/Ε και αναζήτηση αλφαριθμητικών του ταχύτερου σταθμού εργασίας για την υλοποίηση P3

m	$S_{E/E}$	S_{search}
5	39848898.15	3254197.391
10	29666201.05	3458715.671
20	29438025.77	3559279.002
30	29346732.2	3583320.366
60	29107986.58	3629279.082

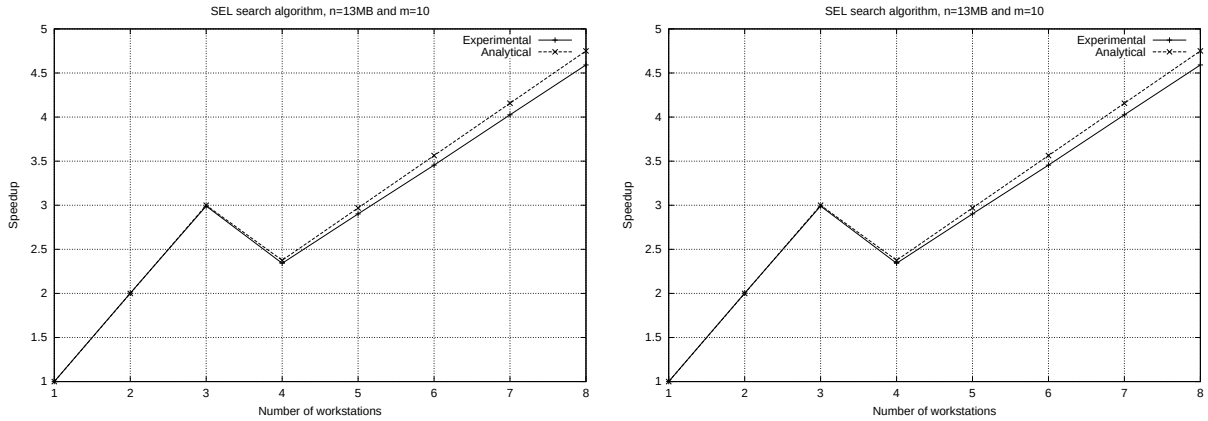
k	$S_{E/E}$	S_{search}
1	29430103.53	3465758.273
3	29666201.05	3458715.671
6	29375403.38	3459712.362
9	29523524.37	3432285.977

Τέλος, τα Σχήματα 6.25, 6.26, 6.27 και 6.28 παρουσιάζουν τις επιταχύνσεις χρησιμοποιώντας τους αλγορίθμους της αναζήτησης αλφαριθμητικών με k διαφορές όπως TUD, WM, MULTIWM και MYE.

Από τα παραπάνω παρατηρούμε ότι τα πειραματικά αποτελέσματα των παράλληλων υλοποιήσεων για τους αλγορίθμους που έχουμε χρησιμοποιήσει επιβεβαιώνονται από τα θεωρητικά αποτελέσματα του μοντέλου μας πρόβλεψης απόδοσης που παρουσιάσαμε προηγουμένως.

Πίνακας 6.9: Αναλυτικοί χρόνοι εκτέλεσης (σε δευτερόλεπτα) για μέγεθος κειμένου 13MB και $k = 3$ για διαφορετικά μήκη προτύπων χρησιμοποιώντας τον αλγόριθμο SEL

m/p	Παρ. Υλοπ.	1	2	3	4	5	6	7	8
5	P1	21.870	10.936	7.291	9.107	7.286	6.072	5.205	4.555
	P2	20.867	11.978	8.558	7.460	6.665	6.064	5.593	5.214
	P3	21.181	10.702	7.172	6.023	5.195	4.570	4.083	3.691
	P4	21.870	10.936	7.291	6.076	5.208	4.557	4.051	3.646
10	P1	40.146	20.073	13.383	16.863	13.491	11.243	9.637	8.433
	P2	39.436	21.321	14.826	12.716	11.195	10.047	9.151	8.430
	P3	39.674	19.949	13.337	11.170	9.613	8.441	7.526	6.792
	P4	40.146	20.073	13.383	11.168	9.582	8.390	7.463	6.720
20	P1	77.053	38.527	25.685	32.463	25.971	21.643	18.551	16.233
	P2	76.561	39.886	27.203	23.055	20.072	17.825	16.071	14.664
	P3	76.678	38.451	25.672	21.473	18.459	16.191	14.421	13.003
	P4	77.053	38.527	25.685	21.444	18.405	16.121	14.341	12.915
30	P1	114.327	57.164	38.110	48.037	38.430	32.026	27.451	24.020
	P2	114.029	58.620	39.693	33.467	28.999	25.637	23.016	20.915
	P3	114.022	57.123	38.119	31.911	27.446	24.080	21.453	19.344
	P4	114.327	57.164	38.110	31.803	27.287	23.895	21.252	19.136
60	P1	224.885	112.444	74.964	94.646	75.717	63.099	54.085	47.325
	P2	224.931	114.077	76.667	64.367	55.540	48.897	43.717	39.565
	P3	224.709	112.467	75.015	62.654	53.795	47.135	41.944	37.786
	P4	224.885	112.444	74.964	62.574	53.699	47.029	41.833	37.671



Σχήμα 6.13: Αναλυτική και πειραματική επιτάχυνση σε συνάρτηση με τον αριθμό των σταθμών εργασίας για την υλοποίηση P1 χρησιμοποιώντας τον αλγόριθμο SEL

Πίνακας 6.10: Αναλυτικοί χρόνοι εκτέλεσης (σε δευτερόλεπτα) για μέγεθος κειμένου 13MB και $m = 10$ για διαφορετικά τιμές του k χρησιμοποιώντας τον αλγόριθμο SEL

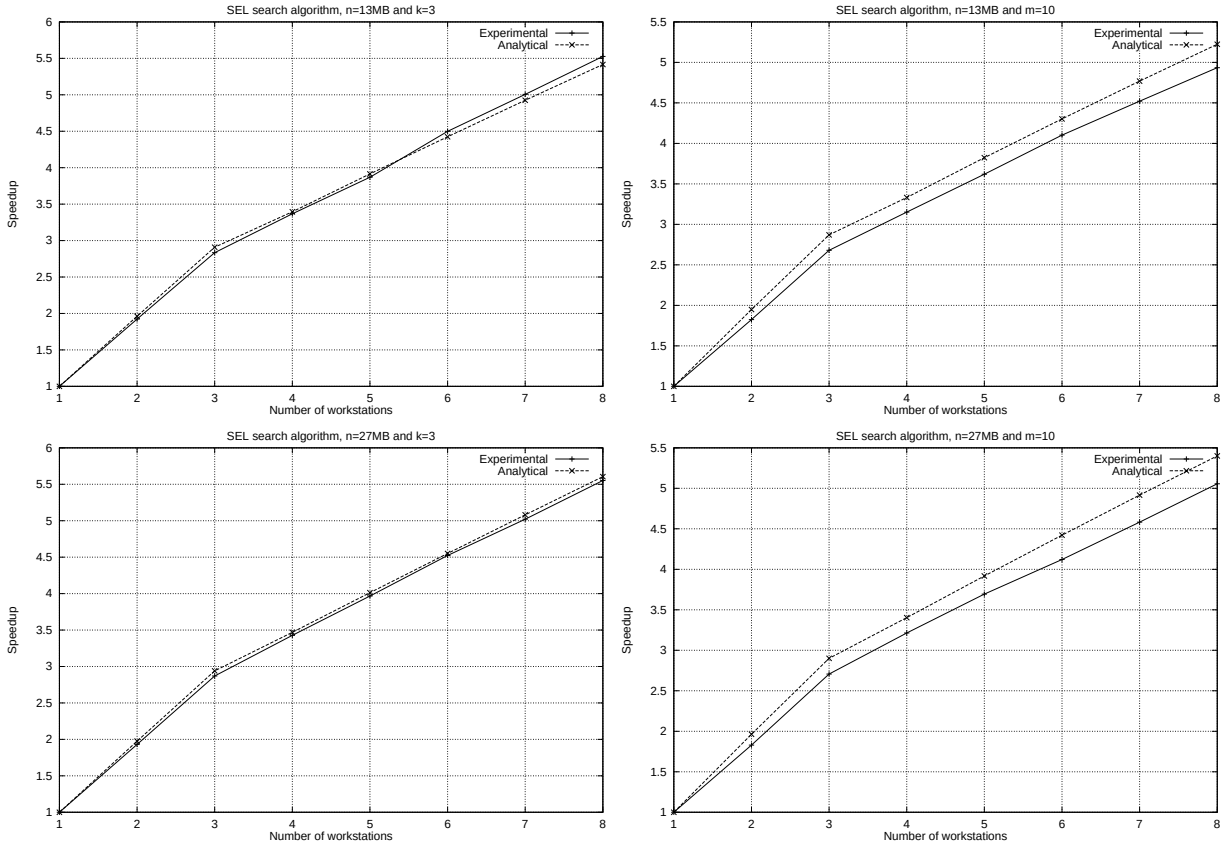
k/p	Παρ. Υλοπ.	1	2	3	4	5	6	7	8
1	P1	40.275	20.138	13.426	16.964	13.572	11.310	9.695	8.483
	P2	39.433	21.320	14.825	12.714	11.193	10.045	9.148	8.428
	P3	39.598	19.911	13.311	11.158	9.609	8.441	7.529	6.797
	P4	40.275	20.138	13.426	11.209	9.620	8.426	7.496	6.751
3	P1	40.146	20.073	13.383	16.863	13.491	11.243	9.637	8.433
	P2	39.436	21.321	14.826	12.716	11.195	10.047	9.151	8.430
	P3	39.674	19.949	13.337	11.170	9.613	8.441	7.526	6.792
	P4	40.146	20.073	13.383	11.168	9.582	8.390	7.463	6.720
6	P1	40.280	20.140	13.428	16.953	13.563	11.303	9.689	8.478
	P2	39.446	21.326	14.829	12.717	11.196	10.048	9.151	8.430
	P3	39.667	19.946	13.335	11.175	9.622	8.451	7.537	6.804
	P4	40.280	20.140	13.428	11.209	9.619	8.425	7.494	6.749
9	P1	40.483	20.242	13.495	17.061	13.650	11.375	9.751	8.532
	P2	39.692	21.449	14.911	12.786	11.254	10.099	9.196	8.471
	P3	39.978	20.101	13.438	11.258	9.691	8.511	7.589	6.850
	P4	40.483	20.242	13.495	11.268	9.671	8.471	7.536	6.787

Πίνακας 6.11: Αναλυτικοί χρόνοι εκτέλεσης (σε δευτερόλεπτα) για μέγεθος κειμένου 27MB και $k = 3$ για διαφορετικά μήκη προτύπων χρησιμοποιώντας τον αλγόριθμο SEL

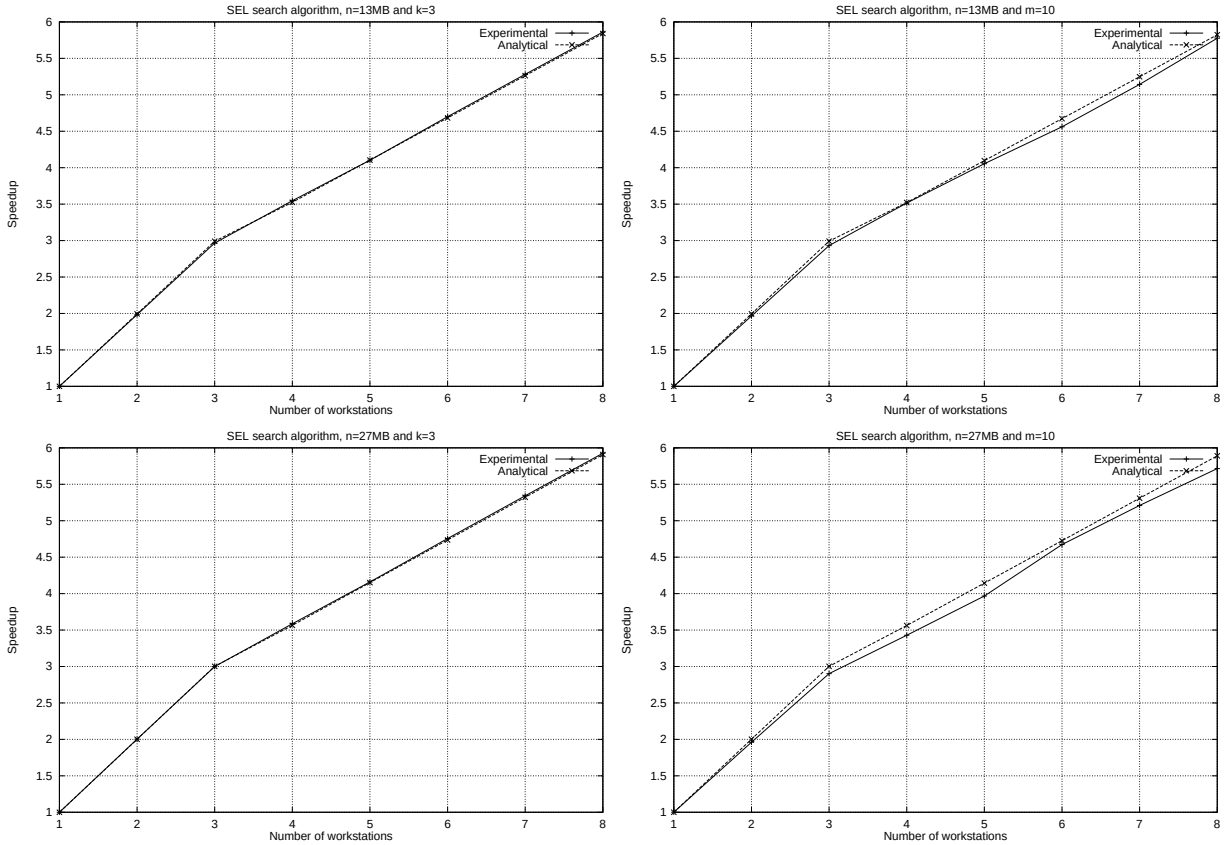
m/p	Παρ. Υλοπ.	1	2	3	4	5	6	7	8
5	P2	41.734	23.946	17.105	14.909	13.320	12.118	11.176	10.418
	P3	43.362	21.404	14.344	12.044	10.388	9.140	8.164	7.381
	P4	43.740	21.871	14.581	12.150	10.414	9.112	8.099	7.289
10	P2	78.873	42.633	29.641	25.421	22.380	20.084	18.290	16.850
	P3	79.348	39.897	26.673	22.338	19.225	16.880	15.050	13.583
	P4	80.292	40.146	26.765	22.334	19.162	16.779	14.924	13.438
20	P2	153.122	79.761	54.395	46.099	40.134	35.640	32.132	29.317
	P3	153.357	76.902	51.342	42.945	36.917	32.380	28.841	26.004
	P4	154.107	77.054	51.370	42.887	36.809	32.240	28.680	25.828
30	P2	228.058	117.231	79.376	66.923	57.988	51.264	46.022	41.820
	P3	228.044	114.245	76.238	63.820	54.890	48.159	42.904	38.687
	P4	228.655	114.328	76.220	63.605	54.573	47.787	42.503	38.270
60	P2	449.862	228.143	153.324	128.724	111.070	97.784	87.424	79.119
	P3	449.419	224.932	150.029	125.307	107.589	94.268	83.887	75.571
	P4	449.770	224.887	149.925	125.146	107.396	94.056	83.664	75.339

Πίνακας 6.12: Αναλυτικοί χρόνοι εκτέλεσης (σε δευτερόλεπτα) για μέγεθος κειμένου 27MB και $m = 10$ για διαφορετικά τιμές του k χρησιμοποιώντας τον αλγόριθμο SEL

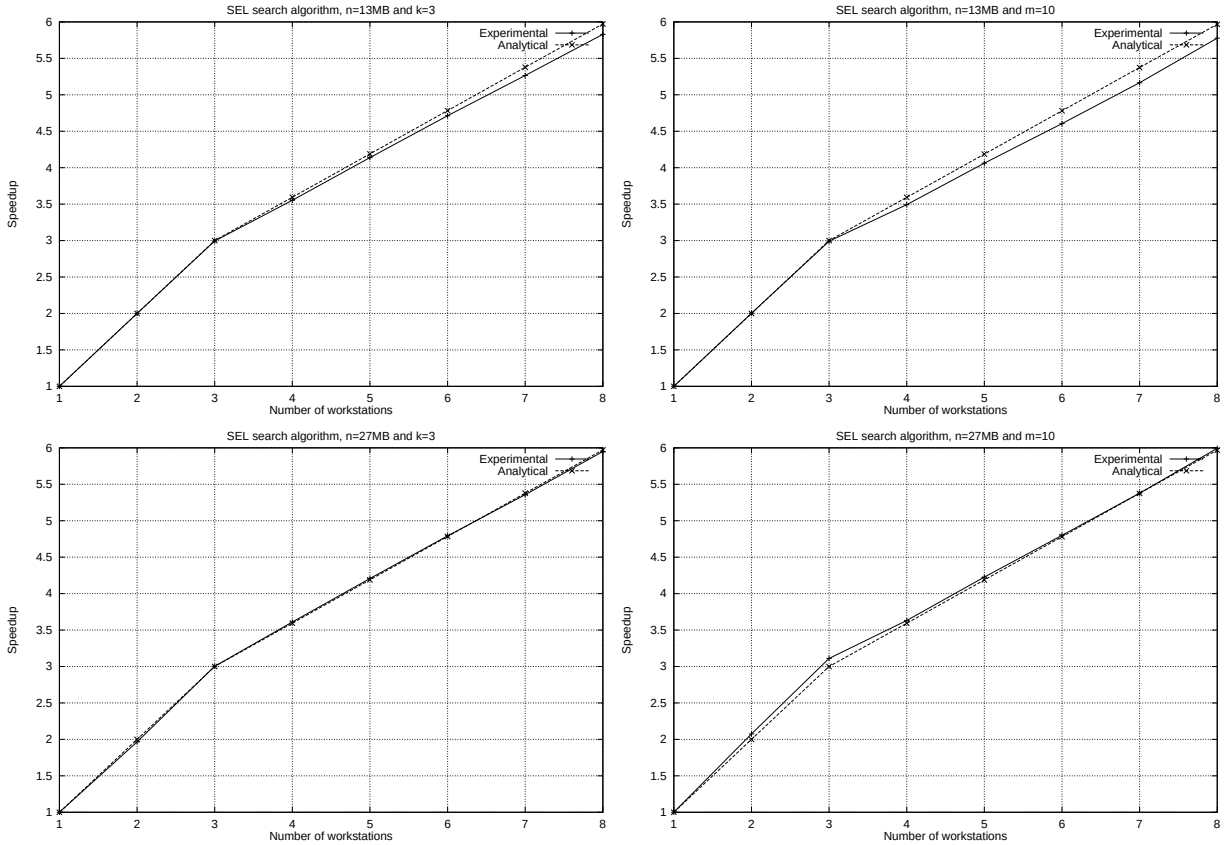
k/p	Παρ. Υλοπ.	1	2	3	4	5	6	7	8
1	P2	78.867	42.630	29.639	25.417	22.376	20.080	18.286	16.845
	P3	79.196	39.821	26.622	22.316	19.218	16.881	15.057	13.593
	P4	80.550	40.275	26.851	22.416	19.239	16.851	14.990	13.499
3	P2	78.873	42.633	29.641	25.421	22.380	20.084	18.290	16.850
	P3	79.348	39.897	26.673	22.338	19.225	16.880	15.050	13.583
	P4	80.292	40.146	26.765	22.334	19.162	16.779	14.924	13.438
6	P2	78.892	42.643	29.648	25.424	22.382	20.086	18.291	16.850
	P3	79.334	39.890	26.668	22.349	19.243	16.901	15.073	13.606
	P4	80.560	40.281	26.854	22.416	19.237	16.848	14.987	13.496
9	P2	79.384	42.889	29.812	25.561	22.498	20.187	18.381	16.931
	P3	79.956	40.201	26.875	22.515	19.381	17.020	15.177	13.698
	P4	80.967	40.484	26.990	22.534	19.341	16.942	15.071	13.573



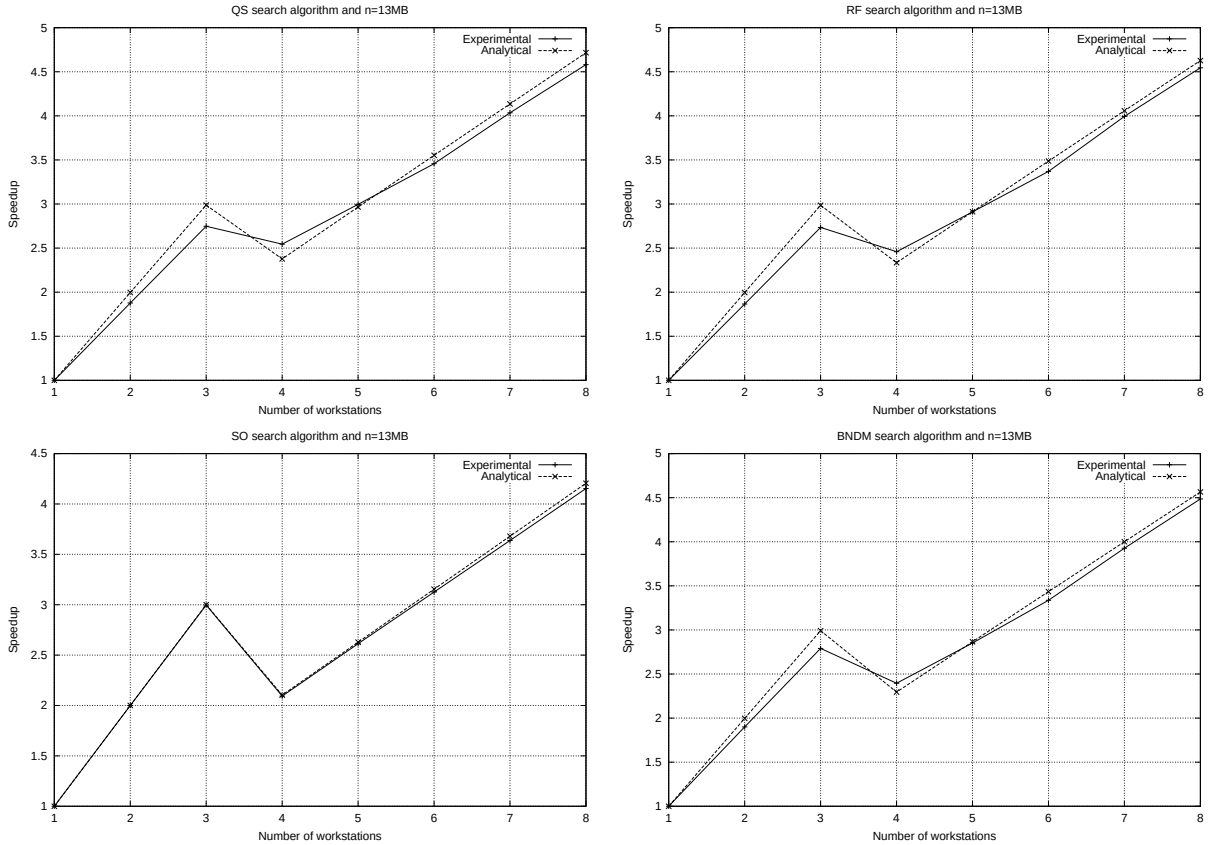
Σχήμα 6.14: Αναλυτική και πειραματική επιτάχυνση σε συνάρτηση με τον αριθμό των σταθμών εργασίας για την υλοποίηση P2 χρησιμοποιώντας τον αλγόριθμο SEL



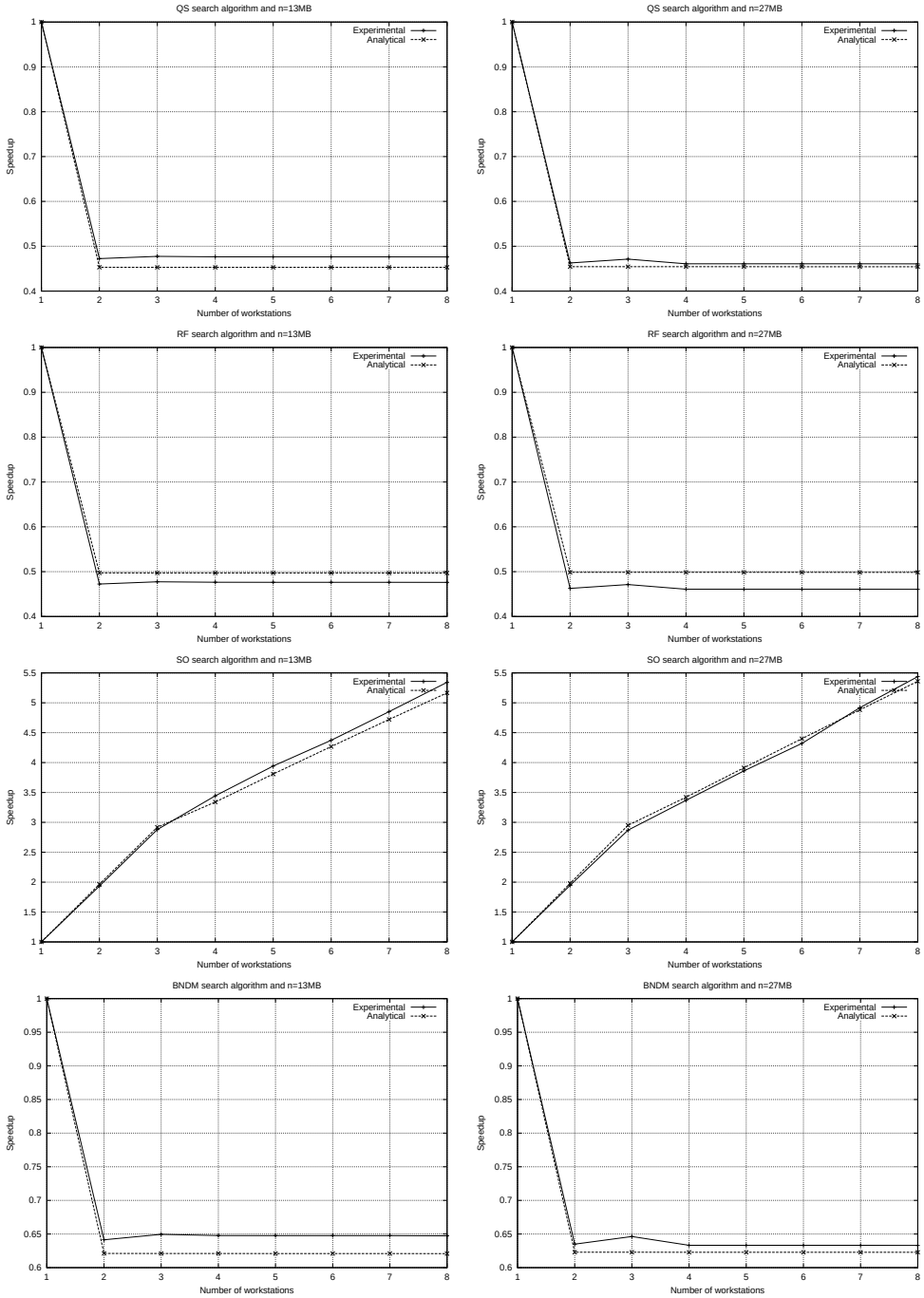
Σχήμα 6.15: Αναλυτική και πειραματική επιτάχυνση σε συνάρτηση με τον αριθμό των σταθμών εργασίας για την υλοποίηση P3 χρησιμοποιώντας τον αλγόριθμο SEL



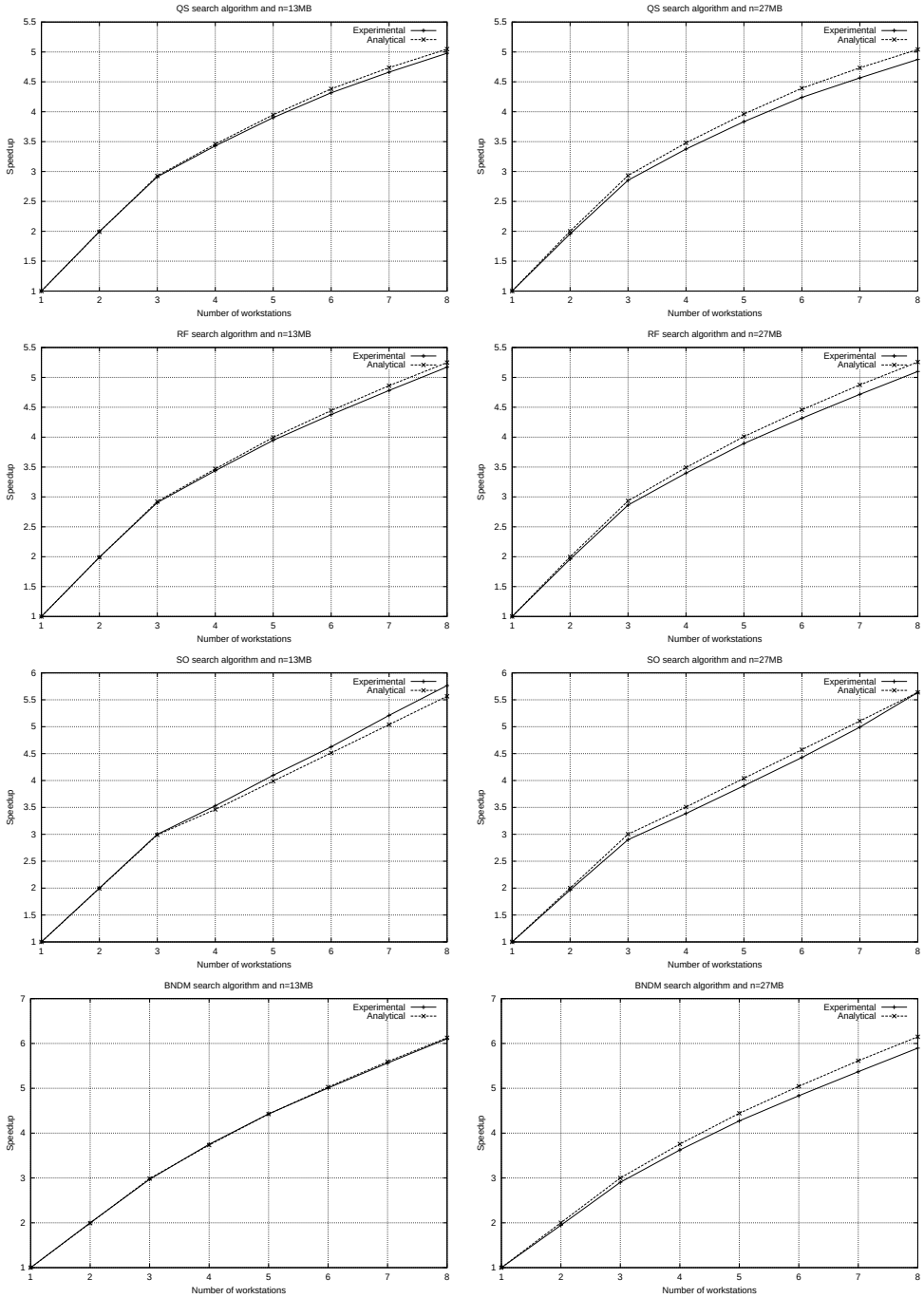
Σχήμα 6.16: Αναλυτική και πειραματική επιτάχυνση σε συνάρτηση με τον αριθμό των σταθμών εργασίας για την υλοποίηση P4 χρησιμοποιώντας τον αλγόριθμο SEL



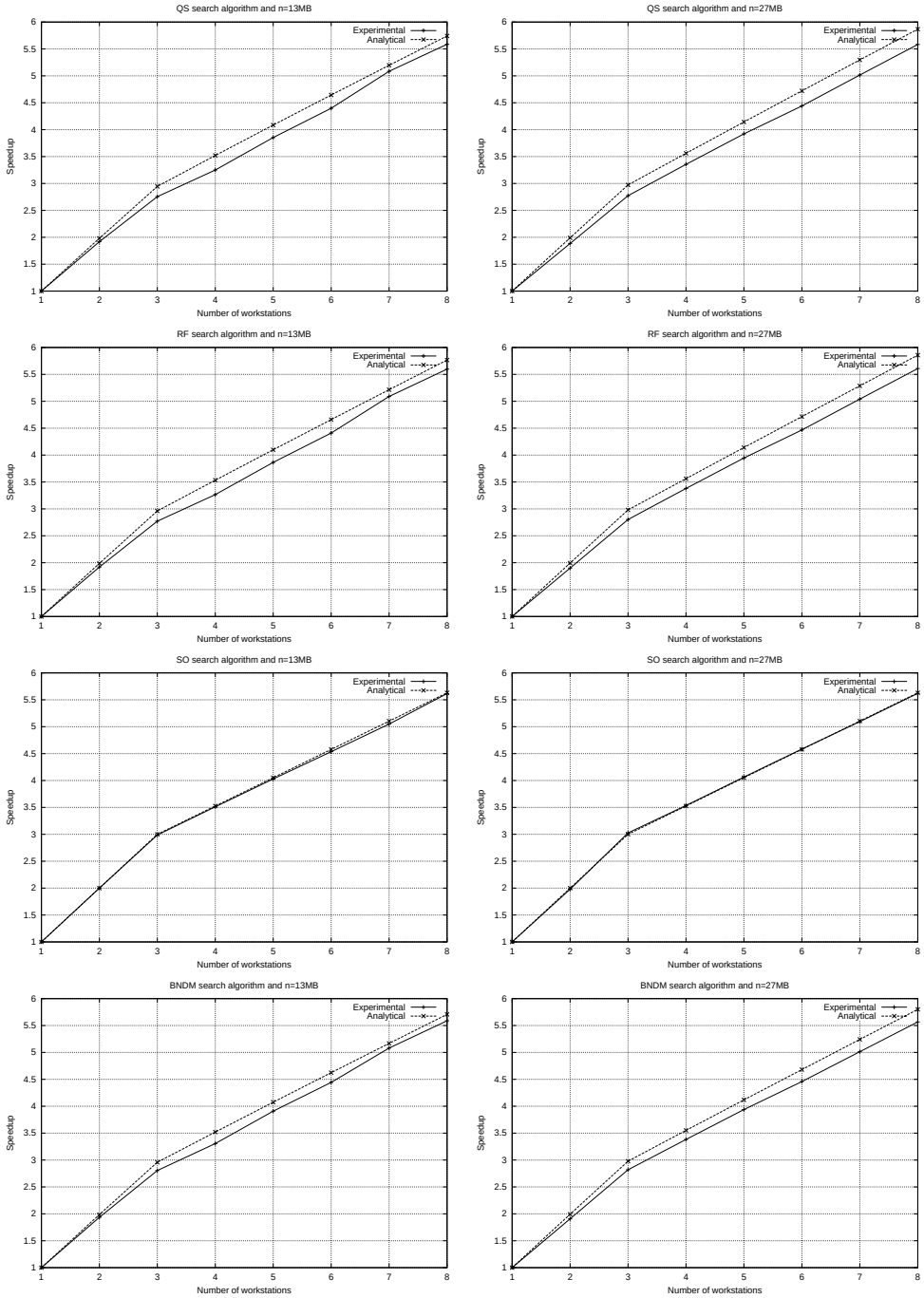
Σχήμα 6.17: Αναλυτική και πειραματική επιτάχυνση σε συνάρτηση με τον αριθμό των σταθμών εργασίας για την υλοποίηση P1 χρησιμοποιώντας τους αλγορίθμους QS, RF, SO και BNDM



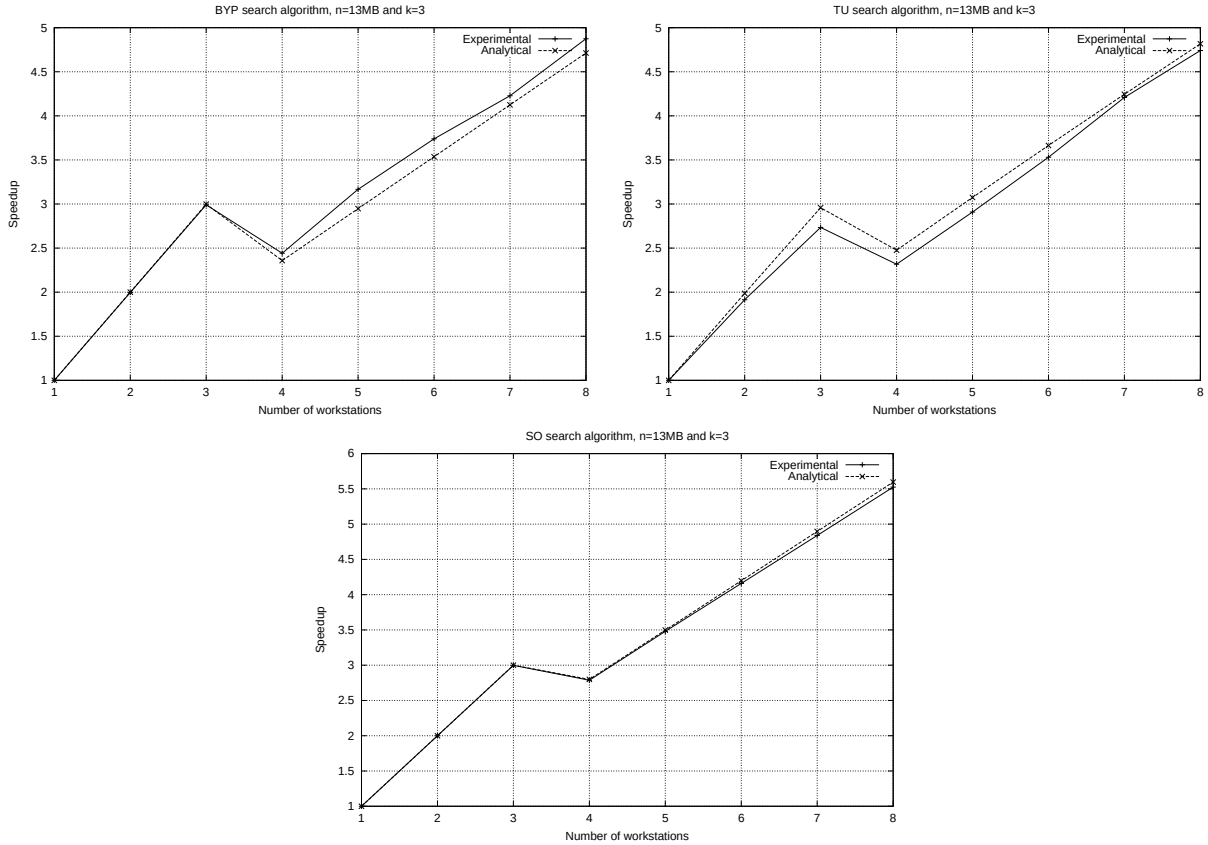
Σχήμα 6.18: Αναλυτική και πειραματική επιτάχυνση σε συνάρτηση με τον αριθμό των σταθμών εργασίας για την υλοποίηση P2 χρησιμοποιώντας τους αλγορίθμους QS, RF, SO και BNDM



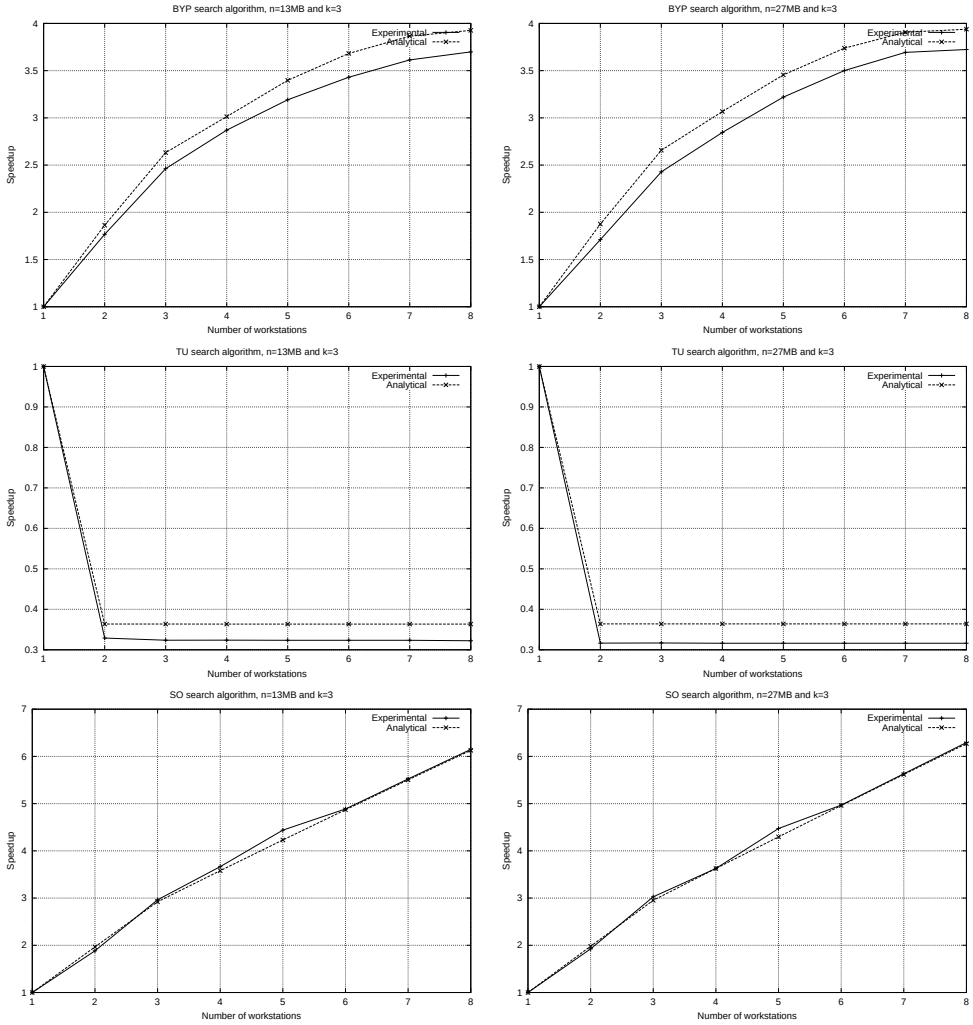
Σχήμα 6.19: Αναλυτική και πειραματική επιτάχυνση σε συνάρτηση με τον αριθμό των σταθμών εργασίας για την υλοποίηση P3 χρησιμοποιώντας τους αλγορίθμους QS, RF, SO και BNDM



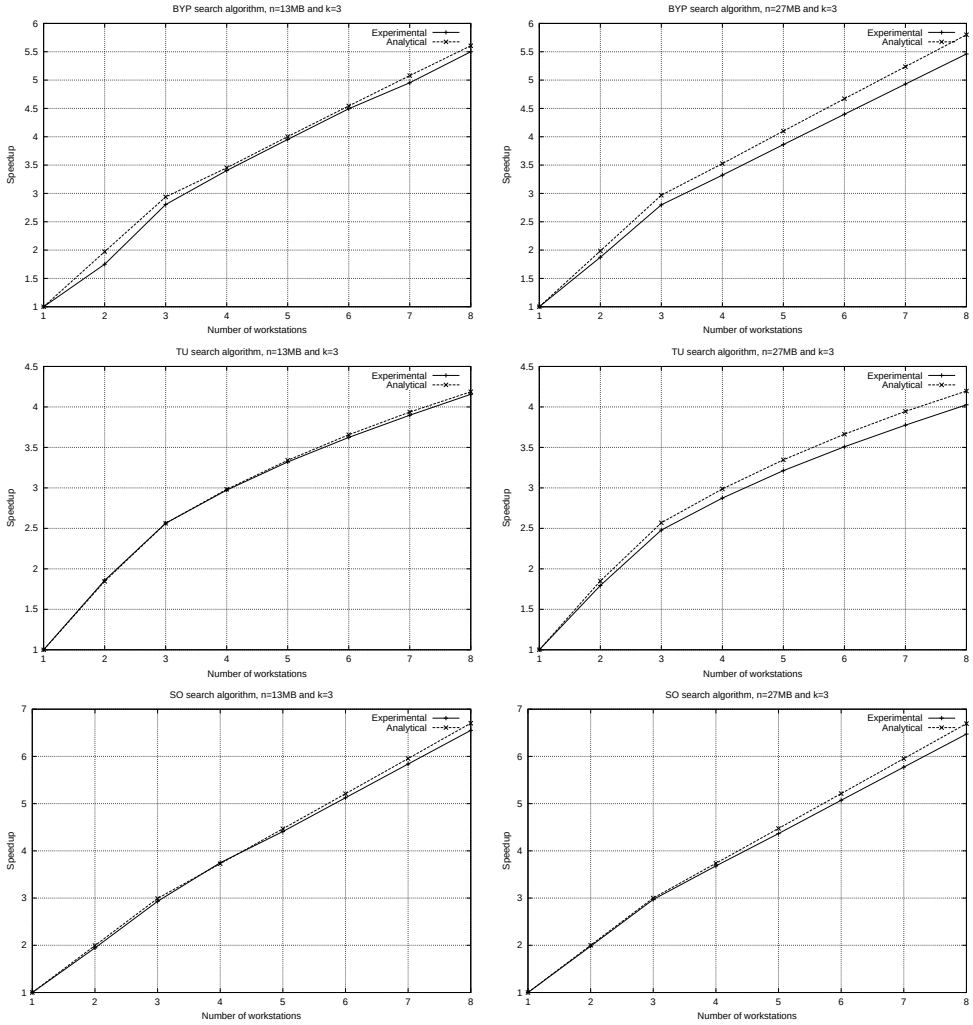
Σχήμα 6.20: Αναλυτική και πειραματική επιτάχυνση σε συνάρτηση με τον αριθμό των σταθμών εργασίας για την υλοποίηση P4 χρησιμοποιώντας τους αλγορίθμους QS, RF, SO και BNDM



