ΑΝΑΦΟΡΑ ΕΡΓΑΣΙΑΣ PROJECT ΣΤΗΝ ΑΝΑΠΤΥΞΗ ΛΟΓΙΣΜΙΚΟΥ ΓΙΑ ΠΛΗΡΟΦΟΡΙΑΚΑ ΣΥΣΤΗΜΑΤΑ

Ομάδα 3 ατόμων:

- Κοκκινάκος Παναγιώτης 1115201400069
- Ρούβαλης Γεώργιος 1115201400173
- Στέφου Θεόδωρος 1115201400193

Η εργασία έγινε στη γλώσσα C και για τη μεταγλώττιση χρησιμοποιήθηκε ο gcc με -O2 optimization flag.

Επίσης χρησιμοποιήθηκαν τα: gitlab για version control, Unity για unit testing και valgrind για έλεγχο memory leaks.

Όλα τα αποτελέσματα έχουν ελεγχθεί με την εντολή diff και δεν έχουν βγάλει διαφορές.

Part 1

Η πρώτη προσέγγισή μας ήταν κάθε κόμβος να έχει πίνακα από δείκτες στους κόμβους-παιδιά και κατά τη εισαγωγή ή τη διαγραφή ενός στοιχείου του πίνακα κάναμε αντιγραφές με αναθέσεις.
Επίσης, για εισαγωγή και διαγραφή στοιχείων από τους πίνακες κάναμε γραμμική αναζήτηση.
Σε αυτή την φάση το πρόγραμμα έτρεχε γύρω στο 1 λεπτό και 30 δευτερόλεπτα.
Από το πρώτο κιόλας part αλλάξαμε τα παραπάνω με πίνακα από κόμβους-παιδιά αντί για δείκτες ώστε να εκμεταλλευτούμε την τοπικότητα της μνήμης καθώς και με τη χρήση της συνάρτησης memmove αντί των αναθέσεων. Αυτές οι δύο αλλαγές γλίτωσαν από το πρόγραμμά μας 30 δευτερόλεπτα φέρνοντας πλέον την εκτέλεσή του λίγο πιο κάτω από το 1 λεπτό. Αμέσως μετά, μεταγλωττίσαμε και με τη flag -O2, κάτι που μας γλίτωσε άλλα 25 δευτερόλεπτα. Τελικά, αλλάξαμε και τη γραμμική αναζήτηση με δυαδική όπου είδαμε και τη μεγάλη διαφορά: το πρόγραμμα έτρεχε πλέον σε λίγο παραπάνω από 1 δευτερόλεπτο.

Γενικότερα, στο πρώτο μέρος επικεντρωθήκαμε κυρίως στο να παίρνουμε σωστά αποτελέσματα, οπότε δεν υλοποιήσαμε κάποια επιπλέον δομή για τη βελτίωση του χρόνου.

Part 2

Στο δεύτερο μέρος προστέθηκαν οι δομές BloomFilter και Linear Hash Table για τη βελτίωση της ταχύτητας καθώς και η έννοια του στατικού trie για τη μείωση του χώρου που καταλαμβάνει. Επίσης προστέθηκε η λειτουργία Top K για την εύρεση των K δημοφιλέστερων n-grams σε κάθε ριπή.

Συγκεκριμένα, το BloomFilter είχε έναν πίνακα με κελιά από bytes(char) αντί για int ώστε να μην σπαταλάμε τον τετραπλάσιο χώρο, εφόσον οι τιμές είναι boolean. Για το "καθάρισμα" του δέντρου δεν χρησιμοποιήθηκαν αναθέσεις αλλά η συνάρτηση memset.
Για το hashing του BloomFilter χρησιμοποιήθηκε η συνάρτηση MurmurHash2 στην οποία εκμεταλλευτήκαμε το γεγονός ότι κάνει hash σε διαφορετικές θέσεις ανάλογα το φύτρο που θα λάβει ως όρισμα και έτσι είχαμε στη διάθεση μας όσες hash συναρτήσεις χρειαζόμασταν.
Αναλόγως το μέγεθος του αρχείου αρχικοποίησης του δέντρου, βρήκαμε διαφορετικές τιμές για το μέγεθος τους πίνακα και το πλήθος των συναρτήσεων, ώστε τα αποτελέσματα με τις συγκεκριμένες εισόδους να μην βγάζουν κανένα απολύτως λάθος.
Συγκεκριμένα:
Για το small 9280 κελιά και 4 συναρτήσεις
Για το medium 57500 κελιά και 5 συναρτήσεις

