ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ
ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ

Μελέτη τεχνικών βελτιστοποίησης για εφαρμογές με μη κανονικές προσβάσεις στη μνήμη

Διπλωματική Εργασία

Κωνσταντίνος Θ. Κανελλόπουλος

Επιβλέπων: Επ. Καθηγητής Γεώργιος Γκούμας

Αθήνα, Μάρτιος 2018

ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ
ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ

Μελέτη τεχνικών βελτιστοποίησης για εφαρμογές με μη κανονικές προσβάσεις στη μνήμη

Διπλωματική Εργασία

Κωνσταντίνος Θ. Κανελλόπουλος

Επιβλέπων: Επ. Καθηγητής Γεώργιος Γκούμας

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 18η Δεκεμβρίου 2017.

.....	.....	.....
Γεώργιος Γκούμας	Νεκτάριος Κοζύρης	Νικόλαος Παπασπύρου
Επ. Καθηγητής Ε.Μ.Π.	Καθηγητής Ε.Μ.Π.	Αναπ. Καθηγητής Ε.Μ.Π.

Ευχαριστίες

Αρχικά, θα ήθελα να εκφράσω τις ευχαριστίες μου στον επιβλέποντα της διπλωματικής μου εργασίας Επίκουρο Καθηγητή κ.Γεώργιο Γκόυμα ο οποίος μου έδωσε την ευκαιρία να εργαστώ στο Εργαστήριο Υπολογιστικών Συστημάτων και του οποίου η καθοδήγηση ήταν καταλύτικη για την εκπόνηση της. Επίσης, θα ήθελα να ευχαριστήσω τον ΚαΘηγητή κ.Νεκτάριο Κοζύρη για το υψηλό επίπεδο των διαλέξεών του χάρις στις οποίες αγάπησα την Αρχιτεκτονική Υπολογιστών. Ευχαριστώ ιδιαίτερα τον υποψήφιο διδάκτωρ Δημήτριο Σιακαβάρα χωρίς τον οποίο δεν θα είχα καταφέρει να εκπονήσω την παρούσα εργασία. Ακόμη, οφείλω πολλά στον Μεταδιδακτορικό ερευνητή Κωνσταντίνο Νίκα ο οποίος με έκανε καλύτερο επιστήμονα και με βοήθησε από την πρώτη στιγμή τόσο στην επίτευξη της ολοκλήρωσης της διπλωματικής μου εργασίας όσο και των προσωπικών μου στόχων. Η επιτυχία μου στη σχολή οφείλεται σε πολύ μεγάλο βαθμό στην Ελεάνα, η οποία δεν σταμάτησε ούτε λεπτό να με στηρίζει και να με παρακινεί για το καλύτερο. Το ίδιο ισχύει και για τον Δημοσθένη με τον οποίο υπήρξαμε φίλοι και συνεργάτες τα τελευταία 5 χρόνια. Τέλος, ένα πολύ μεγάλο ευχαριστώ αξίζει στους γονείς μου και στους φίλους μου, οι οποίοι ήταν πάντα εκεί για εμένα.

Περίληψη

Πολλές σύγχρονες εφαρμογές χαρακτηρίζονται ως memory-bound λόγω ακανόνιστων πρόσβασεων στη μνήμη. Ο αλγόριθμος του Dijkstra ανήκει σε αυτή την κατηγορία εφαρμογών που πάσχουν από αυτού του είδους τις προσβάσεις. Πραγματοποιήσαμε εκτεταμένο profiling στον αλγόριθμο για να ανακαλύψουμε το bottleneck του και χρησιμοποιήσαμε prefetching λογισμικού για να το αντιμετωπίσουμε. Στις περιπτώσεις ακανόνιστων προσβάσεων στη μνήμη, το prefetching λογισμικού παρέχει την δυνατότητα prefetching εκμεταλλευόμενοι τα χαρακτηριστικά του αλγορίθμου. Εν αντιθέσει, οι hardware prefetchers δεν είναι τόσο ευέλικτοι. Σε αυτή τη διατριβή, παρουσιάζουμε το σχήμα Prefetch-Process-Thread-Alternation, το οποίο βασίζεται στο πιο απλό Prefetching Helper Thread σχήμα. Το PPTA χρησιμοποιεί δύο νήματα που εναλλάσσονται μεταξύ μίας φάσης prefetch και μίας φάσης εκτέλεσης, για να αποκρύψει την καθυστέρηση απόκρισης μνήμης που προκαλείται από τα ακανόνιστα μοτίβα πρόσβασης. Αξιολογούμε και τα δύο σχήματα χρησιμοποιώντας δύο διαφορετικές πλατφόρμες, μία από τις οποίες υποστηρίζει simultaneous multithreading. Τα πειράματά μας, σε γράφους με αυξανόμενες πυκνότητες, δείχνουν ότι το PPTA επιτυγχάνει επιτάχυνση έως 1.82 για αραιούς και 1.62 για πυκνούς γράφους.

Λέξεις κλειδιά: Μνήμη, Ακανόνιστες προσβάσεις, Συντομότερα μονοπάτια, Prefetching

Abstract

Many modern applications are memory-bound due to irregular memory access patterns. Dijkstra's algorithm belongs in this class of applications that suffer from this kind of accesses. We performed an extended profiling of the algorithm to discover its bottleneck and employed software prefetching to compromise it. When complex data access patterns are considered, the software approach provides the opportunity of performing sophisticated prefetching based on the characteristics of the algorithm whereas hardware schemes are not so flexible. In this thesis, we introduced the Prefetch-Process-Thread-Alternation scheme which is based on the Prefetching Helper Thread scheme. PPTA involves two threads that alternately switch between a prefetching and a process phase, to hide the memory latency caused by cache misses. We evaluate both schemes using two different platforms, one of which supports simultaneous multithreading. Our experiments, on graphs with increasing densities, show that PPTA achieves performance speedup up to 1.82 for sparse and 1.62 for dense graphs.

Keywords: Memory-bound, Irregular accesses, Dijkstra, Prefetching

Περιεχόμενα

Κατάλογος Σχημάτων

Κεφάλαιο 1

Εισαγωγή

Η βελτιστοποίηση της απόδοσης των εφαρμογών που περιορίζονται από τη μνήμη είναι μία δύσκολη εργασία και καθίσταται ακόμα πιο δύσκολη όταν δεν εκθέτουν παραλληλισμό με απλό τρόπο. Αυτές οι εφαρμογές τυπικά λειτουργούν σε μεγάλα σύνολα δεδομένων με περιορισμένη temporal locality. Το κόστος της εντατικής κίνησης των δεδομένων και των μη κανονικών προσβάσεων στη μνήμη είναι σημαντικό, αφού οι περισσότερες από αυτές τις προσπελάσεις εξυπηρετούνται από την κύρια μνήμη. Ο αποκαλούμενος τοίχος-μνήμης τονίζει την ανάγκη χρήσης τεχνικών που μειώνουν την καθυστέρηση μνήμης.

1.1 Σύγχρονα συστήματα

Τα σύγχρονα συστήματα έχουν πολλές δυνατότητες που μειώνουν με την καθυστέρηση μνήμης και αυξάνουν την απόδοση:

- Η τεχνική prefetching υλικού και λογισμικού έχει προταθεί ως μια από τις πιο σημαντικές λύσεις για την απόκρυψη της καθυστέρησης μνήμης στα συστήματα. Τα σύγχρονα συστήματα παρέχουν αρχιτεκτονική υποστήριξη για το prefetching λογισμικού και ένα μικρό αριθμό από prefetchers υλικού.

- Πολλαπλοί πυρήνες για την εκμετάλλευση του παραλληλισμού και την κλίμακα αποτελεσματικά.

- Η τεχνολογία HBM επιτυγχάνει υψηλότερο εύρος ζώνης, ενώ χρησιμοποιεί λιγότερη ισχύ σε πολύ μικρότερο μέγεθος από το DDR4 ή το GDDR5. Αυτό επιτυγχάνεται χρησιμοποιώντας 3D stacked τεχνολογία μνήμης, συμπεριλαμβανομένης μίας προαιρετικής βάσης μνήμης με ελεγκτή μνήμης, οι οποίες διασυνδέονται με θύρες μέσω πυριτίου TSV και των μικροβυθισμάτων.