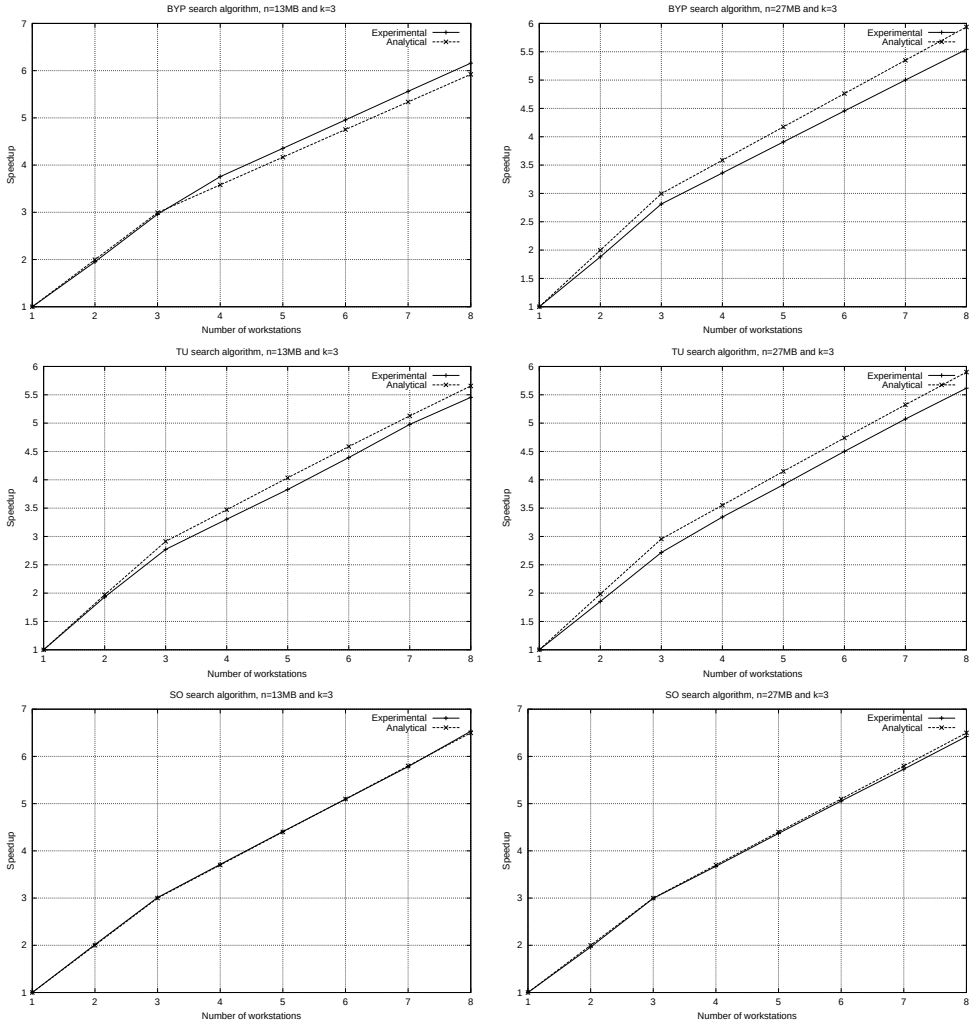
Σχήμα 6.21: Αναλυτική και πειραματική επιτάχυνση σε συνάρτηση με τον αριθμό των σταθμών εργασίας για την υλοποίηση P1 χρησιμοποιώντας τους αλγορίθμους BYP, TU και SO



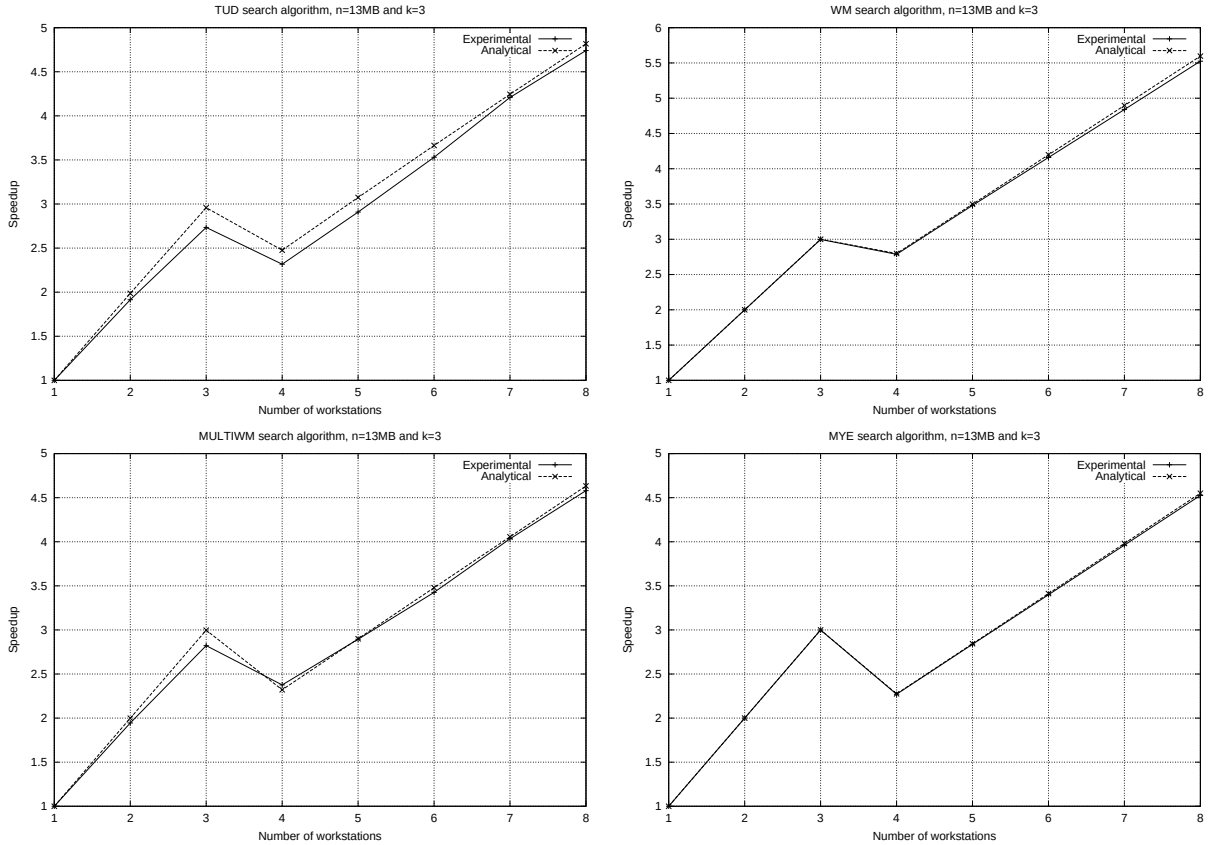
Σχήμα 6.22: Αναλυτική και πειραματική επιτάχυνση σε συνάρτηση με τον αριθμό των σταθμών εργασίας για την υλοποίηση P2 χρησιμοποιώντας τους αλγορίθμους BYP, TU και SO



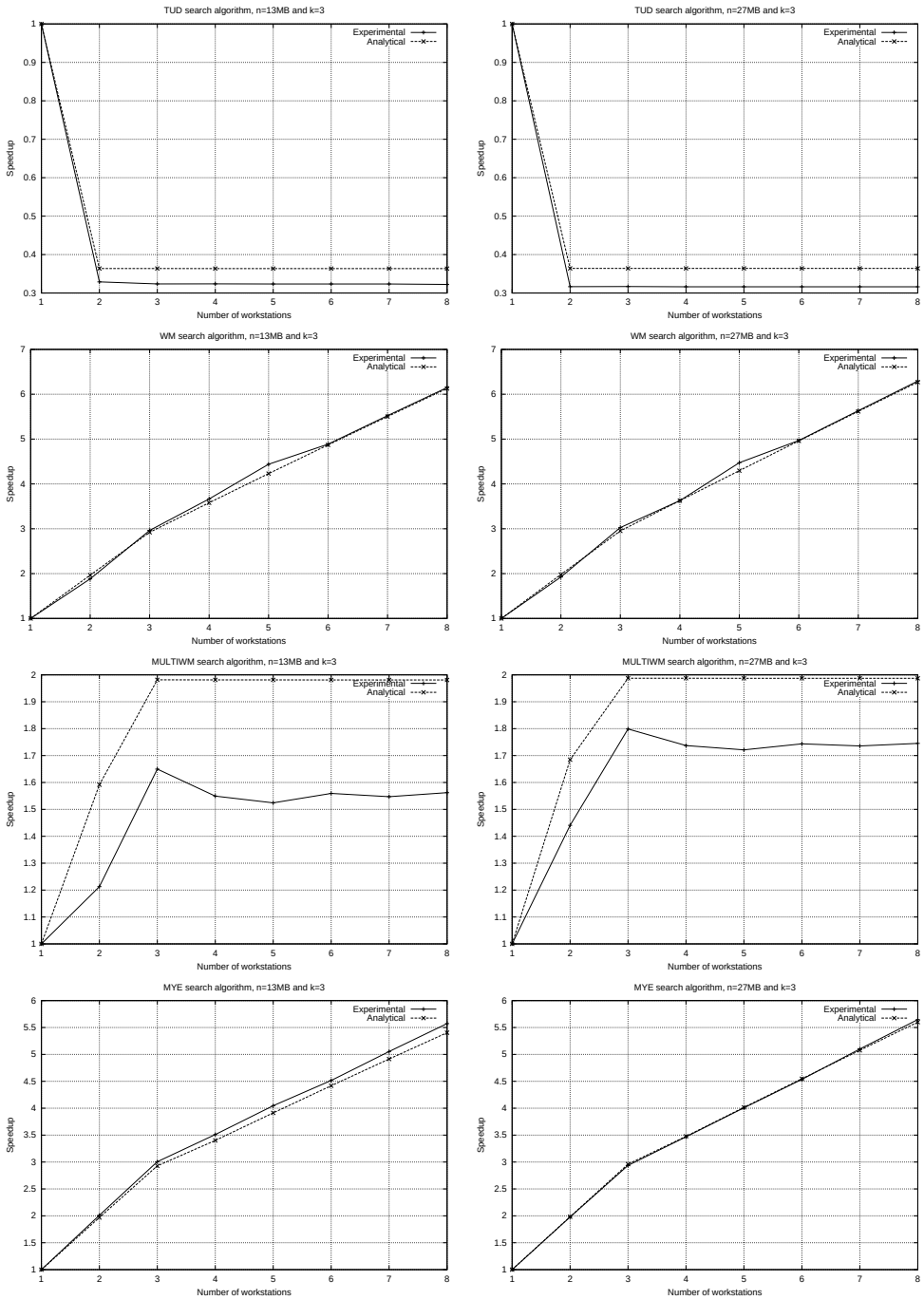
Σχήμα 6.23: Αναλυτική και πειραματική επιτάχυνση σε συνάρτηση με τον αριθμό των σταθμών εργασίας για την υλοποίηση P3 χρησιμοποιώντας τους αλγορίθμους BYP, TU και SO



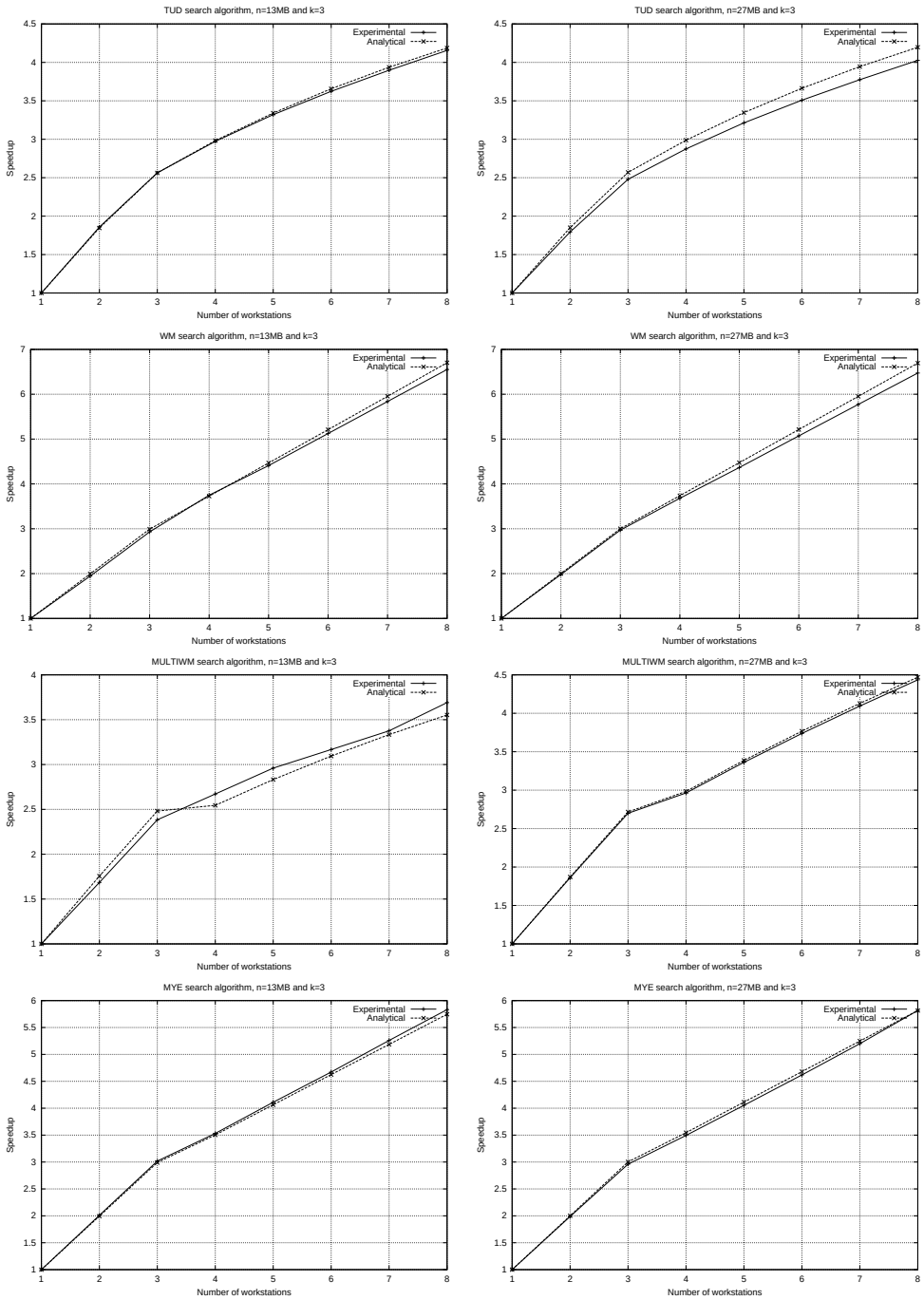
Σχήμα 6.24: Αναλυτική και πειραματική επιτάχυνση σε συνάρτηση με τον αριθμό των σταθμών εργασίας για την υλοποίηση P4 χρησιμοποιώντας τους αλγορίθμους BYP, TU και SO



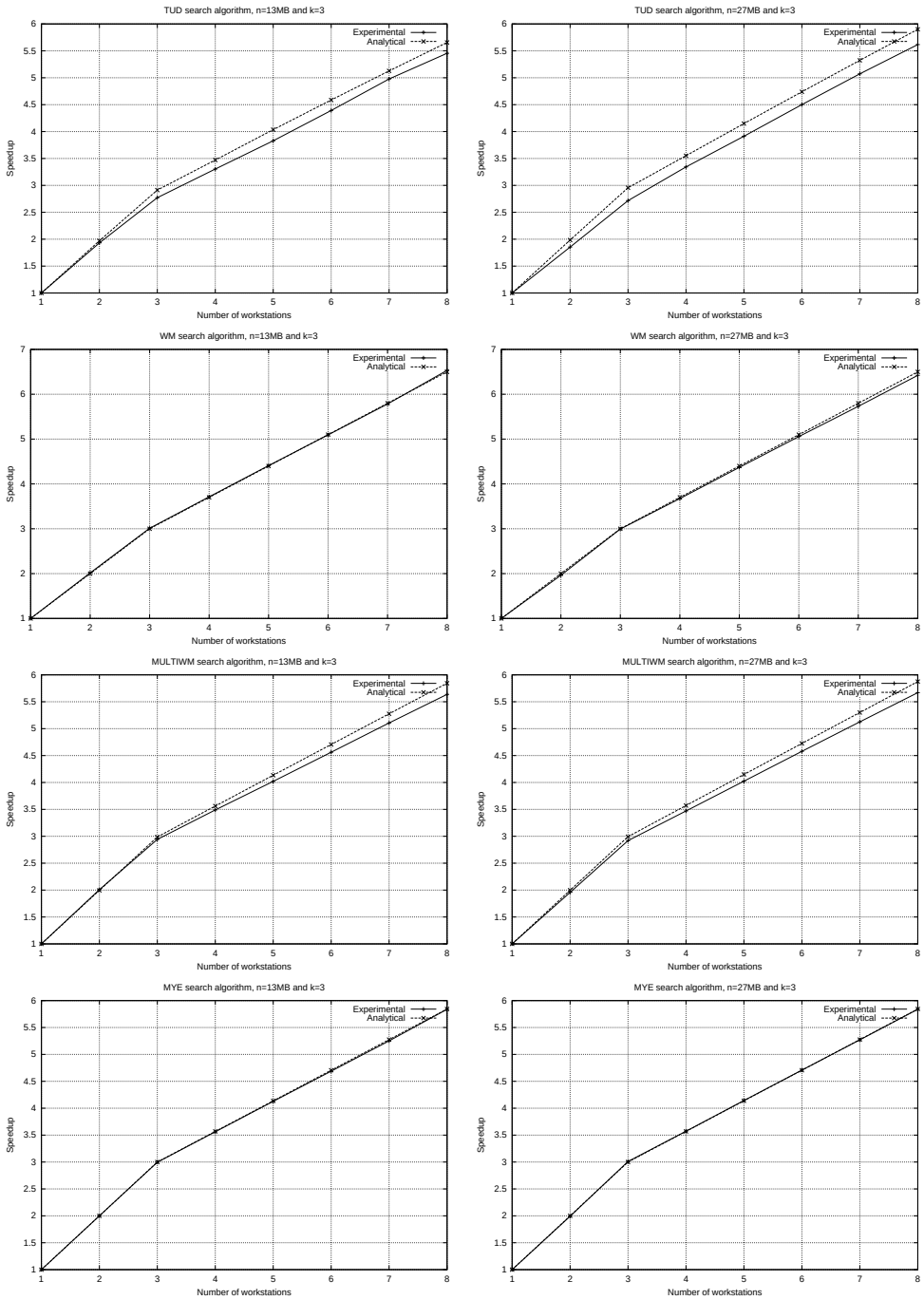
Σχήμα 6.25: Αναλυτική και πειραματική επιτάχυνση σε συνάρτηση με τον αριθμό των σταθμών εργασίας για την υλοποίηση P1 χρησιμοποιώντας τους αλγορίθμους TUD, WM, MULTIWM και MYE



Σχήμα 6.26: Αναλυτική και πειραματική επιτάχυνση σε συνάρτηση με τον αριθμό των σταθμών εργασίας για την υλοποίηση P2 χρησιμοποιώντας τους αλγορίθμους TUD, WM, MULTIWM και MYE



Σχήμα 6.27: Αναλυτική και πειραματική επιτάχυνση σε συνάρτηση με τον αριθμό των σταθμών εργασίας για την υλοποίηση P3 χρησιμοποιώντας τους αλγορίθμους TUD, WM, MULTIWM και MYE



Σχήμα 6.28: Αναλυτική και πειραματική επιτάχυνση σε συνάρτηση με τον αριθμό των σταθμών εργασίας για την υλοποίηση P4 χρησιμοποιώντας τους αλγορίθμους TUD, WM, MULTIWM και MYE

Κεφάλαιο 7

Υλοποιήσεις Αλγορίθμων Αναζήτησης Αλφαριθμητικών σε Διατάξεις Επεξεργαστών

7.1 Εισαγωγή

Οι αλγόριθμοι αναζήτησης αλφαριθμητικών περιλαμβάνουν κυρίως πράξεις και υπολογισμούς που είναι επαναληπτικοί (repititive) και κανονικοί (regular). Οι αλγόριθμοι αυτοί μπορούν να απεικονιστούν αποτελεσματικά σε παράλληλες αρχιτεκτονικές πολύ μεγάλης κλίμακας ολοκλήρωσης (Very Large Scale Integration - VLSI). Οι τελευταίες εξελίξεις στην τεχνολογία VLSI έχουν καταστήσει δυνατή την ανάπτυξη διατάξεων επεξεργαστών ειδικού σκοπού (special purpose array processors) για πολύπλοκα και απαιτητικά υπολογιστικά προβλήματα. Τα χαρακτηριστικά του παραλληλισμού (parallelism), της ταυτοχρονικότητας (concurrency), της διασωλήνωσης (pipelining), της ακεραιότητας (modularity), της κανονικότητας (regularity) και της τοπικής διασύνδεσης (local interconnection) έχουν γίνει πρότυπα στους σχεδιασμούς VLSI. Οι διατάξεις επεξεργαστών ειδικού σκοπού επιτρέπουν μια άμεση απεικόνιση του αλγορίθμου αναζήτησης αλφαριθμητικών πάνω στο υλικό (hardware). Περισσότερες λεπτομέρειες για την αρχιτεκτονική των διατάξεων επεξεργαστών ειδικού σκοπού παρουσιάσαμε στο κεφάλαιο 4.

Σε αυτό το κεφάλαιο παρουσιάζουμε υλοποιήσεις αλγορίθμων απλής και προσεγγιστικής αναζήτησης αλφαριθμητικών και μακρύτερης κοινής υποακολουθίας (Longest Common Subsequence - LCS) σε διατάξεις επεξεργαστών χρησιμοποιώντας την μεθοδολογία απεικόνισης που βασίζεται στο γράφο εξάρτησης δεδομένων (data dependence graph) [72]. Οι αλγόριθμοι που απεικονίζονται στις διατάξεις επεξεργαστών ανήκουν σε δύο κατηγορίες: 1) δυναμικού προγραμματισμού και 2) bit-παραλληλισμού.

Το κεφάλαιο αυτό είναι διαρθρωμένο ως εξής: στην επόμενη ενότητα περιγράφουμε τους βασικούς αλγορίθμους δυναμικού προγραμματισμού για τα προβλήματα απλής και προσεγγιστικής αναζήτησης αλφαριθμητικών. Στην ενότητα 7.3 περιγράφουμε τους αλγορίθμους bit-παραλληλισμού. Στην ενότητα 7.5 παρουσιάζουμε έναν απλό αλγόριθμο bit-παραλληλισμού για την λύση του προβλήματος LCS. Στην ενότητα 7.6 παρουσιάζουμε μια σύντομη επισκόπηση σχετικά με προηγούμενες διατάξεις επεξεργαστών για τα προβλήματα απλής και προσεγγιστικής αναζήτησης αλφαριθμητικών και μακρύτερης κοινής υποακολουθίας. Στην ενότητα 7.7 δίνουμε τους γράφους εξάρτησης δεδομένων για τους αλγορίθμους δυναμικού προγραμματισμού και bit-παραλληλισμού για το πρόβλημα της προσεγγιστικής αναζήτησης αλφαριθμητικών. Επίσης, στην ίδια ενότητα παρουσιάζουμε γράφο εξάρτησης δεδομένων για τον αλγόριθμο της μακρύτερης κοινής υποακολουθίας. Στην ενότητα 7.8 παρουσιάζουμε την διαδικασία απεικόνισης της φάσης της αναζήτησης των αλγορίθμων δυναμικού προγραμματισμού και bit-παραλληλισμού σε διατάξεις επεξεργαστών. Επίσης, παρουσιάζουμε διάταξη επεξεργαστών για τον αλγόριθμο LCS.

7.2 Αλγόριθμοι Δυναμικού Προγραμματισμού

Σε αυτή την παράγραφο περιγράφουμε πιο αναλυτικά τους ακολουθιακούς αλγορίθμους δυναμικού προγραμματισμού για την λύση των προβλημάτων απλής και προσεγγιστικής αναζήτησης αλφαριθμητικών.

7.2.1 Αλγόριθμος βασισμένος στην Κλασσική Λύση

Όπως γνωρίζουμε ένας αλγόριθμος αναζήτησης αλφαριθμητικών αποτελείται από δύο φάσεις: τη φάση της προεπεξεργασίας και τη φάση της αναζήτησης. Η φάση της προεπεξεργασίας περιλαμβάνει τη συγκέντρωση πληροφορίας για το πρότυπο που μπορεί να χρησιμοποιηθεί για μια γρήγορη υλοποίηση

στη φάση της αναζήτησης. Από την άλλη μεριά, η φάση της αναζήτησης περιλαμβάνει είτε τη σάρωση του κειμένου ή την κατασκευή ενός πίνακα δυναμικού προγραμματισμού για να βρεθούν όλες οι απλές ή προσεγγιστικές εμφανίσεις του προτύπου μέσα στο κείμενο. Συνεπώς, σε αυτή η φάση βασίζεται στην κατασκευή του πίνακα δυναμικού προγραμματισμού D.

Φάση Προεπεξεργασίας

Κατά τη διάρκεια της φάσης προεπεξεργασίας το αλφαριθμητικό p κωδικοποιείται πάνω σε ένα χάρτη μνήμης bit-επιπέδου (bit-level memory map) R των $(m \times |\Sigma|)$ bits, όπου $|\Sigma|$ είναι το μέγεθος του αλφαβήτου. Το χάρτη μνήμης μπορούμε να τον δούμε σαν διδιάστατο πίνακα bit-επιπέδου όπου κάθε γραμμή αντιστοιχεί σε ένα χαρακτήρα του προτύπου και κάθε στήλη αντιστοιχεί σε ένα χαρακτήρα του αλφαβήτου. Συνεπώς, η στήλη R_j^T, για $1 \leq j \leq |\Sigma|$, κρατάει την πληροφορία του j-οστού χαρακτήρα του αλφαβήτου, που θα συμβολίζεται με σ_j. Τη στήλη R_j^T μπορούμε να τη δούμε σαν διάνυσμα bit-επιπέδου των m bits όπου το i-οστό bit, για $1 \leq i \leq m$, δηλαδή το $R_{i,j}$ κρατάει την πληροφορία σχετικά με το j-οστό χαρακτήρα σ_j και την i-οστή θέση του προτύπου. Η βασική πληροφορία που μπορεί να καταγραφεί είναι αν το j-οστό χαρακτήρα του αλφαβήτου είναι ίδιο με το i-οστό χαρακτήρα του προτύπου, δηλαδή αν είναι $p_i = \sigma_j$. Η φάση της προεπεξεργασίας κατασκευάζει το χάρτη μνήμης R και ο αλγόριθμος παρουσιάζεται στον Αλγόριθμο 7.1.

```
for j = 1 to |Σ| do
    for i = 1 to m do
        R_{i,j} ← 1;
        if p_i = σ_j then R_{i,j} ← 0;
    end
end
```

Αλγόριθμος 7.1: Φάση προεπεξεργασίας

Λαμβάνοντας σαν αλφάβητο αναφοράς το σύνολο χαρακτήρων UNICODE είναι απλό να υπολογίζουμε τις απαιτήσεις μνήμης του bit-χάρτη R. Έτσι $|\Sigma| = 64K$ και συνεπώς ο χάρτης μνήμης ενός m προτύπου απαιτεί $16m$ Kbytes. Οι συνολικές απαιτήσεις χώρου είναι $\lceil m|\Sigma|/16 \rceil$ bytes, υποθέτοντας ότι ένας χαρακτήρας κωδικοποιείται σε δύο bytes. Επίσης, η φάση προεπεξεργασίας μπορεί να εκτελεστεί

περίπου σε $m|\Sigma|$ βήματα.

Η απόδοση του αλγορίθμου μπορεί να βελτιωθεί αν εισάγουμε μια εξειδικευμένη διευθυνσιοδότηση (specialised addressing) τέτοια ώστε ένας χαρακτήρας να απεικονίζεται άμεσα στην κατάλληλη στήλη του χάρτη μνήμης. Μια τέτοια διευθυνσιοδότηση μπορεί να πάρει την μορφή μιας συνάρτησης απεικόνισης (mapping function) ενός χαρακτήρα ch του αλφαβήτου Σ, $map(ch, \Sigma)$, επιστρέφοντας τον αριθμό στήλης του χάρτη μνήμης. Περισσότερες λεπτομέρειες για την συνάρτηση απεικόνισης παρουσιάζονται στην εργασία [97].

Παρακάτω δίνεται ένα παράδειγμα για την κατασκευή του χάρτη μνήμης R. Το αλφάβητο Σ είναι {a,b,c,d,e} με $|\Sigma| = 5$. Το πρότυπο αλφαριθμητικό p είναι "abba" με $m = 4$. Στον Πίνακα 7.1 παρουσιάζεται ο χάρτης μνήμης bit-επιπέδου R.

Πίνακας 7.1: Χάρτης μνήμης bit-επιπέδου R

R	a	b	c	d	e
a	0	1	1	1	1
b	1	0	1	1	1
b	1	0	1	1	1
a	0	1	1	1	1

Απλή Αναζήτηση Αλφαριθμητικών

Ο αλγόριθμος δυναμικού προγραμματισμού για την απλή αναζήτηση αλφαριθμητικών είναι ίδιος με τον αλγόριθμο δυναμικού προγραμματισμού για την προσεγγιστική αναζήτηση αλφαριθμητικών με k αποτυχίες όπου το $k = 0$. Βλέπε στην επόμενη ενότητα.

Προσεγγιστική Αναζήτηση Αλφαριθμητικών με k Αποτυχίες

Υπολογίζουμε έναν $(m + 1) \times (n + 1)$ πίνακα δυναμικού προγραμματισμού $D_{0..m,0..n}$ τέτοιο ώστε $D_{i,j}$, $0 \leq i \leq m$ και $0 \leq j \leq n$ είναι η ελάχιστη απόσταση Hamming ανάμεσα σε $p_{1..i}$ και σε ένα υποαλφαριθμητικό του t που λήγει στην θέση t_j και έχει μήκος i. Συνεπώς, οι τιμές του πίνακα D υπολογίζονται από τον Αλγόριθμο 7.2.

```
for i = 0 to m do D_{i,0} ← i;
for j = 1 to n do
    D_{0,j} ← 0;
    c ← map(t_j, Σ);
    for i = 1 to m do
        D_{i,j} ← D_{i-1,j-1} + R_{i,c};
    end
end
```

Αλγόριθμος 7.2: Δυναμικού προγραμματισμού με k αποτυχίες

Η έκφραση $D_{i-1,j-1} + R_{i,c}$ αντιστοιχεί στην πράξη αντικατάστασης t_j για p_i. Αν $D_{m,j} \leq k$, $1 \leq j \leq n$ τότε σημαίνει ότι υπάρχει μια προσεγγιστική εμφάνιση του προτύπου που λήγει στη θέση j μέσα στο κείμενο με αριθμό αποτυχιών λιγότερο από ή ίσον με k. Ο αλγόριθμος αυτός εκτελείται σε $O(mn)$ βήματα και οι απαιτήσεις χώρου είναι $\lceil mn/16 \rceil$ bytes. Μια άλλη σημαντική παρατήρηση είναι ότι ο υπολογισμός της στήλης j εξαρτάται μόνο από τις τιμές της προηγούμενης στήλης $j-1$. Συνεπώς, δεν υπάρχει ανάγκη να αποθηκεύσουμε ολόκληρο τον πίνακα και έτσι οι απαιτήσεις χώρου μειώνονται σε $\lceil m/16 \rceil$ bytes.

Ακολουθεί ένα παράδειγμα για την κατασκευή του πίνακα δυναμικού προγραμματισμού D. Το παράδειγμα αυτό θα χρησιμοποιηθεί και στις παρακάτω ενότητες. Το αλφάβητο Σ είναι {a,b,c,d,e} με $|\Sigma| = 5$. Το πρότυπο αλφαριθμητικό p είναι "abba", με $m = 4$, το κείμενο t είναι "eabcdbbabbacd" με $n = 13$ και $k = 2$. Στον Πίνακα 7.2 δείχνει τον πίνακα δυναμικού προγραμματισμού D.

Πίνακας 7.2: Πίνακας δυναμικού προγραμματισμού D για την αναζήτηση του προτύπου p ="abba" μέσα στο κείμενο t ="eabcdbbabbacd" με δύο αποτυχίες. Τα έντονα κελιά δείχνουν την εμφάνιση των θέσεων στο κείμενο.

D		e	a	b	c	d	b	b	a	b	b	a	c	d
	0	0	0	0	0	0	0	0	0	0	0	0	0	0
a	1	1	0	1	1	1	1	1	0	1	1	0	1	1
b	2	2	2	0	2	2	1	1	2	0	1	2	1	2
b	3	3	3	2	1	3	2	1	2	2	0	2	3	2
a	4	4	3	4	3	**2**	4	3	**1**	3	3	**0**	3	4

Προσεγγιστική Αναζήτηση Αλφαριθμητικών με k Διαφορές

Υπολογίζουμε έναν $(m+1) \times (n+1)$ πίνακα δυναμικού προγραμματισμού $D_{0..m,0..n}$ τέτοιο ώστε $D_{i,j}$, $0 \leq i \leq m$ και $0 \leq j \leq n$ είναι η ελάχιστη απόσταση διόρθωσης (δηλαδή, ο ελάχιστος αριθμός διαφορών) ανάμεσα σε $p_{1..i}$ και σε οποιοδήποτε υποαλφαριθμητικό του t που λήγει στην t_j. Οι τιμές του πίνακα D υπολογίζονται από τον Αλγόριθμο 7.3 [145, 159, 163]:

```
for i = 0 to m do D_{i,0} ← i;
for j = 1 to n do
    D_{0,j} ← 0;
    c ← map(t_j, Σ);
    for i = 1 to m do
        D_{i,j} ← min(D_{i-1,j-1} + R_{i,c}, D_{i,j-1} + 1, D_{i-1,j} + 1) ;
    end
end
```

Αλγόριθμος 7.3: Δυναμικού προγραμματισμού με k διαφορές

Οι τρεις πρώτες εκφράσεις στην σχέση min αντιστοιχούν στην αντικατάσταση t_j για p_i, στην εισαγωγή t_j στο πρότυπο και στην διαγραφή p_i από τον πρότυπο αντίστοιχα. Αν $D_{m,j} \leq k$, $1 \leq j \leq n$ τότε σημαίνει ότι υπάρχει μια προσεγγιστική εμφάνιση του προτύπου που λήγει στην θέση j μέσα στο κείμενο με απόσταση διόρθωσης λιγότερη από ή ίση με k διαφορές. Επίσης, ο παραπάνω υπολογισμός αναζήτησης εκτελείται σε $O(mn)$ βήματα και απαιτεί χώρο $\lceil mn/16 \rceil$ bytes. Ο Πίνακας 7.3 δείχνει τα σχετικά αποτελέσματα.

Πίνακας 7.3: Πίνακας δυναμικού προγραμματισμού D για την αναζήτηση του προτύπου $p =$"abba" μέσα στο κείμενο $t =$"eabcdbbabbacd" με δύο διαφορές. Τα έντονα κελιά δείχνουν την εμφάνιση των θέσεων στο κείμενο.

D		e	a	b	c	d	b	b	a	b	b	a	c	d
	0	0	0	0	0	0	0	0	0	0	0	0	0	0
a	1	1	0	1	1	1	1	1	0	1	1	0	1	1
b	2	2	1	0	1	2	1	1	1	0	1	1	1	2
b	3	3	2	1	1	2	2	1	2	1	0	1	2	2
a	4	4	3	**2**	**2**	**2**	3	**2**	**1**	**2**	**1**	**0**	**1**	**2**

Προσεγγιστική Αναζήτηση Αλφαριθμητικών με k Διαφορές με βάση το Αλγόριθμο Myers

Πρέπει να αναφέρουμε ότι ο πίνακας δυναμικού προγραμματισμού D έχει μια σημαντική γεωμετρική ιδιότητα [159, 98]: τα γειτονικά κελιά κατά οριζόντια και κάθετη κατεύθυνση διαφέρουν κατά 1, 0 ή -1. Ορίζουμε το οριζόντιο δέλτα (horizontal delta) $\Delta h_{i,j}$ στο (i,j) σαν διαφορά $D_{i,j} - D_{i,j-1}$ και το κάθετο δέλτα (vertical delta) $\Delta v_{i,j}$ σαν διαφορά $D_{i,j} - D_{i-1,j}$ για $1 \leq i \leq m$ και $1 \leq j \leq n$. Έχουμε $-1 \leq \Delta h_{i,j}, \Delta v_{i,j} \leq 1$ για $1 \leq i \leq m$ και $1 \leq j \leq n$.

Μπορούμε τώρα να αντικαταστήσουμε το πρόβλημα υπολογισμού D με το πρόβλημα υπολογισμού των πινάκων δυναμικού προγραμματισμού Δh και Δv. Θα δείξουμε πώς υπολογίζουμε τις τιμές του δέλτα σε μια στήλη από την προηγούμενη στήλη. Για να αρχίσουμε, θεωρούμε ένα ξεχωριστό κελί του πίνακα δυναμικού προγραμματισμού που αποτελείται από το τετράγωνο $(i-1,j-1)$,$(i-1,j)$,$(i,j-1)$ και (i,j). Υπάρχουν δύο οριζόντιες και δύο κάθετες τιμές του δέλτα - $\Delta v_{i,j}$, $\Delta v_{i,j-1}$, $\Delta h_{i,j}$ και $\Delta h_{i-1,j}$- που συσχετίζονται με τις πλευρές του κελιού. Χρησιμοποιώντας τον ορισμό των δέλτα και την βασική σχέση για τις τιμές D καταλήγουμε στην ακόλουθη εξίσωση για το $\Delta v_{i,j}$ ως συνάρτηση των $R_{i,c}$, $\Delta v_{i,j-1}$ και $\Delta h_{i-1,j}$[125, 168]:
$\Delta v_{i,j} = min\{1, R_{i,c} - \Delta h_{i-1,j}, \Delta v_{i,j-1} - \Delta h_{i-1,j} + 1\}$.
Παρόμοια για το οριζόντιο δέλτα $\Delta h_{i,j}$:
$\Delta h_{i,j} = min\{1, R_{i,c} - \Delta v_{i,j-1}, \Delta h_{i-1,j} - \Delta v_{i,j-1} + 1\}$.
Συνεπώς, οι τιμές των πινάκων Δv και Δh υπολογίζονται από το Αλγόριθμο 7.4 [125, 168].

Για να υπολογίσουμε τη διαφορά της κάθε στήλης του πίνακα Δv συντηρούμε την τιμή του $e_j = D_{m,j}$ καθώς υπολογίζουμε τις τιμές Δv_j χρησιμοποιώντας το γεγονός ότι $e_0 = m$ και $e_j = e_{j-1} + \Delta h_{m,j}$. Αν $e_j \leq k$ τότε υπάρχει μια προσεγγιστική εμφάνιση του προτύπου που λήγει στην θέση j μέσα στο κείμενο με απόσταση διόρθωσης λιγότερη από ή ίση με k. Τέλος, ο χρόνος αναζήτησης του παραπάνω υπολογισμού είναι $O(mn)$ βήματα.

Στον Πίνακα 7.4 φαίνεται η λειτουργία του αλγορίθμου δυναμικού προγραμματισμού Δv. Πρέπει να σημειώσουμε ότι στον Πίνακα 7.4 υπάρχουν τιμές Δh για την τελευταία γραμμή του πίνακα.

```
for i = 1 to m do Δv_{i,0} ← 1;
e_0 ← m;
for j = 1 to n do
    Δh_{0,j} ← 0;
    c ← map(t_j, Σ);
    for i = 1 to m do
        Δv_{i,j} ← min{1, R_{i,c} − Δh_{i−1,j}, Δv_{i,j−1} − Δh_{i−1,j} + 1};
        Δh_{i,j} ← min{1, R_{i,c} − Δv_{i,j−1}, Δh_{i−1,j} − Δv_{i,j−1} + 1};
    end
    e_j ← e_{j−1} + Δh_{m,j};
end
```

Αλγόριθμος 7.4: Δυναμικού προγραμματισμού με k διαφορές

7.3 Αλγόριθμοι Bit-Παραλληλισμού

Σε αυτή την ενότητα, παρουσιάζουμε αλγορίθμους που βασίζονται στην προσομοίωση ενός μη-ντετερμινιστικού πεπερασμένου αυτομάτου (nondeterministic finite automaton - NFA) που κατασκευάζεται από το πρότυπο και χρησιμοποιεί το κείμενο σαν είσοδο.

Πίνακας 7.4: Πίνακας δυναμικού προγραμματισμού Δv για την αναζήτηση του προτύπου p ="abba" μέσα στο κείμενο t ="eabcdbbabbacd" με δύο διαφορές. Τα έντονα κελιά δείχνουν την εμφάνιση των θέσεων στο κείμενο.

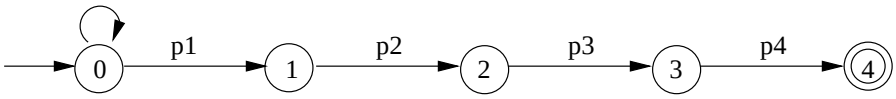
Δv		e	a	b	c	d	b	b	a	b	b	a	c	d
		0	0	0	0	0	0	0	0	0	0	0	0	0
a	1	1	0	1	1	1	1	1	0	1	1	0	1	1
b	1	1	1	-1	0	1	0	0	1	-1	0	1	0	1
b	1	1	1	1	0	0	1	0	1	1	-1	0	1	0
a	1	1	1	1	1	0	1	1	-1	1	1	-1	-1	0
Δh		0	-1	-1	0	0	1	-1	-1	1	-1	-1	1	1
e		4	3	**2**	**2**	**2**	3	**2**	**1**	**2**	**1**	**0**	**1**	**2**

7.3.1 Αλγόριθμος βασισμένος στις Γραμμές του Αυτομάτου

Οι αλγόριθμοι αυτής της κατηγορίας αποτελούνται από δύο φάσεις: τη φάση της προεπεξεργασίας και τη φάση της αναζήτησης. Η φάση της προεπεξεργασίας είναι ίδια όπως στον αλγόριθμο δυναμικού προγραμματισμού και οι λεπτομέρειες περιγράφονται στην ενότητα 7.2.1. Η φάση της αναζήτησης βασίζεται στην προσομοίωση του μη-ντετερμινιστικού πεπερασμένου αυτομάτου κατά γραμμές.

Απλή Αναζήτηση Αλφαριθμητικών

Θεωρούμε το αυτόματο NFA για πρότυπο μήκους $m = 4$ χωρίς διαφορές όπως φαίνεται στο Σχήμα 7.1. Κάθε στήλη παριστάνει την ταύτιση του προτύπου μέχρι σε μια ορισμένη θέση. Ο αυτό-βρόχος (self-loop) στην αρχική κατάσταση μας επιτρέπει να θεωρήσουμε οποιοδήποτε χαρακτήρα ως πιθανό αρχικό σημείο μιας ταύτισης. Το αυτόματο NFA έχει $m + 1$ καταστάσεις. Αναθέτουμε τον αριθμό i στην κατάσταση στην στήλη i, $0 \leq i \leq m$. Αρχικά, η ενεργή κατάσταση (active state) είναι η στήλη 0.



Σχήμα 7.1: NFA για απλή αναζήτηση αλφαριθμητικών

Έχουμε ένα διάνυσμα F^0 που αντιστοιχεί στο αυτόματο NFA. F_i^0 είναι 0 αν η κατάσταση i είναι ενεργή και 1 διαφορετικά. Το διάνυσμα αλλάζει καθώς διαβάζει κάθε χαρακτήρα του κειμένου. Οι νέες τιμές του διανύσματος $F_i'^0$, $0 \leq i \leq m$ αφού διαβάσουμε ένα καινούργιο χαρακτήρα κειμένου t_j, $1 \leq j \leq n$, υπολογίζονται από τον Αλγόριθμο 7.5 [7].

Ο πρώτος όρος του τύπου F'^0 παριστάνει την ταύτιση. Αν $F_m'^0 = 0$ τότε υπάρχει μια εμφάνιση του προτύπου μέσα στο κείμενο. Τέλος, αυτή η προσομοίωση του NFA απαιτεί $O(mn)$ βήματα.

Ο Πίνακας 7.5 δείχνει το διάνυσμα F'^0. Τα διανύσματα F'^1 και F'^2 εξηγούνται αργότερα.

```
for i = 0 to m do F_i^0 ← 1;
for i = 0 to 0 do F_i^0 ← 0;
for j = 1 to n do
    c ← map(t_j, Σ);
    for i = 1 to m do
        F'_i^0 ← F_{i-1}^0 OR R_{i,c};
    end
    for i = 0 to m do
        F_i^0 ← F'_i^0;
    end
end
```

Αλγόριθμος 7.5: Bit-παραλληλισμού για απλή αναζήτηση αλφαριθμητικών

Πίνακας 7.5: Διανύσματα F' για την αναζήτηση του προτύπου p="abba" μέσα στο κείμενο t="eabcdbbabbacd" με $k = 0$, $k = 1$ και $k = 2$ αποτυχίες.