Προτιμήσαμε να έχουμε περισσότερα κελιά αντί για συναρτήσεις ώστε να διατηρούμε την ταχύτητα του προγράμματος χαμηλή. Αρχικά, επειδή είχαμε πολλές συναρτήσεις αντί για πολλά κελιά, η δομή του BloomFilter επιβράδυνε το πρόγραμμα. Όταν το αλλάξαμε, είδαμε διαφορά με μέσο όρο στο medium dataset μείον 4 δευτερόλεπτα και για το small dataset μείον 0.3 δευτερόλεπτα.
Κάθε trie χρησιμοποιούσε 1 Bloom Filter.

Επίσης, προστέθηκε ο νέος τύπος στατικού δέντρου για τον οποίο χρησιμοποιήσαμε διαφορετικούς τύπους κόμβων και διαφορετική συνάρτηση για το Q query.
Λόγω του ότι δεν ήταν γνωστός την ώρα της μεταγλώττισης ο τύπος του δέντρου, χρησιμοποιήσαμε void* για το δείκτη στο δέντρο και function pointers ώστε να πετύχουμε ένα μικρό μοντέλο αντικειμενοστραφούς υποτυπισμού. Κατά το διάβασμα της πρώτης σειράς του αρχείου, αναθέτουμε στους function pointers μας τις συναρτήσεις που θα χρησιμοποιήσουμε κατά τη διάρκεια του υπόλοιπου προγράμματος. Για να το πετύχουμε αυτό, οι συναρτήσεις των δέντρων παίρνουν πλέον void* για να είναι ίδιες και να δηλωθούν σωστά οι αντίστοιχοι function pointers και μέσα στο σώμα κάνουν cast το δέντρο σε ό,τι τελικά θα χρησιμοποιήσουν. Για το στατικό προσθέσαμε και κάποιες ψεύτικες συναρτήσεις για τα add και delete, ώστε σε περίπτωση που υπάρχει λάθος σε κάποιο αρχείο και γίνει

π.χ. A query σε στατικό δέντρο να μην χρησιμοποιηθεί NULL function pointer και καταρρεύσει το πρόγραμμα αλλά απλά να μην γίνει τίποτα.

Πέρα από τα παραπάνω, προστέθηκε στο πρώτο επίπεδο και των δύο δέντρων ένα Linear Hashtable το οποίο βελτίωσε το χρόνο εκτέλεσης του small dataset κατά μισό δευτερόλεπτο. Μία σχεδιαστική επιλογή που κάναμε για να αποφύγουμε τα διαδοχικά splits κατά την υπερχείλιση κάποιου bucket, ήταν το καινούριο bucket που δημιουργείται σε κάθε υπερχείλιση, να έχει ίδιο μέγεθος με το bucket που θα γίνει split. Στα υπόλοιπα dataset είδαμε μία βελτίωση με μέσο όρο 12 δευτερόλεπτα.

Τέλος, προστέθηκε και η λειτουργία Top k, η οποία αν και επιβράδυνε κάπως το πρόγραμμα, κάτι το οποίο είναι απόλυτα φυσιολογικό αφού προστίθεται παραπάνω λειτουργία, η αλλαγή δεν ήταν ιδιαίτερα σημαντική. Για τα medium και large datasets ο χρόνος αυξήθηκε γύρω στο 1.5 με 2 δευτερόλεπτα, ενώ στο small δεν παρατηρήθηκε καμία αλλαγή.