- Processing-in-Memory που παρέχει βασικές λειτουργίες υπολογιστικής μονάδας στην κύρια μνήμη και μειώνει τις μεγάλες μεταφορές δεδομένων.

- Οι μονάδες επεξεργασίας γραφικών (GPU),η ιδιαίτερα παράλληλη δομή των οποίων, τις καθιστά αποδοτικότερες από τις CPU γενικής χρήσης για τους αλγόριθμους όπου η επεξεργασία μεγάλων ομάδων δεδομένων γίνεται παράλληλα.

1.2 Βελτιστοποίηση memory-bound εφαρμογών

Υπάρχουν επίσης τρόποι βελτιστοποίησης των εφαρμογών που συνδέονται με τη μνήμη:

- Εξαγωγή του παραλληλισμού και αξιοποιήση του συνολικού εύρου ζώνης μνήμης.
- Σχεδίαση δομών δεδομένων με υψηλή απόδοση.
- Profiling της εφαρμογής και βελτιστοποιήσεις μέσω του μεταγλωττιστή.
- Επιλογή του κατάλληλου αρχιτεκτονικού μοντέλου που ταιριάζει στις ανάγκες της εφαρμογής.

1.3 Ο αλγόριθμος του Dijkstra

Οι εφαρμογές γραφημάτων χρησιμοποιούνται ευρέως στις μέρες μας. Υπάρχουν πολλές εφαρμογές στην οικονομία, την αεροναυπηγική, τη φυσική, τη βιολογία (για την ανάλυση του DNA),τα μαθηματικά και άλλες περιοχές που χρησιμοποιούν τη θεωρία γραφημάτων για να μοντελοποιήσουν πολύπλοκα προβλήματα. Σε αυτή τη διπλωματική εργασία, εφαρμόζουμε prefetchingλογισμικού στον αλγόριθμο του Dijkstra , καθώς είναι ένα παράδειγμα memory-bound εφαρμογής της οποίας η απόδοση επηρεάζεται αρνητικά από μη κανονικές προσβάσεις στη μνήμη.

Κεφάλαιο 2

Αλγόριθμος του Dijkstra

2.1 Πρόβλημα συντομότερης διαδρομής

Στη θεωρία γραφημάτων, το πρόβλημα της μικρότερης διαδρομής είναι το πρόβλημα της εύρεσης μιας διαδρομής μεταξύ δύο κορυφών (ή κόμβων) σε ένα γράφημα, έτσι ώστε να ελαχιστοποιείται το άθροισμα των βαρών των συνιστώντων άκρων.

Το συντομότερο πρόβλημα διαδρομής μπορεί να οριστεί για γραφήματα ανεξάρτητα από το εάν είναι κατευθυνόμενα ή μικτά. Για κατευθυνόμενα γραφήματα, ο ορισμός της διαδρομής απαιτεί τη σύνδεση των διαδοχικών κορυφών με κατάλληλη κατευθυνόμενη ακμή.

Δύο κορυφές είναι γειτονικές εάν συνδέονται με μία ακμή. Μια διαδρομή σε ένα μη κατευθυνόμενο γράφημα είναι μια ακολουθία κορυφών $P = (v_1, v_2, ..., v_n) \epsilon V \times V \times ... \times V$ ώστε η κορυφή v_i να είναι γειτονική της v_{i+1} για $1 \leq i < n$. Αυτό το μονοπάτι P ορίζεται ως μονοπάτι μήκους $n-1$ το οποίο ξεκινάει από τον κόμβο v_1 και τερματίζει στον v_n.

Έστω $e_{i,j}$ η ακμή που ενώνει v_i και v_j. Δεδομένης μίας πραγματικής συνάρτησης βάρους $f : E \to \Re$, και ένα μη κατευθυνόμενο γράφο G, η συντομότερη διαδρομή από το v στο v' είναι το μονοπάτι $P = (v_1, v_2, ..., v_n)$ (όπου $v_1 = v$ και $v_n = v'$) το οποίο ελαχιστοποιεί το άθροισμα $\sum_{i=1}^{n-1} f(e_{i,i+1})$.

Υπάρχουν ορισμένες παραλλαγές του προβλήματος συντομότερης διαδρομής:

- Πρόβλημα συντομότερης διαδρομής με αφετηριακό κόμβο, στο οποίο πρέπει να βρούμε το συντομότερο μονοπάτι από έναν αφετηριακό κόμβο v προς όλους τους υπόλοιπους.

- Πρόβλημα συντομότερης διαδρομής με μοναδικό τελικό κόμβο, στο οποίο πρέπει να βρούμε τη συντομότερη διαδρομή από όλους τους κόμβους προς έναν κόμβο-προορισμό.

- Πρόβλημα συντομότερων μονοπατιών για όλα τα ζεύγη κορυφών, στο οποίο πρέπει να υπολογίσουμε το συντομότερο μονοπάτι κάθε ζευγαριού κόμβων v, v' του γράφου.

Οι πιο σημαντικοί αλγόριθμοι που λύνουν τα παραπάνω προβλήματα είναι οι εξής:

- **Ο Αλγόριθμος του Dijkstra** : λύνει το πρόβλημα συντομότερης διαδρομής με αφετηριακό κόμβο.

- **Ο Αλγόριθμος του Bellman-Ford** : λύνει το πρόβλημα συντομότερης διαδρομής με αφετηριακό κόμβο ακόμα και αν υπάρχουν ακμές με αρνητικά βάρη.
- **Ο Αλγόριθμος A*** : λύνει το πρόβλημα συντομότερης διαδρομής για ένα ζευγάρι κόμβων, χρησιμοποιώντας ευριστικές συναρτήσεις για την επιτάχυνση της εκτέλεσης.
- **Ο Αλγόριθμος των Floyd–Warshall** : λύνει το πρόβλημα συντομότερης διαδρομής για όλα τα ζεύγη κορυφών.
- **Ο Αλγόριθμος του Johnson** : λύνει το πρόβλημα συντομότερης διαδρομής για όλα τα ζεύγη κορυφών.

2.1.1 Εφαρμογές

Ο αλγόριθμος του Dijkstra είναι σε θέση να υπολογίσει τη συντομότερη διαδρομή από έναν κόμβο s σε κάθε άλλο κόμβο u σε ένα γράφημα με μη αρνητικά βάρη. Το εύρος των εφαρμογών στις οποίες μπορεί να χρησιμοποιηθεί είναι μεγάλο. Τα πρωτόκολλα δρομολόγησης δικτύου (RIPM, BGP), ο σχεδιασμός VLSI , η στοιχειοθέτηση Latex και τα κοινωνικά δίκτυα είναι μερικές από τις εφαρμογές στις οποίες χρησιμοποιείται ο αλγόριθμος. Ορισμένες περιπτώσεις χρήσης είναι επίσης:

- Πλοήγηση GPS & χάρτες
- Σχεδιασμός της κυκλοφορίας
- Ροβοτ Μοτιον ανδ Ναιγατιον
- Κίνηση ρομπότ και πλοήγηση
- Δίκτυα Μεταφορών
- Βιντεοπαιχνίδια και Εικονικές Περιηγήσεις

2.2 Υλοποίηση του αλγορίθμου του Dijkstra

2.2.1 Ψευδοκώδικας & Λειτουργίες & Δομές Δεδομένων

Ο αλγόριθμος του Dijkstra υπολογίζει τις πιο σύντομες διαδρομές μιας πηγής (SSSP) για κατευθυνόμενα γραφήματα με μη αρνητικές άκμές. Είναι ασυμπτωτικά ο γρηγορότερος γνωστός αλγόριθμος σύντομης διαδρομής μονής πηγής για γραφήματα με μη περιορισμένα και μη αρνητικά βάρη. Ο αλγόριθμος βασίζεται στην παρατήρηση ότι κάθε υποδιαδρομή οποιασδήποτε συντομότερης διαδρομής είναι η ίδια η συντομότερη διαδρομή (βέλτιστη υποδομή). Συγκεκριμένα, έστω $G = (V, E)$ ένα κατευθυνόμενο γράφημα μς $n = |V|$ κόμβους και συνάρτηση βάρους $w : E \to \Re^+$ η οποία αποδίδει μη αρνητικά πραγματικά βάρη στις ακμές του G. Για κάθε κορυφή v, το πρόβλημα SSSP υπολογίζει $\delta(v)$, την τιμή της συντομότερης διαδρομής από την κορυφή προέλευσης s έως τον κόμβο v. Για κάθε κορυφή v, ο αλγόριθμος του Dijkstra διατηρεί μια εκτίμηση βραχύτερης διαδρομής (ή προσωρινή απόσταση) $d(v)$, η οποία είναι ένα ανώτερο όριο για την πραγματική τιμή της συντομότερης διαδρομής από s σε v, $\delta(v)$. Αρχικά, το $d(v)$ έχει οριστεί σε $+\infty$ και μέσω διαδοχικών χαλαρώσεων στις ακμές

μειώνεται σταδιακά και συγκλίνει σε $d(v)$. Η χαλάρωση μίας ακμής (v, w) ορίζει την τιμή $d(w)$ σε $min\{d(w), d(v)+w(v, w)\}$, αν μπορεί να μειώσει το βάρος της μικρότερης διαδρομής από το s στο w μέσω του v. Τέλος, ο αλγόριθμος αυτός ενημερώνει έναν πίνακα που ονομάζεται $previous$, έτσι ώστε να μπορεί να ανασυγκροτηθεί αναδρομικά η συντομότερη διαδρομή προς μια κορυφή v.