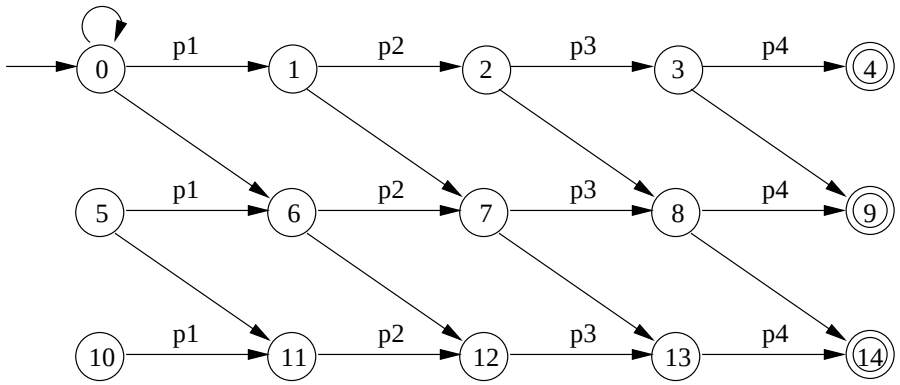
F'^0		e	a	b	c	d	b	b	a	b	b	a	c	d
	0	0	0	0	0	0	0	0	0	0	0	0	0	0
a	1	1	0	1	1	1	1	1	0	1	1	0	1	1
b	1	1	1	0	1	1	1	1	1	0	1	1	1	1
b	1	1	1	1	1	1	1	1	1	1	0	1	1	1
a	1	1	1	1	1	1	1	1	1	1	1	0	1	1

F'^1		e	a	b	c	d	b	b	a	b	b	a	c	d
	0	0	0	0	0	0	0	0	0	0	0	0	0	0
a	0	0	0	0	0	0	0	0	0	0	0	0	0	0
b	1	1	1	0	1	1	0	0	1	0	0	1	0	1
b	1	1	1	1	0	1	1	0	1	1	0	1	1	1
a	1	1	1	1	1	1	1	1	0	1	1	0	1	1

F'^2		e	a	b	c	d	b	b	a	b	b	a	c	d
	0	0	0	0	0	0	0	0	0	0	0	0	0	0
a	0	0	0	0	0	0	0	0	0	0	0	0	0	0
b	0	0	0	0	0	0	0	0	0	0	0	0	0	0
b	1	1	1	0	0	1	0	0	0	0	0	0	1	0
a	1	1	1	1	1	0	1	1	0	1	1	0	1	1

Προσεγγιστική Αναζήτηση Αλφαριθμητικών με k Αποτυχίες

Θεωρούμε το αυτόματο NFA για πρότυπο μήκους $m = 4$ με $k = 2$ αποτυχίες όπως φαίνεται στο Σχήμα 7.2.



Σχήμα 7.2: NFA για αναζήτηση αλφαριθμητικών με αποτυχίες

Κάθε γραμμή συμβολίζει τον αριθμό των αποτυχιών. Η πρώτη γραμμή αντιστοιχεί στην αποτυχία 0, η δεύτερη γραμμή αντιστοιχεί στην αποτυχία 1 και ουτώ καθεξής. Οι οριζόντιες γραμμές παριστάνουν την ταύτιση ενός χαρακτήρα και οι διαγώνιες γραμμές παριστάνουν την αντικατάσταση χαρακτήρα του κειμένου με τον αντίστοιχο χαρακτήρα στο πρότυπο (που είναι μια αποτυχία). Το αυτόματο NFA έχει $(m+1) \times (k+1)$ καταστάσεις. Αναθέτουμε τον αριθμό (l, i) στην κατάσταση της γραμμής l και της στήλης i, $0 \leq l \leq k$, $0 \leq i \leq m$. Αρχικά, οι ενεργές καταστάσεις στην γραμμή l είναι οι στήλες από 0 έως l.

Θεωρούμε $k+1$ διανύσματα F^l, $0 \leq l \leq k$ που αντιστοιχούν στις γραμμές του αυτομάτου. F_i^l είναι 0 αν η κατάσταση (l, i) είναι ενεργή και 1 διαφορετικά. Οι νέες τιμές των διανυσμάτων $F_i'^l$, $0 \leq l \leq k$, $0 \leq i \leq m$, αφού διαβάσουμε ένα καινούργιο χαρακτήρα κειμένου t_j, $1 \leq j \leq n$ υπολογίζονται από τον Αλγόριθμο 7.6:

```
for l = 0 to k do
    for i = 0 to m do
        F_i^l ← 1;
    end
end
for l = 0 to k do
    for i = 0 to l do
        F_i^l ← 0;
    end
end
for j = 1 to n do
    c ← map(t_j, Σ);
    for i = 1 to m do
        F'_i^0 ← F_{i-1}^0 OR R_{i,c};
    end
    for l = 1 to k do
        for i = 1 to m do
            F'_i^l ← (F_{i-1}^l OR R_{i,c}) AND F_{i-1}^{l-1};
        end
    end
    for l = 0 to k do
        for i = 0 to m do
            F_i^l ← F'_i^l;
        end
    end
end
```

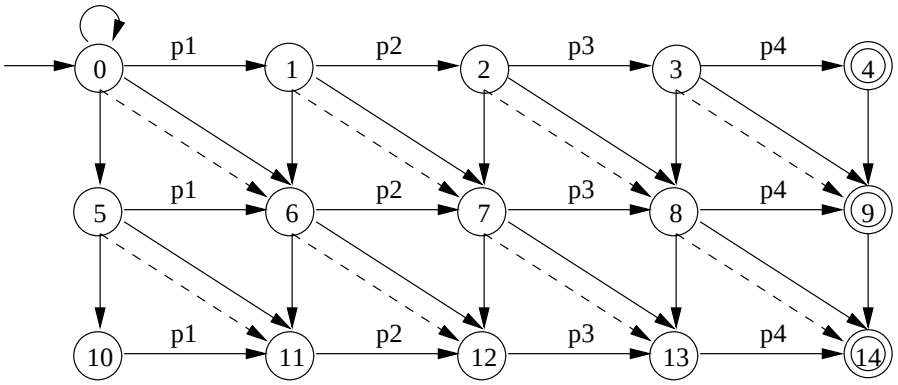
Αλγόριθμος 7.6: Bit-παραλληλισμού για προσεγγιστική αναζήτηση αλφαριθμητικών με k αποτυχίες

Στον τύπο F'^l, ο πρώτος όρος παριστάνει την ταύτιση και ο δεύτερος όρος παριστάνει την πράξη αντικατάστασης. Αν $F'^k_m = 0$ τότε υπάρχει μια προσεγγιστική εμφάνιση του προτύπου μέσα στο κείμενο με k αποτυχίες. Η προηγούμενη προσομοίωση του NFA εκτελείται σε $O(nmk)$ βήματα. Στον

Πίνακα 7.5 φαίνονται τα διανύσματα F'^l.

Προσεγγιστική Αναζήτηση Αλφαριθμητικών με k Διαφορές

Θεωρούμε το αυτόματο NFA για αναζήτηση του προτύπου μήκους $m = 4$ με $k = 2$ διαφορές όπως φαίνεται στο Σχήμα 7.3. Κάθε γραμμή συμβολίζει τον αριθμό των διαφορών όπως στην περίπτωση για τις αποτυχίες. Η δομή του αυτομάτου είναι παρόμοια με εκείνη της περίπτωσης των αποτυχιών προσθέτοντας δύο επιπλέον μεταπτώσεις ανά κατάσταση. Οι διακεκομένες διαγώνιες γραμμές παριστάνουν την διαγραφή ενός χαρακτήρα από το πρότυπο, ενώ οι κάθετες γραμμές παριστάνουν την εισαγωγή ενός χαρακτήρα στο πρότυπο. Θέτουμε F_i^l σε 0 αν η κατάσταση στην γραμμή l και στην στήλη i είναι ενεργή. Οι νέες τιμές των διανυσμάτων $F_i'^l$, $0 \leq l \leq k$, $0 \leq i \leq m$, αφού διαβάσουμε ένα καινούργιο χαρακτήρα κειμένου t_j, $1 \leq j \leq n$, υπολογίζονται από τον Αλγόριθμο 7.7 [167]:



Σχήμα 7.3: NFA για αναζήτηση αλφαριθμητικών με διαφορές

```
for l = 0 to k do
    for i = 0 to m do
        F_i^l ← 1;
    end
end
for l = 0 to k do
    for i = 0 to l do
        F_i^l ← 0;
    end
end
for j = 1 to n do
    c ← map(t_j, Σ);
    for i = 1 to m do
        F'_i^0 ← F_{i-1}^0 OR R_{i,c};
    end
    for l = 1 to k do
        for i = 1 to m do
            F'_i^l ← (F_{i-1}^l OR R_{i,c}) AND F_i^{l-1} AND F_{i-1}^{l-1} AND F'_{i-1}^{l-1};
        end
    end
    for l = 0 to k do
        for i = 0 to m do
            F_i^l ← F'_i^l;
        end
    end
end
```

Αλγόριθμος 7.7: Bit-παραλληλισμού για προσεγγιστική αναζήτηση αλφαριθμητικών με k διαφορές

Στον τύπο F'^l, ο πρώτος όρος παριστάνει την ταύτιση, ο δεύτερος όρος παριστάνει την πράξη εισαγωγής, ο τρίτος όρος παριστάνει την πράξη αντικατάστασης και ο τελευταίος όρος παριστάνει την πράξη διαγραφής. Αν $F'^k_m = 0$ τότε υπάρχει μια προσεγγιστική εμφάνιση του προτύπου μέσα στο

κείμενο με k διαφορές. Η εκτέλεση απαιτεί $O(mnk)$ βήματα. Στον Πίνακα 7.6 φαίνονται τα διανύσματα F'^l.

Πίνακας 7.6: Διανύσματα F' για την αναζήτηση του προτύπου p="abba" μέσα στο κείμενο t="eabcdbbabbacd" με $k = 0$, $k = 1$ και $k = 2$ διαφορές.

F'^0		e	a	b	c	d	b	b	a	b	b	a	c	d
	0	0	0	0	0	0	0	0	0	0	0	0	0	0
a	1	1	0	1	1	1	1	1	0	1	1	0	1	1
b	1	1	1	0	1	1	1	1	1	0	1	1	1	1
b	1	1	1	1	1	1	1	1	1	1	0	1	1	1
a	1	1	1	1	1	1	1	1	1	1	1	0	1	1

F'^1		e	a	b	c	d	b	b	a	b	b	a	c	d
	0	0	0	0	0	0	0	0	0	0	0	0	0	0
a	0	0	0	0	0	0	0	0	0	0	0	0	0	0
b	1	1	0	0	0	1	0	0	0	0	0	0	0	1
b	1	1	1	0	0	1	1	0	1	0	0	0	1	1
a	1	1	1	1	1	1	1	1	0	1	0	0	0	1

F'^2		e	a	b	c	d	b	b	a	b	b	a	c	d
	0	0	0	0	0	0	0	0	0	0	0	0	0	0
a	0	0	0	0	0	0	0	0	0	0	0	0	0	0
b	0	0	0	0	0	0	0	0	0	0	0	0	0	0
b	1	1	0	0	0	0	0	0	0	0	0	0	0	0
a	1	1	1	0	0	0	1	0	0	0	0	0	0	0

7.3.2 Αλγόριθμος βασισμένος στις Στήλες του Αυτομάτου

Η φάση αναζήτησης αυτού του αλγορίθμου βασίζεται στην προσομοίωση του αυτομάτου κατά στήλες.

Απλή Αναζήτηση Αλφαριθμητικών

Η προσομοίωση του NFA κατά στήλες για την απλή αναζήτηση αλφαριθμητικών είναι ίδια με την προσομοίωση του NFA για την προσεγγιστική αναζήτηση αλφαριθμητικών με k αποτυχίες όπου $k = 0$. Βλέπε στην παρακάτω ενότητα.

Προσεγγιστική Αναζήτηση Αλφαριθμητικών με k Αποτυχίες

Ας ορίσουμε τους $m+1$ αριθμούς C_i, $0 \leq i \leq m$ στο διάστημα $0...k+1$ που αντιστοιχούν στις στήλες του αυτομάτου. Κάθε τιμή του C_i παριστάνει το μικρότερο επίπεδο ενεργής κατάστασης ανά στήλη. Οι νέες τιμές της κάθε στήλης C'_i αφού διαβάσουμε ένα καινούργιο χαρακτήρα κειμένου υπολογίζονται από τον παρακάτω Αλγόριθμο 7.8 [9]:

```
for i = 0 to m do C_i ← i;
for j = 1 to n do
    c ← map(t_j, Σ);
    for i = 1 to m do
        C'_i ← C_{i-1} + R_{i,c};
    end
    for i = 0 to m do
        C_i ← C'_i ;
    end
end
```

Αλγόριθμος 7.8: Προσεγγιστική αναζήτηση αλφαριθμητικών με k αποτυχίες

Στον τύπο C'_i, ο πρώτος όρος παριστάνει είτε την ταύτιση ή την αντικατάσταση. Αν $C'_m \leq k$, τότε υπάρχει μια προσεγγιστική εμφάνιση του προτύπου μέσα στο κείμενο με k αποτυχίες. Η λύση αυτή απαιτεί $O(mn)$ βήματα. Αναγνωρίζουμε την λύση αυτή σαν παραλλαγή της δημοφιλούς προσέγγισης δυναμικού προγραμματισμού για το πρόβλημα της προσεγγιστικής αναζήτησης αλφαριθμητικών με k αποτυχίες. Για ένα δοσμένο χαρακτήρα κειμένου αν συλλέξουμε σε κάθε στήλη του αυτομάτου τις μικρότερες ενεργές γραμμές τότε προκύπτει το κάθετο διάνυσμα από τον πίνακα δυναμικού προγραμματισμού. Για παράδειγμα, αν συλλέξουμε τις μικρότερες ενεργές γραμμές για τον τελευταίο χαρακτήρα κειμένου d από τον Πίνακα 7.5 τότε προκύπτει το κάθετο διάνυσμα [1,2,2,3] που είναι παρόμοιο με την τελευταία στήλη του Πίνακα 7.2.

Προσεγγιστική Αναζήτηση Αλφαριθμητικών με k Διαφορές

Παρόμοια, ορίζουμε τους $m+1$ αριθμούς C_i, $0 \leq i \leq m$ στο διάστημα $0...k+1$ που αντιστοιχούν στις στήλες του αυτομάτου. Κάθε τιμή του C_i παριστάνει την ελάχιστη γραμμή του NFA ανά στήλη. Οι νέες τιμές της κάθε στήλης C'_i αφού διαβάσουμε ένα καινούργιο χαρακτήρα κειμένου υπολογίζονται από το παρακάτω Αλγόριθμο 7.9 [9]:

```
for i = 0 to m do
    C_i ← i;
    C'_i ← C_i;
end
for j = 1 to n do
    c ← map(t_j, Σ);
    for i = 1 to m do
        C'_i ← min{C_{i-1} + R_{i,c}, C_i + 1, C'_{i-1} + 1};
    end
    for i = 0 to m do
        C_i ← C'_i;
    end
end
```

Αλγόριθμος 7.9: Προσεγγιστική αναζήτηση αλφαριθμητικών με k διαφορές

Στον τύπο C'_i, ο πρώτος όρος παριστάνει είτε την ταύτιση ή την αντικατάσταση, ο δεύτερος όρος παριστάνει την εισαγωγή και ο τελευταίος όρος παριστάνει την διαγραφή. Αν $C'_m \leq k$, τότε υπάρχει μια προσεγγιστική εμφάνιση του προτύπου μέσα στο κείμενο με k διαφορές. Ο παραπάνω υπολογισμός αναζήτησης απαιτεί $O(mn)$ βήματα. Αν συλλέξουμε τις μικρότερες ενεργές γραμμές για τον τελευταίο χαρακτήρα κειμένου d από τον Πίνακα 7.6 τότε προκύπτει το κάθετο διάνυσμα [1,2,2,2] που είναι παρόμοιο με την τελευταία στήλη του Πίνακα 7.3.

7.3.3 Αλγόριθμος βασισμένος στις Διαγώνιες του Αυτομάτου

Η φάση αναζήτησης του αλγορίθμου βασίζεται στην προσομοίωση του αυτομάτου κατά διαγωνίους. Στην επόμενη παράγραφο περιγράφουμε την προσομοίωση του αυτομάτου κατά διαγωνίους για το γενικό πρόβλημα προσεγγιστικής αναζήτησης αλφαριθμητικών.

Προσεγγιστική Αναζήτηση Αλφαριθμητικών με k Διαφορές

Υποθέτουμε ότι χρησιμοποιούμε τις πλήρεις διαγώνιες γραμμές του αυτομάτου μήκους $k+1$. Η κατάσταση της κάθε διαγωνίου i παριστάνεται από ένα αριθμό D_i, την μικρότερη ενεργή γραμμή σε εκείνη τη διαγώνιο (δηλαδή, την μικρότερη διαφορά). Τότε η κατάσταση αυτής της προσομοίωσης αποτελείται από $m-k+1$ αριθμούς στο διάστημα $0...k+1$. Οι νέες τιμές για D_i, $1 \leq i \leq m-k$, αφού διαβάσουμε έναν καινούργιο χαρακτήρα κειμένου t_j, $1 \leq j \leq n$ υπολογίζονται από τον παρακάτω Αλγόριθμο 7.10 [9, 13]:

```
D_0 ← 0;
for i = 1 to m − k do
    D_i ← k + 1;
end
for j = 1 to n do
    c ← map(t_j, Σ);
    for i = 1 to m − k do
        D'_i ← min{D_i + 1, D_{i+1} + 1, g(D_{i−1}, t_j)};
    end
    for i = 0 to m − k do
        D_i ← D'_i;
    end
end
```

Αλγόριθμος 7.10: Προσεγγιστική αναζήτηση αλφαριθμητικών με k διαφορές

όπου $g(D_i, t_j)$ παίρνει την τιμή D_i αν υπάρχει μια θέση r τέτοια ώστε $r \geq D_i$ και $p_{i+r} = t_j$, διαφορετικά παίρνει την τιμή $k+1$. Ο πρώτος όρος του τύπου D' παριστάνει την αντικατάσταση, που

ακολουθεί στην ίδια διαγώνιο. Ο δεύτερος όρος παριστάνει την εισαγωγή ενός χαρακτήρα. Τέλος, ο τελευταίος όρος παριστάνει την ταύτιση ενός χαρακτήρα: επιλέγουμε την ελάχιστη ενεργή κατάσταση (το min του τύπου g) της προηγούμενης διαγωνίου που ταυτίζει το κείμενο και μπορεί να μετακινηθεί στην τρέχουσα διαγώνιο. Βρίσκουμε μια προσεγγιστική εμφάνιση μέσα στο κείμενο οπουδήποτε $D_{m-k} \leq k$. Τέλος, ο αλγόριθμος αναζήτησης εκτελείται σε $O((m-k)n)$ βήματα. Όμως, αυτή η προσομοίωση έχει τον πλεονέκτημα ότι μπορεί να υπολογίζει παράλληλα για όλες τις τιμές i και η πολυπλοκότητα της αναζήτησης μειώνεται σε $O(n)$ βήματα. Ο Πίνακας 7.7 δείχνει τις καταστάσεις D'.

Πίνακας 7.7: Καταστάσεις D' για την αναζήτηση του προτύπου p="abba" μέσα στο κείμενο t="eabcdbbabbacd" με $k = 2$ διαφορές. Τα έντονα κελιά δείχνουν την εμφάνιση των θέσεων στο κείμενο.

D'		e	a	b	c	d	b	b	a	b	b	a	c	d
D'_1	3	3	0	0	1	2	0	0	0	0	0	0	1	2
D'_2	3	3	3	**0**	**1**	**2**	3	**0**	**0**	**0**	**0**	**1**	**2**	3

7.4 Περιορισμένες Εκφράσεις

Σε αυτή την ενότητα περιγράφουμε τις περιορισμένες εκφράσεις που μπορούν να υποστηρίξουν οι προηγούμενοι αλγόριθμοι αναζήτησης αλφαριθμητικών. Μια περιορισμένη έκφραση [168] είναι ένα πρότυπο που ταυτίζει όχι μόνο ένα χαρακτήρα αλλά ένα σύνολο χαρακτήρων και είναι ένα υποσύνολο από τα πολύπλοκα πρότυπα που παρουσιάζονται στις εργασίες [7, 167]. Χρησιμοποιούμε τα σύμβολα '*', '^', '[', και ']' για να συμβολίζουν κοινούς τύπους συμβόλων ως εξής (που είναι παρόμοια με τα γνωστά προγράμματα του Unix *grep* και *agrep*):

- Το αδιάφορο σύμβολο (don't care) είναι το '*' που παριστάνει την ταύτιση με ένα οποιοδήποτε χαρακτήρα.
- Το σύμβολο άρνησης (complement) είναι το '^' που παριστάνει την ταύτιση με όλους τους χαρακτήρες εκτός από τον χαρακτήρα που δεν θέλουμε να εμφανίζεται.
- Το σύμβολο της κλάσης χαρακτήρων (class of characters) είναι ένα ζεύγος από '[' και ']' που παριστάνει την ταύτιση που ορίζει η κλάση ή το διάστημα χαρακτήρων.

Για να αναζητήσουμε τις περιορισμένες εκφράσεις τροποποιούμε μόνο την φάση προεπεξεργασία χωρίς επιπλέον πολυπλοκότητα αναζήτησης. Επομένως κάνουμε την ακόλουθη τροποποίηση:

$$R_{i,j} = \begin{cases} 0, & \text{αν } p_i{=}*, \text{ για } 1{\leq}\ j\ \leq |\Sigma| \\ 0, & \text{αν } p_i{=}\hat{}\sigma_1, \text{ για } 1{\leq}\ j\ \leq |\Sigma| \text{ και } j\ {\neq}map(\sigma_1, \Sigma) \text{ για } 1{\leq}\ i\ \leq m \\ 0, & \text{αν } p_i{=}[\sigma_1\sigma_2], \text{ για } map(\sigma_1, \Sigma)\ \leq j\ {\leq}map(\sigma_2, \Sigma) \end{cases}$$

όπου σ_j είναι ο j-οστό χαρακτήρας του αλφαβήτου.

Παρακάτω παρουσιάζεται ένα παράδειγμα για την κατασκευή του χάρτη μνήμης R για τις τρεις περιπτώσεις των περιορισμένων εκφράσεων. Το αλφάβητο Σ είναι {a,b,c,d,e} με $|\Sigma| = 5$. Το πρότυπο αλφαριθμητικό p είναι "abba", με $m = 4$. Αν ο πρώτος χαρακτήρας του προτύπου αντικαθίστανται από ένα αδιάφορο χαρακτήρα τότε ο χάρτης μνήμης bit-επιπέδου R τροποποιείται όπως φαίνεται στον Πίνακα 7.8. Παρόμοια, αν ο πρώτος χαρακτήρας του προτύπου είναι η άρνηση του χαρακτήρα 'a' τότε ο χάρτης μνήμης bit-επιπέδου R τροποποιείται όπως φαίνεται στον Πίνακα 7.8. Τέλος, αν ο πρώτος χαρακτήρας του προτύπου είναι το διάστημα χαρακτήρων [ad] τότε ο χάρτης μνήμης bit-επιπέδου R τροποποιείται όπως φαίνεται στον Πίνακα 7.8.

Πίνακας 7.8: Χάρτες μνήμης bit-επιπέδου για τις περιορισμένες εκφράσεις

R	a	b	c	d	e
*	0	0	0	0	0
b	1	0	1	1	1
b	1	0	1	1	1
a	0	1	1	1	1

R	a	b	c	d	e
^a	1	0	0	0	0
b	1	0	1	1	1
b	1	0	1	1	1
a	0	1	1	1	1

R	a	b	c	d	e
[ad]	0	1	1	0	1
b	1	0	1	1	1
b	1	0	1	1	1
a	0	1	1	1	1

7.5 Αλγόριθμος Bit-Παραλληλισμού για το Πρόβλημα LCS

Σε αυτή την παράγραφο περιγράφεται ο ακολουθιακός αλγόριθμος bit-παραλληλισμού για το πρόβλημα LCS που αναφέρεται στις εργασίες [32, 33]. Ο αλγόριθμος έχει δύο φάσεις: τη φάση της προεπεξεργασίας και τη φάση του υπολογισμού LCS.

7.5.1 Φάση Προεπεξεργασίας

Η φάση προεπεξεργασίας του αλγορίθμου LCS κατασκευάζει δύο χάρτες μνήμης bit-επιπέδου M και MN. Ο χάρτης μνήμης bit-επιπέδου M είναι παρόμοιος με το χάρτη μνήμης R που περιγράφεται στην ενότητα 7.2.1. Επίσης ορίζουμε τον χάρτη μνήμη bit-επιπέδου MN ως άρνηση του M. Συνεπώς, παρακάτω παρουσιάζουμε τον Αλγόριθμο 7.11 που κατασκευάζει τους χάρτες μνήμης M και MN.

```
for j = 1 to |Σ| do
    for i = 1 to m do
        M_{i,j} ← 0;
        if x_i = σ_j then M_{i,j} ← 1 και MN_{i,j} ← NOT M_{i,j};
    end
end
```

Αλγόριθμος 7.11: Φάση Προεπεξεργασίας

7.5.2 Φάση Υπολογισμού LLCS και LCS

Μια απλή κλασσική λύση για την φάση υπολογισμού LLCS που υπολογίζει το μήκος μιας ακολουθίας LCS των αλφαριθμητικών x και y, είναι ο αλγόριθμος δυναμικού προγραμματισμού. Έστω $L_{0..m,0..n}$ ο πίνακας δυναμικού προγραμματισμού, όπου $L_{i,j}$ παριστάνει το μήκος του LCS για $x_1...x_i$ και $y_1...y_j$, $1 \leq i \leq m$, $1 \leq j \leq n$. Ο πίνακας L μπορεί να υπολογιστεί με τον Αλγόριθμο 7.12 [54].

Για να ανακτήσουμε μια ακολουθία LCS των αλφαριθμητικών x και y χρησιμοποιούμε τον πίνακα δυναμικού προγραμματισμού L. Έστω $LCS_{0..m,0..n}$ ο πίνακας δυναμικού προγραμματισμού, όπου $LCS_{i,j}$ παριστάνει την ακολουθία LCS των δύο αλφαριθμητικών $x_1...x_i$ και $y_1...y_j$, $1 \leq i \leq m$, $1 \leq j \leq n$. Ο πίνακας LCS μπορεί να υπολογιστεί με τον Αλγόριθμο 7.13.

Ο Αλγόριθμος 7.12 που υπολογίζει τον πίνακα L μπορεί να συνδυαστεί με τον Αλγόριθμο 7.13 που υπολογίζει τον πίνακα LCS σε μια μόνο φάση.

Ο Crochemore και άλλοι [32, 33] παρουσίασαν έναν καινούργιο τρόπο για τον υπολογισμό του μήκους της ακολουθίας LCS δύο αλφαριθμητικών χρησιμοποιώντας πράξεις bit-διανυσμάτων. Η ιδιότητα της μονοτονικότητας (monotonicity property) στον πίνακα L μας επιτρέπει να αποθηκεύουμε κάθε

```
for i = 0 to m do L_{i,0} ← 0;
for j = 1 to n do
    L_{0,j} ← 0;
    for i = 1 to m do
        if x_i = y_j then
            L_{i,j} ← L_{i-1,j-1} + 1;
        else
            L_{i,j} ← max(L_{i,j-1}, L_{i-1,j});
        end
    end
end
```

Αλγόριθμος 7.12: Δυναμικού προγραμματισμού για τον υπολογισμό του μήκους της ακολουθίας LCS

```
for i = 0 to m do LCS_{i,0} ← ε;
for j = 1 to n do
    LCS_{0,j} ← ε;
    for i = 1 to m do
        if L_{i,j} = L_{i,j-1} then
            LCS_{i,j} ← LCS_{i,j-1};
        else
            if x_i = y_j και L_{i,j} = 1 + L_{i-1,j} then
                LCS_{i,j} ← LCS_{i-1,j} x_i;
            else
                LCS_{i,j} ← LCS_{i-1,j};
            end
        end
    end
end
```

όπου ε συμβολίζει το κενό αλφαριθμητικό.

Αλγόριθμος 7.13: Δυναμικού προγραμματισμού για την ανάκτηση της ακολουθίας LCS

στήλη στον πίνακα L χρησιμοποιώντας bit-διανύσματα. Θα συμβολίσουμε με ΔL και c την σχετική κωδικοποίηση του πίνακα L ως εξής: $\Delta L_{i,j} = L_{i,j} - L_{i-1,j}$, $c_{i,j} = L_{i,j} - L_{i,j-1}$ και $0 \leq \Delta L_{i,j}, c_{i,j} \leq 1$ για $1 \leq i \leq m$ και $1 \leq j \leq n$. Επίσης, ορίζουμε το ΔLN ως άρνηση του ΔL (ΔLN=NOT ΔL). Η φάση υπολογισμού LLCS παίρνει την μορφή μιας σειράς από 3 πράξεις bit (AND, OR και AND) και μιας πράξης πρόσθεσης που εξαρτώνται από την προηγούμενη στήλη $\Delta LN_{i,j-1}$ και τους χάρτες μνήμης bit- επιπέδου (M και MN) για τα σημεία ταύτισης (match points) στην τρέχουσα στήλη. Για τη φάση υπολογισμού LCS αντικαθιστούμε το ψευδοκώδικα που υπολογίζει τον πίνακα LCS χρησιμοποιώντας τους ορισμούς των ΔL, c και M. Επίσης, ορίζουμε $LCS_{i,j}$ την δυαδική αναπαράσταση των m bits για την ακολουθία LCS των δύο αλφαριθμητικών $x_1...x_i$ και $y_1...y_j$ ($1 \leq i \leq m$ και $1 \leq j \leq n$) ως εξής: κάθε bit του $LCS_{i,j}$ παίρνει την τιμή 1 αν το i-οστό χαρακτήρα του αλφαριθμητικού x ανήκει στην ακολουθία LCS διαφορετικά παίρνει την τιμή 0. Συνεπώς, η φάση υπολογισμού LLCS και LCS δίνεται από τον παρακάτω Αλγόριθμο 7.14:

```
for i = 1 to m do
    ΔLN_{i,0} ← 1;
    LCS_{i,0} ← 0^m;
end
for j = 1 to n do
    k ← map(y_j, Σ);
    c_{0,j} ←0;
    LCS_{0,j} ← 0^m;
    for i = 1 to m do
        ΔLN_{i,j} ← (c_{i-1,j}+ΔLN_{i,j-1}+(ΔLN_{i,j-1} AND M_{i,k})) OR (ΔLN_{i,j-1} AND MN_{i,k});
        c_{i,j} ← (c_{i-1,j} + ΔLN_{i,j-1} + (ΔLN_{i,j-1} AND M_{i,k})) mod 2;
        if c_{i,j} = 0 then
            LCS_{i,j} ← LCS_{i,j-1};
        else
            if M_{i,k} = 1 και NOT ΔLN_{i,j} = 1 then
                LCS_{i,j} ← LCS_{i-1,j} OR 0^{i-1} M_{i,k} 0^{m-i};
            else
                LCS_{i,j} ← LCS_{i-1,j};
            end
        end
    end
end
```

Αλγόριθμος 7.14: Bit-παραλληλισμού για τον πρόβλημα LCS

Το bit-διάνυσμα $c_{m,j}$ για $1 \leq j \leq n$ δίνει το μήκος της ακολουθίας LCS, δηλαδή, το πλήθος των μονάδων στην τελευταία γραμμή του c. Το στοιχείο $LCS_{m,n}$ δίνει την ακολουθία LCS δύο αλφαριθμητικών. Οι απαιτήσεις χώρου είναι $\lceil mn/2 \rceil$ bytes επειδή απαιτούνται δύο bits για ΔLN και c αντίστοιχα. Τέλος, ο χρόνος υπολογισμού εκτελείται περίπου σε $O(nm)$ βήματα (για $m \leq n$).

Στη συνέχεια παρουσιάζεται ένα παράδειγμα του παραπάνω αλγορίθμου. Το αλφάβητο Σ είναι $\{a, b, c\}$ με $|\Sigma| = 3$. Το αλφαριθμητικό x είναι "abccb" με $m = 5$ και το αλφαριθμητικό y είναι "bcabcb" με $n = 6$. Οι χάρτες μνήμης bit-επιπέδου M και MN φαίνονται στον Πίνακα 7.9.

Πίνακας 7.9: Χάρτες μνήμης bit-επιπέδου M και MN για x=abccb

M	a	b	c
a	1	0	0
b	0	1	0
c	0	0	1
c	0	0	1
b	0	1	0

MN	a	b	c
a	0	1	1
b	1	0	1
c	1	1	0
c	1	1	0
b	1	0	1

Πίνακας 7.10: Φάση Υπολογισμού LLCS

ΔLN	ε	b	c	a	b	c	b
a	1	1	1	0	0	0	0
b	1	0	0	1	0	0	0
c	1	1	0	0	1	0	0
c	1	1	1	1	1	1	1
b	1	1	1	1	0	1	0

c	b	c	a	b	c	b
0	0	0	0	0	0	0
a	0	0	1	0	0	0
b	1	0	0	1	0	0
c	1	1	0	0	1	0
c	1	1	0	0	1	0
b	1	1	0	1	0	1

Στον Πίνακα 7.10 φαίνονται οι πίνακες bit-επιπέδου ΔLN και c. Η τελευταία γραμμή του διανύσματος c έχει συνολικά 5 στοιχεία, υπάρχουν τέσσερις μονάδες και το μήκος της ακολουθίας LCS για δύο αλφαριθμητικά x και y είναι τέσσερα. Ο Πίνακας 7.11 δείχνει τον πίνακα bit-επιπέδου LCS για την ανάκτηση μιας ακολουθίας LCS δύο αλφαριθμητικών x και y. Από το στοιχείο $LCS_{5,6}$ του πίνακα LCS προκύπτει μια ακολουθία LCS που είναι 11101 και αντιστοιχεί στο αλφαριθμητικό LCS "abcb".

7.6 Επισκόπηση

Σε αυτή την ενότητα παρουσιάζουμε σύντομα διατάξεις επεξεργαστών για τα προβλήματα απλής ή προσεγγιστικής αναζήτησης αλφαριθμητικών και μακρύτερης κοινής υποακολουθίας.

7.6.1 Πρόβλημα Προσεγγιστικής Αναζήτησης Αλφαριθμητικών

Στη βιβλιογραφία έχουν εμφανιστεί πολλές διατάξεις επεξεργαστών και αρχιτεκτονικές για το πρόβλημα της απλής ή προσεγγιστικής αναζήτησης αλφαριθμητικών. Οι Foster και Kung [40] πρότειναν μια γραμμική συστολική διάταξη VLSI για το πρόβλημα της απλής αναζήτησης αλφαριθμητικών. Αυτή η

Πίνακας 7.11: Φάση υπολογισμού LCS

LCS	ε	b	c	a	b	c	b
ε	00000	00000	00000	00000	00000	00000	00000
a	00000	00000	00000	10000	10000	10000	10000
b	00000	01000	01000	01000	11000	11000	11000
c	00000	01000	01100	01100	01100	11100	11100
c	00000	01000	01100	01100	01100	11100	11100
b	00000	01000	01100	01100	01101	01101	11101

διάταξη επιτρέπει την ταυτόχρονη διασωλήνωση (pipelining) του κειμένου και του προτύπου αλφαριθμητικού διαμέσου της αμφίδρομης ροής δεδομένων (bi-directional data flow). Κατά τη διάρκεια κάθε ζεύγους διαδοχικών κύκλων χρόνου η διάταξη λαμβάνει δύο χαρακτήρες και επιστρέφει μια έξοδο ή ένα αποτέλεσμα. Όμως, ο απαιτούμενος χρόνος για να επεξεργαστεί ολόκληρο το αλφαριθμητικό κειμένου μήκους n είναι $2n$ βήματα και είναι διπλάσιος του μήκους του αλφαριθμητικού. Για να ελαχιστοποιήσουμε τον αριθμό των κελιών σε αριθμό χαρακτήρων του προτύπου, η διάταξη αυτή χρησιμοποίησε τη μέθοδο ανακύκλωσης του προτύπου.

Οι Cheng και Fu [25] πρότειναν μια αρχιτεκτονική VLSI που υπολογίζει την απόσταση διόρθωσης και επίσης την ακολουθία διόρθωσης S ανάμεσα σε δύο αλφαριθμητικά. Αυτή η αρχιτεκτονική είναι μια άμεση απεικόνιση του ακολουθιακού αλγορίθμου δυναμικού προγραμματισμού πάνω σε μια διδιάστατη συστολική διάταξη (two dimensional systolic array) και απαιτεί mn επεξεργαστές για να επεξεργαστεί αλφαριθμητικά μήκους m και n. Επίσης, η ακολουθία διόρθωσης προκύπτει μέσω ενός υλικού οπισθοδρόμησης και περισσότερες λεπτομέρειες για αυτό το υλικό υπάρχουν στην αρχική εργασία [25]. Συνεπώς, ολόκληρη η πράξη υπολογισμού της απόστασης διόρθωσης και τον προσδιορισμό της ακολουθίας διόρθωσης παίρνει $2(m + n + 1)$ χρονικές μονάδες. Ένα μειονέκτημα με την αρχιτεκτονική αυτή είναι ότι χρειάζεται $m+n$ εισόδους ώστε να επιτρέπεται παράλληλη επικοινωνία κατά την διάρκεια κάθε χρονικής στιγμής.

Στα [88, 90] παρουσιάζεται το P-NAC (Princeton Nucleic Acid Comparator) που χρησιμοποιεί γραμμική συστολική διάταξη για σύγκριση ακολουθιών DNA, η οποία είναι μια παραλληλοποίηση του αλγορίθμου δυναμικού προγραμματισμού. Σε αυτή τη γραμμική προσέγγιση, για να συγκριθούν δύο αλφαριθμητικά εισάγονται από τις αντίθετες πλευρές της διάταξης κελιών. Όμως, η αρχιτεκτονική αυτή επιδεικνύει πολύ περιορισμένη ευελιξία εξαιτίας του σχήματος κωδικοποίησης που χρησιμοποιεί-

ται. Συγκεριμένα, η αρχιτεκτονική αυτή περιορίζεται για την περίπτωση όπου τα κόστη διόρθωσης για αντικατάσταση, εισαγωγή και διαγραφή είναι σταθερά σε 2, 1 και 1 αντίστοιχα. Παρόμοια προσέγγιση ακολούθησαν οι εργασίες [143, 142] χρησιμοποιώντας ξανά έναν αλγόριθμο δυναμικού προγραμματισμού, αλλά πρότειναν βελτιωμένο σχήμα κωδικοποίησης και περιορισμένη επιβάρυνση επικοινωνίας και ελέγχου. Συνεπώς, αυτή η τελευταία αρχιτεκτονική μπορεί να εκτελέσει υπολογισμούς απόστασης διόρθωσης για μεταβλητά κόστη διόρθωσης και δεν επιβάλει κανένα περιορισμό για τα μήκη των αλφαριθμητικών που μπορεί να επεξεργαστεί. Οι δύο παραπάνω γραμμικές προσεγγίσεις απαιτούν $m+n-1$ στοιχεία επεξεργασίας (processing elements) ή κελιά (cells) για να επεξεργαστούν αλφαριθμητικά μήκους n και m. Μια ιδέα κατάτμησης (partition) περιγράφεται στην εργασία [142] για να χειριστεί περιπτώσεις όπου το μέγεθος του προβλήματος είναι μεγαλύτερο από το μέγεθος της διάταξης.

Ο Megson [99] πρότεινε δύο καινούργιες ευριστικές αναζήτησης αλφαριθμητικών οι οποίες μειώνουν τις απαιτήσεις υλικού και βελτιώνουν την απόδοση της συστολικής διάταξης αναζήτησης αλφαριθμητικών των Lipton και Lopresti [88, 90]. Γνωρίζουμε ότι η αμφίδρομη ροή δεδομένων επιβάλει χαμηλή χρήση του επεξεργαστή (περίπου το 50%) και αυξάνει τον χρόνο υπολογισμού. Συνεπώς, ο Megson βελτίωσε την πολυπλοκότητα χρόνου για τις αρχιτεκτονικές της αμφίδρομης ροής δεδομένων οι οποίες μειώνουν τον χρόνο υπολογισμού περίπου σε n χρονικές μονάδες με την κατάτμηση του προβλήματος σε δύο υποπροβλήματα τα οποία λύνονται ταυτόχρονα πάνω στην ίδια διάταξη. Η τεχνική αυτή κατάτμησης κάνει το κελί 100% αποτελεσματικό.

Στα [77, 51, 78] παρουσιάζεται το SAMBA (Systolic Accelerator for Molecular Biological Applications) που χρησιμοποιεί γραμμική συστολική διάταξη για να επιταχύνει τις συγκρίσεις βιολογικών ακολουθιών. Το SAMBA υλοποιεί μια παραμετροποιημένη έκδοση του αλγορίθμου Smith και Waterman [151]. Επίσης, ακολουθεί τη μονόδρομη ροή δεδομένων (unidirectional data flow) όπου η ακολουθία προτύπου αποθηκεύεται στη διάταξη (ένα χαρακτήρα ανά κελί) και η άλλη ακολουθία ρέει από αριστερά προς τα δεξιά διαμέσου της διάταξης. Μια παρόμοια διάταξη επεξεργαστών VLSI προτάθηκε στην εργασία [97] για ευέλικτη αναζήτηση αλφαριθμητικών, η οποία είναι μια παραλληλοποίηση του αλγορίθμου [7]. Όμως, αυτή η διάταξη επεξεργαστών υποστηρίζει μια περιορισμένη ποικιλία περιορισμένων εκφράσεων. Οι δύο παραπάνω γραμμικές προσεγγίσεις με μονόδρομη ροή δεδομένων πραγματοποιούν παρόμοια πολυπλοκότητα χώρου και χρόνου όπως στη βελτιωμένη διάταξη του [99] χωρίς επιπλέον πολυπλοκότητα δηλαδή χωρίς να χρησιμοποιούν τεχνικές κατάτμησης.

Οι Hughey και Lopresti πρότειναν μια προγραμματιζόμενη συστολική διάταξη, την B-SYS (Brown Systolic Array) που μπορεί να χρησιμοποιηθεί για να υπολογίζει την απόσταση διόρθωσης ανάμεσα σε δύο αλφαριθμητικά [60]. Επίσης, μια υλοποίηση σύγκρισης των γενετικών ακολουθιών χρησιμοποιώντας Field Programmable Gate Arrays (FPGAs) περιγράφεται στις εργασίες [55, 56, 57, 91]. Αυτές οι προγραμματιζόμενες αρχιτεκτονικές χρησιμοποιούν το ίδιο σχήμα κωδικοποίησης που προτάθηκε στις εργασίες [88, 90] και συνεπώς έχουν τους ίδιους περιορισμούς.

Πρόσφατα, οι ερευνητές [146] και [147] παρουσίαζαν υλοποιήσεις FPGAs για απλή αναζήτηση αλφαριθμητικών και αναζήτηση κανονικών εκφράσεων (regular expressions) αντίστοιχα.

7.6.2 Πρόβλημα Μακρύτερης Κοινής Υποακολουθίας

Διάφορες γραμμικές διατάξεις επεξεργαστών για τα προβλήματα LLCS και LCS έχουν προταθεί στις εργασίες [140, 170, 86, 82, 83, 87, 94]. Οι Yang και Lee [170] πρότειναν δύο γραμμικές συστολικές διατάξεις για το πρόβλημα LLCS. Μια από αυτές επιτρέπει τα δύο αλφαριθμητικά x και y να ρέουν από τις αντίθετες πλευρές της διάταξης κελιών. Η διάταξη αυτή υπολογίζει το μήκος μιας ακολουθίας LCS σε $2n + m - 1$ χρονικές μονάδες και ο αριθμός των κελιών που απαιτούνται είναι $2n$. Όμως, το κύριο μειονέκτημα είναι η χαμηλή χρήση των επεξεργαστών. Επίσης, οι Yang και Lee πρότειναν μια βελτιωμένη συστολική διάταξη που ακολουθεί τη μονόδρομη ροή δεδομένων όπου το ένα αλφαριθμητικό φορτώνεται στη διάταξη και το άλλο αλφαριθμητικό ρέει από αριστερά προς τα δεξιά διαμέσου της διάταξης. Η βελτιωμένη αυτή διάταξη απαιτεί $n + 2m - 1$ χρονικές μονάδες και ο αριθμός των κελιών που απαιτούνται μειώνεται σε m κελιά. Όμως, οι δύο παραπάνω γραμμικές προσεγγίσεις απαιτούν πολλούς καταχωρητές σε κάθε Στοιχείο Επεξεργασίας (ΣΕ - Processing Element) ή κελί και οι πράξεις του ΣΕ είναι πολύπλοκες.

Οι Robert και Tchuente [140] πρότειναν μια γραμμική μονόδρομη συστολική διάταξη των m κελιών για να υπολογίζει το μήκος μιας ακολουθίας LCS σε $n + 2m$ χρονικές μονάδες. Όμως, ο σχεδιασμός αυτός δεν ήταν ικανός να ανακτήσει μια ακολουθία LCS. Έτσι, για να ανακτήσει μια ακολουθία LCS, οι συγγραφείς επέκτειναν την προηγούμενη διάταξη σε μια διδιάστατη διάταξη των m ΣΕς ή κελιών, που κάθε κελί είναι εξοπλισμένο με μια συστολική στοίβα (systolic stack) για να αποθηκεύσει τις ταυτίσεις (δηλαδή, $x_i = y_j$) στις οποίες εμφανίζεται ενώ υπολογίζει το μήκος l. Κατόπιν λειτουργεί

μια δεύτερη φάση σε αντίστροφη σειρά για να ιχνηλατηθεί μια λύση που είναι έξοδος από το τελευταίο κελί μετά από $3m$ χρονικά βήματα. Συνεπώς, η διάταξη αυτή βρίσκει μια ακολουθία LCS σε $3m-2$ περισσότερες χρονικές μονάδες αλλά η αρχιτεκτονική στοίβας δεν είναι κατάλληλη για τεχνολογίες wafer-scale integration (WSI).

Ο Lin [86] παρουσίασε μια γραμμική συστολική διάταξη των m κελιών που απαιτεί λιγότερες θύρες (ports) Ε/Ε (Εισόδου/Εξόδου) από την διάταξη των Robert και Tchuente [140] για να υπολογίσει το μήκος μιας ακολουθίας LCS σε $n+2m-1$ χρονικές μονάδες. Για να ανακτήσουμε μια ακολουθία LCS, ο συστολικός σχεδιασμός του Lin είναι ίδιος όπως την συστολική διάταξη που παρουσιάστηκε στην εργασία [140] αλλά χρησιμοποιεί μια content-addressable memory (CAM) σε κάθε ΣΕ αντί για συστολικές στοίβες. Η παραπάνω αρχιτεκτονική παίρνει $2m-1$ χρονικές μονάδες οι οποίες είναι λιγότερες από την αρχιτεκτονική που παρουσιάστηκε στην εργασία [140].