Για το στατικό δέντρο, μαζέψαμε τα παρακάτω στατιστικά για πριν και μετά τη συμπίεση:

Small dataset:
Before compression: 997,230 bytes 2,358 nodes
After compression: 841,529 bytes 1,981 nodes
Compression time: 0.013s

Medium dataset:
Before compressing: 18,950,786 bytes 44,376 nodes
After compressing: 15,828,919 bytes 36,817 nodes
Compression time: 0.059s

Large dataset:
Before compressing: 1,292,915,312 bytes 3,007,804 nodes
After compressing: 511,564,285 bytes 1,122,525 nodes
Compression time: 0.539s

Από τα παραπάνω παρατηρούμε ότι η συμπίεση έχει νόημα όσο αυξάνεται το μέγεθος της εισόδου. Συγκεκριμένα:

Dataset/%Βελτίωσης	Bytes	Κόμβοι
Small	15.6%	15.9%
Medium	16.4%	17%
Large	60.4%	62.6%

Επίσης, σημειώνεται ότι ο χρόνος της συμπίεσης είναι ελάχιστος σε σχέση με την εξοικονόμιση χώρου που προσφέρει.

Στο τέλος της εκπόνησης του δεύτερου μέρους, μαζέψαμε τα παρακάτω στατιστικά χρόνων σε Intel Core i5 CPU 750 2.67GHz × 4.

Για να υπάρχει μεγαλύτερη αντικειμενικότητα στις μετρήσεις, τα στατικά δέντρα μετρήθηκαν συμπιεσμένα αλλά και μη.

Small static compressed : 0.546 sec
Small static not compressed : 0.509 sec
Small dynamic : 0.582 sec

Medium input compressed : 23.192 sec
Medium input not compressed :19.746 sec
Medium dynamic: 27.756 sec

Από τα παραπάνω παρατηρούμε ότι η συμπίεση προκαλεί μία μικρή αύξηση του χρόνου, όχι όμως μόνο λόγω της συμπίεσης αλλά και επειδή σε συμπιεσμένα δέντρα αυξάνονται οι συγκρίσεις και οι συνθήκες που υπολογίζονται για τα question queries.

Τα δυναμικά δέντρα είναι πιο αργά στην εκτέλεση επειδή εκτελούν και πράξεις πρόσθεσης και διαγραφής.

Στις παρακάτω μετρήσεις φαίνεται η χρονική διαφορά μεταξύ του 1ου μέρους και του 2ου για τα small και medium datasets
(με dynamic trie αφού στο part 1 δεν υπήρχε η έννοια του static).

1ο μέρος small: 1.059 sec

2ο μέρος small: 0.582 sec

1ο μέρος medium : 63.042 sec

2ο μέρος medium : 27.756 sec

Φαίνεται ότι η προσθήκη των δομών BloomFilter και Linear Hashtable βελτίωσε αισθητά το χρόνο εκτέλεσης.

Part 3

Στο τρίτο μέρος, αρχικά μετατρέψαμε τον πίνακα από bytes του BloomFilter σε bit vector. Πλέον κάθε θέση είναι ένα bit στο οποίο έχουμε πρόσβαση μέσω ειδικών συναρτήσεων. Με αυτό, φέραμε το χώρο που καταλαμβάνει το BloomFilter στο 1/8 του προηγουμένου.

Επίσης αλλάξαμε τον τρόπο με τον οποίο συλλέγονται τα αποτελέσματα της κάθε ριπής. Πλέον οι Queues με τα αποτελέσματα που επιστρέφονται από κάθε question δεν αποθηκεύονται σε Queue, αλλά σε πίνακα για λόγο ο οποίος τεκμηριώνεται παρακάτω.

Υλοποίησαμε έναν job scheduler γενικού σκοπού ο οποίος δημιουργεί έναν αριθμό από νήματα τα οποία περιμένουν σε μία condition variable μέχρι ο scheduler να σηματοδοτήσει την αρχή της λειτουργίας τους. Για κάθε question, η δουλειά προστίθεται στην ουρά του scheduler και όταν βρεθεί το F και τελειώσει η ριπή αυτός στέλνει σήμα στα threads να αρχίσουν να δουλεύουν. Τα threads ξυπνάνε το ένα το άλλο μέχρις ότου να μην υπάρχουν άλλες δουλειές στην ουρά. Το main thread περιμένει όλες τις δουλειές να τελειώσουν προτού προχωρήσει στην εκτύπωση των αποτελεσμάτων και την εύρεση των Top K n-grams. Όταν βρίσκουμε εισαγωγή ή διαγραφή ngrams(στα δυναμικά tries), αυτές πραγματοποιούνται από το main thread επί τόπου, καθώς δεν γίνεται παραλληλοποίηση για αυτά. Το κάθε question σε κάθε ριπή έχει μοναδικό id, το οποίο ξεκινάει απο το 0 και αυξάνεται γραμμικά, οπότε γράφει το αποτέλεσμά του στην θέση του πίνακα αποτελεσμάτων που αντιστοιχεί στο id του. Με αυτό τον τρόπο, δεν μας ενδιαφέρει με ποιά σειρά τελείωσαν τα questions, αφού στην εκτύπωση των αποτελεσμάτων εκτυπώνουμε γραμμικά τα περιεχόμενα του πίνακα με αυτά.