Ο αλγόριθμος παρουσιάζεται με περισσότερη λεπτομέρεια στον Αλγόριθμο 1. Διατηρεί ένα υποσύνολο του V σε κατακρημνισμένους, εισαχθέντες στην ουρά προτεραιότητας και μη καταγεγραμμένους κόμβους. Οι καταγεγραμμένοι κόμβοι έχουν $d(v) = \delta(v)$; οι κόμβοι της ουράς έχουν $d(v) > \delta(v)$ και οι μη καταγεγραμμένοι $d(v) = \infty$. Αρχικά, μόνο ο s βρίσκεται στην ουρά με $d(s) = 0$ και όλοι οι υπόλοιποι κόμβοι είναι μη καταγεγραμμένοι. Σε κάθε επανάληψη του αλγορίθμου, η κορυφή με την μικρότερη εκτίμηση της συντομότερης διαδρομής επιλέγεται και η κατάσταση της μετατρέπεται μόνιμα σε εγκατεστημένη ενώ όλες οι εξερχόμενες ακμές χαλαρώνουν, προκαλώντας την είσοδο στην ουρά οποιουδήποτε από τους γείτονες της εν λόγω κορυφής, σε περίπτωση που ο αλγόριθμος δεν τον έχει επισκεφθεί ακόμα.

Η βασική δομή δεδομένων που βρίσκεται στην καρδιά του αλγορίθμου είναι μια ουρά προτεραιότητας των κόμβων, η οποία κατατάσσεται με βάση τις $d()$ τιμές. Η ουρά διατηρεί όλες τις κορυφές του γραφήματος εκτός από τις διευθετημένες και πρέπει να είναι ουρά ελάχιστης προτεραιότητας για να υποστηρίξει τη λειτουργία Extract-Min.

Σε κάθε επανάληψη, η κορυφή με το ελάχιστο κλειδί εξάγεται από την ουρά προτεραιότητας (**Extract-Min**) και η κατάστασή της μετατρέπεται σε $visited$. Στη συνέχεια, οι εξερχόμενες άκμές της εξαχθείσας κορυφής του χαλαρώνονται (**Update**) και τα κλειδιά d των γειτόνων, των οποίων το $state \neq visited$, μειώνονται αν χρειαστεί. Είναι επίσης απαραίτητο να ενημερωθεί ο $previous$ πίνακας και να μειωθεί το κλειδί κάθε γειτονικού κόμβου στην ουρά προτεραιότητας για να διατηρηθεί η ελάχιστη ιδιότητά του (**Decrease-Key**).

Η πολυπλοκότητα του αλγορίθμου εξαρτάται από το είδος της ουράς προτεραιότητας και επομένως από την πολυπλοκότητα των λειτουργιών της. Εξαρτάται επίσης από τη δομή δεδομένων που χρησιμοποιείται για την απεικόνιση του γραφήματος.

Η πολυπλοκότητα της λειτουργίας Extract-Min είναι $\mathcal{O}(V)$ δεδομένου ότι όλες οι κορυφές πρέπει να εξαχθούν από την ουρά προτεραιότητας. Η χρήση ενός κοινού δυαδικού σωρού έχει σαν αποτέλεσμα $\mathcal{O}(VlogV)$ αφού η λειτουργία Extract-Min αυτής της ουράς προτεραιότητας είναι ανάλογη προς το ύψος του σωρού. Κατά τη διάρκεια της φάσης χαλάρωσης χαλαρώνουμε τις ακμές που εξέρχονται από την εξαγόμενη κορυφή και ενημερώνουμε τον σωρό. Η λεγόμενη Decrease-Key λειτουργία της ουράς προτεραιότητας εκτελείται σε όλες τις ακμές του γράφου εισόδου και κοστίζει $\mathcal{O}(ElogV)$. Συνεπώς, ο χρόνος εκτέλεσης του αλγορίθμου χρησιμοποιώντας έναν κοινό δυαδικό σωρό είναι $\mathcal{O}(ElogV + VlogV)$.

2.2.2 Αναζήτηση ενιαίου κόστους

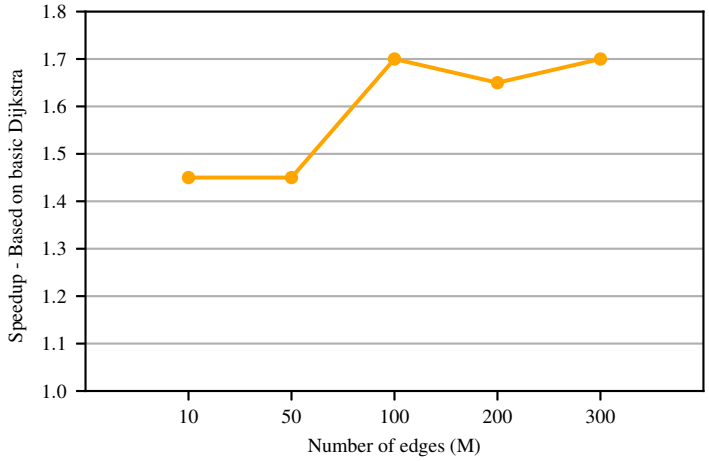
Στις απλές υλοποιήσεις του αλγορίθμου του Διθκστρα, αρχικά όλοι οι κόμβοι εισάγονται στην ουρά προτεραιότητας. Αυτό όμως δεν είναι απαραίτητο: ο αλγόριθμος μπορεί να ξεκινήσει με μια ουρά προτεραιότητας που περιέχει μόνο ένα στοιχείο και να εισάγει νέα στοιχεία καθώς ανακαλύπτονται (αντί να κάνουμε μία μείωση κλειδιού, ελέγχουμε εάν το κλειδί βρίσκεται στην ουρά, και εάν είναι μέσα, το μειώνουμε, διαφορετικά τοποθετούμε τον κόμβο με το κλειδί του). Αυτή η παραλλαγή έχει τα ίδια περιθώρια χειρότερης περίπτωσης με την απλή υλοποίηση, διατηρεί όμως στην ουρά μια ουρά

Algorithm 1 : Dijkstra's Algorithm

Require: Graph $G(V, E)$, Source vertex S
Ensure: Predecessors array *previous*
Shortest distance array d

1: **for all** $v \in V[G]$ **do**
2: $d[v] \leftarrow +\infty$
3: previous$[v] \leftarrow undefined$
4: $d[s] \leftarrow 0$
5: $S \leftarrow$ empty set } **Insert Operation**
6: $Q \leftarrow V[G]$
7: **while** Q is not empty **do**
8: $u \leftarrow$ **ExtractMin**(Q) // **Extract Min operation**
9: $S \leftarrow S \cup \{u\}$
10: **for all** edge (u, v) outgoing from u **do**
11: **if** $v \notin S$ **then**
12: $sum \leftarrow d[u] + w(u, v)$
13: **if** $sum < d[v]$ **then**
14: $u \leftarrow$ **DecreaseKey**(Q,v,sum) } **Relaxation**
15: $d[v] \leftarrow d[u] + w(u, v)$
16: $previous[v] := u$

προτεραιότητας μικρότερης προτεραιότητας, επιταχύνοντας τις λειτουργίες ουράς. Επιπλέον, η εισαγωγή όλων των κόμβων σε ένα γράφημα καθιστά δυνατή την επέκταση του αλγόριθμου ώστε να βρεθεί η συντομότερη διαδρομή από μία μόνο πηγή στο πλησιέστερο από ένα σύνολο κόμβων στόχου σε άπειρα γραφήματα ή εκείνα που είναι πολύ μεγάλα για να αναπαρασταθούν στη μνήμη. Ο αλγόριθμος που προκύπτει ονομάζεται αναζήτηση ενιαίου κόστους (UCS) στη βιβλιογραφία τεχνητής νοημοσύνης και περιγράφεται με χρήση ψευδοκώδικα στον Αλγ. 2.