Ο Lecroq και άλλοι [82, 83] παρουσίασαν μια μονόδρομη συστολική διάταξη των m κελιών για να λύσουν τα προβλήματα LLCS και LCS ταυτόχρονα σε $n+2m$ χρονικές μονάδες. Για να υπάρξει βελτίωση του χρόνου στην παραπάνω αρχιτεκτονική κάθε ΣΕ εξοπλίζεται με m καταχωρητές για να αποθηκεύει την ακολουθία LCS. Οι καταχωρητές αυτοί αντικαθιστούν τις συστολικές στοίβες ή τις content-addressable memories (CAMs) που χρησιμοποιήθηκαν στις εργασίες [140] και [86] αντίστοιχα, για την ανάκτηση μιας ακολουθίας LCS. Η αρχιτεκτονική που πρότειναν ο Lecroq και άλλοι είναι η πρώτη αρχιτεκτονική που υπολογίζει το μήκος μιας ακολουθίας LCS και την ανάκτηση της ακολουθίας LCS σε μόνο μια φάση ή πέρασμα αντί δύο φάσεις όπως στις εργασίες [140, 86].

Επίσης, οι Lin και Chen [87] ακολούθησαν τη συστολική λύση των [82] για τον υπολογισμό των προβλημάτων LLCS και LCS σε μια φάση αλλά εφάρμοσαν τη συστολική διάταξη του [86] για να υπολογίσουν το μήκος μιας ακολουθίας LCS.

Τέλος, οι Luce και Myoupo [94] πρότειναν μια μονόδρομη αρχιτεκτονική ημι-πλέγματος (semi-mesh architecture) των $\frac{m(m+1)}{2}$ κελιών για να ανακτήσουν μια ακολουθία LCS δύο αλφαριθμητικών σε $n+3m+l$ χρονικές μονάδες. Η αρχιτεκτονική αυτή χρησιμοποιεί λιγότερα κελιά και τρέχει ταχύτερα σε σχέση με την διδιάστατη διάταξη της [140]. Όλοι οι προηγούμενοι συστολικοί σχεδιασμοί βασίστηκαν στην παραλληλοποίηση του κλασσικού αλγορίθμου δυναμικού προγραμματισμού [54].

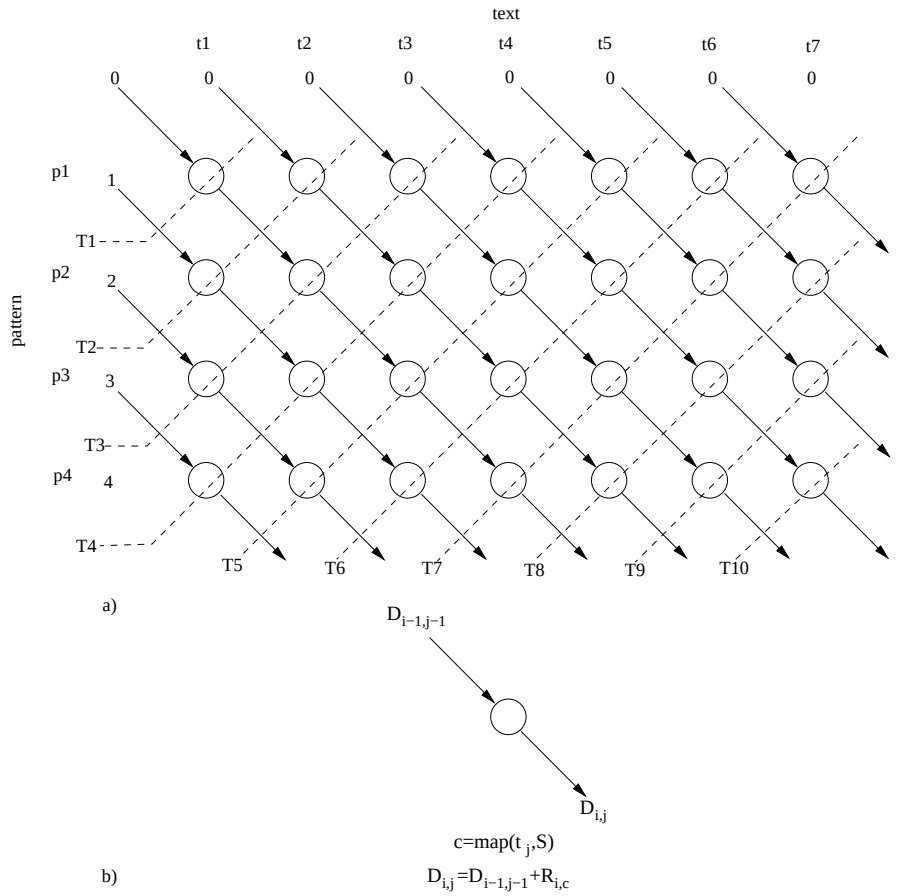
7.7 Γράφοι Εξάρτησης Δεδομένων

Σε αυτή την ενότητα εξάγουμε γράφους εξάρτησης δεδομένων (data dependence graphs) για τους αλγορίθμους δυναμικού προγραμματισμού, bit-παραλληλισμού και του αλγορίθμου LCS που παρουσιάστηκαν στις ενότητες 7.2, 7.3 και 7.5 αντίστοιχα. Ένας γράφος εξάρτησης δεδομένων είναι ένας κατευθυνόμενος γράφος (directed graph) που αντιστοιχεί τις εξαρτήσεις δεδομένων ενός αλγορίθμου. Αποτελείται από κόμβους και ακμές, όπου κάθε κόμβος αντιστοιχεί σε μια πράξη ή υπολογισμό και κάθε ακμή αντιστοιχεί σε μια εξάρτηση δεδομένων ανάμεσα τους υπολογισμούς.

7.7.1 Γράφοι Εξάρτησης για τους Αλγορίθμους Δυναμικού Προγραμματισμού

Τα Σχήματα 7.4, 7.5 και 7.6 δείχνουν τους γράφους εξάρτησης και τα διαγράμματα χρονισμού (parallel timing diagrams) για τους τρεις αλγορίθμους δυναμικού προγραμματισμού. Όλοι οι κόμβοι των γράφων οι οποίοι βρίσκονται στην ίδια γραμμή χρησιμοποιούν τον ίδιο χαρακτήρα του προτύπου αλφαριθμητικού p. Παρόμοια, όλοι οι κόμβοι οι οποίοι βρίσκονται στην ίδια στήλη χρησιμοποιούν τον ίδιο χαρακτήρα του κειμένου t. Για παράδειγμα, όλοι οι κόμβοι της πρώτης γραμμής των γράφων αποθηκεύουν τον πρώτο χαρακτήρα p_1 του προτύπου p. Παρόμοια, όλοι οι κόμβοι της πρώτης στήλης των γράφων φορτώνουν τον πρώτο χαρακτήρα t_1 του κειμένου t. Επίσης, στον κόμβο (i, j) των γράφων 7.4, 7.5 και 7.6 $(1 \leq i \leq m, 1 \leq j \leq n)$ ανατίθεται μια ολόκληρη γραμμή i του χάρτη μνήμης bit-επιπέδου R για τον χαρακτήρα p_i του προτύπου. Με άλλα λόγια, στους κόμβους της πρώτης γραμμής των γράφων ανατίθεται η πρώτη γραμμή του R που αντιστοιχεί στον πρώτο χαρακτήρα του προτύπου. Στους κόμβους της δεύτερης γραμμής των γράφων ανατίθεται η δεύτερη γραμμή του R που αντιστοιχεί στο δεύτερο χαρακτήρα του προτύπου και ούτω καθεξής.

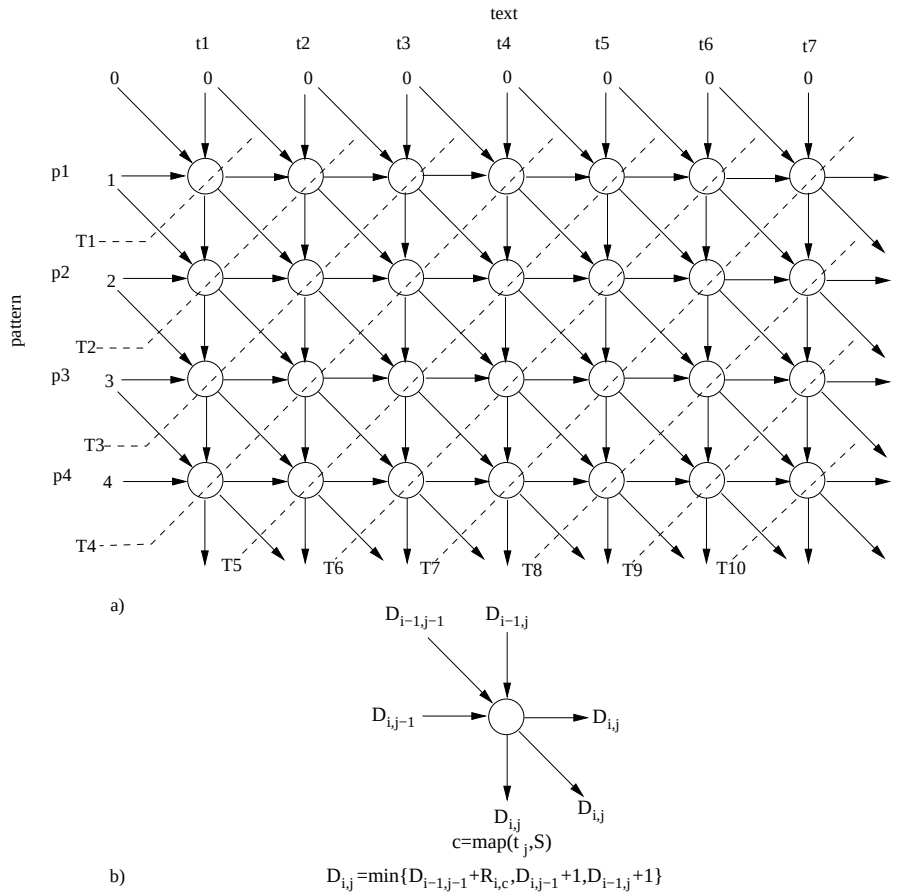
Στο Σχήμα 7.4α για να υπολογίσουμε ένα στοιχείο του πίνακα δυναμικού προγραμματισμού $D_{i,j}$ $(1 \leq i \leq m, 1 \leq j \leq n)$ χρειαζόμαστε το προηγούμενο υπολογισμένο διαγώνιο στοιχείο $D_{i-1,j-1}$ όπως φαίνεται στο Σχήμα 7.4β. Κάθε κόμβος είναι υπεύθυνος να υπολογίζει την μια γραμμή του πίνακα δυναμικού προγραμματισμού D. Ο γράφος εξάρτησης μας επιτρέπει να υπολογίζουμε τα στοιχεία που ανήκουν στην ίδια διαγώνιο (από αριστερά-κάτω προς τα δεξιά-πάνω) παράλληλα. Αυτό φαίνεται με τις διακεκομένες διαγώνιες γραμμές του γράφου 7.4α. Για παράδειγμα, στην χρονική στιγμή T4 υπολογίζονται τα στοιχεία $D_{4,1}, D_{3,2}, D_{2,3}$ και $D_{1,4}$ ταυτόχρονα.



Σχήμα 7.4: (α) Γράφος εξάρτησης και διάγραμμα χρονισμού για αναζήτηση αλφαριθμητικών με αποτυχίες (β) Υπολογισμός κόμβου

Στο Σχήμα 7.5α για να υπολογίσουμε ένα στοιχείο του πίνακα δυναμικού προγραμματισμού $D_{i,j}$ $(1 \leq i \leq m, 1 \leq j \leq n)$ χρειαζόμαστε τις τρεις προηγούμενες υπολογισμένες τιμές $D_{i-1,j}, D_{i,j-1}$ και $D_{i-1,j-1}$ όπως φαίνεται στο Σχήμα 7.5β. Παρόμοια με το γράφο του Σχήματος 7.4α, ο γράφος εξάρτησης του Σχήματος 7.5α μας επιτρέπει να υπολογίσουμε όλα τα m στοιχεία που ανήκουν στην ίδια διαγώνιο στην ίδια χρονική στιγμή. Αυτό φαίνεται με τις διακεκομένες διαγώνιες γραμμές του Σχήματος 7.5α.

Κάθε κόμβος του Σχήματος 7.6α υπολογίζει την οριζόντια διαφορά $\Delta h_{i,j}$ και την κάθετη διαφορά $\Delta v_{i,j}$ $(1 \leq i \leq m, 1 \leq j \leq n)$ των πινάκων δυναμικού προγραμματισμού Δh και Δv αντίστοιχα. Για να υπολογίσουμε αυτές τις διαφορές, κάθε κόμβος χρειάζεται να λάβει τις προηγούμενες υπολογισμένες

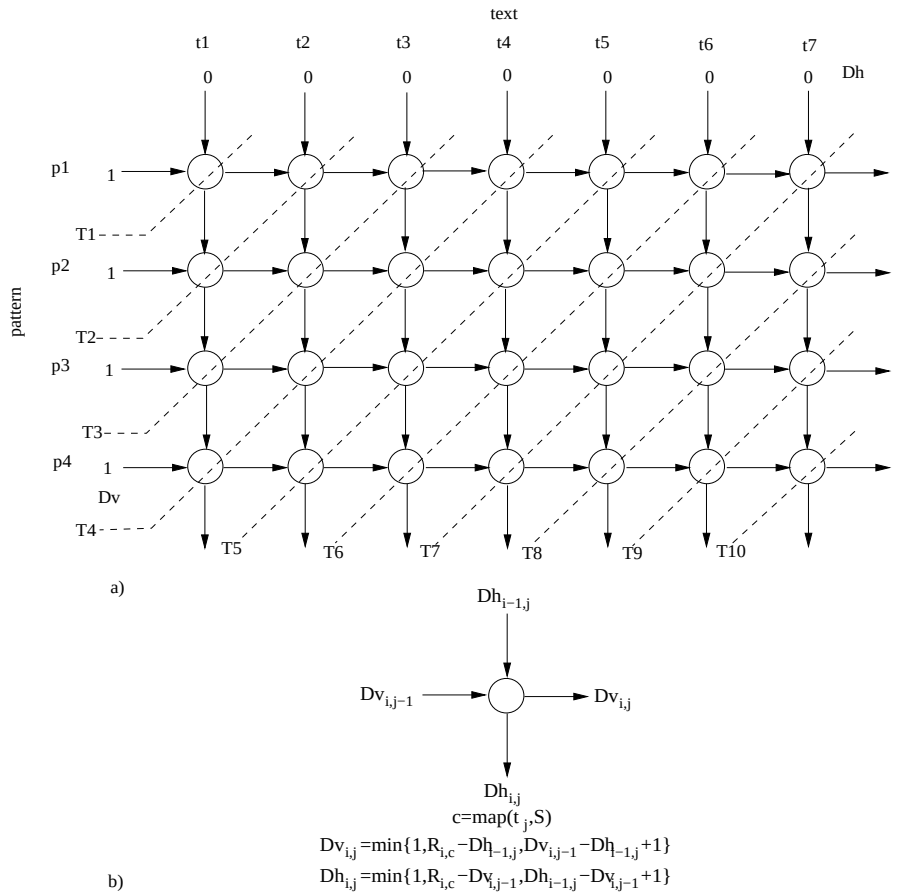


Σχήμα 7.5: (α) Γράφος εξάρτησης και διάγραμμα χρονισμού για αναζήτηση αλφαριθμητικών με διαφορές (β) Υπολογισμός κόμβου Min

διαφορές οριζόντιας και κάθετης, $\Delta h_{i-1,j}$ και $\Delta v_{i,j-1}$ αντίστοιχα, όπως φαίνεται στο Σχήμα 7.6β. Τέλος, οι υπολογισμοί κατά 45 μοίρες διαγώνια εκτελούνται ταυτόχρονα.

7.7.2 Γράφοι Εξάρτησης για τους Αλγορίθμους Bit-Παραλληλισμού

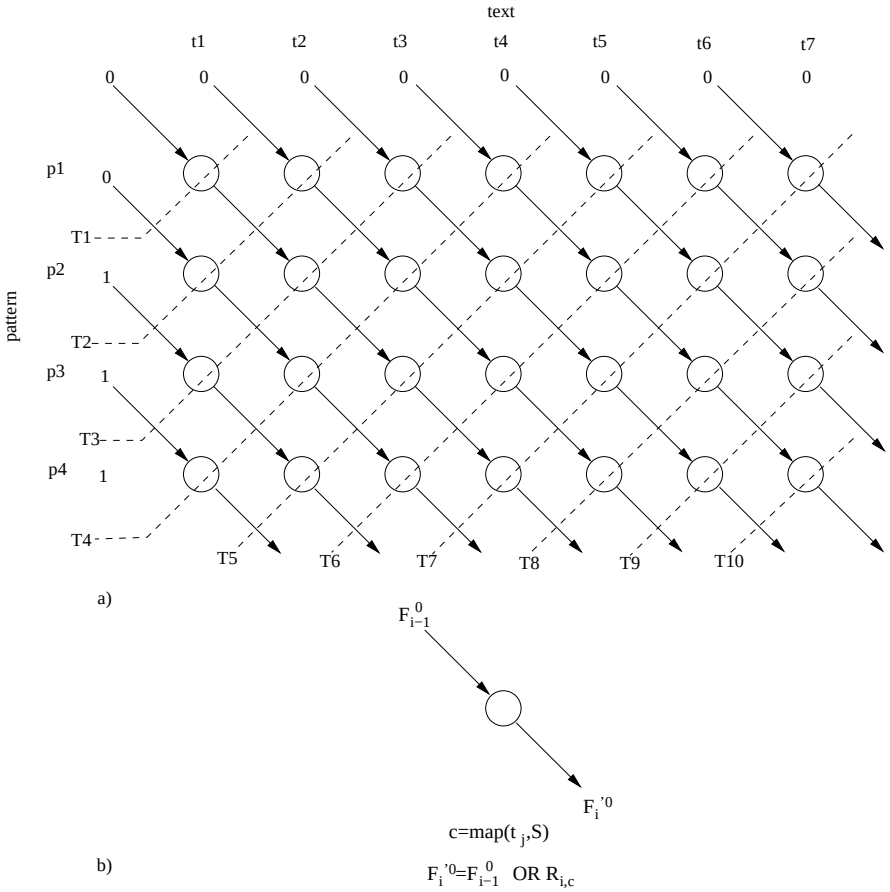
Τα Σχήματα 7.7, 7.8 και 7.9 δείχνουν τους γράφους εξάρτησης και τα διαγράμματα χρονισμού για τους αλγορίθμους αναζήτησης αλφαριθμητικών που προσομοιώνουν το αυτόματο NFA κατά γραμμές. Ο κόμβος (i, j) των γράφων 7.7, 7.8 και 7.9 $(1 \leq i \leq m, 1 \leq j \leq n)$ αποθηκεύει το χαρακτήρα p_i του προτύπου αλφαριθμητικού p και τον χαρακτήρα t_j του κειμένου t. Επίσης, σε κάθε κόμβο (i, j) των γράφων ανατίθεται μια γραμμή i του R για το χαρακτήρα προτύπου p_i.



Σχήμα 7.6: (α) Γράφος εξάρτησης και διάγραμμα χρονισμού για αναζήτηση αλφαριθμητικών με διαφορές βασισμένος στον αλγόριθμο του Myers (β) Υπολογισμοί κόμβου Min

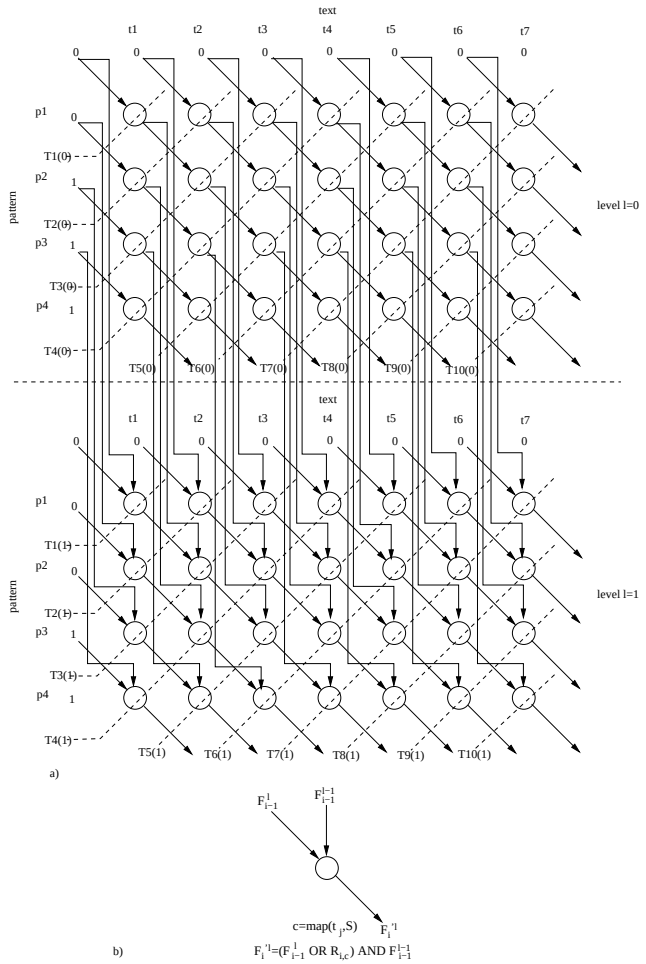
Στο Σχήμα 7.7α για να υπολογίσουμε ένα στοιχείο του διανύσματος $F_i'^0$ $(1 \leq i \leq m)$ μετά την ανάγνωση ενός καινούργιου χαρακτήρα κειμένου t_j $(1 \leq j \leq m)$, χρειαζόμαστε την προηγούμενη τιμή F_{i-1}^0 όπως φαίνεται στο Σχήμα 7.7β. Υποθέτουμε ότι κάθε κόμβος είναι υπεύθυνος για να υπολογίζει όλους τους κόμβους της κάθε γραμμής του γράφου 7.7α. Από το γράφο εξάρτησης συμπεραίνουμε ότι όλοι οι κόμβοι οι οποίοι βρίσκονται στην ίδια διαγώνιο (από αριστερά-κάτω προς δεξιά-πάνω) μπορούν να εκτελεστούν στην ίδια χρονική στιγμή όπως φαίνεται στο Σχήμα 7.7α με τις διακεκομένες διαγώνιες γραμμές. Τέλος, παρατηρούμε ότι το διάγραμμα χρόνου αυτού του γράφου είναι παρόμοιο με τα διάγραμματα χρόνου των Σχημάτων 7.4, 7.5 και 7.6.

Το Σχήμα 7.8 δείχνει το γράφο εξάρτησης για τα δύο επίπεδα, δηλαδή επιτρέπει την αναζήτηση



Σχήμα 7.7: (α) Γράφος εξάρτησης και διάγραμμα χρονισμού για απλή αναζήτηση αλφαριθμητικών (β) Υπολογισμός κόμβου

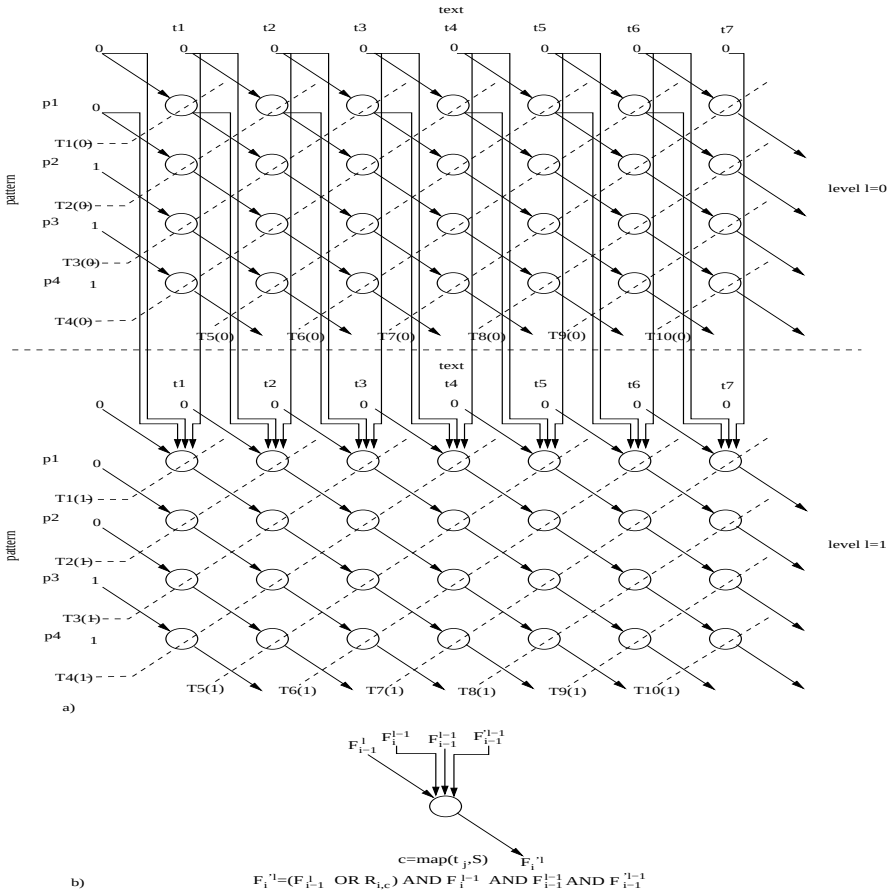
αλφαριθμητικών με 1 αποτυχία. Το επίπεδο $l = 0$ αντιστοιχεί στους υπολογισμούς του διανύσματος $F_i'^0$, ενώ το επίπεδο $l = 1$ αντιστοιχεί στους υπολογισμούς του διανύσματος $F_i'^1$. Οι υπολογισμοί και το διάγραμμα χρονισμού για το επίπεδο $l = 0$ είναι παρόμοια με το Σχήμα 7.7. Στο επίπεδο $l = 1$ του Σχήματος 7.8α, για να υπολογίσουμε ένα στοιχείο του διανύσματος $F_i'^1$ $(1 \leq i \leq m)$ μετά την ανάγνωση ενός χαρακτήρα κειμένου t_j $(1 \leq j \leq n)$, πρέπει να λάβουμε δύο προηγούμενες υπολογισμένες τιμές F^l_{i-1} και F^{l-1}_{i-1} όπως φαίνεται στο Σχήμα 7.8β. Το Σχήμα 7.8α δείχνει το διάγραμμα χρονισμού, όπου οι υπολογισμοί κατά 45 μοίρες διαγώνια εκτελούνται ταυτόχρονα. Τέλος, παρατηρούμε από τα διαγράμματα χρονισμού ότι οι κόμβοι οι οποίοι βρίσκονται στις ίδιες διακεκομένες διαγώνιες γραμμές για τα δύο επίπεδα μπορούν να εκτελεστούν παράλληλα. Για παράδειγμα, οι χρονικές στιγμές T4(0) και T4(1) για τα επίπεδα 0 και 1 αντίστοιχα μπορούν να εκτελεστούν ταυτόχρονα. Πρέπει να σημειώ-



Σχήμα 7.8: (α) Γράφος εξάρτησης και διάγραμμα χρονισμού για αναζήτηση αλφαριθμητικών με αποτυχίες (β) Υπολογισμός κόμβου

σουμε εδώ ότι αν έχουμε αναζήτηση αλφαριθμητικών με 3 αποτυχίες τότε ο γράφος εξάρτησης θα έχει τέσσερα επίπεδα δηλαδή $k+1$. Σε αυτή την περίπτωση οι συνδέσεις του κάθε επιπέδου με τον προηγούμενο επίπεδο θα είναι παρόμοιες με αυτές του Σχήματος 7.8. Έτσι, λόγω της πολυπλοκότητας του γράφου εξάρτησης με τέσσερα επίπεδα παρουσιάσαμε ένα γράφο εξάρτησης με μόνο δύο επίπεδα όπως φαίνεται στο Σχήμα 7.8.

Το Σχήμα 7.9 δείχνει το γράφο εξάρτησης για τα δύο επίπεδα, δηλαδή επιτρέπει την αναζήτηση αλφαριθμητικών με 1 διαφορά. Οι υπολογισμοί και το διάγραμμα χρονισμού για το επίπεδο $l = 0$ είναι όμοια με το Σχήμα 7.7. Κάθε κόμβος του Σχήματος 7.9α για το επίπεδο $l = 1$ υπολογίζει ένα



Σχήμα 7.9: (α) Γράφος εξάρτησης και διάγραμμα χρονισμού για αναζήτηση αλφαριθμητικών με διαφορές (β) Υπολογισμοί κόμβου

στοιχείο του διανύσματος $F_i'^1$ $(1 \leq i \leq m)$ μετά την ανάγνωση ενός καινούργιου χαρακτήρα κειμένου t_j $(1 \leq j \leq n)$. Προκειμένου να υπολογίσουμε αυτό το στοιχείο, κάθε κόμβος πρέπει να λάβει τέσσερα προηγούμενα υπολογισμένα στοιχεία $F_{i-1}^l, F_i^{l-1}, F_{i-1}^{l-1}$ και $F_{i-1}'^{l-1}$ όπως φαίνεται στο Σχήμα 7.9β. Από το γράφο εξάρτησης προκύπτει ότι οι κατά 45 μοίρες διαγώνιοι κόμβοι εκτελούνται παράλληλα. Λόγω της πολυπλοκότητας του Σχήματος 7.9β παραλείπουμε όλες τις συνδέσεις των υπολοίπων κόμβων του επίπεδου $l = 1$ αφού οι συνδέσεις είναι παρόμοιες με την πρώτη γραμμή του επιπέδου 1. Τέλος, από το γράφο εξάρτησης του Σχήματος 7.9 παρατηρούμε ότι οι χρονικές στιγμές T4(0) και T4(1), για παράδειγμα, μπορούν να υπολογιστούν παράλληλα. Επίσης, παρουσιάζουμε και εδώ γράφους εξάρτησης μόνο με δύο επίπεδα αντί για πολλά επίπεδα λόγω του μεγάλου σχήματος.

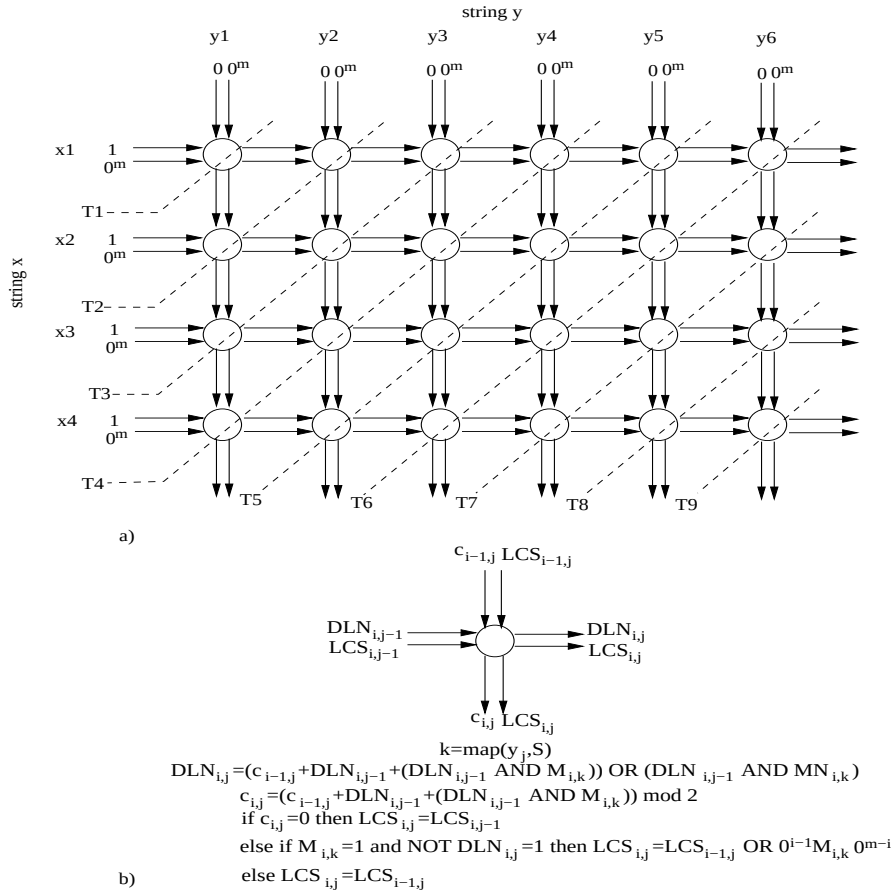
Για τους αλγορίθμους που προσομοιώνουν το αυτόματο NFA κατά στήλες δεν εξάγουμε τους γράφους εξάρτησης επειδή οι γράφοι αυτοί είναι παρόμοιοι με τους γράφους των αλγορίθμων δυναμικού προγραμματισμού.

7.7.3 Γράφος Εξάρτησης για τον Αλγόριθμο LCS

Το Σχήμα 7.10 δείχνει το γράφο εξάρτησης και το διάγραμμα χρονισμού για τον αλγόριθμο LCS. Σε κάθε κόμβο (i, j) $(1 \leq i \leq m, 1 \leq j \leq n)$ του γράφου αποθηκεύεται ο χαρακτήρας x_i του αλφαριθμητικού x και ο χαρακτήρας y_j του αλφαριθμητικού y. Επίσης, στον ίδιο κόμβο ανατίθεται μια ολόκληρη γραμμή i των χαρτών μνήμης bit-επιπέδου M και MN για τον χαρακτήρα x_i του αλφαριθμητικού x. Για να υπολογίσουμε ένα στοιχείο του bit-διανύσματος $c_{i,j}$ $(1 \leq i \leq m, 1 \leq j \leq n)$ και τα στοιχεία των πινάκων δυναμικού προγραμματισμού $\Delta LN_{i,j}$ και $LCS_{i,j}$ $(1 \leq i \leq m, 1 \leq j \leq n)$, χρειαζόμαστε να λάβουμε σαν εισόδους τις τέσσερις προηγουμένες υπολογισμένες τιμές $c_{i-1,j}, \Delta LN_{i,j-1}, LCS_{i-1,j}$ και $LCS_{i,j-1}$ όπως φαίνεται στο Σχήμα 7.10β. Επίσης, υποθέτουμε ότι κάθε κόμβος είναι υπεύθυνος για να υπολογίζει μια γραμμή των πινάκων δυναμικού προγραμματισμού $\Delta LN_{i,j}, c_{i,j}$ και $LCS_{i,j}$ ή του γράφου. Στον ίδιο γράφο του Σχήματος 7.10α παρουσιάζεται το διάγραμμα χρονισμού όπου οι κόμβοι οι οποίοι βρίσκονται στην ίδια διαγώνιο (από αριστερά-κάτω προς δεξιά-πάνω) μπορούν να υπολογιστούν ταυτόχρονα. Αυτό φαίνεται με τις διακεκομμένες γραμμές του Σχήματος 7.10α.

7.8 Απεικόνιση Αλγορίθμων Αναζήτησης Αλφαριθμητικών σε Διατάξεις Επεξεργαστών

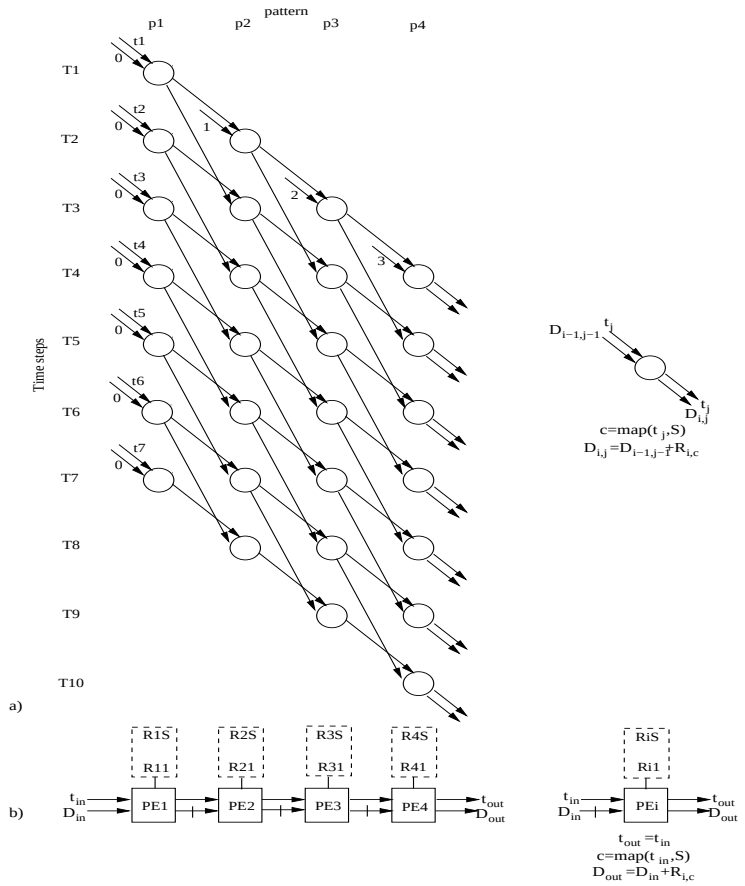
Αρχικά, περιγράφονται οι υλοποιήσεις της φάσης αναζήτησης των αλγορίθμων δυναμικού προγραμματισμού, bit-παραλληλισμού και LCS σε διάταξη επεξεργαστών, οι οποίες έχουν απαιτητικό υπολογιστικό φορτίο αφού $m << n$.



Σχήμα 7.10: (α) Γράφος εξάρτησης και διάγραμμα χρονισμού για LCS (β) Υπολογισμοί κόμβου

7.8.1 Διατάξεις Επεξεργαστών για τους Αλγορίθμους Δυναμικού Προγραμματισμού

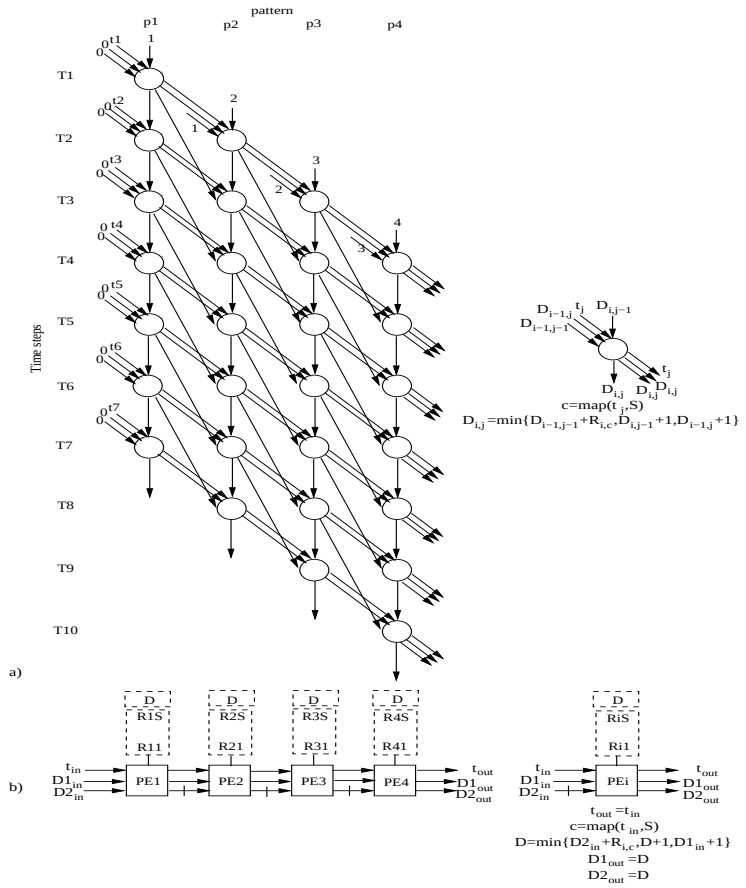
Πρώτα, μετασχηματίζουμε τους αρχικούς γράφους εξάρτησης δεδομένων των Σχημάτων 7.4α, 7.5α και 7.6α σε τοπικούς γράφους εξάρτησης όπως φαίνονται στα Σχήματα 7.11α, 7.12α και 7.13α έτσι ώστε οι χαρακτήρες του κειμένου να ρέουν μέσα από τις τοπικές ακμές, ενώ ο κάθετος και οριζόντιος άξονας των γράφων παριστάνουν τα χρονικά βήματα και τους χαρακτήρες του προτύπου αντίστοιχα. Ως πολυπλοκότητα χώρου των αλγορίθμων ορίζεται ο αριθμός των κελιών (ή Στοιχείων Επεξεργασίας, ΣΕ) που είναι ενεργά σε οποιοδήποτε χρονικό βήμα. Αυτή η πληροφορία δίνεται από τον αριθμό των στηλών του τοπικού γράφου εξάρτησης ενώ τα χρονικά βήματα που απαιτούνται δίνονται από τον αριθμό των γραμμών του γράφου.



Σχήμα 7.11: (α) Μετασχηματισμένος Γράφος (β) Διάταξη Επεξεργαστών

Οι διάταξεις επεξεργαστών σχηματίζονται από την προβολή του κάθετου άξονα του γράφου εξάρτησης των Σχημάτων 7.11α, 7.12α και 7.13α σε μια γραμμή επεξεργαστών. Στα Σχήματα 7.11β, 7.12β και 7.13β φαίνεται η προβολή αυτή για ένα γενικό πρόβλημα με $m = 4$ και για ανεξάρτητες τιμές του n και $|\Sigma|$.

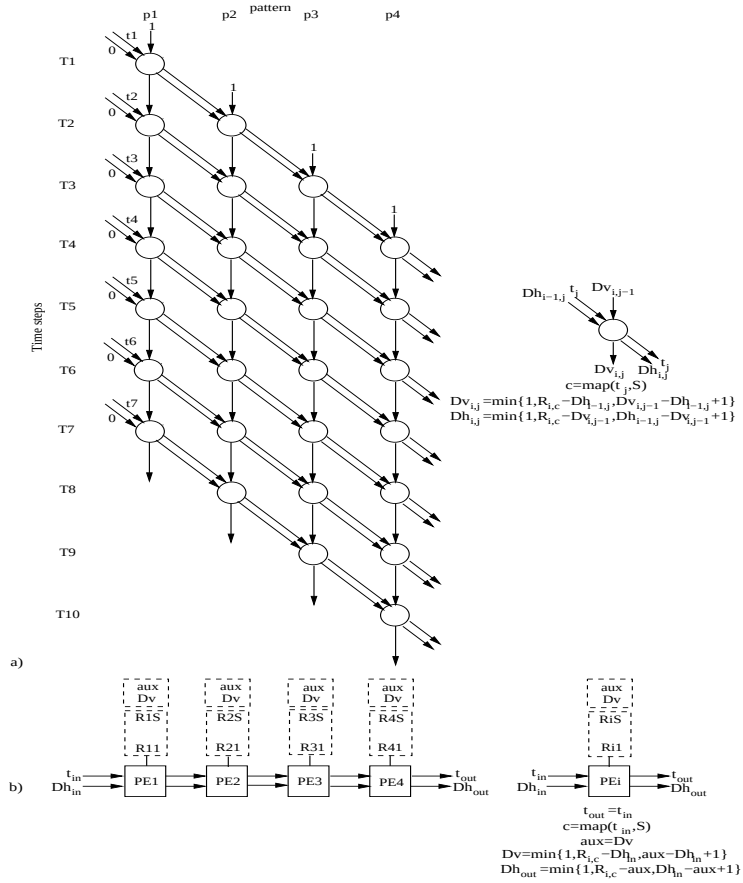
Στο Σχήμα 7.11β φαίνεται μια γραμμική διάταξη επεξεργαστών που αποτελείται από m κελιά που συνδέονται μεταξύ τους διαμέσου δύο καναλιών επικοινωνίας, όπου το ένα κανάλι μεταφέρει την δυαδική αναπαράσταση των χαρακτήρων κειμένου και το άλλο κανάλι μεταφέρει τα δέλτα αποτελέσματα byte-επιπέδου. Κάθε γραμμή R_i, $1 \leq i \leq m$ του χάρτη μνήμης bit-επιπέδου R διανέμεται σε ένα ΣΕ, έτσι ώστε η ανάγνωση μέσω της ειδικής συνάρτησης διευθυνσιοδότησης $map(ch, \Sigma)$ να παράγει ένα bit ανά ΣΕ. Για την υλοποίηση της συνάρτησης $map(ch, \Sigma)$ εισάγουμε ένα προγραμματιζόμενο υλικό



Σχήμα 7.12: (α) Μετασχηματισμένος Γράφος (β) Διάταξη Επεξεργαστών

όπως ένας αποκωδικοποιητής σε κάθε κελί. Η πράξη αυτή υλοποιείται σε χρόνο $2m$. Κάθε κελί εκτελεί ένα πλήρες βήμα υπολογισμού όπως φαίνεται στο Σχήμα 7.11β δεξιά, δηλαδή τις πράξεις απεικόνισης και πρόσθεσης. Ένα χρονικό βήμα ολοκληρώνεται με την επικοινωνία μεταξύ των γειτονικών ΣΕ έτσι ώστε τα μερικά αποτελέσματα διοχετεύονται προς το τελευταίο κελί. Σημειώνεται ότι τα αποτελέσματα δρομολογούνται προς την ίδια κατεύθυνση όπως οι χαρακτήρες κειμένου αλλά σε μισή ταχύτητα, που επιτυγχάνεται με την ενδιάμεση καθυστέρηση ανάμεσα στα κελιά.

Στο Σχήμα 7.12β φαίνεται μια γραμμική διάταξη των m κελιών που συνδέονται μεταξύ τους διαμέσου τριών καναλιών επικοινωνίας, το πρώτο κανάλι μεταφέρει την δυαδική αναπαράσταση των χαρακτήρων κειμένου και τα άλλα δύο κανάλια μεταφέρουν αποτελέσματα byte-επιπέδου. Κάθε κελί της διάταξης διανέμεται όπως και στην προηγούμενη διάταξη σε μια γραμμή του R και έναν καταχωρητή D ο οποίος



Σχήμα 7.13: (α) Μετασχηματισμένος Γράφος (β) Διάταξη Επεξεργαστών

αποθηκεύει την τιμή $D_{i,j-1}$. Επίσης, κάθε κελί εκτελεί πράξεις απεικόνισης, ελαχίστου και ανάθεσης όπως φαίνεται στο Σχήμα 7.12β δεξιά. Αφού κάθε κελί ενημερώσει την τιμή D, αποθηκεύεται στον καταχωρητή D και αποστέλεται προς στο επόμενο κελί μέσω των δύο καναλιών επικοινωνίας που μεταφέρουν τα αποτελέσματα. Σημειώνεται ότι τα αποτελέσματα του τρίτου καναλιού επικοινωνίας ρέουν από αριστερά προς τα δεξιά με το ίδιο τρόπο όπως τα πρώτα δύο κανάλια αλλά σε μισή ταχύτητα, που επιτυγχάνεται με την ενδιάμεση καθυστέρηση ανάμεσα στα κελιά.