Τα στατικά δέντρα παραλληλοποιήθηκαν πολύ πιο εύκολα από τα δυναμικά επειδή δεν υπάρχουν εισαγωγές και διαγραφές στο work αρχείο τους.

Για τα δυναμικά δέντρα υλοποιήσαμε το versioning, δηλαδή σε κάθε query αντιστοιχίσαμε ένα id ώστε να ξέρουμε πότε προστέθηκε και πότε διαγράφηκε ένα n-gram, για να μην έχουμε εκτύπωση λάθος αποτελεσμάτων, αφού όλα τα question κάθε ριπής εκτελούνται μετά από όλες τις εντολές εισαγωγής και διαγραφής. Κάθε κόμβος του δέντρου έχει δύο καινούριες μεταβλητές οι οποίες εκφράζουν πότε προστέθηκε και πότε διαγράφηκε ο κόμβος αυτός. Άν δεν έχει διαγραφεί η μεταβλητή που δείχνει το πότε διαγράφηκε είναι ίση με το -1. Οι μεταβλητές αυτές μας ενδιαφέρουν μόνο για τους final κόμβους. Κάθε Question περιλαμβάνει στο αποτέλεσμά της όποια n-grams έχουν προστεθεί πιο νωρίς από όταν έγινε το question, δηλαδή όποια έχουν id εισαγωγής μικρότερο από το id του question, και ταυτόχρονα όταν έχουν id διαγραφής -1 ή μεγαλύτερο από το id του question.

Πλέον, όταν εμφανίζεται εντολή διαγραφής, στην πραγματικότητα δεν διαγράφεται κατευθείαν το n-gram, αλλά αλλάζουμε τη μεταβλητή που εκφράζει το χρόνο στον οποίο διαγράφηκε. Αυτό συμβαίνει έτσι ώστε να λειτουργήσει ομαλά το versioning. Όταν βρούμε εντολή F, τότε καθαρίζουμε και ετοιμάζουμε το trie για την επόμενη ριπή, δηλαδή διαγράφουμε όσα n-gram πρέπει να διαγραφούν, και αρχικοποιούμε πάλι τις μεταβλητές εισαγωγής και διαγραφής σε 0 και -1 αντίστοιχα. Επίσης, όταν βρεθεί F, η αρίθμηση των id των queries αλλά και τον questions αρχίζει πάλι από την αρχή για την επόμενη ριπή.

Πλέον, λόγω της παραλληλοποίησης των questions, δεν μπορούμε να έχουμε μόνο 1 Bloom Filter για όλο το trie, αφού έτσι θα χρησιμοποιούταν ταυτόχρονα από n σε αριθμό questions, όπου n το πλήθος των threads. Για να αντιμετωπιστεί αυτό, κάθε thread έχει δικό του Bloom Filter, και αν το job το οποίο έχει αναλάβει είναι question, τότε του το δίνει σαν όρισμα.

Ένα bug που συναντήσαμε προκλήθηκε από τη χρήση της strtok μέσα στις questions όταν έτρεχαν παράλληλα 2 ή περισσότερα νήματα. Με λίγη έρευνα καταλάβαμε ότι η strtok δεν είναι thread safe και για αυτό την αντικαταστήσαμε με την strtok_r.