Σχήμα 2.1: Επιτάχυνση με την χρήση της αναζήτησης ενιαίου κόστους.

Στο σχήμα 2.1 παρουσιάζουμε την επιτάχυνση που επιτεύχθηκε χρησιμοποιώντας τη βελτιστοποίηση UCSαντί του απλού αλγορίθμου Dijkstra. Η ουρά προτεραιότητας που χρησιμοποιήσαμε είναι ένας απλός δυαδικός σωρός και το γράφημα αποτελείται

Algorithm 2 : Dijkstra with UCS optimization

Require: Graph $G(V, E)$, Source vertex S
Ensure: Predecessors array *previous*
Shortest distance array d

```
1: for all v ∈ V[G] do
2:    d[v] ← +∞
3:    previous[v] ← undefined
4: d[s] ← 0 // Initialization
5: S ← empty set
6: Q ← s
7: while Q is not empty do
8:    u ← Extract-Min(Q) // Extract-Min
9:    S ← S ∪ {u}
10:   for all edge (u, v) outgoing from u do
11:      if v ∉ S then
12:         sum ← d[u] + w(u, v)
13:         if v ∉ Q then
14:            Insert(Q,v,sum)
15:         else
16:            if sum < d[v] then // Update
17:               u ← DecreaseKey(Q,v,sum)
18:               d[v] ← d[u] + w(u, v)
19:               previous[v] ← u
```

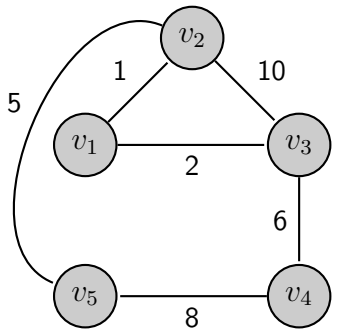
από 10Mκόμβους και 500Mακμές. Το σύστημα που χρησιμοποιήσαμε είναι ένας Intel-Broadwell-EPτου οποίου η διάταξη περιγράφεται στο Κεφάλαιο 5. Επιτυγχάνουμε επιτάχυνση 1.7 επειδή η ουρά προτεραιότητας παραμένει μικρότερη καθ΄όλη την εκτέλεση του αλγορίθμου. Με αυτόν τον τρόπο, δαπανάται λιγότερος χρόνο για την ενημέρωση της ουράς προτεραιότητας.

2.2.3 Αναπαράσταση γράφου

Υπάρχουν διάφοροι τρόποι για να αναπαραστήσουμε γράφους στη μνήμη, ο καθένας από τους οποίους έχει πλεονεκτήματα και μειονεκτήματα. Ορισμένα σημαντικά κριτήρια που πρέπει να ληφθούν υπόψη είναι το αποτύπωμα της μνήμης και ο χρόνος που χρειάζεται για να προσδιοριστεί αν μία συγκεκριμένη άκμή βρίσκεται στο γράφημα. Θα παρουσιάσουμε δύο τρόπους για να αποθηκεύσουμε έναν γράφο στη μνήμη.

Πίνακας Γειτνίασης

Με δεδομένο ένα γράφημα με $|V|$ κόμβους, ένας πίνακας γειτνίασης είναι ένας πίνακας $|V|times|V|$ πραγματικών ή δυαδικών τιμών (ανάλογα με τον τύπο του γραφήματος), όπου η καταχώρηση στη σειρά i και η στήλη j είναι 1 αν και μόνο αν η άκρη (i, j) γραφική παράσταση. Αν πρέπει να υποδείξουμε ένα βάρος ακμής, η καταχώρηση που αναφέρθηκε προηγουμένως περιέχει μια πραγματική τιμή w αν το κόστος της ακμής (i, j) είναι w. Δείχνουμε τον πίνακα γειτνίασης για ένα γράφο με ακμές δυο κατευθύνσεων:



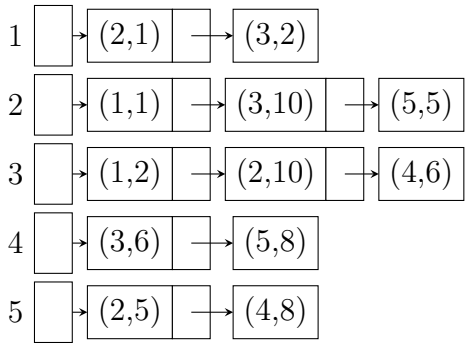
$$A_G = \begin{pmatrix} 0 & 1 & 2 & 0 & 0 \\ 1 & 0 & 10 & 0 & 5 \\ 2 & 10 & 0 & 6 & 0 \\ 0 & 0 & 6 & 0 & 8 \\ 0 & 5 & 0 & 8 & 0 \end{pmatrix}$$

Σχήμα 2.2: Παράδειγμα γράφου με τον αντίστοιχο πίνακα γειτνίασης A_G.

Λίστα γειτνίασης

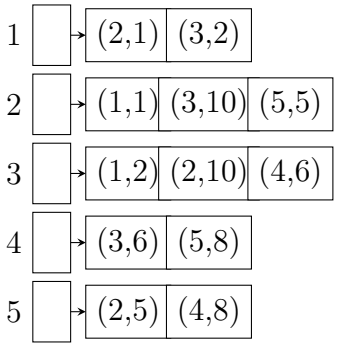
Ένας πιο αποδοτικός, ως προς τον χώρο αποθήκευσης, τρόπος για να υλοποιήσουμε ένα αραιά συνδεδεμένο γράφημα είναι να χρησιμοποιήσουμε μια λίστα γειτνίασης. Μια λίστα γειτνίασης αποτελείται από μια λίστα δομών δεδομένων που χρησιμοποιούν κλειδιά-τιμές όπου το κλειδί είναι μια κορυφή και η τιμή είναι το βάρος μιας ακμής. Για κάθε κορυφή i, αποθηκεύουμε μια σειρά από κορυφές, γειτονικές σε αυτήν. Συνήθως έχουμε μια σειρά από $|V|$ λίστες γειτνίασης, δηλαδή μία λίστα γειτνίασης ανά κορυφή. Για ένα κατευθυνόμενο γράφημα, η λίστα γειτνίασης περιέχει ένα σύνολο στοιχείων $|E|$, ένα στοιχείο ανά κατευθυνόμενη ακμή. Υπάρχουν πολλές παραλλαγές αυτής της βασικής ιδέας, που διαφέρουν στις λεπτομέρειες του τρόπου υλοποίησης των συλλογών. Θα παρουσιάσουμε δύο τυπικές υλοποιήσεις λιστών γειτνίασης.

Συνδεδεμένη Λίστα Στο σχήμα 2.3 παρουσιάζουμε τη λίστα γειτνίασης που αντιπροσωπεύει το γράφημα του Σχήματος 2.2. Δημιουργείται με τη χρήση συνδεδεμένων λιστών.



Σχήμα 2.3: Λίστα γειτνίασης με τη χρήση απλά συνδεδεμένων λιστών

Δυναμικός Πίνακας Στο σχήμα 2.4 η λίστα γειτνίασης δημιουργείται χρησιμοποιώντας δυναμικούς πίνακες.



Σχήμα 2.4: Λίστα γειτνίασης με χρήση δυναμικών πινάκων.