Στο Σχήμα 7.13β φαίνεται μια γραμμική διάταξη των m κελιών που συνδέονται μεταξύ τους διαμέσου δύο καναλιών επικοινωνίας, το ένα μεταφέρει την δυαδική αναπαράσταση των χαρακτήρων κειμένου και το άλλο μεταφέρει τα οριζόντια δέλτα αποτελέσματα bit-επιπέδου. Κάθε κελί της διάταξης διανέμεται σε μια γραμμή του R και δύο καταχωρητές, Δv και aux, οι οποίες αποθηκεύουν την τρέχουσα κάθετη

διαφορά $\Delta v_{i,j}$ και την προηγούμενη διαφορά $\Delta v_{i,j-1}$ μετά την επεξεργασία του t_j αντίστοιχα. Επίσης, κάθε κελί εκτελεί ένα πλήρες βήμα υπολογισμού όπως φαίνεται στο Σχήμα 7.13β δεξιά, δηλαδή πράξεις απεικόνισης, ελαχίστου και ανάθεσης. Συνεπώς, κάθε κελί υπολογίζει δυο τιμές διαφοράς. Η μία κάθετη διαφορά ενημερώνεται και αποθηκεύεται στον καταχωρητή Δv, ενώ η άλλη οριζόντια διαφορά επίσης ενημερώνεται και διοχετεύεται στο διπλανό κελί. Οι χαρακτήρες κειμένου ρέουν από αριστερά προς στα δεξιά με το ίδιο τρόπο όπως στα μερικά αποτελέσματα, χωρίς ενδιάμεση καθυστέρηση ανάμεσα στα κελιά σε σχέση με τις προηγούμενες διατάξεις επεξεργαστών.

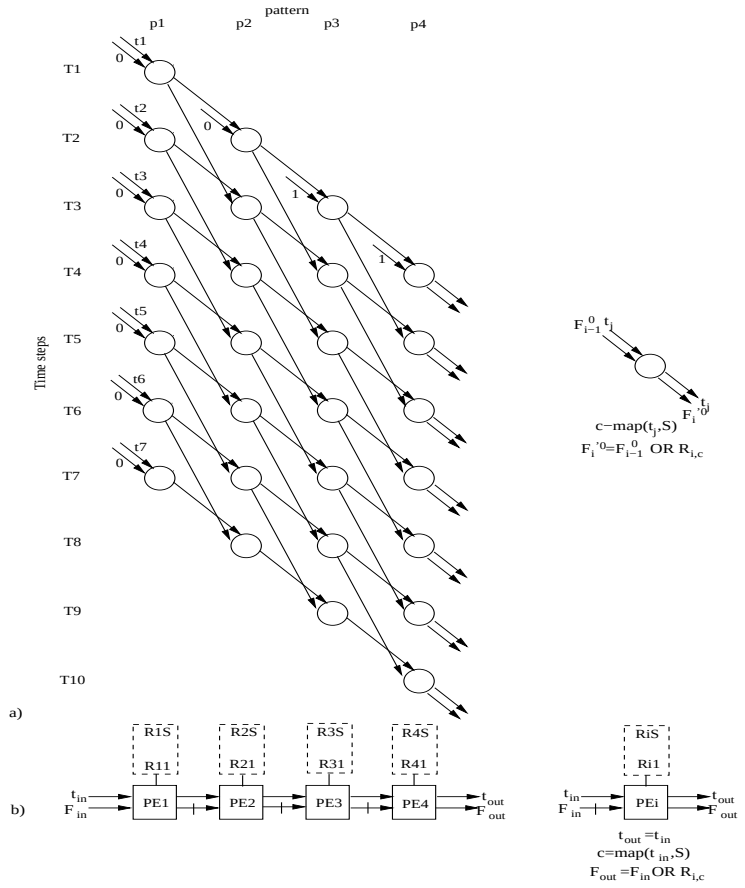
Ο συνολικός χρόνος υπολογισμού για τις τρεις παραπάνω γραμμικές προσεγγίσεις εκτελείται σε $n + m - 1$ βήματα χρόνου. Μπορούμε να συμφωνήσουμε ότι ο χρόνος υπολογισμού είναι n βήματα περίπου δεδομένου του γεγονότος ότι $m << n$. Ο χώρος που απαιτείται είναι m ΣΕ για τις τρεις διατάξεις επεξεργαστών. Λαμβάνοντας υπόψη το παράδειγμα του κώδικα χαρακτήρων UNICODE, οι απαιτήσεις τοπικής μνήμης είναι 16 Kbytes ανά κελί.

7.8.2 Διατάξεις Επεξεργαστών για τους Αλγορίθμους Bit-παραλληλισμού

Παρόμοια, μετασχηματίζουμε τους αρχικούς γράφους εξάρτησης των Σχημάτων 7.7α, 7.8α και 7.9α σε τοπικούς γράφους εξάρτησης όπως φαίνονται στα Σχήματα 7.14α, 7.15α και 7.16α με τον ίδιο τρόπο που επεξεργαστήκαμε τους γράφους για τους αλγορίθμους δυναμικού προγραμματισμού. Σημειώνουμε ότι οι γράφοι των Σχημάτων 7.15α και 7.16α αντιστοιχούν στο επίπεδο $l = 1$ και οι γράφοι που αντιστοιχούν στο επίπεδο $l = 0$ δεν παρουσιάζονται επειδή είναι παρόμοιοι με το γράφο του Σχήματος 7.14α. Οι διάταξεις επεξεργαστών σχηματίζονται από την προβολή του κάθετου άξονα του γράφου των Σχημάτων 7.14α, 7.15α και 7.16α και φαίνονται στα Σχήματα 7.14β, 7.15β και 7.16β για ένα γενικό πρόβλημα με $m = 4, k = 1$ και για ανεξάρτητες τιμές του n και $|\Sigma|$.

Στο Σχήμα 7.14β φαίνεται μια γραμμική διάταξη επεξεργαστών η οποία είναι παρόμοια με τη διάταξη του Σχήματος 7.11β. Όμως, το δεύτερο κανάλι επικοινωνίας μεταφέρει αποτελέσματα bit-επιπέδου αντί αποτελέσματα byte-επιπέδου. Επίσης, κάθε κελί εκτελεί μια λογική πράξη αντί την πράξη της πρόσθεσης όπως φαίνεται στο Σχήμα 7.14β δεξιά, δηλαδή πράξεις απεικόνισης και OR. Συνεπώς, κάθε κελί ενημερώνει το αποτέλεσμα και το διοχετεύει στο διπλανό κελί.

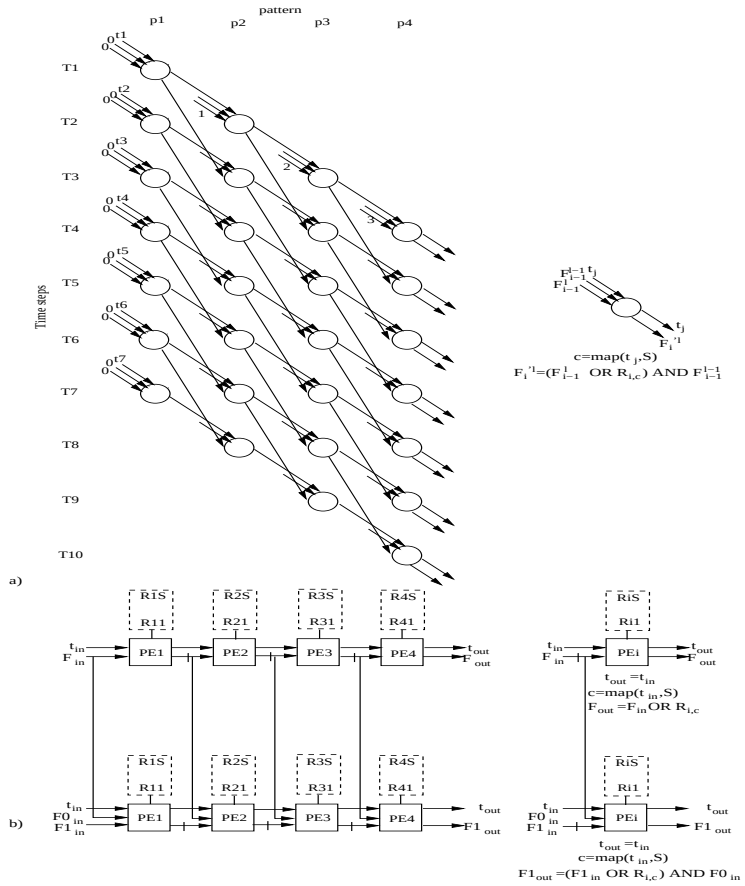
Στο Σχήμα 7.15β παρουσιάζονται $k + 1$ ή δύο γραμμικές διατάξεις που αποτελούνται από m κελιά.



Σχήμα 7.14: (α) Μετασχηματισμένος Γράφος (β) Διάταξη Επεξεργαστών

Η πρώτη διάταξη αντιστοιχεί στο επίπεδο $l = 0$ και είναι παρόμοια με την διάταξη του Σχήματος 7.14β. Η πρώτη και η δεύτερη διάταξη συνδέονται μεταξύ τους διαμέσου $k + 2$ καναλιών επικοινωνίας, όπου το πρώτο κανάλι μεταφέρει τη δυαδική αναπαράσταση των χαρακτήρων κειμένου και τα υπόλοιπα $k+1$ κανάλια μεταφέρουν αποτελέσματα bit-επιπέδου. Κάθε γραμμή του χάρτη μνήμης bit-επιπέδου R διανέμεται σε ένα ΣΕ. Επίσης, κάθε κελί εκτελεί ένα πλήρες βήμα υπολογισμού όπως φαίνεται στο Σχήμα 7.15β δεξιά, δηλαδή, τις πράξεις απεικόνισης και OR/AND. Το τελευταίο κανάλι επικοινωνίας μεταφέρει τα αποτελέσματα για το επίπεδο $l = 1$ από αριστερά προς τα δεξιά με τον ίδιο τρόπο όπως τα πρώτα δύο κανάλια αλλά σε μισή ταχύτητα.

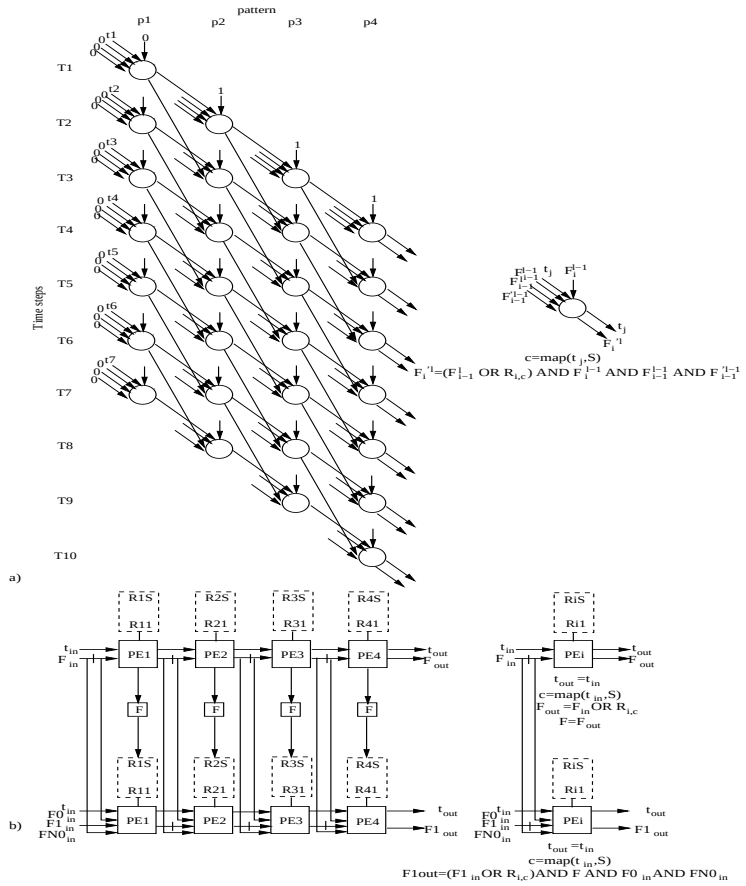
Στο Σχήμα 7.16β παρουσιάζονται $k + 1$ ή δύο γραμμικές διατάξεις που αποτελούνται από m κελιά. Η πρώτη διάταξη αντιστοιχεί στο επίπεδο $l = 0$ και είναι παρόμοια με την διάταξη του Σχήματος 7.14β.



Σχήμα 7.15: (α) Μετασχηματισμένος Γράφος (β) Διάταξη Επεξεργαστών

Πρέπει να σημειώσουμε ότι χρησιμοποιούμε έναν επιπλέον καταχωρητή F σε κάθε κελί της πρώτης διάταξης ο οποίος αποθηκεύει την τρέχουσα τιμή F όπως φαίνεται στο Σχήμα 7.14. Το περιεχόμενο του καταχωρητή F μπορεί να χρησιμοποιηθεί σαν είσοδος για το αντίστοιχο κελί της δεύτερης διάταξης. Η δεύτερη διάταξη συνδέεται με τα γειτονικά κελιά διαμέσου $k+3$ καναλιών επικοινωνίας (για $k=1$ έχουμε 4 κανάλια). Το πρώτο κανάλι μεταφέρει τη δυαδική αναπαράσταση των χαρακτήρων κειμένου και τα υπόλοιπα $k+2$ κανάλια μεταφέρουν αποτελέσματα bit-επιπέδου. Κάθε γραμμή του R διανέμεται σε ένα ΣΕ όπως στις προηγούμενες διατάξεις. Επίσης, κάθε κελί εκτελεί υπολογισμούς όπως φαίνονται στο Σχήμα 7.16β δεξιά.

Ο συνολικός χρόνος υπολογισμού για τις τρεις προηγούμενες υλοποιήσεις είναι $n+m-1$ βήματα χρόνου. Επίσης, ο χώρος που απαιτείται για τη διάταξη του Σχήματος 7.14β είναι m ΣΕ ενώ ο χώρος



Σχήμα 7.16: (α) Μετασχηματισμένος Γράφος (β) Διάταξη Επεξεργαστών

που απαιτείται για τις διατάξεις των Σχημάτων 7.15β και 7.16β είναι $(k+1)m$ ΣΕ.

7.8.3 Διάταξη Επεξεργαστών για τον Αλγόριθμο LCS

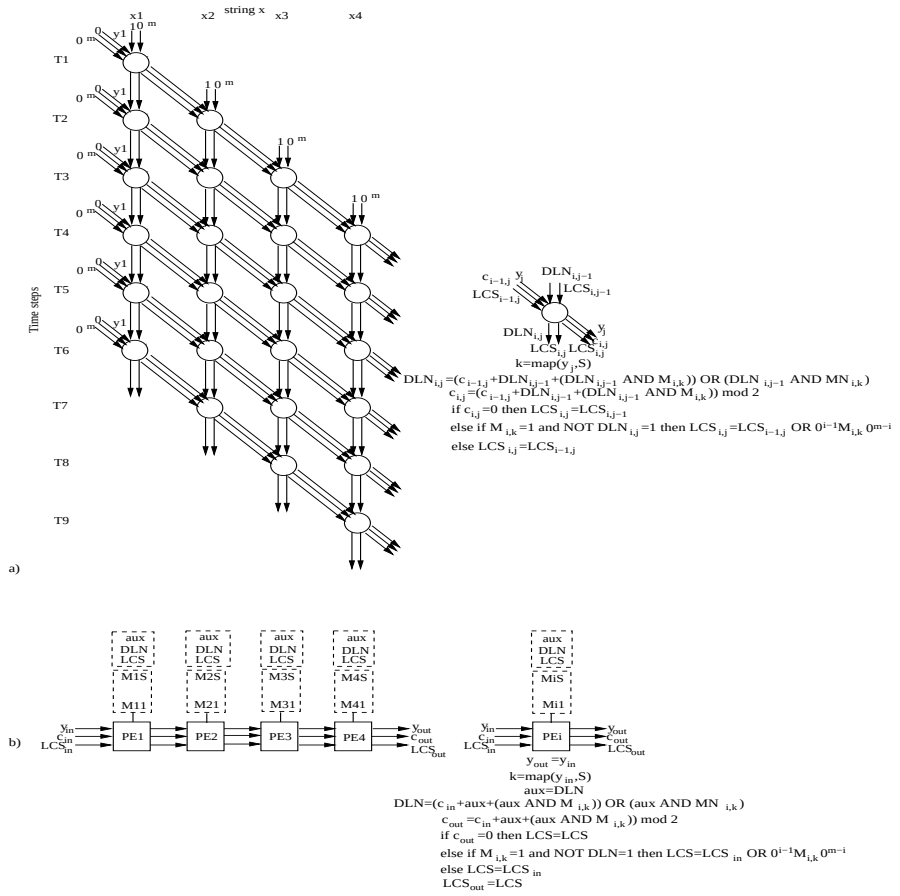
Για να μετασχηματίσουμε τον αρχικό γράφο του Σχήματος 7.10α για τον αλγόριθμο LCS ακολουθούμε παρόμοια διαδικασία όπως είδαμε και στους προηγούμενους γράφους. Ο τοπικός γράφος εξάρτησης μετά τον μετασχηματισμό φαίνεται στο Σχήμα 7.17α. Μια γραμμική διάταξη επεξεργαστών σχηματίζεται από την προβολή του κάθετου άξονα του γράφου εξάρτησης του Σχήματος 7.17α και φαίνεται στο Σχήμα 7.17β για ένα γενικό πρόβλημα με $m = 4$ και για ανεξάρτητες τιμές του n και $|\Sigma|$. Η παραπάνω γραμμική διάταξη αποτελείται από m κελιά που συνδέονται μεταξύ τους διαμέσου τριών καναλιών επικοινωνίας, όπου το πρώτο κανάλι μεταφέρει τη δυαδική αναπαράσταση των χαρακτήρων του

αλφαριθμητικού y και τα άλλα δύο κανάλια μεταφέρουν αποτελέσματα όπως το διάνυσμα bit-επιπέδου c και LCS. Κάθε γραμμή M_i, $1 \leq i \leq m$ του χάρτη μνήμης bit-επιπέδου M αποθηκεύεται σε κάθε ΣΕ της γραμμικής διάταξης έτσι ώστε χρησιμοποιώντας την ειδική συνάρτηση διευθυνσιοδότησης $map(ch, \Sigma)$ να παράγεται ένα μοναδικό bit ανά ΣΕ. Για την υλοποίηση της συνάρτησης $map(ch, \Sigma)$ χρησιμοποιούμε προγραμματιζόμενο υλικό τύπου αποκωδικοποιητή σε κάθε κελί. Επίσης, σε κάθε κελί απαιτούνται τρεις καταχωρητές $\Delta LN, aux$ και LCS, που αποθηκεύουν την τρέχουσα τιμή $\Delta LN_{i,j}$, την προηγούμενη τιμή $\Delta LN_{i,j-1}$ και τον προηγούμενο δυαδικό αλφαριθμητικό $LCS_{i,j-1}$ μετά την επεξεργασία του y_j αντίστοιχα. Κάθε ΣΕ εκτελεί ένα πλήρες βήμα υπολογισμού LCS όπως φαίνεται στο Σχήμα 7.17β δεξιά. Συνεπώς, κάθε ΣΕ υπολογίζει νέες τιμές $\Delta LN_{i,j}$ και $LCS_{i,j}$ που αποθηκεύονται στους καταχωρητές ΔLN και LCS αντίστοιχα, ενώ το αποτέλεσμα $c_{i,j}$ ενημερώνεται και διοχετεύεται προς στο γειτονικό κελί. Η τιμή του καταχωρητή LCS επίσης διοχετεύεται προς στο διπλανό κελί. Οι χαρακτήρες του αλφαριθμητικού y μεταδίδονται από αριστερά προς τα δεξιά με τον ίδιο τρόπο όπως τα αποτελέσματα bit-επιπέδου, χωρίς ενδιάμεση καθυστέρηση ανάμεσα στα κελιά.

Ο συνολικός χρόνος υπολογισμού είναι $n+m-1$ βήματα χρόνου, ενώ ο χώρος που απαιτείται είναι m ΣΕ.

7.9 Υλοποίηση της Φάσης Προεπεξεργασίας

Στην ενότητα αυτή παρουσιάζεται μια κοινή υλοποίηση της φάσης προεπεξεργασίας για τα δύο προβλήματα, δηλαδή, την κατασκευή του χάρτη μνήμης bit-επιπέδου R για το πρόβλημα της προσεγγιστικής αναζήτησης αλφαριθμητικών και την κατασκευή του χάρτη μνήμης bit-επιπέδου M για το πρόβλημα της μακρύτερης κοινής υποακολουθίας. Ο στόχος είναι να χρησιμοποιήσουμε τις ίδιες διατάξεις επεξεργαστών που παρουσιάσαμε στις προηγούμενες ενότητες για τα δύο προβλήματα προκειμένου να κατασκευάσουμε τους χάρτες μνήμης bit-επιπέδου με τα στοιχεία τους που διανέμονται στα κατάλληλα κελιά. Σε αυτή την υλοποίηση της φάσης προεπεξεργασίας πρέπει να ενσωματώσουμε όλες τις περιπτώσεις της ευέλικτης αναζήτησης αλφαριθμητικών όπως, αδιάφορους χαρακτήρες, κλάσεις χαρακτήρων και άρνηση ενός χαρακτήρα ή μιας κλάσης χαρακτήρων. Για αυτό τον λόγο κάθε χαρακτήρας $p_i, 1 \leq i \leq m$ του προτύπου πρέπει να συνοδεύεται από μερικές πληροφορίες ελέγχου, που θα συμβολίζουν την παρουσία του αδιάφορου συμβόλου, του συμβόλου άρνησης και του συμβόλου κλάσης



Σχήμα 7.17: (α) Μετασχηματισμένος Γράφος (β) Διάταξη Επεξεργαστών

χαρακτήρων. Ειδικά στην περίπτωση του συμβόλου κλάσης χαρακτήρων απαιτούνται δύο χαρακτήρες που συμβολίζουν τα όρια της κλάσης χαρακτήρων. Συνεπώς, κάθε χαρακτήρας p_i του προτύπου αποτελείται από (1) δύο αλφαριθμητικά που αντιστοιχούν στους δυαδικούς κώδικες των δύο χαρακτήρων που ανήκουν στο αλφάβητο Σ και (2) τις απαραίτητες πληροφορίες ελέγχου.

Οι πράξεις που εκτελούνται από την φάση προεπεξεργασίας είναι ουσιαστικά πράξεις εγγραφής σε μια κατάλληλη γραμμή R_i ή M_i, $1 \leq i \leq m$ των χαρτών μνήμης R και M αντίστοιχα που διανέμονται στο i-οστό κελί. Γνωρίζουμε ότι κάθε θέση μνήμης είναι ένα μοναδικό bit, όπου η πράξη εγγραφής αποτελείται την τοποθέτηση του bit είτε σε 0 είτε σε 1. Χρησιμοποιώντας το παράδειγμα του κώδικα χαρακτήρων UNICODE, προτείνουμε το ακόλουθο πρωτόκολλο στον οποίο φαίνεται στο Πίνακα 7.12. Χρησιμοποιούμε τρία bits ελέγχου: το bit 0 χρησιμοποιείται για την τοποθέτηση bit στην μνήμη είτε

Πίνακας 7.12: Πρωτόκολλο της φάσης προεπεξεργασίας

Λειτουργίες	Ctrl	2	1	0	Χαρ. 0/15	16/31
Καθαρισμός (αποθήκευση 0)		0	0	0	-	-
Αδιάφορος χαρακτήρας (αποθήκευση 1)		0	0	1	-	-
Απλός χαρακτήρας		0	1	0	σ_1	-
Άρνηση ενός χαρακτήρα		0	1	1	σ_1	-
Κλάση χαρακτήρων		1	1	0	σ_1	σ_2

σε 0 είτε σε 1. Τα bits 1 και 2 συμβολίζουν την παρουσία των 0, 1 ή 2 χαρακτήρων που πρέπει να διαβάσουμε.

Για να υλοποιήσουμε το πρωτόκολλο της φάσης προεπεξεργασίας πάνω στις γραμμικές διατάξεις επεξεργαστών που παρουσιάσαμε στις προηγούμενες ενότητες κάνουμε τις ακόλουθες παραδοχές. Πρώτον, υποθέτουμε ότι το κανάλι κειμένου χρησιμοποιείται για την μεταφορά κώδικα χαρακτήρων με ρυθμό ενός χαρακτήρα ανά βήμα μεταφοράς. Επίσης, τα κανάλια αποτελεσμάτων είναι ικανά να μεταφέρουν τα απαραίτητα bits ελέγχου. Δεύτερον, υποθέτουμε ότι το αλφάβητο Σ, η αποκωδικοποίηση του και το μέγεθος $|\Sigma|$ φορτώνονται εκ των προτέρων στα κελιά. Τρίτον, υποθέτουμε ότι το μέγιστο μήκος του προτύπου είναι ίσο με το μέγεθος της διάταξης επεξεργαστών m και είναι γνωστό στο σύστημα. Τέλος, το i-οστό ΣΕ γνωρίζει την θέση του στην διάταξη. Η φάση προεπεξεργασίας αρχίζει από το σήμα επανατοποθέτησης (reset signal) που περνάει από την είσοδο ελέγχου και τότε κάθε κελί μετράει τα βήματα φόρτωσης που είναι i βήματα για το i ΣΕ. Συνεπώς, κάθε κελί εκτελεί τον παρακάτω Αλγόριθμο 7.15 για την κατασκευή της γραμμής του χάρτη μνήμης bit-επιπέδου R. Παρόμοιος αλγόριθμος ακολουθείται για την κατασκευή της γραμμής του M.

Υπάρχουν $|\Sigma|$ πράξεις εγγραφής σε διαδοχικές θέσεις στη γραμμή της μνήμης R ή M. Η χαμηλότερη διεύθυνση μνήμης συμβολίζεται από το $map(0, \Sigma)$ ενώ η υψηλότερη διεύθυνση μνήμης συμβολίζεται από το $map(|\Sigma|, \Sigma)$. Κατά τη διάρκεια της φάσης προεπεξεργασίας προσπελαύνονται όλες οι θέσεις μνήμης από τρεις διαδοχικούς βρόχους FOR. Ο χρόνος υπολογισμού της φάσης προεπεξεργασίας είναι $2m + |\Sigma|$ βήματα εγγραφής. Η ολοκλήρωση της φάσης προεπεξεργασίας ενεργοποιεί την εκτέλεση της φάσης αναζήτησης για τους αλγορίθμους αναζήτησης αλφαριθμητικών ή τη φάση υπολογισμού LCS για τον αλγόριθμο μακρύτερης κοινής υποακολουθίας. Από την προηγούμενη περιγραφή συμπεραίνουμε ότι η φάση προεπεξεργασίας υλοποιείται στο ίδιο κελί που εκτελεί την φάση αναζήτησης ή την φάση

$bit \leftarrow ctrl_0$;
if $ctrl_1 = 0$ *and* $ctrl_2 = 0$ **then** $lo \leftarrow map(0, \Sigma)$, $hi \leftarrow map(|\Sigma|, \Sigma)$;
if $ctrl_1 = 1$ *and* $ctrl_2 = 0$ **then** $lo \leftarrow map(\sigma_1, \Sigma)$, $hi \leftarrow map(\sigma_1, \Sigma)$;
if $ctrl_1 = 1$ *and* $ctrl_2 = 1$ **then** $lo \leftarrow map(\sigma_1, \Sigma)$, $hi \leftarrow map(\sigma_2, \Sigma)$;
for $i = map(0, \Sigma)$ ***to*** $lo - 1$ **do**
  $R_i \leftarrow$ NOT bit;
end
for $i = lo$ ***to*** hi **do**
  $R_i \leftarrow bit$;
end
for $i = hi + 1$ ***to*** $map(|\Sigma|, \Sigma)$ **do**
  $R_i \leftarrow$ NOT bit;
end

Αλγόριθμος 7.15: Φάση Προεπεξεργασίας

υπολογισμού LCS με την προσθήκη ενός περιορισμένου προγραμματιζόμενου υλικού. Συνεπώς, ολόκληρος ο αλγόριθμος για τη φάση προεπεξεργασίας μαζί με τη φάση αναζήτησης ή υπολογισμού LCS μπορεί να εκτελεστεί σε μια διάταξη επεξεργαστών ειδικού σκοπού.

7.10 Σύγκριση με Προηγούμενες Διατάξεις Επεξεργαστών

Σε αυτή την ενότητα, συγκρίνουμε τις προτεινόμενες διατάξεις επεξεργαστών για τα προβλήματα απλής/προσεγγιστικής αναζήτησης αλφαριθμητικών και μακρύτερης κοινής υποακολουθίας με προηγούμενες υλοποιήσεις.

7.10.1 Πρόβλημα Προσεγγιστικής Αναζήτησης Αλφαριθμητικών

Οι βασικές διαφορές των αρχιτεκτονικών που παρουσιάστηκαν στην προηγούμενη ενότητα σε σύγκριση με άλλες αρχιτεκτονικές που έχουν προταθεί είναι οι εξής: Πρώτον, πολλές αρχιτεκτονικές που παρουσιάστηκαν στην βιβλιογραφία είναι κατάλληλες για διαφορετικές εφαρμογές της προσεγγιστι-

κής αναζήτησης αλφαριθμητικών, οι περισσότερες από αυτές υλοποιήθηκαν κυρίως για αλγορίθμους για ανάλυση ακολουθίας DNA, όπως [88, 90, 143, 142, 99, 77, 51, 78]. Οι αρχιτεκτονικές αυτές είναι βελτιστοποιημένες για την αναζήτηση πολύ μεγάλων αλφαριθμητικών του ίδιου μήκους με το κείμενο. Σε αντίθεση, οι προτεινόμενες διατάξεις επεξεργαστών εκτελούν αναζήτηση αλφαριθμητικών για μικρά πρότυπα σε μεγάλες αδόμητες βάσεις κειμένων. Συνεπώς, οι αρχιτεκτονικές μας ακολουθούν την μονόδρομη ροή δεδομένων σε σχέση με τις προηγούμενες αρχιτεκτονικές που ακολουθούν την αμφίδρομη ροή δεδομένων. Δεύτερον, οι διατάξεις επεξεργαστών που παρουσιάσαμε εκτελούν ευέλικτη προσεγγιστική αναζήτηση αλφαριθμητικών σε σχέση με τις προηγούμενες αρχιτεκτονικές [88, 90, 143, 142, 99, 77, 51, 78] που εκτελούν απλή προσεγγιστική αναζήτηση αλφαριθμητικών, δηλαδή, χωρίς πολύπλοκα πρότυπα. Συνεπώς, προσθέσαμε στις αρχιτεκτονικές μας την υλοποίηση ενός σχήματος αποκωδικοποίησης, δηλαδή, την εισαγωγή του αποκωδικοποιητή και τους χάρτες μνήμης bit-επιπέδου. Το σχήμα αυτό αποκωδικοποίησης μας επιτρέπει την αποτελεσματική υλοποίηση της ευέλικτης προσεγγιστικής αναζήτησης αλφαριθμητικών δηλαδή επιτρέπει την αναζήτηση για πολύπλοκα πρότυπα σε αντίθεση με την περιορισμένη ευελιξία των σχημάτων αποκωδικοποίησης που χρησιμοποιούνται στις προηγούμενες αρχιτεκτονικές. Τέλος, παρουσιάσαμε διατάξεις επεξεργαστών που παραλληλοποιούν μια καινούργια κατηγορία αλγορίθμων αναζήτησης αλφαριθμητικών, τους αλγορίθμους bit-παραλληλισμού σε σχέση με προηγούμενες διατάξεις επεξεργαστών που παραλληλοποιούν τους κλασσικούς αλγορίθμους δυναμικού προγραμματισμού.

7.10.2 Πρόβλημα LCS

Στην ενότητα αυτή παρουσιάζουμε συγκρίσεις της διάταξης επεξεργαστών που προτείναμε για το πρόβλημα LCS σε σχέση με τις προηγούμενες αρχιτεκτονικές που έχουν προταθεί στην βιβλιογραφία. Οι βασικές διαφορές της αρχιτεκτονικής μας σε σύγκριση με προηγούμενες υλοποιήσεις είναι οι εξής: Πρώτον, με βάση το βασικό αλγόριθμο δυναμικού προγραμματισμού που περιγράφεται στην εργασία [54] παρατηρείται ότι τα στοιχεία του πίνακα παίρνουν μεγάλες τιμές όταν συγκρίνονται μεγάλα αλφαριθμητικά. Οι περισσότερες εφαρμογές για την μακρύτερη κοινή υποακολουθία απαιτούν να συγκρίνονται κυρίως μεγάλα αλφαριθμητικά. Για παράδειγμα, οι ακολουθίες DNA αποτελούνται από εκατομμύρια μόρια. Πολλές διατάξεις επεξεργαστών [140, 170, 86, 82, 83, 87, 94] που έχουν προταθεί στη βιβλιογραφία βασίζονται στην παράλληλη υλοποίηση του αλγορίθμου δυναμικού προγραμματισμού που

απαιτεί κάθε κελί να προσθέτει και να συγκρίνει σχετικά μεγάλες τιμές της τάξης των $log(n)$ bits για αλφαριθμητικά μήκους n. Επίσης, τα πλάτη των καναλιών επικοινωνίας που απαιτούνται για να ανταλλάξουν δεδομένα μεταξύ των γειτονικών κελιών είναι πολύ μεγάλα. Σε αντίθεση, η αρχιτεκτονική μας βασίζεται στον αλγόριθμο bit-παραλληλισμού που ελαχιστοποιεί τον αριθμό των bits που απαιτούνται για να αναπαραστήσει ένα οποιοδήποτε στοιχείο του πίνακα δυναμικού προγραμματισμού και εκτελεί υπολογισμούς LCS για μεγάλα αλφαριθμητικά. Επίσης, η αρχιτεκτονική μας ελαχιστοποιεί τη ροή δεδομένων ανάμεσα στα γειτονικά κελιά.

Η αρχιτεκτονική μας που βασίζεται στον νέο αλγόριθμο bit-παραλληλισμού απαιτεί κάθε κελί ή επεξεργαστής να εκτελεί απλές αριθμητικές και λογικές πράξεις bit-επιπέδου σε αντίθεση με τις πράξεις σύγκρισης, δηλαδή, πολλές πράξεις if που χρησιμοποιούνται στις διατάξεις επεξεργαστών [140, 170, 86, 82, 83, 87, 94]. Το πλεονέκτημα των αριθμητικών και λογικών πράξεων bit-επιπέδου είναι ότι εκτελούνται αρκετά γρήγορα σε σχέση με τις πράξεις σύγκρισης. Έτσι, η αρχιτεκτονική μας χρησιμοποιεί τις δύο πράξεις σύγκρισης συνολικά που χρησιμοποιούνται για την ανάκτηση της ακολουθίας LCS.

Μια άλλη βασική διαφορά της αρχιτεκτονικής μας σε σχέση με τις προηγούμενες αρχιτεκτονικές που έχουν παρουσιαστεί στην διεθνή βιβλιογραφία είναι η εισαγωγή του αποκωδικοποιητή και των χαρτών μνήμης bit-επιπέδου, όπως στις αρχιτεκτονικές που προτείναμε για το πρόβλημα της προσεγγιστικής αναζήτησης αλφαριθμητικών. Το σχήμα αυτό αποκωδικοποίησης μας επιτρέπει την αποτελεσματική υλοποίηση του καινούργιου αλγορίθμου bit-παραλληλισμού LCS.

Επίσης, η διάταξη μας επεξεργαστών χρειάζεται δύο καταχωρητές σε κάθε κελί και δύο κανάλια επικοινωνίας για τον υπολογισμό του μήκους της ακολουθίας LCS σε σύγκριση με την αρχιτεκτονική [87] που απαιτεί τρεις καταχωρητές και δύο κανάλια επικοινωνίας και με την αρχιτεκτονική [82] που απαιτεί δύο καταχωρητές και τρία κανάλια επικοινωνίας. Τέλος, η αρχιτεκτονική μας για την ανάκτηση της ακολουθίας LCS απαιτεί ένα καταχωρητή σε κάθε κελί και ένα κανάλι επικοινωνίας κατάλληλου μεγέθους για να αποθηκεύσει και να μεταφέρει τα m bits του LCS αντίστοιχα αντί για m καταχωρητές και m κανάλια όπως χρησιμοποιούνται στις αρχιτεκτονικές [82, 83, 87]. Συμπερασματικά, η αρχιτεκτονική μας χρησιμοποιεί 3 καταχωρητές και κανάλια συνολικά σε σύγκριση με την αρχιτεκτονική [82] που χρειάζεται $m + 2$ καταχωρητές και $m + 3$ κανάλια συνολικά και με την αρχιτεκτονική [87] που απαιτεί $m + 3$ καταχωρητές και $m + 2$ κανάλια συνολικά.

Κεφάλαιο 8

Προγραμματιζόμενη Αρχιτεκτονική για Αλγορίθμους Αναζήτησης Αλφαριθμητικών

8.1 Εισαγωγή

Στο προηγούμενο κεφάλαιο παρουσιάσαμε διατάξεις επεξεργαστών ειδικού σκοπού για απλούς και πρόσφατους ευέλικτους αλγορίθμους αναζήτησης αλφαριθμητικών. Όμως, οι αρχιτεκτονικές ειδικού σκοπού παρουσιάζουν ένα μειονέκτημα: ότι οι αρχιτεκτονικές αυτές περιορίζονται στην εκτέλεση ενός συγκεκριμένου αλγορίθμου και δεν προσφέρουν την απαιραίτητη ευελεξία για να εκτελέσουν μια ποικιλία αλγορίθμων που απαιτούνται για να αναζητηθεί ένα πρότυπο αλφαριθμητικό σε μεγάλες αδόμητες βάσεις κειμένων. Συνεπώς, σε αυτό το κεφάλαιο υπάρχουν τρεις βασικοί στόχοι. Ο πρώτος στόχος είναι να προτείνουμε μια ενιαία προγραμματιζόμενη αρχιτεκτονική διάταξη επεξεργαστών (programmable array processor architecture) κατάλληλη για αποτελεσματική εκτέλεση μιας κλάσης αλγορίθμων αναζήτησης αλφαριθμητικών. Υπάρχει ένας μεγάλος αριθμός αλγορίθμων αναζήτησης αλφαριθμητικών για ανάκτηση κειμένου και απαιτείται μια προγραμματιζόμενη αρχιτεκτονική για να συνοδεύσει όλους τους διαφορετικούς αλγορίθμους μέσα σε ένα ενιαίο σύστημα. Καθώς αναπτύσσονται καινούργιοι αλγόριθμοι, η προτεινόμενη αρχιτεκτονική να είναι ικανή να εκτελέσει πολλούς από αυτούς τους

αλγορίθμους χωρίς την ανάγκη να επανασχεδιαστεί.

Ο δεύτερος στόχος είναι η προγραμματιζόμενη αρχιτεκτονική που προτείνουμε να προσφέρει ευελιξία και συγχρόνως να παρέχει υψηλή απόδοση σε ένα επίπεδο παρόμοιο με τις αρχιτεκτονικές ειδικού σκοπού.

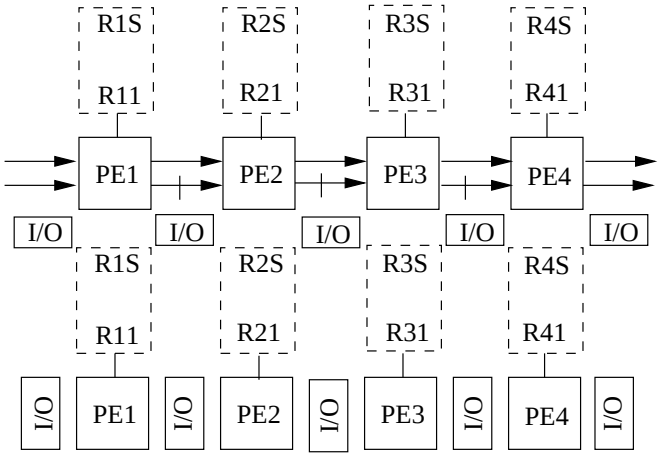
Ο τρίτος στόχος του κεφαλαίου αυτού είναι να αναπτύσσουμε ένα μαθηματικό μοντέλο πρόβλεψης απόδοσης για την προτεινόμενη προγραμματιζόμενη αρχιτεκτονική. Το μαθηματικό μοντέλο που προτείνουμε μπορεί να προβλέψει την υπολογιστική συμπεριφορά όλων των αλγορίθμων αναζήτησης αλφαριθμητικών που μπορεί να εκτελέσει η προγραμματιζόμενη αρχιτεκτονική.

Η δομή του κεφαλαίου αυτού είναι η εξής: στην επόμενη ενότητα παρουσιάζεται η ενιαία αρχιτεκτονική της διάταξης επεξεργαστών. Στην ενότητα 8.3 περιγράφουμε την αρχιτεκτονική του κάθε επεξεργαστή ή κελιού της διάταξης. Στην ενότητα 8.4 παρουσιάζεται η μαθηματική ανάλυση απόδοσης για την προγραμματιζόμενη αρχιτεκτονική. Τέλος, στην ενότητα 8.5 παρουσιάζεται η επιβεβαίωση του μοντέλου απόδοσης με τα πειραματικά αποτελέσματα για την προγραμματιζόμενη αρχιτεκτονική.

8.2 Αρχιτεκτονική της Διάταξης Επεξεργαστών

Το Σχήμα 8.1 φαίνεται η αρχιτεκτονική της γραμμικής προγραμματιζόμενης διάταξης επεξεργαστών. Η παραπάνω δομή της αρχιτεκτονικής προέκυψε από την εφαρμογή της μεθόδου διαμερισμένης υλοποίησης [72, 121], μιας ποικιλίας αλγορίθμων αναζήτησης αλφαριθμητικών. Σε όλες τις περιπτώσεις, η μέθοδος αυτή κατέληξε σε διατάξεις επεξεργαστών με την ίδια αρχιτεκτονική, και έτσι φτάσαμε στο συμπέρασμα ότι μια διάταξη επεξεργαστών είναι ικανή για την αποτελεσματική εκτέλεση μιας κλάσης αλγορίθμων αναζήτησης αλφαριθμητικών (class-specific processor array). Ολόκληρη η αρχιτεκτονική προσδιορίστηκε από τη διαδικασία απεικόνισης που παρουσιάσαμε στο προηγούμενο κεφάλαιο, ενώ τα χαρακτηριστικά των επεξεργαστών ή των κελιών προσδιορίζονται από τις απαιτήσεις των συγκεριμένων αλγορίθμων που συμπεριλαμβάνονται στην κλάση.

Η διάταξη αποτελείται από $k+1$ ή δύο γραμμικές δομές των m στοιχείων επεξεργασίας (ΣΕ), όπως φαίνεται στο Σχήμα 8.1 για $m = 4$. Επίσης, υπάρχουν $2(m+1)$ επιπλέον διεπαφές Ε/Ε- I/O (Εισόδου/Εξόδου - Input/Output) που τοποθετούνται ανάμεσα στις δύο διατάξεις και χρησιμοποιούν-



Σχήμα 8.1: Γραμμική προγραμματιζόμενη διάταξη επεξεργαστών για διάφορους αλγορίθμους αναζήτησης αλφαριθμητικών

ται για να απλοποιήσουν τις επικοινωνίες δεδομένων που μεταφέρουν τη δυαδική αναπαράσταση των χαρακτήρων κειμένου και τα αποτελέσματα bit ή byte επιπέδου αλλά σε καμιά περίπτωση δεν εκτελούν υπολογισμούς. Η επάνω διάταξη του Σχήματος 8.1 χρησιμοποιείται για την εκτέλεση των αλγορίθμων bit-παραλληλισμού με αποτυχίες και διαφορές που αντιστοιχούν στο επίπεδο $l = 0$, ενώ η κάτω διάταξη εκτελεί όλους τους ευέλικτους αλγορίθμους αναζήτησης αλφαριθμητικών, περιλαμβάνοντας τους αλγορίθμους δυναμικού προγραμματισμού, bit-παραλληλισμού που αντιστοιχούν στο επίπεδο $l = 1$ και LCS. Οι επικοινωνίες είναι μονόδρομες μεταξύ των γειτονικών κελιών και γειτονικών μονάδων E/E-I/O.

8.3 Αρχιτεκτονική των Επεξεργαστών

Για το σχεδιασμό της προγραμματιζόμενης αρχιτεκτονικής εξετάσαμε τις απαιτήσεις των κελιών (όπως εισόδους - εξόδους, μνήμη, καταχωρητές και πράξεις) σε μια ποικιλία αλγορίθμων αναζήτησης αλφαριθμητικών και συμπεριλάμβαμε υλικό μέσα στο κελί έτσι ώστε να υπάρχει γρήγορη εκτέλεση των αλγορίθμων και συγχρόνως να διατηρήσουμε την ευελιξία. Ο Πίνακας 8.1 παρουσιάζει τις απαιτήσεις των κελιών για διάφορους αλγορίθμους αναζήτησης αλφαριθμητικών.