Σύνοψη μετρήσεων χρόνων εκτέλεσης για το τρίτο μέρος

Intel Core i5 CPU 750 2.67GHz × 4

Δυναμικά δέντρα:

Dataset/# threads	1 thread	2 threads	3 threads	4 threads	5 threads
Small	0.684s	0.430s	0.314s	0.248s	0.256s
Medium	22.992s	14.048s	11.403s	9.579s	9.741s
Large	69.259s	52.022s	48.626s	46.210s	46.777s

Στατικά δέντρα:

Dataset/# threads	1 thread	2 threads	3 threads	4 threads	5 threads
Small	0.575s	0.371s	0.276s	0.215s	0.234s
Medium	25.965s	12.829s	10.031s	9.975s	10.037s
Large	30.463s	18.054s	14.414s	12.613s	12.990s

AMD Phenom 965 CPU 3.4GHz × 4

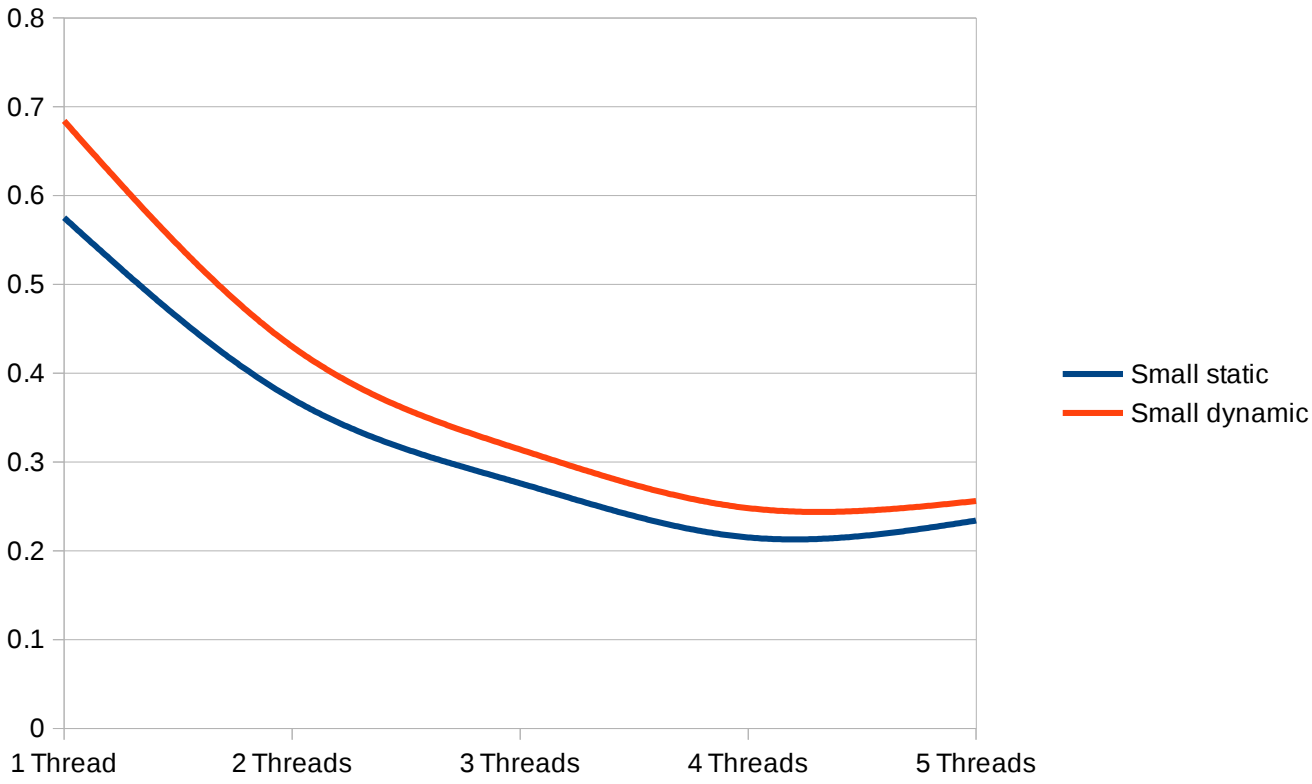
Δυναμικά δέντρα:

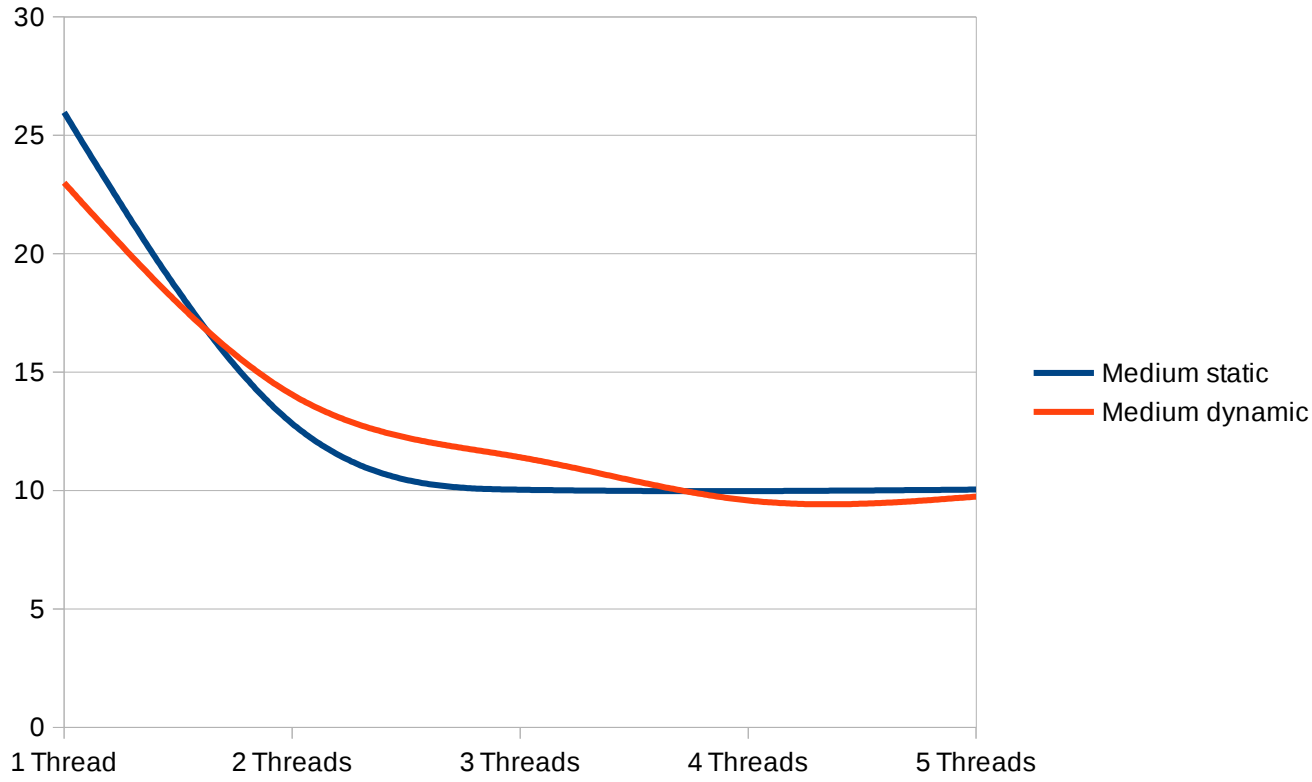
Dataset/# threads	1 thread	2 threads	3 threads	4 threads	5 threads
Small	0.666s	0.388s	0.316s	0.243s	0.246s
Medium	24.972s	15.306s	11.833s	10.180s	10.226s
Large	80.794s	67.290s	59.810s	57.200s	58.270s