Πίνακας γειτνίασης vs Λίστα Γειτνίασης

Χώρος στη μνήμη

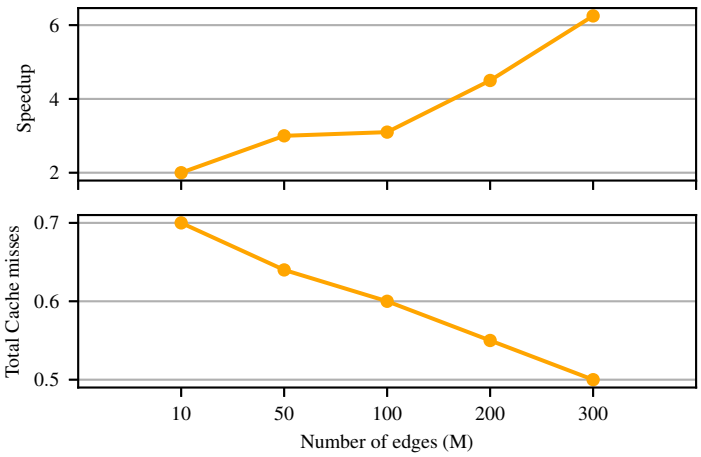
Όσον αφορά τον χώρο μνήμης, ο πίνακας γειτνίασης καταναλώνει $\mathcal{O}(V^2)$ και ακόμη και στην περίπτωση αραιών γραφημάτων, ο καταναλισκόμενος χώρος παραμένει ο ίδιος. Από την άλλη πλευρά, η λίστα γειτνίασης χρησιμοποιεί $\mathcal{O}(|V| + |E|)$ του χώρου. Στη χειρότερη περίπτωση, μπορεί να υπάρχει $C(V, 2)$ αριθμός ακμών σε ένα γράφημα καταναλώνοντας έτσι $\mathcal{O}(V^2)$ χώρο.

Λειτουργίες

Εκτός από το trade-off στον χώρο που καταλαμβάνουν στη μνήμη, οι διάφορες δομές δεδομένων διευκολύνουν επίσης διαφορετικές λειτουργίες. Η εύρεση όλων των κορυφών δίπλα σε μια δεδομένη κορυφή σε μια λίστα γειτνίασης είναι τόσο απλή όσο η ανάγνωση της λίστας και απαιτεί χρόνο ανάλογο με τον αριθμό των γειτόνων. Με ένα πίνακα γειτνίασης, πρέπει να σαρωθεί μια ολόκληρη σειρά, η οποία παίρνει ένα μεγαλύτερο χρονικό διάστημα, ανάλογο του αριθμού των κορυφών σε ολόκληρο το γράφημα. Από την άλλη πλευρά, ο έλεγχος της ύπαρξης μιας άκρης μεταξύ δύο δοσμένων κορυφών μπορεί να προσδιοριστεί ταυτόχρονα με ένα πλέγμα γειτονίας, ενώ απαιτεί χρόνο αναλογικός προς τον ελάχιστο βαθμό των δύο κορυφών με τον κατάλογο γειτνίασης.

Σε αυτή τη διπλωματική εργασία, οι γράφοι που χρησιμοποιούμε είναι αραιοί. Αυτός είναι ο λόγος για τον οποίο θα χρησιμοποιήσουμε την λίστα γειτνίασης για να τα αναπαραστήσουμε. Στο σχήμα 2.5 παρουσιάζουμε την επιτάχυνση που επιτυγχάνεται χρησιμοποιώντας μια δομή δεδομένων δυναμικού πίνακα αντί για συνδεδεμένη λίστα, για ένα γράφημα 10M κόμβων. Δείχνουμε επίσης τη μείωση των αστοχιών της κρυφής μνήμης .

Βλέπουμε ότι η χρήση ενός δυναμικού πίνακα αντί μιας απλά συνδεδεμένης λίστας οδηγεί σε αισθητή επιτάχυνση. Όσον αφορά τα πιο πυκνά γράμματα, κερδίζουμε μια αύξηση των επιδόσεων $\times 6$, η οποία μπορεί να αποδοθεί στο γεγονός ότι οι αστοχίες στην μνήμη ςαςηε μειώνονται κατά 50%. Οι αστοχίες στην κρυφή μνήμης μειώνονται, δεδομένου ότι η υλοποίηση με τον δυναμικό πίνακα ως λίστα γειτνίασης είναι πιο φιλική προς την κρυφή μνήμη σε σχέση με μια απλή-συνδεδεμένη λίστα. Όταν ο αλγόριθμος επισκέπτεται τους γείτονες της εξαχθείσας κορυφής και χαλαρώνει την ακμή $w(u, i)$,



Σχήμα 2.5: Επιτάχυνση χρησιμοποιώντας δυναμικούς πίνακες σε σύγκριση με απλά συνδεδεμένες λίστες.

δεν χρειαζόμαστε pointer chasing για να την αποκτήσουμε, ωστόσο μπορούμε απλά να χρησιμοποιήσουμε ένα ευρετήριο για να αποκτήσουμε πρόσβαση στον δυναμικό πίνακα. Αυτό οδηγεί στη μεταφορά ενός μπλοκ δεδομένων από τη μνήμη στην L1 cache αντί να φέρει μόνο ένα στοιχείο. Αυτό το μπλοκ αποτελείται από συνεχή στοιχεία του πίνακα που θα χρειαστούν κατά τις επόμενες επαναλήψεις. Επομένως, αυτό το κομμάτι των δεδομένων θα είναι στην L1 cache και οι αστοχίες μειώνονται.

2.3 Ουρές προτεραιότητας

Ένας από τους σημαντικότερους σωρούς που έχουν προταθεί και αφορά την υπόθεση αλγορίθμου Dijkstra είναι ο σωρός Fibonacci που βασίζεται στον διωνυμικό σωρό. Αυτός ο σωρός έχει αναπτυχθεί από τους Φρεδμαν et al. [1] και έχει καλύτερα αποσβενήμενο χρόνο λειτουργίας από πολλές άλλες δομές δεδομένων ουράς προτεραιότητας, συμπεριλαμβανομένου του δυαδικού σωρού και του διωνυμικού σωρού. Αν και οι σωροί Φιβοναςςι φαίνονται πολύ αποτελεσματικοί, έχουν τα ακόλουθα δύο μειονεκτήματα: Είναι περίπλοκοι όσον αφορά τον προγραμματισμό τους. Επιπλέον, δεν είναι τόσο αποτελεσματικοί στην πράξη σε σύγκριση με τις θεωρητικά λιγότερο αποτελεσματικές μορφές σωρών, καθώς στην απλούστερη εκδοχή τους απαιτούν αποθήκευση και χειρισμό τεσσάρων δεικτών ανά κόμβο, σε σύγκριση με τους δύο ή τρεις δείκτες ανά κόμβο που απαιτούνται για άλλες δομές.

Επιπλέον, η ουρά Brodal είναι ουρά προτεραιότητας με πολύ χαμηλή χρονική πολυπλοκότητα χειρότερης περίπτωσης [2]. Είναι η πρώτη παραλλαγή σωρού για την επίτευξη αυτών των ορίων χωρίς προσφυγή σε απόσβεση κόστους των λειτουργιών. Αν και έχουν καλύτερα ασυμπτωτικά όρια από άλλες δομές ουράς προτεραιότητας, είναι, κατά τα λόγια του ίδιου του Βροδαλ, ¨αρκετά περίπλοκοι' και ¨δεν εφαρμόζονται στην πράξη¨.

Αρκετές δουλειές έχουν προτείνει ουρές προτεραιότητας που επιτυγχάνουν υψηλές διεκπεραιωτικές ικανότητες κάτω από υψηλό πόλεμο. Στο [3] οι Rihani et al. προτείνουν ένα σχήμα MultiQueue το οποίο διαχειρίζεται μια σειρά από ουρές προτεραιότητας cp όπου το p είναι ο αριθμός των νημάτων του συστήματος. Η αξιολόγησή τους δείχνει ότι η κλίμακωση των MultiQueues είναι πολύ καλή σε συστήματα με ένα socket. Ωστόσο, η σημασιολογία της λειτουργίας Extract-Min είναι χαλαρή και η ενσωμάτωση

στον αλγόριθμο του Dijkstra είναι πολύπλοκη.

Αρκετή έρευνα έχει επίσης διεξαχθεί ως προς τη χρήση παράλληλων ουρών προτεραιότητας βασισμένων σε skiplist παρά στους σωρούς [4, 5]. Η Skip list παρέχει αρκετές δυνατότητες για την εξαγωγή παραλληλισμού, όμως με αυξημένο κόστος στο αποτύπωμα μνήμης και επομένως χρησιμοποιείται περιστασιακά ως ουρά προτεραιότητας.