Πίνακας 8.1: Απαιτήσεις των κελιών για διάφορους αλγορίθμους αναζήτησης αλφαριθμητικών

Αλγόριθμος	Κελιά	Είσοδοι	Έξοδοι	Μνήμη	Καταχωρητές	Πράξεις
Δυν. Προγρ. με k αποτυχίες	m	2	2	R	-	αναφορά μνήμης/πρόσθεση
Δυν. Προγρ. με k διαφορές	m	3	3	R	D	αναφορά μνήμης/πρόσθεση/ελάχ
Δυν. Προγρ. του Myers	m	2	2	R	Dv, aux	αναφορά μνήμης/πρόσθεση/αφαίρεση/ελάχιστ
Bit-παραλ.	m	2	2	R	-	αναφορά μνήμης/OR
Bit-παραλ. με k αποτυχίες	$(k+1)m$	$k+2$	$k+1$	R	-	αναφορά μνήμης/OR/AND
Bit-παραλ. με k διαφορές	$(k+1)m$	$k+3$	$k+1$	R	F	αναφορά μνήμης/OR/AND
LCS	m	3	3	M	DLN, LCS, aux	αναφορά μνήμης/OR/AND/πρόσθεση/αφαίρεσ
LLCS	m	2	2	M	DLN, aux	αναφορά μνήμης/OR/AND/πρόσθεση/αφαίρεσ

Η αρχιτεκτονική του προγραμματιζόμενου στοιχείου επεξεργασίας (ΣΕ) παρουσιάζεται στο Σχήμα 8.2. Τα βασικά μέρη του στοιχείου επεξεργασίας είναι η μονάδα προεπεξεργασίας (preprocessing module), ο αποκωδικοποιητής χαρακτήρα - διεύθυνσης (character to address decoder), η τοπική μνήμη τυχαίας προσπέλασης bit-επιπέδου (local bit-level RAM), η μονάδα ελέγχου (control unit), η αριθμητική μονάδα (arithmetic unit) και οι πολυπλέκτες (multiplexers), που περιγράφονται παρακάτω.

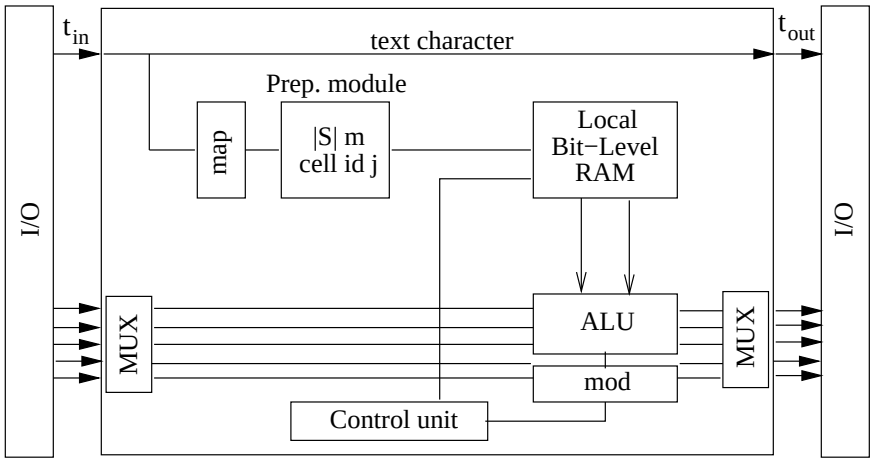
- Μονάδα Προεπεξεργασίας: Η μονάδα αυτή χρησιμοποιείται για να υλοποιήσει τη φάση προεπεξεργασίας διαφόρων αλγορίθμων αναζήτησης αλφαριθμητικών. Συγκεριμένα, η μονάδα αυτή προ-αποθηκεύει τρεις ποσότητες όπως το μέγεθος του αλφαβήτου $|\Sigma|$, το μήκος του προτύπου m και η ταυτότητα του κελιού id, που χρησιμοποιούνται για την εκτέλεση της φάσης προεπεξεργασίας.

- Αποκωδικοποιητής Χαρακτήρα - Διεύθυνσης: Παρατηρούμε ότι το ρεύμα κειμένου του Σχήματος 8.2 που μεταφέρει τον δυαδικό κώδικα του χαρακτήρα κειμένου ch χρησιμοποιείται σαν είσοδος για τον αποκωδικοποιητή Χαρακτήρα - Διεύθυνσης. Ο αποκωδικοποιητής είναι ουσιαστικά μια προγραμματιζόμενη υλοποίηση υλικού της συνάρτησης απεικόνισης $map(ch, \Sigma)$. Η έξοδος του αποκωδικοποιητή είναι η διεύθυνση c του χάρτη μνήμη bit-επιπέδου R ή M.

- Τοπική Μνήμη Τυχαίας Προσπέλασης Bit-Επιπέδου: Η τοπική μνήμη bit-επιπέδου αποθηκεύει μια γραμμή R_i και M_i, $1 \leq i \leq m$, από τους χάρτες μνήμης bit-επιπέδου R και M αντίστοιχα. Συνεπώς, το μέγεθος της τοπικής μνήμης είναι $|\Sigma|$ bits και η διεύθυνση που απαιτείται για να προσπελάσει σε μια μνήμη είναι μεγέθους $log|\Sigma|$ bits. Λαμβάνοντας υπόψη το παράδειγμα του κώδικα χαρακτήρων UNICODE οι απαιτήσεις της τοπικής μνήμης είναι 16Kbytes ανά κελί.

- Μονάδα Ελέγχου: Η μονάδα ελέγχου αποτελείται από έναν μετρητή προγράμματος (program counter), μια μνήμη προγράμματος (program memory) και έναν αποκωδικοποιητή εντολών (instruction decoder). Ο μετρητής προγράμματος δίνει τη διεύθυνση της εντολής που πρόκειται να εκτελεστεί. Η μνήμη προγράμματος αποθηκεύει τις εντολές ενός προγράμματος ή ενός αλγορίθμου και ο αποκωδικοποιητής εντολών αποκωδικοποιεί τις εντολές του προγράμματος και παράγει σήματα ελέγχου για την εκτέλεση της εντολής.

- Αριθμητική μονάδα: Η αριθμητική μονάδα αποτελείται από την αριθμητική και λογική μονάδα (Arithmetic and Logic Unit - ALU) και τους καταχωρητές γενικού σκοπού. Η ALU εκτελεί τις



Σχήμα 8.2: Ορισμός του κελιού για την προγραμματιζόμενη διάταξη επεξεργαστών

πράξεις πρόσθεσης, αφαίρεσης, μετατόπισης και μερικές λογικές πράξεις όπως (NOT, AND, OR). Οι καταχωρητές γενικού σκοπού αποθηκεύουν ενδιάμεσα αποτελέσματα όπως $\Delta, \Delta_v, \Delta LN, F, aux$ και συνήθως συνδέονται άμεσα με το διάδρομο δεδομένων. Η αριθμητική αυτή μονάδα είναι επαρκής για ένα σύνολο ευέλικτων αλγορίθμων αναζήτησης αλφαριθμητικών. Όμως, μια αριθμητική μονάδα με επιπλέον δυνατότητες μπορεί να χρησιμοποιηθεί σε μια διάταξη επεξεργαστών κατάλληλη για άλλους αλγορίθμους.

- Πολυπλέκτες: Όταν επιλέγεται ένας αλγόριθμος για εκτέλεση πάνω στην διάταξη, οι πολυπλέκτες εισόδου και εξόδου φροντίζουν να επιλέξουν τα κατάλληλα κανάλια επικοινωνίας τα οποία μεταφέρουν αποτελέσματα bit ή byte-επιπέδου.

8.4 Ανάλυση Απόδοσης για την Προγραμματιζόμενη Αρχιτεκτονική

Σε αυτή την ενότητα αναλύουμε την απόδοση της προγραμματιζόμενης αρχιτεκτονικής πάνω σε επεξεργαστές γενικού σκοπού όπως Pentium. Εδώ περιγράφουμε δύο μοντέλα απόδοσης: Το πρώτο μοντέλο αναλύει την απόδοση της προγραμματιζόμενης διάταξης επεξεργαστών σε ένα επεξεργαστή και το δεύτερο μοντέλο αναλύει την απόδοση πάνω σε μια γραμμική διάταξη επεξεργαστών.

Σε αυτή την ενότητα χρησιμοποιούμε τους παρακάτω παραμέτρους:

- n_{prep}, ο αριθμός των πράξεων που απαιτούνται από τη φάση προεπεξεργασίας ενός οποιουδήποτε αλγόριθμου αναζήτησης αλφαριθμητικών,
- $n_{searchcomp}$, ο αριθμός των πράξεων που απαιτούνται από τη φάση αναζήτησης ενός οποιουδήποτε αλγόριθμου αναζήτησης αλφαριθμητικών,
- n_{comm}, ο αριθμός των δεδομένων που μεταφέρονται,
- t_{prep}, ο μέσος χρόνος για να εκτελεστεί ένα βήμα προεπεξεργασίας,
- $t_{searchcomp}$, ο μέσος χρόνος για να εκτελεστεί ένα βήμα αναζήτησης,
- t_{comm}, ο μέσος χρόνος για μεταφορά δεδομένων και
- w, ο αριθμός των bits σε μια λέξη του επεξεργαστή.

Οι παράμετροι n_{prep}, $n_{searchcomp}$ και n_{comm} εξαρτώνται από τον αλγόριθμο και από την μέθοδο απεικόνισης, ενώ οι παράμετροι t_{prep}, $t_{searchcomp}$, t_{comm} και w εξαρτώνται από τα χαρακτηριστικά του επεξεργαστή.

Πριν διατυπώσουμε το μαθηματικό μοντέλο για ένα επεξεργαστή και για μια διάταξη επεξεργαστών κάνουμε την υπόθεση ότι κάθε επεξεργαστής επεξεργάζεται ένα τμήμα (block) από w χαρακτήρες προτύπου, όσο είναι το μέγεθος της λέξης του υπολογιστή. Με άλλα λόγια, όταν οι αλγόριθμοι bit-παραλληλισμού και ο αλγόριθμος της μακρύτερης κοινής υποακολουθίας επιλέγονται για να εκτελεστούν στην προγραμματιζόμενη αρχιτεκτονική μπορούν να κάνουν χρήση του εσωτερικού παραλληλισμού των bit πράξεων μέσα σε μια λέξη του επεξεργαστή. Δηλαδή, συνενώνουμε τις w τιμές bit-επιπέδου σε μια λέξη και ενημερώνουμε τις τιμές αυτές με μια μόνο πράξη. Το παραπάνω πλεονέκτημα της χρήσης εσωτερικού παραλληλισμού δεν μπορεί να χρησιμοποιηθεί για τους αλγορίθμους δυναμικού προγραμματισμού, απλά σε κάθε επεξεργαστή θα εκτελούνται οι αλγόριθμοι ακολουθιακά σε w χαρακτήρες του προτύπου.

8.4.1 Ανάλυση Απόδοσης σε ένα Επεξεργαστή

Ο χρόνος εκτέλεσης για τον υπολογισμό των αλγορίθμων αναζήτησης αλφαριθμητικών σε έναν επεξεργαστή της προγραμματιζόμενης αρχιτεκτονικής αποτελείται από δύο όρους:

- T_{prep} είναι ο συνολικός χρόνος που απαιτείται για την εκτέλεση της φάσης προεπεξεργασίας. Ο αριθμός των πράξεων που απαιτούται για την εκτέλεση της φάσης προεπεξεργασίας για όλους τους αλγόριθμους αναζήτησης αλφαριθμητικών είναι περίπου $m|\Sigma|$ βήματα όταν το μέγεθος του προτύπου ταιριάζει σε μια λέξη του υπολογιστή. Στην πράξη, σπάνια συμβαίνει το μέγεθος του προτύπου να ταιριάζει σε μια λέξη του επεξεργαστή. Έτσι όταν το μέγεθος του προτύπου είναι πολύ μεγάλο, τότε απαιτούνται $\lceil \frac{m}{w} \rceil$ περάσματα των w bits. Γενικά, ο χρόνος προεπεξεργασίας δίνεται από:

$$T_{prep} = n_{prep} t_{prep} = (\lceil \frac{m}{w} \rceil w|\Sigma|) t_{prep} \tag{8.1}$$

- $T_{searchcomp}$ είναι ο συνολικός χρόνος αναζήτησης για να εκτελεστεί ένας αλγόριθμος αναζήτησης αλφαριθμητικών σε έναν επεξεργαστή. Στον Πίνακα 8.2 παρουσιάζεται ο συνολικός αριθμός των πράξεων ($n_{searchcomp}$) που απαιτούνται για να εκτελεστούν διάφοροι αλγόριθμοι αναζήτησης αλφαριθμητικών με την υπόθεση ότι το μέγεθος του προτύπου είναι μικρότερο από w. Γενικά, ο χρόνος αναζήτησης δίνεται από:

$$T_{searchcomp} = n_{searchcomp} t_{searchcomp} \tag{8.2}$$

Ο ακολουθιακός χρόνος εκτέλεσης για ένα οποιοδήποτε αλγόριθμο αναζήτησης αλφαριθμητικών είναι το άθροισμα των δύο παραπάνω όρων και δίνεται από:

$$T_{seq} = T_{prep} + T_{searchcomp} \tag{8.3}$$

8.4.2 Ανάλυση Απόδοσης σε μια Διάταξη Επεξεργαστών

Ο συνολικός χρόνος εκτέλεσης για την υλοποίηση διαφόρων αλγορίθμων αναζήτησης αλφαριθμητικών σε μια προγραμματιζόμενη διάταξη επεξεργαστών αποτελείται από τρεις όρους:

Πίνακας 8.2: Ο αριθμός των πράξεων αναζήτησης για διάφορους αλγορίθμους αναζήτησης αλφαριθμητικών σε ένα επεξεργαστή

Αλγόριθμος	$n_{searchcomp}$
Δυν. Προγρ. με k αποτυχίες	$n(\lceil \frac{m}{w} \rceil w)$
Δυν. Προγρ. με k διαφορές	$n(\lceil \frac{m}{w} \rceil w)$
Δυν. Προγρ. του Myers	$n(\lceil \frac{m}{w} \rceil w)$
Bit-παραλ.	$n(\lceil \frac{m}{w} \rceil)$
Bit-παραλ. με k αποτυχίες	$n(\lceil \frac{m}{w} \rceil k)$
Bit-παραλ. με k διαφορές	$n(\lceil \frac{m}{w} \rceil k)$
LCS	$n(\lceil \frac{m}{w} \rceil w)$
LLCS	$n(\lceil \frac{m}{w} \rceil)$

- T_{prep} είναι ο συνολικός χρόνος προεπεξεργασίας για να φορτωθεί το πρότυπο αλφαριθμητικό και στην συνέχεια να κατασκευαστεί ο αντίστοιχος χάρτης μνήμη bit-επιπέδου όπως οι R ή M. Ο συνολικός αριθμός των πράξεων που απαιτούνται από την φάση προεπεξεργασίας για διάφορους αλγορίθμους αναζήτησης αλφαριθμητικών είναι $2\lceil \frac{m}{w} \rceil + w|\Sigma|$ βήματα. Τότε, ο χρόνος προεπεξεργασίας σε μια διάταξη επεξεργαστών δίνεται από:

$$T_{prep} = n_{prep} t_{prep} = (2\lceil \frac{m}{w} \rceil + w|\Sigma|) t_{prep} \tag{8.4}$$

- $T_{searchcomp}$ είναι ο συνολικός χρόνος αναζήτησης για να εκτελεστεί ένας οποιοδήποτε αλγόριθμος αναζήτησης αλφαριθμητικών στην προγραμματιζόμενη αρχιτεκτονική. Στον Πίνακα 8.3 παρουσιάζεται ο συνολικός αριθμός των πράξεων ($n_{searchcomp}$) που απαιτείται για διάφορους αλγορίθμους αναζήτησης αλφαριθμητικών. Συνεπώς, ο χρόνος αναζήτησης δίνεται από την εξίσωση:

$$T_{searchcomp} = n_{searchcomp} t_{searchcomp} \tag{8.5}$$

- T_{comm} είναι ο συνολικός χρόνος επικοινωνίας για να μεταφερθούν οι χαρακτήρες κειμένου και τα αποτελέσματα ανάμεσα στους επεξεργαστές. Θεωρούμε ότι η φάση επικοινωνίας δεν επικαλύπτεται με την φάση αναζήτησης ενός οποιουδήποτε αλγορίθμου αναζήτησης αλφαριθμητικών. Τα βήματα που απαιτούνται είναι $n_{comm} = n + \lceil \frac{m}{w} \rceil - 1$. Ο χρόνος επικοινωνίας δίνεται από την παρακάτω εξίσωση:

$$T_{comm} = n_{comm} t_{comm} = (n + \lceil \frac{m}{w} \rceil - 1) t_{comm} \tag{8.6}$$

Πίνακας 8.3: Ο αριθμός των πράξεων αναζήτησης για διάφορους αλγορίθμους αναζήτησης αλφαριθμητικών σε μια διάταξη επεξεργαστών

Αλγόριθμος	$n_{searchcomp}$
Δυν. Προγρ. με k αποτυχίες	$(n + \lceil \frac{m}{w} \rceil - 1)w$
Δυν. Προγρ. με k διαφορές	$(n + \lceil \frac{m}{w} \rceil - 1)w$
Δυν. Προγρ. του Myers	$(n + \lceil \frac{m}{w} \rceil - 1)w$
Bit-παραλ.	$(n + \lceil \frac{m}{w} \rceil - 1)$
Bit-παραλ. με k αποτυχίες	$(n + \lceil \frac{m}{w} \rceil - 1)$
Bit-παραλ. με k διαφορές	$(n + \lceil \frac{m}{w} \rceil - 1)$
LCS	$(n + \lceil \frac{m}{w} \rceil - 1)w$
LLCS	$(n + \lceil \frac{m}{w} \rceil - 1)$

Ο χρόνος t_{comm} μιας μεταφοράς δεδομένων αντιστοιχεί με τον συνολικό χρόνο επικοινωνίας εισόδου και εξόδου, δηλαδή, το χρόνο επικοινωνίας από αριστερά και δεξιά του κάθε επεξεργαστή. Γνωρίζουμε ότι ο χρόνος επικοινωνίας μιας εισόδου ή μιας εξόδου ορίζεται από τον άθροισμα $\alpha + R\beta$, όπου α είναι ο χρόνος αρχικής καθυστέρησης, β είναι ο χρόνος μετάδοσης ανά λέξη και R είναι το μέγεθος των δεδομένων που μεταφέρονται. Επίσης, ο αριθμός επικοινωνίας εισόδου (n_{input}) και εξόδου (n_{output}) για ένα οποιοδήποτε αλγόριθμο αναζήτησης αλφαριθμητικών που εκτελείται στην προγραμματιζόμενη αρχιτεκτονική φαίνεται στο Πίνακα 8.1 από τις στήλες είσοδοι και έξοδοι αντίστοιχα. Συνεπώς, ο μέσος χρόνος t_{comm} για μια μεταφορά δεδομένων ορίζεται ως εξής:

$$t_{comm} = n_{input}(\alpha + R\beta) + n_{output}(\alpha + R\beta) = (n_{input} + n_{output})(\alpha + R\beta) \tag{8.7}$$

Ο χρόνος εκτέλεσης ενός οποιοδήποτε αλγορίθμου αναζήτησης αλφαριθμητικών στην προγραμματιζόμενη αρχιτεκτονική είναι το άθροισμα των τριών παραπάνω όρων και δίνεται από:

$$T_{p-array} = T_{prep} + T_{searchcomp} + T_{comm} \tag{8.8}$$

8.5 Επιβεβαίωση του Μοντέλου Απόδοσης

8.5.1 Επιβεβαίωση του Μοντέλου Απόδοσης για ένα Επεξεργαστή

Σε αυτή την ενότητα παρουσιάζουμε την επιβεβαίωση του μοντέλου απόδοσης σε ένα επεξεργαστή Pentium. Για να επιβεβαιώσουμε την εξίσωση σε ένα επεξεργαστή χρησιμοποιήσαμε τους αλγορίθμους

Πίνακας 8.4: Χρόνοι εκτέλεσης (σε δευτερόλεπτα) για τον αλγόριθμο LCS σε Pentium 4 1.5 GHz

n/m	256	512	1024	2048
15MB	1057.049	2119.054	4277.097	8585.065
30MB	2114.087	4238.086	8554.149	17170.042
150MB	10570.389	21190.344	42770.572	85849.852
300MB	21140.768	42380.666	85541.099	171699.614

Πίνακας 8.5: Χρόνοι εκτέλεσης (σε δευτερόλεπτα) για τον αλγόριθμο LLCS σε Pentium 4 1.5 GHz

n/m	256	512	1024	2048
15MB	995.575	1987.293	4027.120	8081.493
30MB	1991.139	3974.563	8054.195	16162.896
150MB	9955.648	19872.726	40270.794	80814.123
300MB	19911.284	39745.430	80541.543	161628.15

LCS και LLCS. Για να πάρουμε τα θεωρητικά αποτελέσματα πρέπει πρώτα να εκτιμήσουμε τις τιμές των παραμέτρων t_{prep}, $t_{searchcomp}$ για τον αλγόριθμο LCS και $t_{searchcomp}$ για τον αλγόριθμο LLCS σε ένα επεξεργαστή Pentium 4 1.5 GHz. Έτσι, οι παράμετροι t_{prep}, $t_{searchcomp}$ για τον αλγόριθμο LCS και $t_{searchcomp}$ για τον αλγόριθμο LLCS που χρησιμοποιούνται στο μοντέλο απόδοσης για ένα επεξεργαστή είναι 1.692E-08, 2.8279E-07 και 2.5474E-07 δευτερόλεπτα αντίστοιχα.

Στους Πίνακες 8.4 και 8.5 παρουσιάζονται οι χρόνοι εκτέλεσης για διαφορετικούς συνδυασμούς των n και m σε ένα Pentium 4 1.5 GHz για την εκτέλεση των αλγορίθμων LCS και LLCS αντίστοιχα. Επίσης, στους Πίνακες 8.6 και 8.7 παρουσιάζονται οι θεωρητικοί χρόνοι εκτέλεσης για διαφορετικούς συνδυασμούς των n και m σε ένα Pentium 4 1.5 GHz για την εκτέλεση των αλγορίθμων LCS και LLCS αντίστοιχα. Οι θεωρητικοί χρόνοι εκτέλεσης προκύπτουν από την εξίσωση του μοντέλου απόδοσης.

Πίνακας 8.6: Θεωρητικοί χρόνοι εκτέλεσης (σε δευτερόλεπτα) για τον αλγόριθμο LCS σε Pentium 4 1.5 GHz

n/m	256	512	1024	2048
15MB	1086.197	2172.395	4344.790	8689.580
30MB	2172.111	4344.222	8688.444	17376.889
150MB	10859.420	21718.839	43437.679	86875.359
300MB	21718.556	43437.112	86874.223	173748.447

Πίνακας 8.7: Θεωρητικοί χρόνοι εκτέλεσης (σε δευτερόλεπτα) για τον αλγόριθμο LLCS σε Pentium 4 1.5 GHz

n/m	256	512	1024	2048
15MB	978.485	1956.971	3913.942	7827.884
30MB	1956.687	3913.374	7826.748	15653.497
150MB	9782.300	19564.600	39129.199	78258.399
300MB	19564.316	39128.632	78257.263	156514.527

Παρατηρούμε ότι τα θεωρητικά αποτελέσματα για την εκτέλεση των αλγορίθμων LCS και LLCS σε ένα επεξεργαστή Pentium επιβεβαιώνουν τα πειραματικά αποτελέσματα. Συνεπώς, το μοντέλο απόδοσης για ένα επεξεργαστή επιβεβαιώνεται από την υπολογιστική συμπεριφορά των πειραματικών αποτελεσμάτων. Τέλος, παρατηρούμε ότι η μέγιστη διαφορά ανάμεσα στις θεωρητικές και πειραματικές τιμές είναι λιγότερη από 3%.

8.5.2 Επιβεβαίωση του Μοντέλου Απόδοσης για μια Διάταξη Επεξεργαστών

Σε αυτή την ενότητα παρουσιάζουμε την επιβεβαίωση του μοντέλου απόδοσης για μια διάταξη επεξεργαστών Pentium χρησιμοποιώντας πάλι τους αλγορίθμους LCS και LLCS. Για να επιβεβαιώσουμε το μοντέλο απόδοσης σε μια διάταξη επεξεργαστών εκτελέσαμε όλα τα πειράματα σε μια συστοιχία σταθμών εργασίας. Η συστοιχία αποτελείται από 8 σταθμούς εργασίας Pentium 200 MHz που συνδέονται μέσω δικτύου 100 Mb/s Fast Ethernet. Επίσης, για την υλοποίηση της γραμμικής διάταξης επεξεργαστών σε συστοιχία χρησιμοποιήσαμε την βιβλιοθήκη MPI έκδοση 1.2.

Οι τιμές των παραμέτρων α και β που χρησιμοποιούνται για να προσδιορίζουμε το χρόνο επικοινωνίας t_{comm} της εξίσωσης είναι 7.74E-04 και 1.06E-07 δευτερόλεπτα αντίστοιχα. Συνεπώς, ο χρόνος επικοινωνίας t_{comm} για τους αλγορίθμους LCS και LLCS είναι 4.647E-03 δευτερόλεπτα.

Στον Πίνακα 8.8 παρουσιάζονται οι χρόνοι εκτέλεσης σε μια διάταξη από επεξεργαστές Pentium 200 MHz για αλφαριθμητικό μεγέθους 300,000 χαρακτήρων για την υλοποίηση του αλγορίθμου LCS. Στον Πίνακα 8.9 αναφέρονται οι θεωρητικοί χρόνοι εκτέλεσης που προκύπτουν από την εξίσωση για την υλοποίηση του αλγορίθμου LCS.

Παρατηρούμε ότι τα θεωρητικά αποτελέσματα σε μια γραμμική διάταξη από επεξεργαστές Pen-

Πίνακας 8.8: Χρόνοι εκτέλεσης (σε δευτερόλεπτα) για τον αλγόριθμο LCS σε μια διάταξη από επεξεργαστές Pentium 200 MHz

n/m	64	256
300KB	1376.261	1196.123
Αριθμ. επεξεργαστών	2	8

Πίνακας 8.9: Θεωρητικοί χρόνοι εκτέλεσης (σε δευτερόλεπτα) για τον αλγόριθμο LCS σε μια διάταξη από επεξεργαστές Pentium 200 MHz

n/m	64	256
300KB	1396.819	1396.847
Αριθ. επεξεργαστών	2	8

tium επιβεβαιώνουν τα πειραματικά αποτελέσματα. Συνεπώς, το μοντέλο απόδοσης για την διάταξη επεξεργαστών που παρουσιάσαμε είναι σωστό και δίνει σχετικά ακριβείς προβλέψεις. Πρέπει να σημειώσουμε εδώ ότι δεν επεκτείναμε τα πειράματα μας για μεγάλα μεγέθη αλφαριθμητικών και μεγάλο αριθμό επεξεργαστών για δύο λόγους. Ένας λόγος είναι ότι η υλοποίηση της διάταξης επεξεργαστών (ή υλοποίηση του μοντέλου διασωλήνωσης) σε μια συστοιχία σταθμών εργασίας με δίκτυο Fast Ethernet απαιτεί υψηλό κόστος επικοινωνίας. Με άλλα λόγια, ο χρόνος εκτέλεσης της υλοποίησης κυριαρχείται από τον χρόνο επικοινωνίας. Ένας δεύτερος λόγος είναι ότι στόχος μας είναι η απλή επιβεβαίωση του μοντέλου απόδοσης σε μια συστοιχία σταθμών εργασίας. Έτσι, αυτό το μοντέλο να μπορεί να χρησιμοποιηθεί για την πρόβλεψη του χρόνου εκτέλεσης ενός οποιουδήποτε αλγορίθμου αναζήτησης αλφαριθμητικών σε προγραμματιζόμενους επεξεργαστές με γρήγορο δίκτυο επικοινωνίας όπως τα συστήματα ειδικού σκοπού DSP (Digital Signal Processor) ή άλλη διάταξη επεξεργαστών.

Κεφάλαιο 9

Συμπεράσματα

Σε αυτό το κεφάλαιο συνοψίζουμε τα αποτελέσματα της διατριβής μας και επίσης παρουσιάζουμε ορισμένες προτάσεις για μελλοντική έρευνα.

9.1 Αποτελέσματα

Ο στόχος της διδακτορικής μας διατριβής ήταν να αναπτύξουμε μεθόδους και τεχνικές υψηλής απόδοσης για να επιταχύνουμε τις εφαρμογές της ακριβούς και προσεγγιστικής αναζήτησης αλφαριθμητικών σε μεγάλες αδόμητες βάσεις κειμένων. Τα κύρια σημεία αυτής της διατριβής είναι τα εξής:

1. Μελετήσαμε και κατηγοριοποιήσαμε τους ακολουθιακούς αλγορίθμους της ακριβούς και προσεγγιστικής αναζήτησης αλφαριθμητικών. Στην συνέχεια εκτελέσαμε πειράματα των αλγορίθμων και με βάση τα αποτελέσματα δώσαμε πειραματικούς χάρτες τόσο για το πρόβλημα της ακριβούς αναζήτησης όσο για το πρόβλημα της προσεγγιστικής αναζήτησης αλφαριθμητικών. Οι πειραματικοί χάρτες δείχνουν τις περιοχές υπεροχής διαφόρων αλγορίθμων σύμφωνα με ορισμένες παραμέτρους όπως το μέγεθος του αλφαβήτου, το μήκος του προτύπου και ο αριθμός των αποτυχιών ή διαφορών. Η παραπάνω μελέτη συμπίπτει χρονικά με την αντίστοιχη έρευνα του Navvaro [129], αλλά είναι προφανώς ανεξάρτητη. Η μελέτη μας διαφέρει από την έρευνα του Navvaro στο γεγονός ότι χρησιμοποιήσαμε τέσσερις διαφορετικούς τύπους δεδομένων όπως δυαδικό αλφάβητο,

αλφάβητο μεγέθους 8, αγγλικό αλφάβητο και αλφάβητο DNA σε σχέση με το Navvaro που χρησιμοποίησε δύο τύπους κειμένων δηλαδή αλφάβητο DNA και αγγλικό αλφάβητο. Επίσης, στην έρευνα μας συμπεριλάβαμε βασικούς αλγορίθμους από κάθε κατηγορία για τα δύο προβλήματα αναζήτησης ενώ ο Navvaro χρησιμοποίησε περισσότερο τους πρόσφατους (όπως οι αλγόριθμοι bit-parallelism) και λιγότερο τους παλιότερους αλγορίθμους μόνο για το πρόβλημα αναζήτησης με k διαφορές. Τέλος, τα αποτελέσματα της πειραματικής μας μελέτης συμφωνούν γενικά με τα αποτελέσματα του Navvaro.

2. Παρουσιάσαμε τέσσερις παράλληλες και κατανεμημένες υλοποιήσεις για αναζήτηση αλφαριθμητικών σε μια σύγχρονη παράλληλη αρχιτεκτονική κατανεμημένης μνήμης όπως η συστοιχία από ομοιογενείς σταθμούς εργασίας. Επίσης, αναπτύξαμε ένα μαθηματικό μοντέλο απόδοσης για τις παράλληλες και κατανεμημένες υλοποιήσεις που επιβεβαιώθηκε με τα πειραματικά αποτελέσματα. Η μέγιστη διαφορά ανάμεσα στα θεωρητικά και πειραματικά αποτελέσματα ήταν λιγότερη από 10%. Η παραπάνω έρευνα διαφέρει από τις προηγούμενες σχετικές έρευνες στο γεγονός ότι δεν έχουν γίνει συγκεριμένες και εκτεταμένες μελέτες για υλοποιήσεις αλγορίθμων αναζήτησης αλφαριθμητικών σε συστοιχίες υπολογιστών. Προηγούμενες έρευνες [62, 65, 148, 172, 79] αφορούσαν κυρίως υλοποιήσεις αλγορίθμων βιολογικής ανάλυσης ακολουθίας που είναι παραλλαγές του προβλήματος προσεγγιστικής αναζήτησης σε παραδοσιακά παράλληλα συστήματα, όπως Transputer arrays, Cray Y-MP, Intel iPSC/860, nCUBE, Sequent Symmetry και Intel Hypercube. Οι παραπάνω υλοποιήσεις βασίστηκαν μόνο στο δυναμικό μοντέλο συντονιστής - εργαζόμενος χωρίς τη παρουσίαση γενικών μοντέλων και αποτελεσμάτων ενώ σε αυτή την διατριβή παρουσιάσαμε μελέτες και αποτελέσματα για υλοποιήσεις βασισμένες σε στατικά, δυναμικά και υβριδικά μοντέλα.

3. Υλοποιήσαμε τις τεχνικές του σημείου 2 σε μια πιο γενική παράλληλη και κατανεμημένη αρχιτεκτονική όπως η συστοιχία από ετερογενείς σταθμούς εργασίας. Στην συνέχεια, περιγράψαμε ένα θεωρητικό μοντέλο ανάλυσης απόδοσης σε συστοιχία ετερογενών σταθμών εργασίας. Το μοντέλο αυτό έχει σημαντικά πλεονεκτήματα όπως ότι είναι απλό και ότι προβλέπει με ακρίβεια την απόδοσης μιας παράλληλης και κατανεμημένης υλοποίησης αναζήτησης αλφαριθμητικών. Σε αυτό το σημείο δεν έχει γίνει προηγούμενη έρευνα για το πρόβλημα αναζήτησης αλφαριθμητικών ή για το πρόβλημα της βιολογικής ανάλυσης ακολουθίας σε ετερογενή συστοιχία υπολογιστών. Όμως,

έχουν πραγματοποιηθεί έρευνες για υλοποιήσεις αλγορίθμων σε ετερογενή συστοιχία σταθμών εργασίας σε άλλες περιοχές προβλημάτων, όπως η γραμμική άλγεβρα [17, 18]. Επίσης, σε αυτή τη διατριβή τροποποιήσαμε την υλοποίηση του υβριδικού μοντέλου προκειμένου να λάβουμε τις διαφορετικές ταχύτητες επεξεργασίας που έχουν οι σταθμοί εργασίας. Τέλος, το θεωρητικό μοντέλο απόδοσης που αναπτύξαμε βασίζεται στην προηγούμενη εργασία [169] η οποία προβλέπει τους χρόνους εκτέλεσης μιας εφαρμογής με θεωρία ουρών και προσομοίωσης ενώ στο δικό μας μοντέλο πραγματοποιεί προβλέψεις με μαθηματικό τρόπο.

4. Παρουσιάσαμε αρχιτεκτονικές ειδικού σκοπού για την λύση των προβλημάτων ακριβούς και προσεγγιστικής αναζήτησης αλφαριθμητικών χρησιμοποιώντας την μεθοδολογία απεικόνισης που βασίζεται στο γράφο εξάρτησης δεδομένων [72]. Οι αρχιτεκτονικές μας υλοποιούν αλγορίθμους κυρίως από την περιοχή ανάκτησης πληροφοριών σε σύγκριση με άλλες αρχιτεκτονικές όπως [88, 90, 143, 142, 99, 77, 51, 78] που οι περισσότερες από αυτές υλοποιήθηκαν για αλγορίθμους για ανάλυση ακολουθίας DNA. Επίσης, οι αρχιτεκτονικές μας υλοποιούν μια καινούργια κατηγορία αλγορίθμων αναζήτησης αλφαριθμητικών, τους αλγορίθμους bit-παραλληλισμού σε σχέση με προηγούμενες αρχιτεκτονικές που παραλληλοποιούν τους κλασσικούς αλγορίθμους δυναμικού προγραμματισμού. Οι υλοποιήσεις των αλγορίθμων bit-παραλληλισμού σε αρχιτεκτονικές αντιμετωπίζουν το πρόβλημα της αναζήτησης περιορισμένων εκφράσεων αποτελεσματικά. Επιπλέον, οι αρχιτεκτονικές έχουν το πλεονέκτημα ότι μπορούν να εκτελέσουν συνδυαστική αναζήτηση όπως προσεγγιστική αναζήτηση περιορισμένων εκφράσεων. Τέλος, έχουμε παρουσιάσει μια αρχιτεκτονική για την υλοποίηση ενός πρόσφατου και απλού αλγορίθμου για το πρόβλημα της μακρύτερης κοινής υποακολουθίας σε σύγκριση με άλλες αρχιτεκτονικές, όπως [140, 170, 86, 82, 83, 87, 94] που βασίζονται στην παράλληλη υλοποίηση ενός κλασσικού αλγορίθμου δυναμικού προγραμματισμού.

5. Περιγράψαμε μια ευέλικτη προγραμματιζόμενη αρχιτεκτονική γενικού σκοπού που μπορεί να εκτελεί μια κλάση αλγορίθμων αναζήτησης αλφαριθμητικών. Η αρχιτεκτονική αυτή έχει τη δυνατότητα να εκτελεί όλα τα προβλήματα αναζήτησης (όπως ακριβούς, προσεγγιστικής και περιορισμένων εκφράσεων) χωρίς να απαιτείται επανασχεδίαση ξεχωριστά για κάθε αλγόριθμο. Τέλος, παρουσιάσαμε ένα μοντέλο απόδοσης για την προγραμματιζόμενη αρχιτεκτονική το οποίο επιβεβαιώθηκε με τα πειραματικά αποτελέσματα σε μια συστοιχία υπολογιστών. Με βάση αυτό, το

μοντέλο που αναπτύξαμε μπορεί να χρησιμοποιηθεί για να προβλέψουμε με ακρίβεια την απόδοση ενός αλγορίθμου αναζήτησης σε συστήματα ειδικού σκοπού όπως DSP ή άλλη διάταξη επεξεργαστών. Πρέπει να σημειωθεί εδώ ότι ακολουθήσαμε την μεθοδολογία του Moreno και Lang [121] για την σχεδίαση της προγραμματιζόμενη αρχιτεκτονικής γενικού σκοπού και του μοντέλου απόδοσης. Επίσης, οι Moreno και Lang είχαν σχεδιάσει μια παρόμοια αρχιτεκτονική κυρίως για αλγορίθμους γραμμικής άλγεβρας.

9.2 Μελλοντική Έρευνα

Σε αυτή την ενότητα παρουσιάζουμε ορισμένες προτάσεις για περαιτέρω έρευνα.

9.2.1 Υβριδική Αρχιτεκτονική

Σκοπεύουμε να αναπτύξουμε μια υβριδική αρχιτεκτονική υψηλής απόδοσης για αναζήτηση αλφαριθμητικών που συνδυάζει τις προσεγγίσεις της συστοιχίας από ομοιογενείς σταθμούς εργασίας και της προτεινόμενης προγραμματιζόμενης αρχιτεκτονικής έτσι ώστε να πετύχουμε μια υψηλή ταχύτητα σε χαμηλό κόστος. Έτσι, συνδυάζουμε τις μηχανές SIMD και MIMD μέσα σε μια ενιαία παράλληλη αρχιτεκτονική. Με άλλα λόγια, μέσα στους επεξεργαστές μιας συστοιχίας σταθμών εργασίας (MIMD) ενσωματώνεται η προγραμματιζόμενη διάταξη επεξεργαστών (SIMD) για να επιταχύνουμε τις απαιτητικές υπολογιστικές εργασίες. Το βασικό κίνητρο για την υιοθέτηση του υβριδικού υπολογισμού προήλθε από τον λόγο τιμής/απόδοσης. Είναι γνωστό ότι η χρήση συστοιχίας σταθμών εργασίας είναι ένας από τους απλούς και αποτελεσματικούς τρόπους για να αποκτήσουμε υπερυπολογιστική ισχύ σε λογικό κόστος. Η ισχύς μπορεί να βελτιωθεί σημαντικά και συγχρόνως το κόστος να παραμείνει σχετικά χαμηλό ενσωματώνοντας μια προγραμματιζόμενη διάταξη επεξεργαστών (τύπου DSP) σε κάθε σταθμό εργασίας της συστοιχίας. Η διάταξη μπορεί να έχει τη μορφή μιας κάρτας επέκτασης συνεπεξεργαστή που τοποθετείται σε κάποια αντίστοιχη θέση του σταθμού εργασίας.

Η διαδικασία απεικόνισης μιας εφαρμογής στο υβριδικό σύστημα αποτελείται από δύο επίπεδα παραλληλισμού: την παραλληλοποίηση στην προγραμματιζόμενη αρχιτεκτονική και την παραλληλοποίηση στην συστοιχία σταθμών εργασίας. Η παραλληλοποίηση στην προγραμματιζόμενη αρχιτεκτονική πα-

ραλληλοποιεί ένα συγκεριμένο αλγόριθμο αναζήτησης αλφαριθμητικών από το ρεπετόριο αλγορίθμων που διαθέτει. Στο επίπεδο της παραλληλοποίησης σε συστοιχία ακολουθεί το προγραμματιστικό μοντέλο συντονιστής - εργαζόμενος που ο συντονιστής χωρίζει την αδόμητη βάση κειμένων σε ένα σύνολο από μικρότερα υπο-κείμενα και διανέμει τα υπο-κείμενα στους εργαζόμενους χρησιμοποιώντας μια κατάλληλη στρατηγική διανομής. Οι στρατηγικές διανομής που μπορούν να χρησιμοποιηθούν σε αυτό το σύστημα είναι η στατική, η δυναμική και η υβριδική όπως έχουμε παρουσιάσει στα κεφάλαια 5 και 6. Έτσι, παρακάτω δίνουμε μια γενική περιγραφή για την παραλληλοποίηση της αναζήτησης αλφαριθμητικών στο υβριδικό σύστημα.

Ο συντονιστής της συστοιχίας πρώτα εκπέμπει το πρότυπο αλφαριθμητικό σε κάθε εργαζόμενο ή σταθμό εργασίας και επίσης επιλέγει τον αλγόριθμο αναζήτησης αλφαριθμητικών που θα εκτελέσει ο κάθε σταθμός εργασίας. Στην συνέχεια, ο συντονιστής χωρίζει την αδόμητη βάση κειμένων σε ένα σύνολο από μικρότερα υπο-κείμενα και διανέμει τα υπο-κείμενα στους εργαζόμενους χρησιμοποιώντας μια οποιαδήποτε στρατηγική διανομής (όπως είδαμε στα κεφάλαια 5 και 6). Μετά, το υπο-κείμενο που λαμβάνει κάθε εργαζόμενος μεταφέρεται στην προγραμματιζόμενη αρχιτεκτονική που έχει τοποθετηθεί μέσω της μονάδας διεπαφής. Στην συνέχεια, κάθε εργαζόμενος εκτελεί αναζήτηση αλφαριθμητικών για ένα υπο-κείμενο που λαμβάνει από τη μονάδα διεπαφής χρησιμοποιώντας την προγραμματιζόμενη αρχιτεκτονική. Τέλος, κάθε εργαζόμενος επιστρέφει το αποτέλεσμα από την αναζήτηση αλφαριθμητικών πίσω στην μονάδα διεπαφής και στη συνέχεια στο συντονιστή.

Πρέπει να σημειώσουμε ότι η υβριδική αρχιτεκτονική είναι κατάλληλη μόνο για απαιτητικές εφαρμογές που περιέχουν ένα μεγάλο αριθμό από εργασίες που είναι κατάλληλες για εκτέλεση στην προγραμματιζόμενη διάταξη επεξεργαστών. Επίσης, στην υβριδική αρχιτεκτονική μπορούν να απεικονιστούν μόνο απλοί, επαναληπτικοί και απαιτητικοί αλγόριθμοι που είναι κατάλληλοι για εκτέλεση στη διάταξη επεξεργαστών.

Παρακάτω αναλύουμε σύντομα την απόδοση της υβριδικής αρχιτεκτονικής. Για την ανάλυση υποθέτουμε ότι η συστοιχία της υβριδικής αρχιτεκτονικής μας αποτελείται από ομοιογενείς σταθμούς εργασίας και η στρατηγική διανομής κειμένων που εφαρμόζει ο συντονιστής είναι στατική. Έτσι, η ανάλυση απόδοσης που περιγράφουμε παρακάτω συνδυάζει την ανάλυση απόδοσης της παράλληλης υλοποίησης με στατική ή υβριδική διανομή σε ομοιογενές σύστημα (κεφάλαιο 5) με την ανάλυση απόδοσης της προγραμματιζόμενης υλοποίησης (κεφάλαιο 8).

Ο συνολικός χρόνος εκτέλεσης για την υλοποίηση ενός οποιουδήποτε αλγορίθμου αναζήτησης αλφαριθμητικών σε μια υβριδική αρχιτεκτονική αποτελείται από πέντε φάσεις:

- T_a είναι ο χρόνος επικοινωνίας για να εκπέμψει το πρότυπο αλφαριθμητικό και τον αριθμό των διαφορών ή αποτυχιών k σε όλους τους σταθμούς εργασίας. Ο χρόνος T_a είναι παρόμοιος με την εξίσωση 5.19 και ορίζεται ως εξής:

$$T_a = log_2 p(\alpha + (m+1)\beta) \tag{9.1}$$

- T_b είναι ο συνολικός χρόνος Ε/Ε για να διαβάσει ένα υπο-κείμενο από τον τοπικό δίσκο ενός σταθμού εργασίας. Ο χρόνος T_b είναι παρόμοιος με την εξίσωση 5.20 και ορίζεται ως εξής:

$$T_b = (\lceil \frac{n}{p} \rceil + m - 1)\gamma \tag{9.2}$$

- $T_{interface-in}$ είναι ο χρόνος επικοινωνίας εισόδου για να μεταφερθεί ένα υπο-κείμενο ενός σταθμού εργασίας στην προγραμματιζόμενη διάταξη επεξεργαστών. Γνωρίζουμε ότι το μέγεθος του υπο-κειμένου του κάθε σταθμού εργασίας περιέχει $(\lceil \frac{n}{p} \rceil + m - 1)$ χαρακτήρες. Συνεπώς, ο χρόνος επικοινωνίας εισόδου δίνεται από την εξίσωση:

$$T_{interface-in} = (\lceil \frac{n}{p} \rceil + m - 1)t_{interface} \tag{9.3}$$

  όπου $t_{interface}$ είναι ο μέσος χρόνος για την μεταφορά δεδομένων της μονάδας διεπαφής. Ο χρόνος αυτός εξαρτάται από τα χαρακτηριστικά της μονάδας διεπαφής που χρησιμοποιείται στην υβριδική αρχιτεκτονική.