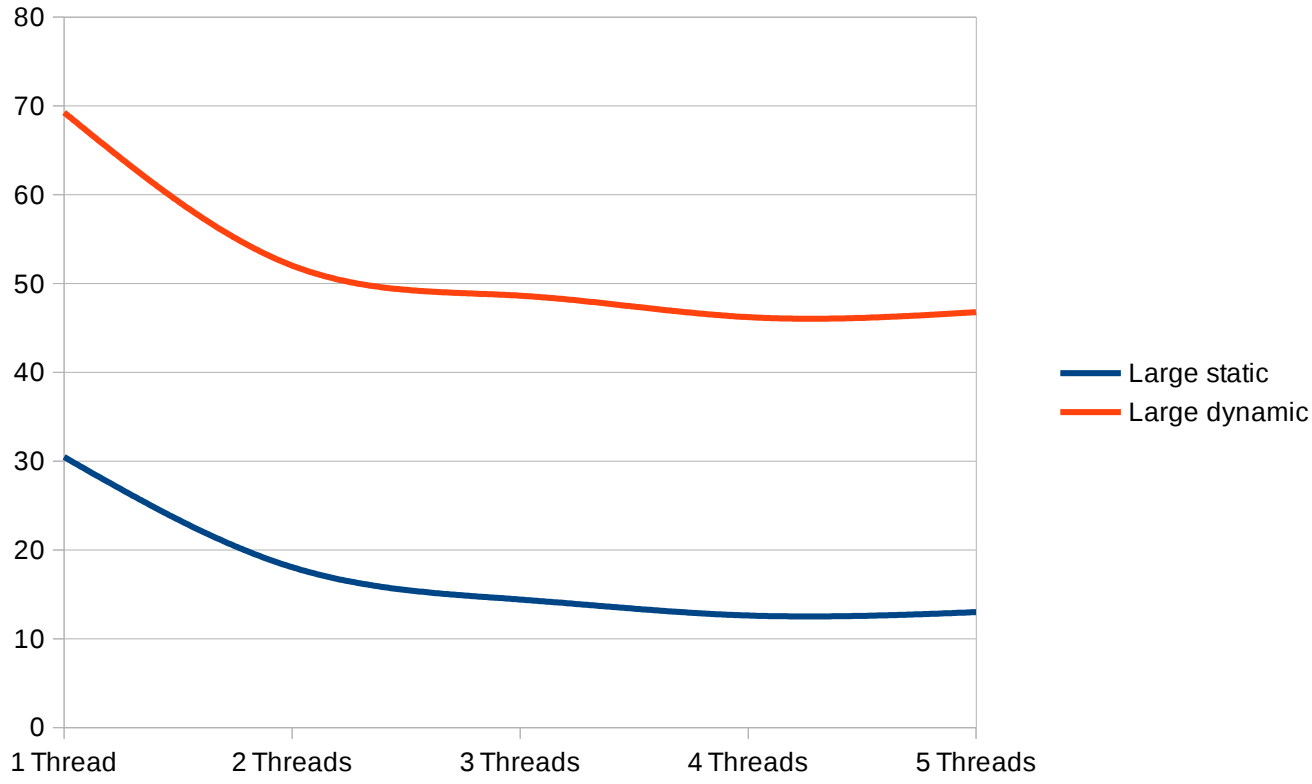
Στατικά δέντρα:

Dataset/# threads	1 thread	2 threads	3 threads	4 threads	5 threads
Small	0.647s	0.374s	0.293s	0.216s	0.217s
Medium	26.854s	13.604s	10.241s	8.408s	8.453s
Large	41.610s	22.236s	17.521s	14.823s	14.899s

Παρακάτω φαίνονται μερικά γραφήματα του χρόνου συναρτήσει του πλήθους των νημάτων. Οι μετρήσεις είναι από τον επεξεργαστή intel που χρησιμοποιήθηκε.







Γενικά συμπεράσματα:

Παρατηρούμε ότι για το ίδιο δέντρο εάν πραγματοποιήσουμε συμπίεση πάνω σε αυτό αυξάνεται ο χρόνος εκτέλεσης αλλά μειώνεται σημαντικά ο αριθμός των κόμβων και των bytes που χρησιμοποιούνται κάτι το οποίο είναι επιθυμητό.

Με την εισαγωγή των threads, οι χρόνοι εκτέλεσης μειώνονται σημαντικά.
Όσο αφορά το πλήθος των threads, στους επεξεργαστές με 4 πυρήνες όπου τρέξαμε το πρόγραμμά μας, ο χρόνος μειώνεται για κάθε επιπλέον thread μέχρι τα 4. Από εκεί και πέρα παραμένει σταθερός και ίσως αυξάνεται ελάχιστα λόγω του κόστους επικοινωνίας, όπως φαίνεται και στα γραφήματα.

Κατά μέσο όρο τα δυναμικά δέντρα έχουν μεγαλύτερο χρόνο εκτέλεσης από τα στατικά για τα αντίστοιχα αρχεία, κάτι που είναι όμως αναμενόμενο, αφού στα .work αρχεία των δυναμικών υπάρχουν και οι χρονοβόρες εντολές εισαγωγής και διαγραφής.