Τέλος, έχουν προταθεί πολλές ουρές προτεραιότητας που λαμβάνουν υπόψιν την δομή της κρυφής μνήμη και χρησιμοποιούνται σε περίπτωση μεγάλων συνόλων δεδομένων [6].

2.4 Παραλληλοποιώντας τον αλγόριθμο του Dijkstra

Για να σχεδιάσουν παράλληλες εκδόσεις του αλγορίθμου, οι ερευνητές ακολουθούν δύο γενικές στρατηγικές.

2.4.1 Παραλληλοποίηση εξωτερικού βρόγχου

Η πρώτη στρατηγική αφορά την χαλάρωση της σειριακής φύσης του αλγορίθμου, παραλληλοποιώντας τον εξωτερικό βρόγχο. Αυτή η τεχνική οδηγεί σε εναλλακτικούς αλγορίθμους όπως ο $\Delta - stepping$ [7] ο οποίος επιτρέπει την παράλληλη εξαγωγή κόμβων από την ουρά προτεραιότητας.

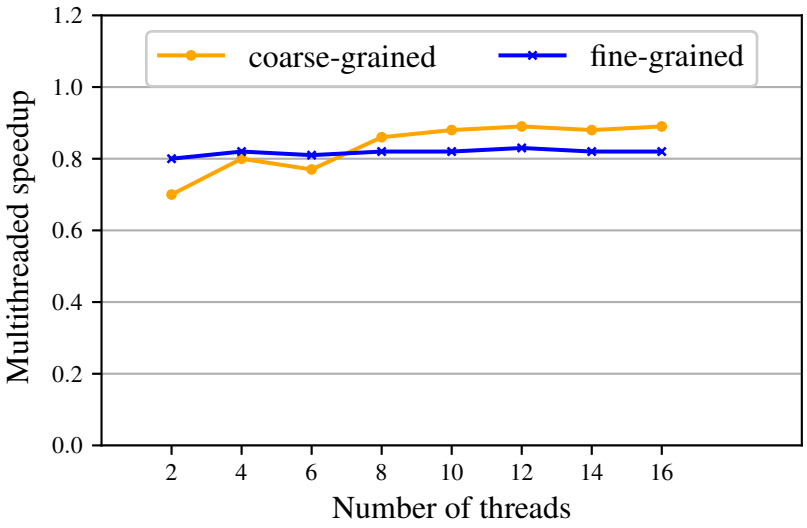
2.4.2 Παραλληλοποίηση εσωτερικού βρόγχου

Μια διαισθητική επιλογή για την παραλληλοποίηση του αλγορίθμου του Διθκστρα είναι να εκμεταλλευτούμε τον παραλληλισμό στον εσωτερικό βρόγχο χαλαρώνοντας όλες τις εξερχόμενες ακμές της κορυφής u, η οποία εξήχθη από τον σώρο, παράλληλα. Μπορούν να γίνουν ορισμένες παρατηρήσεις σχετικά με αυτό το σχήμα παραλληλισμού. Πρώτον, η επιτάχυνση περιορίζεται από τον μέσο βαθμό των κορυφών, δηλ. την πυκνότητα του γραφήματος. Είναι σαφές ότι αν οι κορυφές έχουν έναν μικρό αριθμό γειτόνων κατά μέσο όρο, τότε το παράλληλο τμήμα του αλγορίθμου (γραμμές 10-16) θα καταναλώνει ένα μικρό κλάσμα του συνολικού χρόνου εκτέλεσης, κάνοντας το διαδοχικό τμήμα (γραμμή 8) κυρίαρχο. Η δεύτερη παρατήρηση αφορά τις ταυτόχρονες προσβάσεις στην ουρά προτεραιότητας από τις παράλληλες λειτουργίες Decrease-Key.

Η πρώτη και μάλλον αφελής προσέγγιση που επιτρέπει την παραλληλισμό της φάσης χαλάρωσης είναι η χρήση ενός καθολικού κλειδώματος για να κλειδώνουμε ολόκληρο το σωρό κατά τη διάρκεια κάθε λειτουργίας Decrease-Key. Αυτό συνιστά ένα συντηρητικό, ανδρομερές σχήμα συγχρονισμού που επιτρέπει μόνο μία λειτουργία Decrease-Key κάθε φορά και προφανώς περιορίζει την ταχύτητα. Η εναλλακτική, πιο αισιόδοξη προσέγγιση είναι να επιτρέψουμε σε πολλαπλές ακολουθίες πρόσβασης στον σωρό να εκτελούνται παράλληλα εφ΄όσον έχουν πρόσβαση σε διαφορετικά μέρη του σωρού. Πιο συγκεκριμένα, αντί να χρησιμοποιούμε ένα κλείδωμα για ολόκληρο τον σωρό, μπορεί κανείς να χρησιμοποιήσει ξεχωριστά κλειδώματα για κάθε ζεύγος γονέα-παιδιού. Κάθε φορά που ένα νήμα εκτελεί μια λειτουργία Decrease-Key και απαιτείται ανταλλαγή κόμβου, πρέπει πρώτα να αποκτήσει το κατάλληλο κλείδωμα που προστατεύει αυτό το

συγκεκριμένο ζεύγος κόμβων. Με αυτό τον τρόπο διασφαλίζεται η ατομικότητα και ο αλγόριθμος μπορεί να εκτελεστεί παράλληλα με ασφάλεια.

Στο άρθρο [8] οι Αναστόπουλος et al. υλοποίησαν τα προαναφερθέντα σχήματα και έλαβαν μια εικόνα της απόδοσής τους, καθώς αξιολόγησαν γραφήματα με 10K κορυφές και 100K ακμές. Η απόδοση του ανδρομερούς κλειδώματος δεν είναι αποδοτική, διότι είναι ένα συντηρητικό σχήμα και δεν μπορεί να εκθέσει αρκετό παραλληλισμό. Επιπλέον, το fine-grained κλείδωμα επίσης αποτυγχάνει να ξεπεράσει την σειριακή λόγω των ταυτόχρονων προσβάσεων στον σωρό. Τα αποτελέσματα της αξιολόγησής τους παρουσιάζονται στο Σχήμα 2.6 Σε κάθε περίπτωση που θέλουμε να διατηρήσουμε τη σημασιολογία του αλγορίθμου, τα νήματα είναι σειριοποιημένα, περιορίζοντας έτσι τον συνολικό διαθέσιμο παραλληλισμό.



Σχήμα 2.6: Επιτάχυνση παράλληλων εκδόσεων με χρήση κλειδώματος.

Οι Nikas et al. [9] χρησιμοποίησαν Hardware transactional memory και βοηθητικά νήματα για να εξαγάγουν παραλληλισμό. Αυτά τα βοηθητικά νήματα εκφορτώνουν τις πράξεις από το κύριο νήμα εκμεταλλευόμενοι την ακόλουθη ιδιότητα: οι χαλαρώσεις οδηγούν σε μονοτονικά μειούμενες τιμές για τις αποστάσεις των κόμβων μέχρι κάθε απόσταση να φθάσει στην τελική ελάχιστη τιμή. Η ιδέα είναι ότι τα παράλληλα νήματα μπορούν να χρησιμεύσουν ως βοηθητικά νήματα και να χαλαρώνουν τους γείτονες των κόμβων που ανήκουν στο σύνολο που βρίσκεται στην ουρά. Το φορτίο που αντιστοιχεί σε μερικές από αυτές τις χαλαρώσεις θα αφαιρεθεί από το κύριο νήμα και θα επιταχύνει την εκτέλεση.

Κεφάλαιο 3

Profiling

3.1 Profiling στον αλγόριθμο του Dijkstra

Πραγματοποιήσαμε profiling στις λειτουργίες του αλγορίθμου του Dijkstra με έναν fine-grained τρόπο για να ανακαλύψουμε τη συμφόρηση του αλγορίθμου. Χρησιμοποιήσαμε την εντολή μέτρησης RDTSCP, η οποία διαβάζει έναν χρονικό καταχωρητή, για να μειώσουμε το overhead μέτρησης και τα σφάλματα στην αξιολόγηση.