- $T_{p-array}$ είναι ο συνολικός χρόνος αναζήτησης αλφαριθμητικών στην διάταξη επεξεργαστών του κάθε σταθμού εργασίας. Ο χρόνος $T_{p-array}$ είναι παρόμοιος με την εξίσωση 8.8 και ορίζεται ως εξής:

$$T_{p-array} = T_{prep} + T_{searchcomp} + T_{comm} \tag{9.4}$$

- $T_{interface-out}$ είναι ο χρόνος επικοινωνίας εξόδου για να μεταφερθούν τα αποτελέσματα της αναζήτησης από την διάταξη επεξεργαστών στον σταθμό εργασίας. Τα αποτελέσματα που παράγει η

προγραμματιζόμενη διάταξη επεξεργαστών είναι $(\lceil \frac{n}{p} \rceil + m - 1)$. Συνεπώς, ο χρόνος επικοινωνίας εξόδου δίνεται από την εξίσωση:

$$T_{interface-out} = (\lceil \frac{n}{p} \rceil + m - 1) t_{interface} \tag{9.5}$$

- T_d είναι ο χρόνος επικοινωνίας για να συλλέγουν τα αποτελέσματα από τους p σταθμούς εργασίας. Ο χρόνος T_d είναι παρόμοιος με την εξίσωση 5.22 και ορίζεται ως εξής:

$$T_d = log_2 p(\alpha + \beta) \tag{9.6}$$

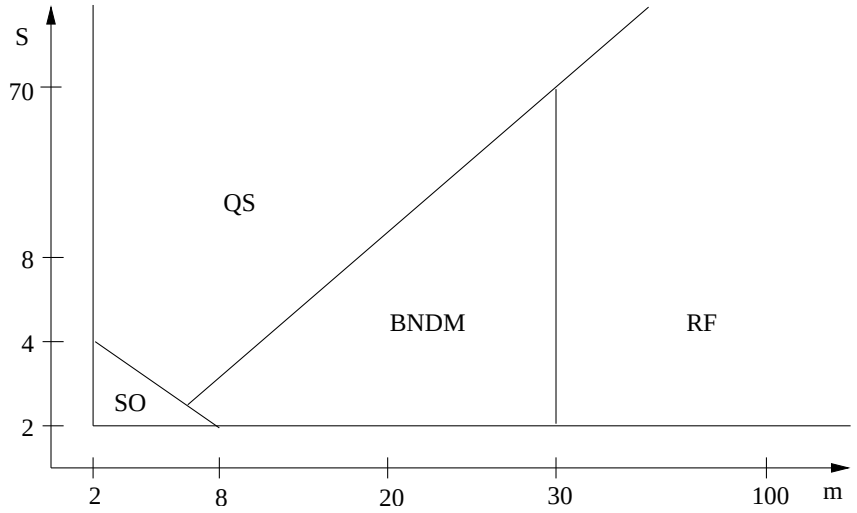
Ο χρόνος εκτέλεσης ενός οποιουδήποτε αλγορίθμου αναζήτησης αλφαριθμητικών στην υβριδική αρχιτεκτονική είναι το άθροισμα των παραπάνω όρων και δίνεται από:

$$T_{hybrid} = T_a + T_b + T_{interface-in} + T_{p-array} + T_{interface-out} + T_d \tag{9.7}$$

9.2.2 Προσαρμοζόμενη Κατανεμημένη Υλοποίηση

Μια άλλη ενδιαφέρουσα επέκταση είναι η ανάπτυξη μιας προσαρμοζόμενης κατανεμημένης υλοποίησης για το πρόβλημα αναζήτησης αλφαριθμητικών σε μια συστοιχία σταθμών εργασίας. Συγκεριμένα, ένας κεντρικός υπολογιστής μέσω μιας συνοπτικής ή διαλογικής διεπαφής μπορεί να προσφέρει στο χρήστη μια σειρά από επιλογές διαφόρων παραμέτρων. Οι παράμετροι αυτοί μπορεί να είναι οι εξής:

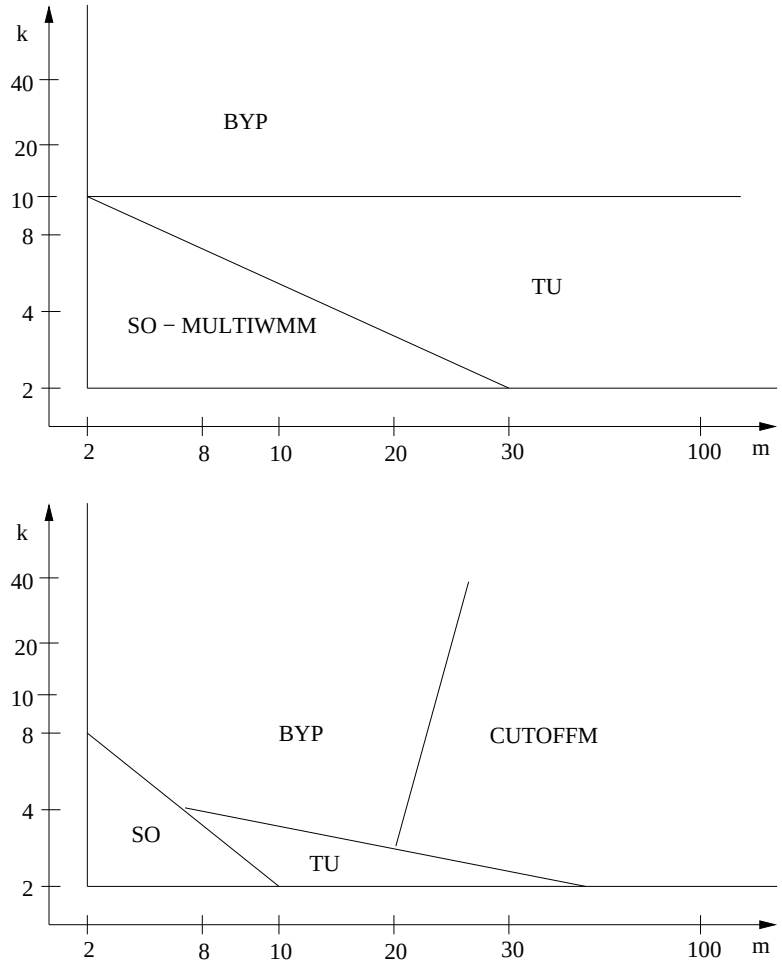
- ορισμός του αλφαβήτου του κειμένου,
- μεταφορά αρχείων κειμένου και απεικόνιση αποτελεσμάτων,
- ορισμός προτύπου ή σειράς προτύπων,
- επιλογή του είδους της αναζήτησης όπως ακριβής ή προσεγγιστική αναζήτηση,
- ορισμός του αριθμού των αποτυχιών ή των διαφορών αν πρόκειται για προσεγγιστική αναζήτηση και
- επιλογή της στρατηγικής διανομής των κειμένων όπως στατική, δυναμική ή υβριδική.



Σχήμα 9.1: Χάρτης πειραματικής απόδοσης για διαφορετικούς αλγορίθμους αναζήτησης αλφαριθμητικών

Επίσης, στο κεντρικό υπολογιστή θα υπάρχει ένα ολοκληρωμένο σύστημα κανόνων που θα περιέχει τους πειραματικούς χάρτες για τα προβλήματα της ακριβούς και προσεγγιστικής αναζήτησης αλφαριθμητικών. Στο Σχήμα 9.1 φαίνεται ο χάρτης για τους αλγορίθμους ακριβούς αναζήτησης ενώ στα Σχήματα 9.2 και 9.3 φαίνονται οι χάρτες για τους αλγορίθμους προσεγγιστικής αναζήτησης. Στη συνέχεια, ο κεντρικός υπολογιστής με βάση τις απαιτήσεις του χρήστη δηλαδή τις παραμέτρους και με την βοήθεια του συστήματος κανόνων μπορεί να επιλέξει τον κατάλληλο αλγόριθμο τον οποίο θα εκτελέσουν οι υπόλοιποι σταθμοί εργασίας της συστοιχίας. Για παράδειγμα, αν το αλφάβητο κειμένου είναι αγγλικό, το μήκος του προτύπου είναι 30 και ο χρήστης επιλέξει ακριβή αναζήτηση, τότε το σύστημα των κανόνων θα επιλέξει τον αλγόριθμο BNDM σύμφωνα με το Σχήμα 9.1. Επίσης, αν το αλφάβητο είναι δυαδικό, το μήκος του προτύπου είναι 5, ο χρήστης επιλέξει προσεγγιστική αναζήτηση και ο αριθμός των διαφορών είναι 6, τότε το σύστημα των κανόνων θα επιλέξει τον αλγόριθμο BYN.

Η παραπάνω κατανεμημένη υλοποίηση μπορεί να επεκταθεί και σε υπολογισμούς πλέγματος (grid computing) που περιέχουν επιμέρους συστοιχίες σταθμών εργασίας. Σε αυτήν την περίπτωση με βάση τις παραμέτρους του χρήστη μπορεί κάθε συστοιχία να εκτελεί το δικό της είδος αναζήτησης δηλαδή μια συστοιχία να εκτελεί ακριβή αναζήτηση και μια άλλη συστοιχία να εκτελεί προσεγγιστική αναζήτηση. Επίσης, μπορεί κάθε συστοιχία να εκτελεί αναζήτηση στο δικό της αλφάβητο δηλαδή μια συστοιχία να εκτελεί αναζήτηση σε αγγλική βάση κειμένων, σε άλλη συστοιχία να εκτελεί αναζήτηση σε βάση

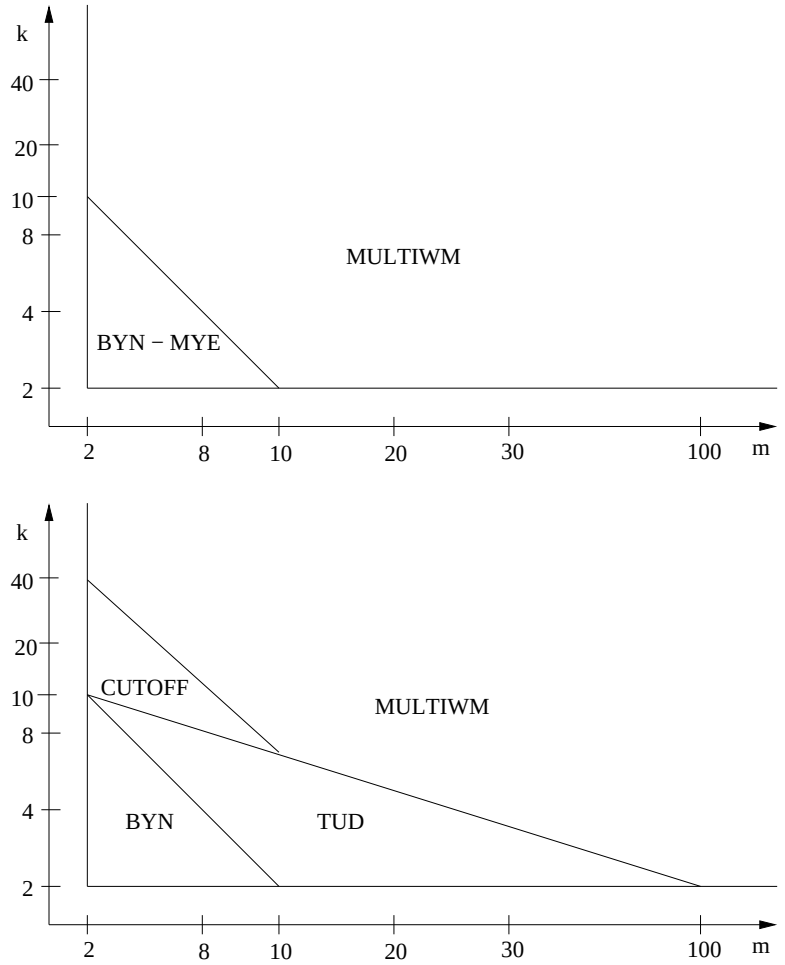


Σχήμα 9.2: Περιοχές όπου κάθε αλγόριθμος για το πρόβλημα με k αποτυχίες είναι αποτελεσματικός. Πάνω είναι για το αγγλικό αλφάβητο και κάτω είναι για το δυαδικό αλφάβητο.

δεδομένων DNA, σε μια άλλη συστοιχία να εκτελεί αναζήτηση σε ελληνική βάση κειμένων κλπ.

9.2.3 Μοντέλα Απόδοσης σε Γενικευμένη Ετερογενή Συστοιχία Υπολογιστών

Παραπέρα, στα πλαίσια του υπολογισμού πλέγματος θα ήταν ενδιαφέρον να αναπτύξουμε ένα θεωρητικό μοντέλο απόδοσης για τις τέσσερις παράλληλες υλοποιήσεις (που παρουσιάσαμε στα κεφάλαια 5 και 6) σε μια πιο γενικευμένη ετερογενή παράλληλη αρχιτεκτονική. Σε ένα τέτοιο σύστημα θα μπορούσε να υπάρχει ετερογένεια όχι μόνο ως προς στις ταχύτητες των σταθμών εργασίας όπως ασχοληθήκαμε σε



Σχήμα 9.3: Περιοχές όπου κάθε αλγόριθμος για το πρόβλημα με k διαφορές είναι αποτελεσματικός. Πάνω είναι για το αγγλικό αλφάβητο και κάτω είναι για το δυαδικό αλφάβητο.

αυτή την διατριβή αλλά και ως προς στο δίκτυο επικοινωνίας καθώς και σε άλλους παραμέτρους όπως ταχύτητες δίσκων, ταχύτητες μνήμης, ταχύτητες πολυεπίπεδων δικτύων κλπ.

Επίσης, θα μπορούσαμε να αναπτύξουμε μια παράλληλη υλοποίηση αναζήτησης διαφόρων προτύπων σε μια πιο γενικευμένη ετερογενή βάση κειμένων. Με άλλα λόγια, σε ορισμένους σταθμούς εργασίας της συστοιχίας να υπάρχουν διαφορετικού τύπου αρχεία όπως συμπιεσμένα αρχεία, αρχεία ιστοσελίδων (html), αρχεία εικόνων και μουσικά αρχεία. Έτσι, η αναζήτηση πάνω σε τέτοια διαφορετικού τύπου αρχεία μπορεί να γίνει χρησιμοποιώντας τους αλγορίθμους αναζήτησης αλφαριθμητικών με κάποιες ελάχιστες τροποποιήσεις όπως φαίνονται στις αναφορές [59, 68, 122, 84, 66, 31, 85, 131].

Επιπλέον, και πάλι στα πλαίσια της τεχνολογίας υπολογιστικού πλέγματος θα ήταν ενδιαφέρον να αναπτύξουμε υλοποιήσεις ειδικά για την αναζήτηση αλφαριθμητικών που να είναι ανεκτές σε σφάλματα (fault tolerance). Παραδείγματα τέτοιων σφαλμάτων είναι τα παρακάτω:

- μεταβατικά σφάλματα υπολογισμού, αποθήκευσης ή επικοινωνίας και
- προσωρινή ή οριστική διακοπή λειτουργίας ή επικοινωνίας κόμβων ή συστοιχιών.

9.2.4 Δισδιάστατη Αναζήτηση και Βιοπληροφορική

Οι τεχνικές που περιγράφηκαν σε αυτή την διατριβή (από το κεφάλαιο 5 μέχρι το κεφάλαιο 9) μπορούν να εφαρμοστούν σε άλλα προβλήματα της αναζήτησης αλφαριθμητικών όπως για παράδειγμα η δισδιάσταστη αναζήτηση αλφαριθμητικών (two dimensional text searching). Επιπλέον, οι τεχνικές μπορούν να εφαρμοστούν σε μια νέα ερευνητική περιοχή όπως αυτής της βιοπληροφορικής. Οι εξελίξεις στην βιοπληροφορική είχαν σαν αποτέλεσμα τη δημιουργία μεγάλων βιολογικών βάσεων δεδομένων όπως η GenBank [19] και η SWISS-PROT [16]. Οι παραπάνω βάσεις δεδομένων περιέχουν ένα μεγάλο αριθμό από ακολουθίες γονιδίων και πρωτεινών καθώς και βιολογικές και βιβλιογραφικές πληροφορίες. Η σάρωση (scanning) των παραπάνων βάσεων δεδομένων είναι μια κοινή και επαναληπτική εργασία στην βιοπληροφορική. Έτσι, στο πεδίο της βιοπληροφορικής υπάρχει ανάγκη να επιταχυνθεί η σάρωση αφού οι βάσεις δεδομένων αυξάνονται με εκθετικό ρυθμό, για παράδειγμα, κάθε χρόνο διπλασιάζονται. Η εργασία σάρωσης αποσκοπεί στην εύρευση των ομοιοτήτων ανάμεσα σε μια ακολουθία προτύπου και σε ακολουθίες της βιολογικής βάσης δεδομένων. Η παραπάνω εργασία δίνει τη δυνατότητα στους βιολόγους να ανακαλύψουν ή να εντοπίσουν ακολουθίες που διαμοιράζουν κοινές υποακολουθίες.

Παράρτημα Α΄

Κώδικας των Αλγορίθμων Αναζήτησης Αλφαριθμητικών

Α.1 Ακολουθιακοί Αλγόριθμοι Ακριβούς Αναζήτησης

Α.1.1 Αλγόριθμος BF

```
void bf_search(char *pattern, char *text, int m, int n, int *count)
{
        int i,j;
        int matches=0;

        i=j=0;
        while(i<=n-m) {
                if (text[i]==pattern[j]) {i++;j++;}
                else {i=i-j+1;j=0;}
                if (j>=m) matches++;
        }
        *count=matches;
```

```
}
```

Α.1.2 Αλγόριθμος KMP

```
void kmp_search(char *pattern, char *text, int m, int n, int *count)
{
        int i,j,kmp_next[ASIZE];
        int matches=0;

        /* preprocessing phase */
        i=0;
        j=kmp_next[0]=-1;
        while (i<m) {
                while (j>-1 && pattern[i]!=pattern[j]) j=kmp_next[j];
                i++;
                j++;
                if (pattern[i]==pattern[j]) kmp_next[i]=kmp_next[j];
                else kmp_next[i]=j;
        }

        /* searching phase */
        i=j=0;
        while (i<n) {
                while (j>-1 && pattern[j]!=text[i]) {j=kmp_next[j];}
                i++;j++;
                if (j==m) {
                        matches++;
                        j=kmp_next[j];
                }
        }
```

```
        *count=matches;
}
```

Α.1.3 Αλγόριθμος BM

```
void bm_search(char *pattern, char *text, int m, int n, int *count)
{
        int delta1[ASIZE],delta2[ASIZE];
        int a,i,j,k,t,t1,q,q1,f[ASIZE];

        /* preprocessing phase */
        for(a=0;a<ASIZE;a++) delta1[a]=m;
        for(j=0;j<m-1;j++) delta1[pattern[j]]=m-j-1;

        for(k=1;k<=m;k++) delta2[k-1]=2*m-k;
        for(j=m,t=m+1;j>0;j--,t--) {
                f[j-1]=t;
                while (t<=m && pattern[j-1]!=pattern[t-1]) {
                        delta2[t-1]=MIN(delta2[t-1],m-j);
                        t=f[t-1];
                }
        }
        q=t;t=m+1-q;q1=1;
        for(j=1,t1=0;j<=t;t1++,j++) {
        f[j-1]=t1;
                while (t1>=1 && pattern[j-1]!=pattern[t1-1]) t1=f[t1-1];
        }
        while (q<m) {
                for(k=q1;k<=q;k++) delta2[k-1]=MIN(delta2[k-1],m+q-k);
                q1=q+1;q=q+t-f[t-1];t=f[t-1];
```

```
        }

        /* searching phase */
        i=m-1;
        while (i<n) {
                j=m-1;
                while (j>=0 && pattern[j]==text[i]) {--j;--i;}
                if (j<0) {
                        matches++;
                        i+=delta2[0]+1;
                }
                else i+=MAX((delta2[j]), (delta1[text[i]]));
        }
        *count=matches;
}
```

A.1.4 Αλγόριθμος Turbo-BM

```
void tbm_search(char *pattern, char *text, int m, int n, int *count)
{
        int delta1[ASIZE], delta2[ASIZE];
        int i,j,u,shift,v,turbo_shift,delta1_shift;
        int matches=0;

        /* preprocessing phase */

        /* searching phase */
        i=u=0;
        shift=m;
        while (i<=n-m) {
```

```
                j=m-1;
                while (j>=0 && pattern[j]==text[i+j]) {
                        --j;
                        if (u!=0 && j==m-1-shift) j-=u;
                }
                if (j<0) {matches++;
                          shift=delta2[0];
                          u=m-shift;}
                else {
                        v=m-1-j;
                        turbo_shift=u-v;
                        delta1_shift=delta1[text[i+j]]-m+j+1;
                        shift=MAX(turbo_shift,delta1_shift);
                        shift=MAX(shift,delta2[j+1]);
                        if (shift==delta2[j+1]) u=MIN((m-shift),v);
                        else {
                                if (turbo_shift<delta1_shift) shift=MAX(shift,(u+1));
                                u=0;
                        }
                }
                i+=shift;
        }
        *count=matches;
}
```

A.1.5 Αλγόριθμος BMH

```
void bmh_search(char *pattern, char *text, int m, int n, int *count)
{
        int d[ASIZE];
```

```
        int i,k,j,matches=0;

        /* preprocessing phase */
        for(j=0;j<ASIZE;j++) d[j]=m;
        for(j=0;j<m-1;j++) d[pattern[j]]=m-j-1;

        /* searching phase */
        i=m-1;
        while (i<=n) {
                k=i;j=m-1;
                while (j>=0 && text[k]==pattern[j]) {j--;k--;}
                if (j<0) matches++;
                i+=d[text[i]];
        }
        *count=matches;
}
```

A.1.6 Αλγόριθμος QS

```
void qs_search(char *pattern, char *text, int m, int n, int *count)
{
        int d1[ASIZE];
        int i,j,k,matches=0;

        /* preprocessing phase */
        for(i=0;i<ASIZE;i++) d1[i]=m+1;
        for(i=0;i<m;i++) d1[pattern[i]]=m-i;

        /* searching phase */
        i=0;
```

```
        while (i<n) {
                k=i;j=0;
                while (j<m && text[k]==pattern[j]) {j++;k++;}
                if (j==m) matches++;
                i+=d1[text[i+m]];
        }
        *count=matches;
}
```

A.1.7 Αλγόριθμος BMS

```
void bms_search(char *pattern, char *text, int m, int n, int *count)
{
        int delta1[ASIZE], td1[ASIZE];
        int k,i,j,matches=0;

        /* preprocessing phase */
        for(i=0;i<ASIZE;i++) {
                delta1[i]=m;
                td1[i]=m+1;
        }
        for(i=0;i<m-1;++i) {
                delta1[pattern[i]]=m-i-1;
                td1[pattern[i]]=m-i;
        }

        /* searching phase */
        i=0;
        while (i<n) {
                k=i;j=0;
```

```
                while (j<m && text[k]==pattern[j]) {k++;j++;}
                if (j==m) {matches++;}
                i+=MAX(delta1[text[i+m-1]],td1[text[i+m]]);
        }
        *count=matches;
}
```

A.1.8 Αλγόριθμος RF

```
struct State {
        int length,suf,pos;
        char terminal;
};

typedef struct State STATE;
int state_counter,period;
STATE states[2*XSIZE+2];
int trans[XSIZE][ASIZE];

int COPY(int p,STATE states[], int trans[XSIZE][ASIZE])
{
        int r;

        r=state_counter++;
        memcpy(trans[r],trans[p],ASIZE*sizeof(int));
        states[r].suf=states[p].suf;
        states[r].pos=states[p].pos;
        return r;
}
```

```
int SUFF(char *pattern, int m, STATE states[], int trans[XSIZE][ASIZE])
{
        int i,art,init,last,p,q,r;
        char a;

        art=state_counter++;
        init=state_counter++;
        states[init].suf=art;
        last=init;
        for(i=m-1;i>=0;--i) {
                a=pattern[i];
                p=last;
                q=state_counter++;
                states[q].length=states[p].length+1;
                states[q].pos=states[p].pos+1;
                while (p!=init && trans[p][a]==0) {
                        trans[p][a]=q;
                        p=states[p].suf;
                }
                if (trans[p][a]==0) {
                        trans[init][a]=q;
                        states[q].suf=init;
                }
                else
                        if (states[p].length+1==states[trans[p][a]].length)
                                states[q].suf=trans[p][a];
                        else {
                                r=COPY(trans[p][a],states,trans);
                                states[r].length=states[p].length+1;
                                states[trans[p][a]].suf=r;
```

```
                                states[q].suf=r;
                                while (p!=art && states[trans[p][a]].length
                                        >= states[r].length) {
                                        trans[p][a]=r;
                                        p=states[p].suf;
                                }
                        }
                        last=q;
                }
                states[last].terminal=1;
                p=last;
                while (p!=init) {
                        p=states[p].suf;
                        states[p].terminal=1;
                }
        return init;
}

void rf_search(char *pattern, char *text, int m, int n, int *count)
{
        int i,j,shift,init,state,state_aux;
        int matches=0;

        /* preprocessing phase */
        memset(trans,0,2*(m+2)*ASIZE*sizeof(int));
        memset(states,0,2*(m+2)*sizeof(STATE));
        state_counter=1;
        period=m;

        /* searching phase */
```

```
        i=0;
        init=SUFF(pattern,m,states,trans);
        while (i<=n-m) {
                j=m-1;
                state=init;
                shift=m;
                while ((state_aux=trans[state][text[i+j]])!=0) {
                        state=state_aux;
                        if (states[state].terminal) {
                                period=shift;
                                shift=j;
                        }
                        j--;
                }
                if (j<0) {matches++;shift=period;}
                i+=shift;
        }
        *count=matches;
}
```

A.1.9 Αλγόριθμος SO

```
void so_search(char *pattern, char *text, int m, int n, int *count)
{
        int i, j, matches=0;
        unsigned int lim, T[ASIZE], state;

        /* preprocessing phase */
        for(i=0;i<ASIZE;i++) T[i]=~0;
        lim=0;
```

```
        for(i=0,j=1;i<m;i++,j<<=1) {
                T[pattern[i]]&=~j;
                lim|=j;
        }
        lim=~(lim>>1);

        /* searching phase */
        state=~0;
        for(i=0;i<n;i++) {
                state=(state<<1)|T[text[i]];
                if (state<lim) matches++;
        }
        *count=matches;
}
```

A.1.10 Αλγόριθμος BNDM

```
void bndm_search(char *pattern, char *text, int m, int n, int *count)
{
        int i,j,pos,last,matches=0;
        unsigned int B[ASIZE];
        unsigned int D, mask, mask1;

        /* preprocessing phase */
        mask=1;
        for(i=0;i<ASIZE;i++) B[i]=0;
        for(i=1;i<=m;i++,mask<<=1) B[pattern[m-i+1]]|=mask;

        /* searching phase */
        pos=0;
```

```
        mask1=1;
        mask1<<=m-1;
        while (pos<=n-m) {
                last=j=m-1;
                D=~0;
                while (D!=0) {
                        D&=B[text[pos+j]];
                        j--;
                        if ((D&mask1)!=0)
                                if (j>0) last=j;
                                else matches++;
                        D<<=1;
                }
                pos+=last;
        }
        *count=matches;
}
```

A.1.11 Αλγόριθμος KR

```
#define D 7
#define Q 16647133

void kr_search(char *pattern, char *text, int m, int n, int *count)
{
         int hp,ht,d,i,j,matches=0;

         d=1;
         for(i=1;i<m;i++) d=(d<<D)%Q;
         ht=hp=0;
```

```
        for(i=0;i<m;i++) {
                hp=((hp<<D)+pattern[i])%Q;
                ht=((ht<<D)+text[i])%Q;
        }

        for(i=0;i<=n-m;i++) {
                if (ht==hp) {
                        for(j=0;j<m && text[i+j]==pattern[j];j++) ;
                        if (j>=m) matches++;
                }
                ht=(ht+(Q<<D)-text[i]*d)%Q;
                ht=((ht<<D)+text[i+m])%Q;
        }
        *count=matches;
}
```

A.2 Ακολουθιακοί Αλγόριθμοι Προσεγγιστικής Αναζήτησης με k αποτυχίες

A.2.1 Αλγόριθμος BF

```
void bf_search(char *pattern, char *text, int m, int n, int k, int *count)
{
        int i,j,l,matches=0;

        for(i=0;i<n-m;i++) {
                for(l=j=1;j<m && l<=m;j++)
                        if (pattern[j]!=text[i+j]) l++;
                if (l<=k) matches++;
        }
```

```
        *count=matches;
}
```

A.2.2 Αλγόριθμος LV

```
#define p2logm          16
#define max_m_2k1       21

int pm[100][max_m_2k1];
int tm[150000][19];

void lv_search(char *pattern, char *text, int m, int n, int k, int *count)
{
        int i,r,j,b,z1,z2,matches=0;
        int p2ll;     /* p2ll=2^(l-1) */

        /* preprocessing phase */
        for(p2ll=1;p2ll<=m;p2ll*=2)
                level1_pat_analysis(pattern,m,k,p2ll);

        /* searching phase */
        for(z1=0;z1<=n-m;++z1)
            for(z2=1;z2<=k+1;++z2)
                tm[z1][z2]=m+1;
        r=j=0;
        for(i=0;i<=n-m;++i) {
                b=0;
                if (i<j)
                        merge(m,k,pattern,text,i,r,j,&b);
                if (b<k+1) {
```

```
                        r=i;
                        extend(m,k,pattern,text,i,&j,&b);
                        if (b<=k) matches++;
                }
        }
        *count=matches;
}

merge(int m, int k, char *pattern, char *text, int i, int r, int j, int *bp)
{
        int b=*bp;
        int d,f,q;

        for(q=1;tm[r][q]<=i-r;++q);
        d=q;
        f=1;
        while (!(b==k+1 || d==k+2 || (i+pm[i-r][f]>j && tm[r][d]==m+1))) {
                if (i+pm[i-r][f]>r+tm[r][d]) {
                        tm[i][++b]=tm[r][d]-(i-r);
                        ++d;
                } else
                if (i+pm[i-r][f]<r+tm[r][d]) {
                        tm[i][++b]=pm[i-r][f];
                        ++f;
                } else {
                if (pattern[pm[i-r][f]]!=text[i+pm[i-r][f]])
                        tm[i][++b]=pm[i-r][f];
                ++f;++d;
                }
        }
```

```
        *bp=b;
}

extend(int m, int k, char *pattern, char *text, int i, int *jp, int *bp)
{
        int j=*jp,b=*bp;

        while (b<k+1 && j-i<m) {
                ++j;
                if (text[j]!=pattern[j-i])
                        tm[i][++b]=j-i;
        }
        *jp=j;*bp=b;
}

level1_pat_analysis(char *pattern, int m, int k, int p2ll)
{
        int p2l,lim,lim2,z1,z2,i,r,j,b;

        p2l=p2ll*2;
        lim=p2logm/p2l*2*k+1;
        if (lim>m-p2ll) lim=m-p2ll;

        lim2=lim;
        if (lim2<2*k+1) lim2=2*k+1;
        for(z1=p2ll;z1<=p2l-1;++z1)
            for(z2=1;z2<=lim2;++z2)
                  pm[z1][z2]=m+1;
        r=j=p2ll;
```

```
        for(i=p2ll;i<=p2l-1;++i) {
                if (i>=m) break;
                b=0;
                if (i<j)
                        pat_merge(pattern,m,i,r,j,&b,lim);
                if (b<lim) {
                        r=i;
                        pat_extend(pattern,m,i,&j,&b,lim);
                }
        }
}

pat_merge(char *pattern, int m, int i, int r, int j, int *bp, int lim)
{
        int b=*bp;
        int d,f,q;

        for(q=1;tm[r][q]<=i-r;++q);
        d=q;
        f=1;
        while (!(b==lim || d==lim+1 || (i+pm[i-r][f]>j && pm[r][d]==m+1))) {
                if (i+pm[i-r][f]>r+pm[r][d]) {
                        pm[i][++b]=pm[r][d]-(i-r);
                        ++d;
                } else
                if (i+pm[i-r][f]<r+pm[r][d]) {
                        pm[i][++b]=pm[i-r][f];
                        ++f;
                } else {
                if (pattern[pm[i-r][f]]!=pattern[i+pm[i-r][f]])
```

```
                        pm[i][++b]=pm[i-r][f];
                ++f;++d;
                }
        }
        *bp=b;
}

pat_extend(char *pattern, int m, int i, int *jp, int *bp, int lim)
{
        int j=*jp,b=*bp;

        while (b<lim && j<m) {
                ++j;
                if (pattern[j]!=pattern[j-i])
                        pm[i][++b]=j-i;
        }
        *jp=j;*bp=b;
}
```

Α.2.3 Αλγόριθμος TU

```
#define ASIZE 128
#define MAXROW 110

void tu_search(char *pattern, char *text, int m, int n, int k, int *count)
{
        int a,i,j,shift,h,neq,matches=0;
        int ready[ASIZE],d[MAXROW][ASIZE];

        /* preprocessing phase */
```

```
        for(a=0;a<ASIZE;a++) ready[a]=m+1;
        for(a=0;a<ASIZE;a++)
            for(i=(m-k);i<=m;i++)
                d[i][a]=m-k;
        for(i=m-1;i>=1;i--) {
            for(j=ready[pattern[i]]-1;j>=MAX(i+1,m-k);j--)
                d[j][pattern[i]]=j-i;
        ready[pattern[i]]=MAX(i+1,m-k);}

        /* searching phase */
        j=m;
        while (j<=n) {
                h=j;i=m;neq=0;
                shift=m-k;
                while (i>0 && neq<=k) {
                        if (i>=m-k) shift=MIN(shift,(d[i][text[h]]));
                        if (text[h]!=pattern[i]) neq++;
                        i--;h--;
                }
                if (neq<=k) {matches++;}
                j+=shift;
        }
        *count=matches;
}
```

A.2.4 Αλγόριθμος BYP

```
#define SIZE 128
#define MOD256 0xff
```

```
typedef struct idxnode {
        int offset;
        struct idxnode *next;
} anode;

void byp_search(char *pattern, char *text, int m, int n, int k, int *count)
{
        int i,j,off1,matches=0;
        anode alpha[SIZE];
        int count1[SIZE];
        anode *aptr;

        /* preprocessing phase */
        for(i=0;i<SIZE;i++) {
                alpha[i].offset=-1;alpha[i].next=NULL; count1[i]=m;
        }
        for(i=1,j=SIZE;i<=m;i++) {
                count1[i]=SIZE;
                if (alpha[pattern[i]].offset==-1) alpha[pattern[i]].offset=m-i-1;
                else {
                        aptr=alpha[pattern[i]].next;
                        alpha[pattern[i]].next=&alpha[j++];
                        alpha[pattern[i]].next->offset=m-i-1;
                        alpha[pattern[i]].next->next=aptr;
                }
        }
        count1[m-1]=m;

        /* searching phase */
        for(i=1;i<=n;i++) {
```

```
                if ((off1=(aptr=&alpha[text[i]])->offset) >=0) {
                        count1[(i+off1)&MOD256]--;
                        for(aptr=aptr->next;aptr!=NULL;aptr=aptr->next)
                                count1[(i+aptr->offset)&MOD256]--;
                }
                if (count1[i&MOD256]<=k) matches++;
                count1[i&MOD256]=m;
        }
        *count=matches;
}
```

A.2.5 Αλγόριθμος SO

```
#define WORD 32
#define MAXSUM 128
#define MATCH 1
#define MISMATCH 0
#define EOS '\0'

void so_search(char *pattern, char *text, int m, int n, int k, int *count)
{
        unsigned int state,overflow,mask,lim,ovmask;
        unsigned int T[MAXSUM];
        int B,type,i,j,matches=0;

        /* preprocessing phase */
        type=MISMATCH;
        if (2*k>m) {
                type=MATCH;k=m-k;
        }
```

```
B=clog2(k+1)+1;
if (m>WORD/*/B*/) printf("Error\n");

lim=k<<((m-1)*B);
ovmask=0;
for(i=1;i<=m;i++) ovmask=(ovmask<<B) | (1<<(B-1));
if (type==MATCH)
        for(i=0;i<MAXSUM;i++) T[i]=0;
else {
        lim+=1<<((m-1)*B);
        for(i=0;i<MAXSUM;i++) T[i]=ovmask>>(B-1);
}
for(j=1,i=1;j<=m;i<<=B,j++) {
        if (type==MATCH)
                T[pattern[j]]+=i;
        else
                T[pattern[j]]&=~i;
}
if (m*B==WORD) mask=~0;
else mask=i-1;

/* searching phase */
if (type==MATCH) {state=0;overflow=0;}
else {state=mask&~ovmask;overflow=ovmask;}
for(i=1;i<=n;i++) {
        state=((state<<B)+T[text[i]])&mask;
        overflow=((overflow<<B)|(state&ovmask))&mask;
        state&=~ovmask;
        if (type==MATCH) {
                 if ((state|overflow)>=lim) matches++;
```

```
                }
                else if ((state|overflow)<lim) matches++;
        }
        *count=matches;
}

int clog2(int x)
{
        int i=0;

        while(x>(1<<i)) i++;
        return i;
}
```

A.3 Ακολουθιακοί Αλγόριθμοι Προσεγγιστικής Αναζήτησης με k διαφορές

A.3.1 Αλγόριθμος SEL

```
#define MAXROWS 110
#define MIN(a,b,c) (a<b)? ((a<c)?a:c):((b<c)?b:c)

void sel_search(char *pattern, char *text, int m, int n, int k, int *count)
{
        int i,j,E,diag,matches=0;
        int D[MAXROWS];

        /* Initialization */
        for(i=0;i<=m;i++) D[i]=i;
```

```
        /* Calculation of the D array */
        for(j=1;j<=n;j++) {
                diag=0;
                for(i=1;i<=m;i++) {
                        if (pattern[i]==text[j])
                                E=diag;
                        else
                                E=(MIN(D[i-1],D[i],diag))+1;
                        diag=D[i];D[i]=E;
                }

               /* Checking the last row of the d array */
               if (D[m]<=k) matches++;
        }
        *count=matches;
}
```

A.3.2 Αλγόριθμος CUTOFF

```
#define MAXSIZE 100
#define MIN(a,b,c) (a<b)? ((a<c)?a:c):((b<c)?b:c)

void cutoff_search(char *pattern, char *text, int m, int n, int k, int *count)
{
        int pn,i,j,diag,E,matches=0;
        int C[MAXSIZE];

        pn=k+1;
        for(i=0;i<=m;i++) C[i]=i;
        for(j=1;j<=n;j++) {
```

```
                diag=0;
                for(i=1;i<=pn;i++) {
                        if (pattern[i]==text[j])
                                E=diag;
                        else
                                E=(MIN(C[i-1],C[i],diag))+1;
                        diag=C[i];
                        C[i]=E;
                }
                while (C[pn]>k) pn--;
                if (pn==m) matches++;else pn++;
        }
        *count=matches;
}
```

A.3.3 Αλγόριθμος GP

```
#define MAXSIZE 110
#define TRUE 1
#define FALSE 0

int u[MAXSIZE],v[MAXSIZE],w[MAXSIZE];

void gp_search(char *pattern, char *text, int m, int n, int k, int *count)
{
        int i,j,c,d,e,col,f,g,r,found,with,b,b1,b2,b3,se;
        int C[MAXSIZE][5], Prefix[MAXSIZE][MAXSIZE];
        int matches=0;

        /* preprocessing phase */
```

```
for(d=1;d<m;d++) {
        i=0;
        while (i+d<m) {
                c=1;
                while (i+c+d<=m && pattern[i+c]==pattern[i+c+d]) c++;
                for(j=1;j<=c;j++) Prefix[i+j][i+j+d]=c-j;
                i+=c;
        }
}

/* searching phase */
for(d=-1;d<=k;d++) {
        C[d][1]=-999;
        C[d][2]=-1;
}
for(e=0;e<=k+1;e++) {
        v[e]=0;w[e]=0;
}
b1=0;b2=1;b3=2;
for(j=0;j<=n-m+k;j++) {
        C[-1][b1]=j;r=0;
        for(e=0;e<=k;e++) {
                d=j-e;
                if (e==0) col=j;
                else col=MAX(C[e-1][b2]+1,C[e-1][b3]+1,C[e-1][b1]);
                se=col+1;
                found=FALSE;
                while (!found) {
                        with=within(col+1,k,&r,u,v);
                        if (with) {
```

```
                                        f=v[r]-col;
                                        g=Prefix[col+1-d][col+1-w[r]];
                                        if (f==g) col+=f;
                                        else {
                                                col+=MIN2(f,g);found=TRUE;}
                                }
                                else if (col-d<m && text[col+1]==pattern[col+1-d])
                                         col++;
                                     else found=TRUE;
                        }
                        C[e][b1]=MIN2(col,m+d);
                        if (C[e][b1]==m+d && C[e-1][b2]<m+d) matches++;
                        if (v[e]>=C[e][b1])
                                if (e==0) u[e]=j+1;
                                else u[e]=MAX2(u[e],v[e-1]+1);
                        else {
                                v[e]=C[e][b1];
                                w[e]=d;
                                if (e==0) u[e]=j+1;
                                else u[e]=MAX2(se,v[e-1]+1);
                        }
                }
                b=b1;b1=b3;b3=b2;b2=b;
        }
        *count=matches;
}

within(int t, int k, int *r_ptr, int u[], int v[])
{
        while (*r_ptr<=k && t>v[*r_ptr]) (*r_ptr)++;
```

```
        if (*r_ptr>k) return(FALSE);
        else if (t>=u[*r_ptr]) return(TRUE);
             else return (FALSE);
}
```

A.3.4 Αλγόριθμος CL

```
#define TRUE 1
#define FALSE 0

void cl_search(char *pattern, char *text, int m, int n, int k, int *count)
{
        int end[MAXSIZE],l[MAXSIZE],loc[MAXSIZE][ASIZE];
        int t,a,s,r,j,running,matches=0;

        /* preprocessing phase */
        for(a=0;a<=ASIZE;a++) {
                loc[m+1][a]=m+2;
                loc[m+2][a]=m+2;
                for(s=m;s>=1;s--){
                        if (pattern[s]==a) loc[s][a]=s;
                        else loc[s][a]=loc[s+1][a];
                }
        }

        /* searching phase */
        end[0]=m; l[0]=m+1;
        t=0;
        for(j=1;j<=n;j++) {
                r=0;
```

```
                running=TRUE;
                while (running) {
                        if (r>t) end[r]=m;
                        else
                                if (l[r]==0) end[r]++;
                                else {
                                        s=loc[end[r]-l[r]+2][text[j]];
                                        if (s<=end[r]+1) end[r]=s-1;
                                        else
                                            if (r+1<=t && l[r+1]!=0) end[r]++;
                                }
                        if (end[r]>=m) {
                                running=FALSE;
                                end[r]=m;
                        }
                        r++;
                }
                t=r-1;
                l[0]=end[0]+1;
                for(r=1;r<=t;r++) l[r]=end[r]-end[r-1];
                if (t>=m-k) matches++;
        }
        *count=matches;
}
```

A.3.5 Αλγόριθμος WMM

```
static int K;
static int *STAB,*DTAB;
static char *CTAB;
```

```
static int pow3[100],pow2[100];
static int *TRAN[128];
static int seg,rem,smax;

static void STATE(register int kapa, register int idx,register int ost,register int car)
{
register int rdx,p3;

        if (kapa<K) {
                rdx=idx+pow2[kapa];
                kapa+=1;
                p3=pow3[kapa];
                if (car<0) {
                        ost+=2*p3;
                        STATE(kapa,idx,ost,1);
                        STATE(kapa,rdx,ost,1);
                        STATE(kapa,idx+=p3,ost,0);
                        STATE(kapa,rdx+=p3,ost,0);
                        STATE(kapa,idx+p3,ost,-1);
                        STATE(kapa,rdx+p3,ost,-1);
                }
                else if (car>0) {
                        STATE(kapa,idx,ost,1);
                        STATE(kapa,rdx,ost,1);
                        STATE(kapa,idx+=p3,ost,0);
                        STATE(kapa,rdx+=p3,ost+p3,1);
                        STATE(kapa,idx+p3,ost,-1);
                        STATE(kapa,rdx+p3,ost+p3,0);
                }
```

```
                else {
                        ost+=p3;
                        STATE(kapa,idx,ost,1);
                        STATE(kapa,rdx,ost,1);
                        STATE(kapa,idx+=p3,ost,0);
                        STATE(kapa,rdx+=p3,ost+p3,1);
                        STATE(kapa,idx+p3,ost,-1);
                        STATE(kapa,rdx+p3,ost+p3,0);
                }
        }
        else {
                STAB[idx]=ost;
                CTAB[idx]=car;
        }
}

void DELTA(register int kapa,register int idx, register int sum)
{
        register int p3;

        if (kapa<K) {
                kapa+=1;
                p3=pow3[kapa];
                DELTA(kapa,idx,sum-1);
                DELTA(kapa,idx+=p3,sum);
                DELTA(kapa,idx+p3,sum+1);
        }
        else
                DTAB[idx]=DTAB[idx+1]=DTAB[idx+2]=sum;
}
```

```
void wmm_search(char *pattern, char *text, int m, int n, int k, int *count)
{
        int i,dik;
        register int *s,*a,*b,c,p,ix,cr,*stab,answr;
        register char *ctab;
        register int *sd,vi;
        int *S,*SE;
        int matches=0;

        /* preprocessing phase */
        K=3;
        pow3[0]=1;
        for(c=1;c<=K+1;c++) pow3[c]=3*pow3[c-1];
        pow2[0]=pow3[K+1];
        for(c=1;c<=K;c++) pow2[c]=2*pow2[c-1];

        STAB=(int *) malloc(sizeof(int)*pow2[K]);
        CTAB=(char *) malloc(sizeof(char)*pow2[K]);
        DTAB=(int *) malloc(sizeof(int)*pow3[K+1]);

        for(c=0;c<=2;c++) STATE(0,c,1,c-1);
        DELTA(0,0,0);

        seg=(m-1)/K+1;
        rem=seg*K-m;
        smax=1;
        for(a=0;a<=K;a++) smax*=3;
        b=(int *) malloc(sizeof(int)*SIGMA*seg);
        for(a=0;a<SIGMA;a++) {
```

```
        TRAN[a]=b;
        for(p=0;p<m;p+=K) {
                answr=0;
                i=p+K;
                if (m<i) i=m;
                for(i--;i>=p;i--) answr=2*answr+(a!=(pattern[i]));
                *b++ = answr*smax;
        }
}
smax-=2;

/* searching phase */
S=(int *)malloc(sizeof(int)*(seg+1))+1;
SE=S+(seg-1);
dik=k+K;
sd=S+(k-1)/K;
for(s=S-1;s<=sd;s++) *s=smax;
vi=(sd-S+1)*K;

stab=STAB;
ctab=CTAB;
for(i=1;i<=n;i++) {
        a=TRAN[text[i]];
        cr=0;
        s=S;
        while (s<=sd) {
                ix=*a++ + *s + cr;
                cr=ctab[ix];
                *s++=stab[ix];
        }
```

```
                if (vi==k && s<=SE) {
                        ix=*a+smax+cr;
                        *++sd=stab[ix];
                        vi+=K+ctab[ix];
                }
                else {
                        vi+=cr;
                        while (vi>dik) vi-=DTAB[*sd--];
                }
                if (sd==SE && vi<=k) matches++;
        }
        if (sd==SE) {
                s=sd;
                a=TRAN[0]+(SE-S);
                for(i=0;i<rem;i++) {
                        ix=*a+*s;
                        *s=stab[ix];
                        vi+=ctab[ix];
                        if (vi<=k) matches++;
                }
        }
        *count=matches;
}
```