3.1.1 Μεθοδολογία

RDTSC

Οι επεξεργαστές Intel και AMD διαθέτουν έναν καταχωρητή χρόνου για την μέτρηση των κύκλων της κεντρικής μονάδας επεξεργασίας. Ξεκινώντας με τον επεξεργαστή Intel Pentium, οι συσκευές έχουν συμπεριλάβει έναν καταχωρητή χρονισμού σημάτων ανά πυρήνα ο οποίος αποθηκεύει την τιμή του μετρητή χρόνου σφραγίδας και ο οποίος μπορεί να διαβαστεί με τις εντολές assembly RDTSC και RDTSCP.

```
// measure cycles of function foo
asm volatile ( "RDTSC\n\t"
      "mov %%edx, %0\n\t"
      "mov %%eax, %1\n\t": "=r" (cycles_high), "=r" (cycles_low));

foo(&variable);

asm volatile ( "RDTSC\n\t"
      "mov %%edx, %0\n\t"
      "mov %%eax, %1\n\t": "=r" (cycles_high1), "=r" (cycles_low1));

start = ( ((uint64_t)cycles_high << 32) | cycles_low );
end = ( ((uint64_t)cycles_high1 << 32) | cycles_low1 );
```

Listing 3.1: Counting cycles with rdtsc

Στην λίστα 3.1, παρουσιάζουμε τις πρόσθετες εντολές assembly που χρησιμοποιούμε για να μετρήσουμε τον αριθμό των κύκλων ρολογιού που χρειάζεται για να καλέσουμε

μια συνάρτηση. Η εντολή RDTSC φορτώνει τα υψηλότερα 32 bits του καταχωρητή χρονικής σήμανσης στον EDX και τα 32 bits χαμηλής τάξης στον EAX. Εκτελείται επίσης μία πράξη λογικού OR για να ανακατασκευαστεί και να αποθηκευτεί η τιμή του καταχωρητή σε μια τοπική μεταβλητή. Καλούμε την εντολή assembly RDTSC για να διαβάσουμε τον καταχωρητή χρονισμού πριν καλέσουμε τη συνάρτηση foo. Στη συνέχεια, καλούμε τη συνάρτηση και διαβάζουμε ξανά τον καταχωρητή χρονισμού (RDTSC) για να δούμε πόσοι κύκλοι ρολογιού έχουν περάσει από την πρώτη ανάγνωση. Οι δύο μεταβλητές, *Start* και *End* αποθηκεύουν τις τιμές του timestamp register στις αντίστοιχες χρονικές στιγμές εκτέλεσης των εντολών RDTSC.

Λογικά ο παραπάνω κώδικας έχει νόημα, αλλά αν προσπαθήσουμε να τον εκτελέσουμε, θα μπορούσαμε να πάρουμε σφάλματα κατάτμησης ή κάποια περίεργα αποτελέσματα. Αυτό συμβαίνει επειδή δεν εξετάσαμε ορισμένα θέματα που σχετίζονται με την εντολή RDTSC.

- Register Overwriting
- Out-Of-Order Execution

RDTSCP

Η εντολή RDTSCP είναι μια εντολή assembly η οποία διαβάζει ταυτόχρονα τον timestamp register και το αναγνωριστικό της CPU. Η τιμή του timestamp register αποθηκεύεται στους καταχωρητές EDX και EAX.Η τιμή του id της CPU αποθηκεύεται στον καταχωρητή ECX. Αυτό που είναι ενδιαφέρον σε αυτή την περίπτωση είναι η ¨ψευδο' σειριακή ιδιότητα της RDTSCP. Η εντολή RDTSCP περιμένει μέχρι να εκτελεστούν όλες οι προηγούμενες εντολές πριν από την ανάγνωση του μετρητή. Ωστόσο, οι επόμενες εντολές ενδέχεται να αρχίσουν να εκτελούνται πριν εκτελεστεί η διαδικασία ανάγνωσης. Αυτό σημαίνει ότι αυτή η εντολή εγγυάται ότι όλα όσα βρίσκονται πάνω από την κλήση στον πηγαίο κώδικα εκτελούνται πριν την κλήση της ίδιας της εντολής. Δεν μπορεί, ωστόσο, να εγγυηθεί ότι η CPU δεν θα εκτελέσει, πριν από την κλήση RDTSCP, εντολές οι οποίες, στον πηγαίο κώδικα, τοποθετούνται μετά από την κλήση της λειτουργίας RDTSCP. Εάν συμβεί κάτι τέτοιο, θα εμφανιστεί μια μόλυνση στις μετρήσεις που προκαλείται από εντολές που έρχονται μετά το RDTSCP.

Βελτιωμένο benchmarking

```
//start_timer()
asm volatile ("CPUID\n\t"
      "RDTSC\n\t"
      "mov %%edx, %0\n\t"
      "mov %%eax, %1\n\t": "=r" (cycles_high), "=r" (cycles_low)::
      "%rax", "%rbx", "%rcx", "%rdx");

function_to_measure();
//stop_timer()
asm volatile("RDTSCP\n\t"
      "mov %%edx, %0\n\t"
```

```
"mov %%eax, %1\n\t"
"CPUID\n\t": "=r" (cycles_high1), "=r" (cycles_low1)::
"%rax", "%rbx", "%rcx", "%rdx");
```

Listing 3.2: Benchmarking-RDTSCP & CPUID

Στον παραπάνω κώδικα, η πρώτη κλήση CPUID εφαρμόζει ένα εμπόδιο για να αποφευχθεί η εκτέλεση των εντολών πάνω και κάτω από την εντολή RDTSC. Παρόλα αυτά, αυτή η κλήση δεν επηρεάζει τη μέτρηση, δεδομένου ότι εκτελείται πριν από την RDTSC (δηλ. Πριν την ανάγνωση του καταχωρητή χρονισμού). Το πρώτο RDTSC διαβάζει τότε τον καταχωρητή χρονισμού και η τιμή αποθηκεύεται στη μνήμη. Στη συνέχεια, εκτελείται ο κώδικας που θέλουμε να μετρήσουμε. Εάν ο κώδικας είναι μια κλήση σε μια συνάρτηση, συνιστάται να δηλωθεί ως inline ώστε να μην υπάρχει επιβάρυνση στην κλήση της συνάρτησης.

Η εντολή RDTSCP διαβάζει για δεύτερη φορά τον timestamp register και εγγυάται ότι ολοκληρώνεται η εκτέλεση του συνόλου του κώδικα που θέλουμε να μετρήσουμε. Οι δύο εντολές 'mov' που εκτελούνται αποθηκεύουν τις τιμές των edx και eax στη μνήμη. Και οι δύο εντολές εγγυούνται πως θα εκτελεστούν μετά το RDTSC (δηλ. Δεν επηρεάζουν την μέτρηση), καθώς υπάρχει λογική εξάρτηση μεταξύ του RDTSCP και των καταχωρητών edx και eax.

Τέλος, μια κλήση CPUID εγγυάται ότι ένα εμπόδιο εφαρμόζεται ξανά έτσι ώστε να είναι αδύνατο να εκτελεστεί οποιαδήποτε εντολή μετά από το CPUID. Με αυτή τη μέθοδο αποφεύγουμε να καλέσουμε μια εντολή CPUID μεταξύ των καταχωρητών χρονισμού.

Profiling των λειτουργιών

Χρησιμοποιήσαμε τις μεθόδους συγκριτικής αξιολόγησης που παρουσιάστηκαν παραπάνω για να καταγράψουμε τη συμπεριφορά των διαφορετικων λειτουργιών του αλγορίθμου Dijkstra.