A.3.6 Αλγόριθμος PDFA

```
#define MAXSIZE 1000
#define MAX 110
#define HASHSIZE 101
#define ASIZE 128
```

```
#define UNKNOWN -1
#define TRUE 1
#define FALSE 0

struct nlist {
        struct nlist *next;
        unsigned long key;
        int state;
};
struct nstate {
        int S[MAX];
};
struct nlist *hashtab[HASHSIZE];
struct nlist *np;
struct nstate cstate[MAXSIZE];
int Cur[MAX];

void pdfa_search(char *pattern, char *text, int m, int n, int k, int *count)
{
        int aut[MAXSIZE][ASIZE];
        int pstate,state,nstate,i,j,c,col,flag;
        unsigned long key;
        int matches=0;

        state=pstate=0;
        for(i=0;i<=m;i++) cstate[state].S[i]=i;
        for(c=0;c<ASIZE;c++) aut[state][c]=UNKNOWN;

        for(i=1;i<=n;i++) {
                nstate=aut[state][text[i]];
```

```
                if (nstate==UNKNOWN) {
                        perform_step(pattern,m,text[i],cstate[state].S);
                        key=radix_conversion(m,i);
                        nstate=search(key);
                        if (nstate==UNKNOWN) {
                                pstate++;
                                nstate=pstate;
                                insert(key,nstate);
                                for(j=0;j<=m;j++) cstate[nstate].S[j]=Cur[j];
                                for(c=0;c<ASIZE;c++) aut[nstate][c]=UNKNOWN;
                        }
                        aut[state][text[i]]=nstate;
                }
                state=nstate;
                if (cstate[state].S[m]<=k) matches++;
        }
        *count=matches;
}

void perform_step(char *p, int m, char textch, int C[MAX])
{
        int i,E,diag;

        Cur[0]=0;
        for(i=1;i<=m;i++) {
                if (p[i]==textch) Cur[i]=C[i-1];
                else Cur[i]=MIN(C[i-1]+1,Cur[i-1]+1,C[i]+1);
        }
}
```

```
search(unsigned long key)
{
        for(np=hashtab[hash(key)];np!=NULL;np=np->next)
                if (key==np->key) return np->state;
        return -1;
}

insert(unsigned long key, int nstate)
{
        unsigned int hashval;

        np=(struct nlist *) malloc(sizeof(*np));
        if (np==NULL || (np->key=strdup(key))==NULL)
                return -1;
        hashval=hash(key);
        np->key=key;
        np->state=nstate;
        np->next=hashtab[hashval];
        hashtab[hashval]=np;
        if (np->state=strdup(nstate)==NULL) return -1;
}

radix_conversion(int m, int j)
{
        int i,sum;

        sum=0;
        for(i=0;i<=m;i++)
                sum+=Cur[i]*power(11,m-i);
        return (sum%6997);
```

```
}

power(int base, int ekt)
{
        int i,p;

        p=1;
        for(i=1;i<=ekt;i++)
                p*=base;
        return p;
}

hash(unsigned long key)
{
        return (key%HASHSIZE);
}
```

A.3.7 Αλγόριθμος TUD

```
#define ASIZE 128
#define MAXSIZE 110
#define TRUE 1
#define FALSE 0

int end,start;

void tud_search(char *pattern, char *text, int m, int n, int k, int *count)
{
        int a,i,j,auj,ch,r,top,col,lastcol,c,next,diag;
        int ready[ASIZE],Ho[MAXSIZE],H[MAXSIZE];
```

```
int d[MAXSIZE][ASIZE],bad[MAXSIZE][ASIZE];
int bad_columns,shift;
int matches=0;

/* preprocessing phase */
for(a=1;a<=ASIZE;a++) ready[a]=m+1;
for(a=1;a<=ASIZE;a++)
        for(i=m-k;i<=k;i++)
                d[i][a]=m;
for(i=m-1;i>=1;i--) {
          for(auj=ready[pattern[i]]-1;auj>=MAX(i+1,m-k);auj--)
        d[auj][pattern[i]]=auj-i;
          ready[pattern[i]]=MAX(i+1,m-k);
}
for(i=1;i<MAXSIZE;i++)
          for(auj=1;auj<ASIZE;auj++)
                bad[i][auj]=TRUE;
for(ch='a';ch<='z';ch++)
          for(i=1;i<=m;i++) {
                for(auj=(i-k);auj<=(i+k);auj++)
                      if (ch==pattern[auj]) bad[i][ch]=FALSE;
          }

/* searching phase */
j=m;
while (j<=n) {
        r=j;i=m;bad_columns=0;shift=m;
        while (i>k && bad_columns<=k) {
                if (i>=m-k) shift=MIN2(shift,d[i][text[r]]);
                if (bad[i][text[r]]) bad_columns++;
```

```
                        i--;r--;
                }
                if (bad_columns<=k) {
                        start=j-m-k+1;
                        if (start<=0) start=1;
                        end=start+m+2*k+1;
                        matches+=searchcut(pattern,text,m,end-start,k);
                        j=j+m+1;
                }
                else j=j+MAX(MIN2(k+1,i+1),shift);
        }
        *count=matches;
}

searchcut(char *pattern, char *text, int m, int n, int k)
{
        int i,r,top,c,d,matches=0;
        int C[MAXSIZE];

        for(i=0;i<=m;i++) C[i]=i;
        top=k+1;
        for(r=start;r<=end;r++) {
                c=0;
                for(i=1;i<=top;i++) {
                        if (pattern[i]==text[r]) d=c;
                        else d=(MIN(C[i-1],C[i],c))+1;
                        c=C[i];C[i]=d;
                }
                while (C[top]>k) top--;
                if (top==m) {matches++;}
```

```
                else top++;
        }
        return matches;
}
```

A.3.8 Αλγόριθμος MULTIWM

```
#define MAXROWS 110
#define ASIZE 128
int start,end;

void multiwm_search(char *pattern, char *text, int m, int n, int k, int *count)
{
        unsigned int state;
        int i,j,s,c,t,B,matches=0;
        unsigned int lim,T[ASIZE];
        unsigned char pat[MAXROWS];
        double r1;

        /* preprocessing phase */
        r1=(m/(k+1));
        t=0;
        c=0;pat[c++]='@';
        B=(k+1);
        for(i=1;i<=r1;i++) {
                t++;s=t;
                for(j=0;j<k+1;j++) {
                        pat[c++]=pattern[s];
                        s+=r1;
                }
```

```
        }
        pat[c]='\0';
        for(i=0;i<ASIZE;i++) T[i]=~0;
        lim=0;
        for(i=1,j=1;i<=m;i++,j<<=1) {
                T[pat[i]]=T[pat[i]]&~j;
                lim|=j;
        }
        lim=~(lim>>B);

        /* searching phase */
        state=~0;
        for(i=1;i<=n;i++) {
                state=(state<<B)|T[text[i]];
                if (state<lim) {
                        start=i-1+r1-m-k;end=i-1+m+k-1;
                        if (start<=0) start=1;
                        if (end>=n) end=n;
                        matches+=searchcut(pattern,text,m,end-start,k);
                }
        }
        *count=matches;
}
```

A.3.9 Αλγόριθμος BYPEP

```
#define ASIZE 128
#define MAXROWS 110
#define MAXSIZE 1000
#define TRUE 1
```

```
#define FALSE 0
#define FAIL -1
#define MAX 100000
int state,newstate,a,s,r,y,q1;
long start,end;
int g[MAXSIZE][ASIZE],f[MAX],out[MAX];
int queue[MAX];

void bypep_search(char *pattern, char *text, int m, int n, int k, int *count)
{
        int i,j,c,matches=0;
        char pat[MAXROWS][MAXROWS];
        double ra;

        /* partition approach */
        ra=(m/(k+1));
        for(j=0;j<=k+1;j++) {
                c=0;pat[j][c++]='@';
                for(i=(ra*j)+1;i<=ra*(j+1);i++)
                        pat[j][c++]=pattern[i];
        }

        /* preprocessing phase */
        for(i=0;i<ASIZE;i++)
                for(a=0;a<ASIZE;a++)
                        g[i][a]=FAIL;
        newstate=0;out[newstate]=FALSE;
        for(i=0;i<k+1;i++) enter(pat[i]);
        for(a=0;a<ASIZE;a++)
                if (g[0][a]==FAIL) g[0][a]=0;
```

```
q1=0;
for(a=0;a<ASIZE;a++) {
        s=g[0][a];
        if (s!=0) {
                y=insert_queue(s);
                if (y==-1) break;
                f[s]=0;
        }
}
while (q1!=0) {
        r=delete();
        if (r==-1) {printf("uderflow\n");break;}
        for(a=0;a<ASIZE;a++) {
                s=g[r][a];
                if (s!=FAIL)  {
                        y=insert_queue(s);
                        if (y==-1) {printf("overflow\n");break;}
                        state=f[r];
                        while (g[state][a]==FAIL) state=f[state];
                        f[s]=g[state][a];
                        if (out[f[s]]) out[s]=TRUE;
                }
        }
}

/* searching phase */
state=0;
for(i=1;i<=n;i++) {
        while (g[state][text[i]]==FAIL) state=f[state];
```

```
                state=g[state][text[i]];
                if (out[state]) {
                        start=i-1+ra-m-k;end=i-1+m+k-1;
                        if (start<=0) start=1;
                        if (end>=n) end=n;
                        matches+=searchcut(pattern,text,m,end-start,k);
                }
        }
        *count=matches;
}

enter(char *pat)
{
        int p,j,mp;

        state=0;j=1;
        while (g[state][pat[j]]!=FAIL) {
                state=g[state][pat[j]];
                j++;
        }
        mp=strlen(pat);
        for(p=j;p<mp;p++) {
                newstate++;
                g[state][pat[p]]=newstate;
                state=newstate;
                out[state]=FALSE;
        }
        out[state]=TRUE;
}
```

```
insert_queue(int element)
{
        if (q1==MAX) return -1;
        else {
                queue[q1]=element;
                q1++;
        }
        return 0;
}

delete()
{
        int element;
        int i;

        if (q1==0) return -1;
        else {
                element=queue[0];
                q1--;
                for(i=0;i<=MAX;i++) queue[i]=queue[i+1];
        }
        return element;
}
```

A.3.10 Αλγόριθμος WM

```
#define ASIZE 128
#define MAXLENGTH_TEXT 1000000

void wm_search(char *pattern, char *text, int m, int n, int k, int *count)
```

```
{
        int i,j,c,d,matches=0;
        int result[MAXLENGTH_TEXT];
        unsigned int mask,mask_tmp,result_mask;
        long *R,*Rprev,S[ASIZE+1];

        /* preprocessing phase */
        mask=(long)1<<(sizeof(long)*CHAR_BIT-1);
        mask_tmp=mask;

        R=(long *) calloc(k+2,sizeof(long));
        Rprev=(long *) calloc(k+2,sizeof(long));

        for(c=0;c<=ASIZE;c++)
                S[c]=(long) 0;
        for(i=1;i<=m;i++) {
                S[pattern[i]]=S[pattern[i]]|mask_tmp;
                mask_tmp>>=1;
        }
        result_mask=mask_tmp<<1;
        R[0]=(long)0;
        for(d=1;d<=k;d++)
                R[d]=mask|R[d-1]>>1;

        /* searching phase */
        for(j=1;j<=n;j++) {
                Rprev[0]=R[0];
                R[0]>>=1;
                R[0]|=mask;
                R[0]&=S[text[j]];
```

```
                for(d=1;d<=k;d++) {
                        Rprev[d]=R[d];
                        R[d]=(((R[d]>>1)|mask)&S[text[j]])|(((Rprev[d-1]|R[d-1])
                                >>1)|mask)|Rprev[d-1];
                }
                d=0;
                while(!(result_mask & R[d]) && d<=k) d++;
                result[j]=(d<=k)?d:m+1;
                if (result[j]<=k) matches++;
        }
        *count=matches;
}
```

A.3.11 Αλγόριθμος BYN

```
#define SIGMA 128

void byn_search(char *pattern, char *text, int m, int n, int k, int *count)
{
        register unsigned long t,f,one,con,D,X;
        unsigned long Pc[SIGMA];
        static unsigned long M1,M2,M3,Din,G,Tm[SIGMA];
        register long p,k2;
        int a,i,matches=0;

        /* preprocessing phase */
        k2=k+2;

        if ((m-k)*k2>8*sizeof(unsigned long))
                printf("Pattern does not fit in a word\n");
```

```
one=1;
M3=1;
for(i=0;i<k;i++) M3=(M3<<1)|one;
Din=M3;
for(i=1;i<m-k;i++) Din|=(Din<<k2);
M1=1;
for(i=1;i<m-k;i++) M1|=(M1<<k2);
M2=M1|M3;

G=(one<<k);
for(a=0;a<SIGMA;a++) Pc[a]=-1;
one=1;
for(p=1;p<=m;p++) {
        con=~one;
        Pc[pattern[p]]&=con;
        one<<=1;
}

for(a=0;a<SIGMA;a++) {
        f=Pc[a];
        t=0;
        for(i=1;i<=m-k-1;i++) {
                t=(t<<k2)|(f&M3);
                f>>=1;
        }
        Tm[a]=t;
}

/* searching phase */
one=1;
```

```
        D=Din;
        i=0;
        while (++i<=n) {
                a=text[i];
                X=(D>>(k+2))|Tm[a];
                D=((D<<1)|M1) & ((D<<(k+3))|M2) & (((X+M1)^X)>>1)&Din;
                if ((D&G)==0) {
                        matches++;
                        D=D|M3;
                }
        }
        *count=matches;
}
```

A.3.12 Αλγόριθμος MYE

```
#define SIGMA 128

void myers_search(char *pattern, char *text, int m, int n, int k, int *count)
{
        int score,i,j;
        unsigned int one,P,M,X,U,Y;
        unsigned long mask=(long) 1<<(sizeof(long)*CHAR_BIT-1);
        unsigned int Ebit,Pc[SIGMA];
        int a,matches=0;

        /* preprocessing phase */
        for(i=0;i<SIGMA;i++) Pc[i]=0;
        one=1;
        for(i=1;i<=m;i++) {
```

```
                Pc[pattern[i]]|=one;
                one<<=1;
        }
        one=1;
        Ebit=(one << (m-1));

        /* searching phase */
        P=-1;
        M=0;
        score=m;
        for(j=1;j<=n;j++) {
                a=text[j];
                U=Pc[a];
                X=(((U&P)+P)^P)|U;
                U|=M;

                Y=P;
                P=M|~(X|Y);
                M=Y&X;
                if (P&Ebit)
                        score+=1;
                else if (M&Ebit)
                        score-=1;
                Y=P<<1;
                P=(M<<1)|~(U|Y);
                M=Y&U;
                if (score<=k) matches++;
        }
        *count=matches;
}
```

Βιβλιογραφία

[1] A. Aho and M. Corasick. Efficient string matching: An aid to bibliographic search. *Communications of the ACM*, 18(6):333–340, 1975.

[2] A. V. Aho. *Algorithms for finding patterns in strings*, chapter 5, pages 255–300. Elsevier Science Publishers, 1990.

[3] A. Apostolico and R. Giancarlo. The Boyer-Moore-Galil string searching strategies revisited. *SIAM Journal on Computing*, 15(1):98–105, 1986.

[4] V.L. Arlazarov, E.A. Dinic, M.A. Kronrod, and I.A. Faradzev. On economic construction of the transitive closure of a directed graph. *Dokl. Akad. Nauk SSSR*, 194:487–488 (in Russian), 1970. English translation in Soviet Math. Dokl., vol. 11, pp. 1209-1210.

[5] R. Baeza-Yates. Algorithms for string searching: A survey. *ACM SIGIR Forum*, 23(3-4):34–58, 1989.

[6] R. Baeza-Yates. Text retrieval: Theory and practice. In *Proceedings of the 12th IFIP World Computer Congress*, pages 465–476, Madrid, Spain, 1992. North-Holland.

[7] R. Baeza-Yates and G. H. Gonnet. A new approach to text searching. *Communications of the ACM*, 35(10):74–82, 1992.

[8] R. Baeza-Yates and B. Ribeiro. *Modern Information Retrieval.* Addison-Wesley, 1999.

[9] R. A. Baeza-Yates. A unified view of string matching algorithms. In *Proceedings of SOFSEM: Theory and Practice of Informatics*, number 1175 in Lecture Notes in Computer Science, pages 1–15. Springer-Verlag, Berlin, 1996.

[10] R. A. Baeza-Yates and G. Gonnet. Fast string matching with mismatches. *Information and Computation*, 108(2):187–199, 1994.

[11] R. A. Baeza-Yates and G. Navarro. A fast heuristic for approximate string matching. In *Proceedings of the 3rd South American Workshop on String Processing*, pages 47–63. Carleton University Press, 1996.

[12] R. A. Baeza-Yates and G. Navarro. A faster algorithm for approximate string matching. In *Proceedings of the 7th Annual Symposium on Combinatorial Pattern Matching*, number 1075 in Lecture Notes in Computer Science, pages 1–23. Spinger-Verlag, Berlin, 1996.

[13] R. A. Baeza-Yates and G. Navarro. A faster algorithm for approximate string matching. *Algorithmica*, 23(2):127–158, 1999.

[14] R. A. Baeza-Yates and C. H. Perleberg. Fast and practical approximate string matching. *Information Processing Letters*, 59(1):21–27, 1996.

[15] R. A. Baeza-Yates and M. Regnier. Fast two dimensional pattern matching. *Information Processing Letters*, 45(1):51–57, 1993.

[16] A. Bairoch and B. Boeckmann. The SWISS-PROT protein sequence data bank. *Nucleic Acids Research*, pages 2019–2022, 1992.

[17] O. Beaumont, V. Boudet, F. Rastello, and Y. Robert. Matrix-matrix multiplication on heterogeneous platforms. Technical Report 2000-02, CNRS-INRIA-ENS LYON, 2000.

[18] O. Beaumont, A. Legrand, and Y. Robert. The master-slave paradigm with heterogeneous processors. Technical Report 2001-13, CNRS-INRIA-ENS LYON, 2001.

[19] D. Benson, D. J. Lipman, and J. Ostell. Genbank. *Nucleic Acids Research*, 21:2963–2965, 1993.

[20] R. S. Boyer and J. S. Moore. A fast string searching algorithm. *Communications of the ACM*, 20(10):762–772, 1977.

[21] R. Buyya. *High Performance Cluster Computing: Architectures and Systems*, volume 1. Prentice Hall, 1999.

[22] R. Buyya. *High Performance Cluster Computing: Programming and Applications*, volume 2. Prentice Hall, 1999.

[23] W. Chang and E. Lawler. Sublinear approximate string matching and biological applications. *Algorithmica*, 12(4/5):327–344, 1994.

[24] W. I. Chang and J. Lampe. Theoretical and empirical comparisons of approximate string matching algorithms. In *Proceedings of the 3rd Annual Symposium on Combinatorial Pattern Matching*, number 664 in Lecture Notes in Computer Science, pages 175–184. Springer-Verlag, Berlin, 1992.

[25] H. D. Cheng and K. S. Fu. VLSI architectures for string matching and pattern matching. *Pattern Recognition*, 20(1):125–141, 1987.

[26] L. Colussi. Correctness and efficiency of the pattern matching algorithms. *Information and Computation*, 95(2):225–251, 1991.

[27] L. Colussi. Fastest pattern matching in strings. *Journal of Algorithms*, 16(2):163–189, 1994.

[28] B. Commentz-Walter. A string matching algorithm fast on the average. In *Proceedings of the 6th International Colloquium on Automata, Languages and Programming*, pages 118–132. Springer-Verlag, Berlin, 1979.

[29] J. K. Cringean, R. England, G. A. Manson, and P. Willett. Parallel text searching in serial files using a processor farm. In *Proceedings of the 13th International Conference on Research and Development in Information Retrieval*, pages 429–453, 1990.

[30] M. Crochemore, A. Czumaj, L. Gasieniec, S. Jarominek, T. Lecroq, W. Plandowski, and W. Rytter. Speeding up two string matching algorithms. *Algorithmica*, 12(4-5):247–267, 1994.

[31] M. Crochemore, C. Iliopoulos, G. Navarro, and Y. Pinzon. A bit-parallel suffix automaton approach for (delta,gamma) matching in music retrieval. In *Proceedings of the International Symposium on String Processing and Information Retrieval*, number 2857 in Lecture Notes in Computer Science, pages 211–223, 2003.

[32] M. Crochemore, C. S. Iliopoulos, Y. J. Pinzon, and J. R. Reid. A fast and practical bit-vector algorithm for the longest common subsequence problem. In *Proceedings of the 11th Australasian Workshop on Combinatorial Algorithms*, pages 75–86, 2000.

[33] M. Crochemore, C. S. Iliopoulos, Y. J. Pinzon, and J. R. Reid. A fast and practical bit-vector algorithm for the longest common subsequence problem. *Information Processing Letters*, 80:279–285, 2001.

[34] M. Crochemore and W. Rytter. *Text algorithms.* Oxford University Press, 1994.

[35] G. Das, R. Fleisher, L. Gasieniek, D. Gunopulos, and J. Karkainen. Episode matching. In *Proceedings of the 8th Annual Symposium on Combinatorial Pattern Matching*, number 1264 in Lecture Notes in Computer Science, pages 12–27. Springer-Verlag, 1997.

[36] G. Davies and S. Bowsher. Algorithms for pattern matching. *Software Practice and Experience*, 16(6):575–601, 1986.

[37] A. Dermouche. A fast algorithm for string matching with mismatches. *Information Processing Letters*, 55(2):105–110, 1995.

[38] N. El-Mabrouk and M. Crochemore. Boyer-moore strategy to efficient approximate string matching. In *Proceedings of the 11th Annual Symposium on Combinatorial Pattern Matching*, number 1075 in Lecture Notes in Computer Science, pages 24–38. Springer-Verlag, Berlin, 1996.

[39] M. Flynn. Very high speed computing systems. *Proceedings of the IEEE*, 54:1901–1909, 1966.

[40] M. J. Foster and H. T. Kung. The design of special purpose VLSI chips. *IEEE Computer*, 13(1):26–40, 1980.

[41] J. French, A. Powell, and E. Schulman. Applications of approximate word matching in information retrieval. In *Proceedings of the ACM CIKM'97*, pages 9–15, 1997.

[42] Z. Galil. On improving the worst case running time of the boyer-moore string searching algorithm. *Communications of the ACM*, 22(9):505–508, 1979.

[43] Z. Galil and R. Giancarlo. Improved string matching with k mismatches. *Sigact News*, 17(4):52–54, 1986.

[44] Z. Galil and R. Giancarlo. Data structures and algorithms for approximate string matching. *Journal of Complexity*, 4(1):33–72, 1988.

[45] Z. Galil and K. Park. An improved algorithm for approximate string matching. *SIAM Journal on Computing*, 19(6):989–999, 1990.

[46] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. *PVM: Parallel Virtual Machine, A User's Guide and Tutorial for Networked Parallel Computing.* Massachusetts: The MIT Press, 1994.

[47] G. H. Gonnet and R. Baeza-Yates. *Handbook of Algorithms and Data Structures in Pascal and C.* Addison-Wesley, Workingham, 2nd edition, 1991.

[48] R. Gonzalez and M. Thomason. *Syntactic pattern recognition.* Addison-Wesley, 1978.

[49] O. Gotoh. An improved algorithm for matching biological sequences. *Journal of Molecular Biology*, 162:705–708, 1982.

[50] R. Grossi and F. Luccio. Simple and efficient string matching with k mismatches. *Information Processing Letters*, 33(3):113–120, 1989.

[51] P. Guerdoux-Jamet, D. Lavenier, C. Wagner, and P. Quinton. Design and implementation of a parallel architecture for biological sequence comparison. In *Proceedings of the European Parallel Processing*, Lecture Notes in Computer Science. Springer-Verlag, Berlin, 1996.

[52] C. Hancart. On simon's string searching algorithm. *Information Processing Letters*, 47(2):95–99, 1993.

[53] M. C. Harrison. Implementation of the substring test by hashing. *Communications of the ACM*, 14(12):777–779, 1971.

[54] D. S. Hirschberg. A linear space algorithm for computing maximal common subsequences. *Communications of the ACM*, 18(6):341–343, 1975.

[55] D. T. Hoang. A systolic array for the sequence alignment problem. Technical Report CS-92-22, Dept. of Computer Science, Brown University, Providence, RI, 1992.

[56] D. T. Hoang. Searching genetic databases on splash 2. In *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, pages 185–191, 1993.

[57] D. T. Hoang and D. P. Lopresti. FPGA implementation of systolic sequence alignment. In *Proceedings of International Workshop on Field Programmable Logic*, 1992.

[58] R. N. Horspool. Practical fast searching in strings. *Software Practice and Experience*, 10(6):501–506, 1980.

[59] W-C. Hu, M. Schmalz, and G. Ritter. Image retrieval using the longest approximate common subsequences. In *Proceedings of the IEEE International Conference on Multimedia Computing and Systems*, volume 2, pages 730–734, 1999.

[60] R. Hughey and D. P. Lopresti. Architecture of a programmable systolic array. In *Proceedings of International Conference on Systolic Arrays*, pages 41–50, 1988.

[61] A. Hume and D. Sunday. Fast string searching. *Software Practice and Experience*, 21(11):1221–1248, 1991.

[62] G. A. Manson J. K. Cringean, R. England and P. Willett. Network design for the implementation of text searching using a multicomputer. *Information Processing and Management*, 27:265–283, 1991.

[63] P. Willett J. K. Cringean, G. A. Manson and G. A. Wilson. Efficiency of text scanning in bibliographic databases using microprocessor-based multiprocessor networks. *Journal of Information Science*, 14:335–345, 1988.

[64] P. Jokinen, J. Tarhio, and E. Ukkonen. A comparison of approximate string matching algorithms. *Software Practice and Experience*, 26(12):1439–1458, 1996.

[65] A. Julich. Implementations of BLAST for parallel computers. *Computer Applications in the Biosciences*, 11(1):3–6, 1995.

[66] J. Kδukkδinen, G. Navarro, and E. Ukkonen. Approximate string matching on ziv-lempel compressed text. *Journal of Discrete Algorithms*, 1(3/4):313–338, 2003.

[67] B. W. Kernighan and D. M. Ritchie. *The C Programming Language*. Prentice Hall, Englewood Cliffs, NJ, 1988.

[68] T. Kida, M. Takeda, A. Shinohara, and S. Arikawa. Shift-and approach to pattern matching in LZW compressed text. In *Proceedings of the 10th Annual Symposium on Combinatorial Pattern Matching*, pages 1–13, 1999.

[69] D. E. Knuth, J. H. Morris, and V. R. Pratt. Fast pattern matching in strings. *SIAM Journal on Computing*, 6(2):323–350, 1977.

[70] K. Kukich. Techniques for automatically correcting words in text. *ACM Computing Surveys*, 24(4):377–439, 1992.

[71] S. Kumar and E. Spafford. A pattern matching model for intrusion detection. In *Proceedings of the National Computer Security Conference*, pages 11–21, 1994.

[72] S. Y. Kung. *VLSI Array Processors*. Prentice-Hall, 1988.

[73] S. Kurtz. Approximate string searching under weighted edit distance. In *Proceedings of the 3rd South American Workshop on String Processing*, pages 156–170. Carleton University Press, 1996.

[74] G. M. Landau and U. Vishkin. Efficient string matching with k mismatches. *Theoretical Computer Science*, 43(2-3):239–249, 1986.

[75] G. M. Landau and U. Vishkin. Fast string matching with k differences. *Journal of Computer and System Sciences*, 37(1):63–78, 1988.

[76] G. M. Landau and U. Vishkin. Fast parallel and serial approximate string matching. *Journal of Algorithms*, 10(2):157–169, 1989.

[77] D. Lavenier. SAMBA: Systolic accelerators for molecular biological applications. Technical Report 988, IRISA, 35042 Rennes Cedex, France, 1996.

[78] D. Lavenier. Speeding up genome computations with a systolic accelerator. *SIAM News*, 31(8):6–7, 1998.

[79] D. Lavenier and J.L. Pacherie. Parallel processing for scanning genomic databases. In *Proceedings of the Parallel Computing 97*, pages 81–88, 1997.

[80] T. Lecroq. A variation on the boyer-moore algorithm. *Theoretical Computer Science*, 92(1):119–144, 1992.

[81] T. Lecroq. Experimental results on string matching algorithms. *Software Practice and Experience*, 25(7):727–765, 1995.

[82] T. Lecroq, G. Luce, and J. F. Myoupo. A faster linear systolic algorithm for recovering a longest common subsequence. *Information Processing Letters*, 61:129–136, 1997.

[83] T. Lecroq, J. F. Myoupo, and D. Seme. A one-phase parallel algorithm for the sequence alignment problem. *Parallel Processing Letters*, 8(4):515–526, 1998.

[84] K. Lemstrom. *String matching techniques for music retrieval.* PhD thesis, University of Helsinki, Department of Computer Science, 2000.

[85] K. Lemstrom and G. Navarro. Flexible and efficient bit-parallel techniques for transposition invariant approximate matching in music retrieval. In *Proceedings of the International Symposium on String Processing and Information Retrieval*, number 2857 in Lecture Note in Computer Science, pages 224–237, 2003.

[86] Y-C. Lin. New systolic arrays for the longest common subsequence problem. *Parallel Computing*, 20:1323–1334, 1994.

[87] Y-C. Lin and J-C. Chen. An efficient systolic algorithm for the longest common subsequence problem. *Journal of Complexity*, 12:373–385, 1998.

[88] R. J. Lipton and D. P. Lopresti. A systolic array for rapid string comparison. In *Proceedings of Chapel Hill Conference on VLSI*, pages 363–376, 1985.

[89] D. Lopresti and A. Tomkins. On the searchability of electronic ink. In *Proceedings of the 4th International Workshop on Frontiers in Handwriting Recognition*, pages 156–165, 1994.

[90] D. P. Lopresti. P-NAC: A systolic array for comparing nucleic acid sequences. *IEEE Computer*, 20(1):98–99, 1987.

[91] D. P. Lopresti. Rapid implementation of a genetic sequence comparator using field-programmable logic arrays. In *Proceedings of Advanced Research in VLSI*, pages 138–152, 1991.

[92] D. Loveman. High-performance fotran. *IEEE Parallel and Distributed Technology*, 1(1), 1993.

[93] R. Lowrance and R. Wagner. An extension of the string-to-string correction problem. *Journal of the ACM*, 22:177–183, 1975.

[94] G. Luce and J. F. Myoupo. Systolic-based parallel architecture for the longest common subsequences problem. *Integration, the VLSI Journal*, 25:53–70, 1998.

[95] T. Luczak and W. Szpankowski. A suboptimal lossy data compression based on approximate pattern matching. *IEEE Transactions on Information Theory*, 1997.

[96] Y. Manolopoulos and C. Faloutsos. Experimenting with pattern matching algorithms. *Information Sciences*, 90(1-4):75–89, 1996.

[97] K. G. Margaritis and D. J. Evans. A VLSI processor array for flexible string matching. *Parallel Algorithms and Applications*, 11:45–60, 1997.

[98] W. J. Masek and M. S. Paterson. A faster algorithm computing string edit distances. *Journal of Computer and System Sciences*, 20:18–31, 1980.

[99] G. M. Megson. Efficient systolic string matching. *Electronic Letters*, 26(24):2040–2042, 1990.

[100] P. D. Michailidis and K. G. Margaritis. String matching algorithms. Technical report, Department of Applied Informatics, University of Macedonia, 1999. (in Greek).

[101] P. D. Michailidis and K. G. Margaritis. A survey of on-line approximate string matching algorithms. Technical report, Department of Applied Informatics, University of Macedonia, 1999.

[102] P. D. Michailidis and K. G. Margaritis. Implementation of the string matching problem on a cluster of workstations. In Y. Manolopoulos and S. Evripidou, editors, *Proceedings of the 8th Panhellenic Conference on Informatics*, volume 2, pages 72–81. Livanis Publishing House, 2001.

[103] P. D. Michailidis and K. G. Margaritis. Implementing string searching algorithms on a network of workstations using MPI. In E. Lipitakis, editor, *Proceedings of the 5th Hellenic European Conference on Computer Mathematics and its Applications*, pages 243–244, 2001.

[104] P. D. Michailidis and K. G. Margaritis. On-line string matching algorithms: Survey and experimental results. *International Journal of Computer Mathematics*, 76(4):411–434, 2001.

[105] P. D. Michailidis and K. G. Margaritis. Parallel text searching application on a heterogeneous cluster of workstations. In T. Pinkston, editor, *Proceedings of the 2001 International Conference on Parallel Processing Workshops*, pages 169–175. IEEE Computer Society Press, 2001.

[106] P. D. Michailidis and K. G. Margaritis. String matching problem on a cluster of personal computers: Experimental results. In A. Popov and R. Romansky, editors, *Proceedings of the 15th International Conference on Systems for Automation of Engineering and Research*, pages 71–75, 2001.

[107] P. D. Michailidis and K. G. Margaritis. String matching problem on a cluster of personal computers: Performance modeling. In A. Popov and R. Romansky, editors, *Proceedings of the 15th International Conference on Systems for Automation of Engineering and Research*, pages 76–81, 2001.

[108] P. D. Michailidis and K. G. Margaritis. Text searching on a heterogeneous cluster of workstations. In Y. Cotronis and J. Dongarra, editors, *Proceedings of the 8th European PVM/MPI Users' Group Meeting*, number 2131 in Lecture Notes in Computer Science, pages 378–385. Springer-Verlag, Berlin, 2001.

[109] P. D. Michailidis and K. G. Margaritis. Implementation of the approximate string matching application on a cluster of heterogeneous workstations. In D. Grigoras, editor, *Proceedings of the 2002 International Symposium on Parallel and Distributed Computing*, pages 193–204, 2002.

[110] P. D. Michailidis and K. G. Margaritis. On-line approximate string searching algorithms: Survey and experimental results. *International Journal of Computer Mathematics*, 79(8):867–888, 2002.

[111] P. D. Michailidis and K. G. Margaritis. Parallel implementations for string matching problem on a cluster of distributed workstations. *Neural, Parallel and Scientific Computations*, 10:287–312, 2002.

[112] P. D. Michailidis and K. G. Margaritis. A performance study of load balancing strategies for approximate string matching on an MPI heterogeneous system environment. In J. Dongarra D. Kranzlmuller, P. Kacsuk and J. Volkert, editors, *Proceedings of the 9th European PVM/MPI Users' Group Meeting*, number 2474 in Lecture Notes in Computer Science, pages 432–440. Springer-Verlag, Berlin, 2002.

[113] P. D. Michailidis and K. G. Margaritis. A processor array for approximate limited expression matching. In *Proceedings of the First Workshop on Application Specific Processors*, pages 137–144, 2002.

[114] P. D. Michailidis and K. G. Margaritis. Bit-level processor array architecture for flexible string matching. In *Proceedings of the 1st Balkan Conference in Informatics*, pages 517–526, 2003.

[115] P. D. Michailidis and K. G. Margaritis. Parallel architecture for flexible approximate text searching. In *CD-ROM Proceedings of the 7th WSEAS International Multiconference on Circuits, Systems, Communications and Computers*, 2003.

[116] P. D. Michailidis and K. G. Margaritis. Performance analysis of approximate string searching implementations for heterogeneous computing platform. In J. Dongarra, D. Laforenza, and S. Orlando, editors, *Proceedings of the 10th European PVM/MPI Users' Group Meeting*, number 2840 in Lecture Notes in Computer Science, pages 242–246. Springer-Verlag, Berlin, 2003.

[117] P. D. Michailidis and K. G. Margaritis. Performance analysis of approximate string searching implementations for heterogeneous computing platform. In *Proceedings of the 2003 International Conference on Parallel Processing Workshops*, pages 173–180. IEEE Computer Society Press, 2003.

[118] P. D. Michailidis and K. G. Margaritis. Performance evaluation of load balancing strategies for approximate string matching application on a MPI cluster of heterogeneous workstations. *Future Generation Computer Systems*, 19(7):1075–1104, 2003.

[119] P. D. Michailidis and K. G. Margaritis. Parallel text searching applications on a heterogeneous cluster architecture. *International Journal of Computational Science and Engineering*, 2004. to appear.

[120] P. D. Michailidis and K.G. Margaritis. Flexible approximate string matching application on a heterogeneous distributed environment. In *Proceedings of the 2003 International Conference on Parallel and Distributed Processing Techniques and Applications*, volume 1, pages 164–172, 2003.

[121] J. H. Moreno and T. Lang. *Matrix Computations on Systolic-Type Arrays*. Kluwer Academic Publishers, 1992.

[122] E. Moura, G. Navarro, N. Ziviani, and R. Baeza-Yates. Fast and flexible word searching on compressed text. *ACM Transactions on Information Systems*, 18(2):113–139, 2000.

[123] G. Myers. Algorithmic advances for searching biosequence databases. pages 121–135. Plenum Press, 1994.

[124] G. Myers. A fast bit-vector algorithm for approximate pattern matching based on dynamic programming. In *Proceedings of the 9th Annual Symposium on Combinatorial Pattern Matching*, number 1448 in Lecture Notes in Computer Science, pages 1–13. Springer-Verlag, Berlin, 1998.

[125] G. Myers. A fast bit-vector algorithm for approximate string matching based on dynamic programming. *Journal of the ACM*, 46(3):395–415, 1999.

[126] G. Navarro. Multiple approximate string matching by counting. In *Proceedings of the 4th South American Workshop on String Processing*, pages 125–139. Carleton University Press, 1997.

[127] G. Navarro. A partial deterministic automaton for approximate string matching. In *Proceedings of the 4th South American Workshop on String Processing*, pages 112–124. Carleton University Press, 1997.

[128] G. Navarro. *Approximate Text Searching.* PhD thesis, University of Chile, Dept. of Computer Science, 1998.

[129] G. Navarro. A guided tour to approximate string matching. *ACM Computing Surveys*, 33(1):31–88, 2001.

[130] G. Navarro and M. Raffinot. A bit-parallel approach to suffix automata: Fast extended string matching. In *Proceedings of the 9th Annual Symposium on Combinatorial Pattern Matching*, number 1448 in Lecture Notes in Computer Science, pages 14–33. Springer-Verlag, Berlin, 1998.

[131] G. Navarro and M. Raffinot. Practical and flexible pattern matching over ziv-lempel compressed text. *Journal of Discrete Algorithms*, 2003. to appear.

[132] S. Needleman and C. Wunsch. A general method applicable to the search for similarities in the amino acid sequences of two proteins. *Journal of Molecular Biology*, 48:444–453, 1970.

[133] J. Nesbit. The accuracy of approximate string matching algorithms. *Journal of Computer-Based Instruction*, 13(3):80–83, 1986.

[134] B. Nichols, D. Buttlar, and J. P. Farrell. *Pthreads Programming.* O'Reilly, 1996.

[135] O. Owolabi and R. McGregor. Fast approximate string matching. *Software Practice and Experience*, 18(4):387–393, 1988.

[136] P. S. Pacheco. *Parallel Programming with MPI.* San Francisco, CA, Morgan Kaufmann, 1997.

[137] P. Pevzner and M. Waterman. Multiple filtration and approximate pattern matching. *Algorithmica*, 13(1-2):135–154, 1995.

[138] G. Pfister. *In Search of Clusters.* Prentice Hall, 1998.

[139] T. Raita. Tunning the boyer-moore-horspool string searching algorithm. *Software Practice and Experience*, 22(10):879–884, 1992.

[140] Y. Robert and M. Tchuente. A systolic array for the longest common subsequence problem. *Information Processing Letters*, 21(4):191–198, 1985.

[141] D. Sankoff and J. Kruskal. *Time Warps, String Edits and Macromolecules: The Theory and Practice of Sequence Comparison.* Addison-Wesley, 1983.

[142] R. Sastry and N. Ranganathan. A systolic array for approximate string matching. In *Proceedings of International Conference on Computer Design*, pages 402–405, 1993.

[143] R. Sastry, N. Ranganathan, and K. Remedios. CASM: A VLSI chip for approximate string matching. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 17(8):824–830, 1995.

[144] P. Sellers. On the theory and computation of evolutionary distances. *SIAM Journal of Applied Mathematics*, 26:787–793, 1974.

[145] P. H. Sellers. The theory and computations of evolutionary distances: pattern recognition. *Journal of Algorithms*, 1:359–373, 1980.

[146] R. Sidhu, A. Mei, and V. K. Prasanna. String matching on multicontext FPGAs using self-reconfiguration. In *Proceedings of International Symposium on Field-Programmable Gate Arrays*, 1999.

[147] R. Sidhu and V. K. Prasanna. Fast regular expression matching using FPGAs. In *Proceedings of IEEE Symposium on Field-Programmable Custom Computing Machines*, 2001.

[148] D. F. Sittig, D. Foulser, N. Carriero, G. McCorkle, and P. L. Miller. A parallel computing approach to genetic sequence comparison: The master-worker paradigm with interworker communication. *Computer and Biomedical Research*, 24:152–169, 1991.

[149] G. Smit and V. De. A comparison of three string matching algorithms. *Software Practice and Experience*, 12(1):57–66, 1982.

[150] P. Smith. Experiments with a very fast substring search algorithm. *Software Practice and Experience*, 21(10):1065–1074, 1991.

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

[152] M. Snir, S. Otto, S. Huss-Lederman, D. W. Walker, and J. Dongarra. *MPI: The complete reference*. Massachusetts: The MIT Press, 1996.

[153] G. A. Stephen. *String searching algorithms*. World Scientific Press, 1994.

[154] D. Sunday. A very fast substring search algorithm. *Communications of the ACM*, 33(8):132–142, 1990.

[155] E. Sutinen and J. Tarhio. On using q-gram locations in approximate string matching. In *Proceedings of the 3rd Annual European Symposium*, number 979 in Lecture Notes in Computer Science, pages 327–340. Springer-Verlag, Berlin, 1995.

[156] T. Takaoka. Approximate pattern matching with samples. In *Proceedings of the 5th International Symposium on Algorithm and Computation*, number 834 in Lecture Notes in Computer Science, pages 234–242. Springer-Verlag, Berlin, 1994.

[157] J. Tarhio and E. Ukkonen. Approximate boyer-moore string matching. *SIAM Journal on Computing*, 22(2):243–260, 1993.

[158] E. Ukkonen. Algorithms for approximate string matching. *Information and Control*, 4(1-3):100–118, 1985.

[159] E. Ukkonen. Finding approximate patterns in strings. *Journal of Algorithms*, 6:132–137, 1985.

[160] E. Ukkonen. Approximate string matching with q-grams and maximal matches. *Theoretical Computer Science*, 92(1):191–211, 1992.

[161] E. Ukkonen and D. Wood. Approximate string matching with suffix automata. *Algorithmica*, 10(5):353–364, 1993.

[162] E. Lusk W. Gropp and A. Skjellum. *Using MPI: Portable parallel programming with the message passing interface.* Massachusetts: The MIT Press, 1994.

[163] R. A. Wagner and M. J. Fisher. The string to string correction problem. *Journal of the ACM*, 21:168–173, 1974.

[164] M. Waterman. *Introduction to Computational Biology.* Chapman and Hall, 1995.

[165] B. Wilkinson and M. Allen. *Parallel Programming: Techniques and applications using networked workstations and parallel computers.* Prentice Hall, 1999.

[166] A. Wright. Approximate string matching using within-word parallelism. *Software Practice and Experience*, 24(4):337–362, 1994.

[167] S. Wu and U. Manber. Fast text searching allowing errors. *Communications of the ACM*, 35(10):83–91, 1992.

[168] S. Wu, U. Manber, and G. Myers. A subquadratic algorithm for approximate limited expression matching. *Algorithmica*, 15:50–67, 1996.

[169] Y. Yan, X. Zhang, and Y. Song. An effective and practical performance prediction model for parallel computing on non-dedicated heterogeneous NOW. *Journal of Parallel and Distributed Computing*, 38(1):63–80, 1996.

[170] C-B. Yang and R. C. T. Lee. Systolic algorithms for the longest common subsequence problem. *Journal of the Chinese Institute of Engineers*, 10(6):691–699, 1987.

[171] T. K. Yap, O. Frieder, and R. L. Martino. Parallel homologous sequence searching in large databases. In *Proceedings of the 5th IEEE Symposium Frontiers of Massively Parallel Computation*, pages 231–237, 1995.

[172] T. K. Yap, O. Frieder, and R. L. Martino. Parallel computation in biological sequence analysis. *IEEE Transactions on Parallel and Distributed Systems*, 9(3):283–293, 1998.

[173] J. Zobel and P. Dart. Phonetic string matching: lessons from information retrieval. In *Proceedings of the SIGIR'96*, pages 166–172, 1996.