Algorithm 3 : Profiling Dijkstra's algorithm

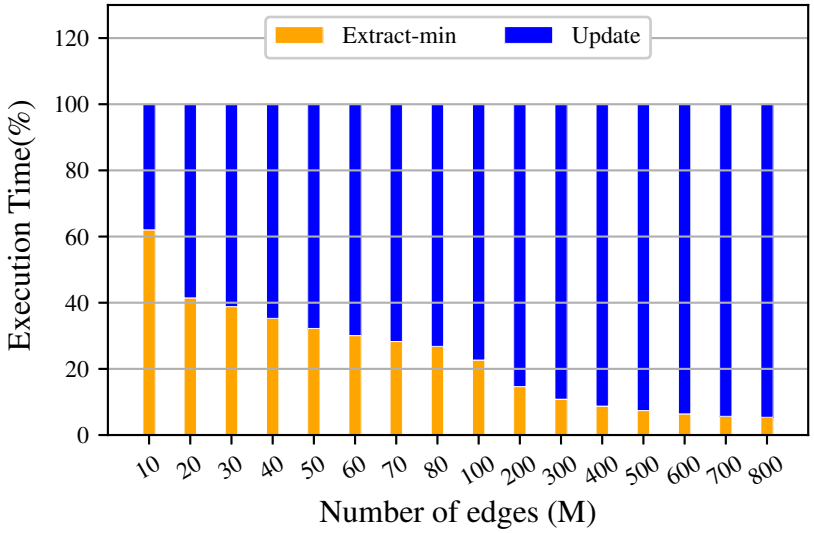
Require: Graph $G(V, E)$, Source vertex S
Ensure: Predecessors array *previous*
Shortest distance array d

```
1: for all v ∈ V[G] do
2:    d[v] ← +∞
3:    previous[v] ← undefined
4: d[s] ← 0 // Initialization
5: S ← empty set
6: Q ← s
7: while Q is not empty do
8:    start_timer()
9:    u ← Extract-Min(Q) // Extract-Min
10:   stop_timer()
11:   S ← S ∪ {u}
12:   for all edge (u, v) outgoing from u do
13:      if v ∉ S then
14:         sum ← d[u] + w(u, v)
15:         if v ∉ Q then
16:            start_timer()
17:            Insert(Q,v,sum)
18:            stop_timer()
19:         else
20:            start_timer()
21:            if sum < d[v] then // Update
22:               start_timer()
23:               u ← DecreaseKey(Q,v,sum)
24:               stop_timer()
25:               d[v] ← d[u] + w(u, v)
26:               previous[v] ← u
27:               stop_timer()
```

3.1.2 Αποτελέσματα Profiling

Στο Σχήμα 3.1 παρουσιάζουμε τη διανομή των χρόνων εκτέλεσης, σε γραφήματα με 10M κόμβους και αυξανόμενο αριθμό ακμών. Για αυτό το πείραμα χρησιμοποιήσαμε έναν απλό δυαδικό σωρό ως ουρά προτεραιότητας. Η λειτουργία Update περιλαμβάνει τις γραμμές 18,17,19 του Αλγορίθμου 2. Είναι φανερό ότι η λειτουργία Update γίνεται το κύριο μέρος της εκτέλεσης καθώς τα γραφήματα γίνονται πυκνότερα και ο χρόνος εκτέλεσης της φάσης αρχικοποίησης είναι ελάχιστος σε σύγκριση με την Extract-Min.

Και οι δύο αυτές λειτουργίες εξαρτώνται από την ουρά προτεραιότητας που χρησιμοποιείται στον αλγόριθμο. Ένας τρόπος για να αυξήσουμε την απόδοση είναι να χρησιμοποιήσουμε ουρές προτεραιότητας που παρέχουν πολυπλοκότητες χαμηλού κόστους και λειτουργούν με μεγάλα throughputs σε μεγάλα σύνολα δεδομένων.



Σχήμα 3.1: Κατανομή του ποσοστού χρόνου εκτέλεσης. Η λειτουργία Update καταλαμβάνει το μεγαλύτερο κομμάτι του χρόνου σε πυκνά γράμματα.

3.2 Ουρές προτεραιότητας

Στην επιστήμη των υπολογιστών, μια ουρά προτεραιότητας είναι ένας αφηρημένος τύπος δεδομένων ο οποίος είναι σαν μια κανονική δομή δεδομένων ουρών ή στοίβας, αλλά όπου επιπλέον κάθε στοιχείο έχει μια ¨προτεραιότητα' που συνδέεται με αυτό. Σε ουρά προτεραιότητας, ένα στοιχείο με υψηλή / χαμηλή προτεραιότητα εξυπηρετείται πριν από ένα στοιχείο με χαμηλή / υψηλή προτεραιότητα. Εάν δύο στοιχεία έχουν την ίδια προτεραιότητα, εξυπηρετούνται σύμφωνα με τη σειρά τους στην ουρά.

Πολλές εφαρμογές απαιτούν να επεξεργαζόμαστε αντικείμενα με ταξινομημένα κλειδιά, αλλά όχι απαραίτητα σε πλήρη ταξινομημένη σειρά και όχι απαραίτητα με ταυτοχρονισμό. Ο αλγόριθμος του Dijkstra είναι μια εφαρμογή στην οποία χρείαζομαστε μια ουρά προτεραιότητας για την εκτέλεση των λειτουργιών της.

Μια ουρά προτεραιότητας πρέπει τουλάχιστον να υποστηρίζει τις ακόλουθες λειτουργίες:

- Εισαγωγή: προσθέστε ένα στοιχείο στην ουρά με μια σχετική προτεραιότητα.
- Αφαίρεση μεγίστου/ελαχίστου: αφαιρέση του στοιχείου το οποίο έχει την υψηλότερη /χαμηλότερη προτεραιότητα.

Σε αυτή την ενότητα θα αναλύσουμε τις λειτουργίες τριών βασικών ουρών προτεραιότητας οι οποίες ανταποκρίνονται διαφορετικά στις λειτουργίες του αλγορίθμου Dijkstra και θα αξιολογήσουμε τη χρήση τους. Οι ουρές προτεραιότητας που θα εξεταστούν είναι ο d-ary σωρός, ο σωρός Fibonacci και η Skip list.

3.2.1 D-ary σωρός

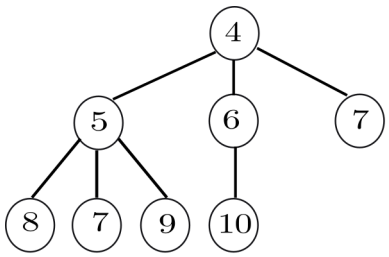
Ο D-ary σωρός είναι ένα πλήρες D-ary δέντρο γεμάτο από αριστερά προς τα δεξιά στο οποίο κάθε γονικός κόμβος έχει υψηλότερη (ή ίση τιμή) από ό,τι όλα οι κόμβοι-παιδιά [10]. Οι σωροί που σέβονται αυτήν την διάταξη ονομάζονται σωροί μεγίστου, επειδή ο

κόμβος με τη μέγιστη τιμή είναι στην κορυφή του δέντρου. Ανάλογα, ο min-heap είναι ένας σωρός, στον οποίο κάθε γονικός κόμβος έχει χαμηλότερη (ή ίση) τιμή από όλους τους απογόνους του. Χάρη σε αυτές τις ιδιότητες, ο d-ary σωρός συμπεριφέρεται ως ουρά προτεραιότητας. Μία ειδική περίπτωση του δ-αρψ σωρού ($d = 2$) είναι ο δυαδικός σωρός.

Ο D-ary σωρός όταν χρησιμοποιείται στην υλοποίηση της ουράς προτεραιότητας επιτρέπει ταχύτερη λειτουργία μείωσης κλειδιού σε σύγκριση με τον δυαδικό σωρό (i)$\mathcal{O}(log_2 n)$ για δυαδικό σωρό vs (ii) $\mathcal{O}(log_k n)$ φορ d-ary σωρός. Παρόλα αυτά, προκαλεί την αύξηση της πολυπλοκότητας της λειτουργίας Extract-Min σε $\mathcal{O}(k \cdot log_k n)$ σε σύγκριση με την πολυπλοκότητα $\mathcal{O}(log_2 n)$ όταν χρησιμοποιείτε δυαδικός σωρός σαν ουρά προτεραιότητας. Αυτό επιτρέπει στον D-ary σωρό να είναι πιο αποδοτικός σε αλγόριθμους όπου οι λειτουργίες μείωσης προτεραιότητας είναι πιο συχνές από τη λειτουργία Extract-Min. Ο αλγόριθμος Dijkstra είναι μια τέτοια περίπτωση. Όταν επεξεργαζόμαστε ένα γράφημα με m άκμές και n κορυφές, ο αλγόριθμος του Dijkstra χρησιμοποιεί ένα μιν-ηεαπ στο οποίο πραγματοποιούνται n λειτουργίες Extract-Min και m λειτουργίες Decrease-Key . Χρησιμοποιώντας d-ary σωρό με $d = m/n$, οι συνολικοί χρόνοι για αυτούς τους δύο τύπους πράξεων μπορεί να ισορροπηθούν μεταξύ τους, οδηγώντας σε ένα συνολικό χρόνο $\mathcal{O}(mlog_{m/n})$.

Ο D-ary σωρός είναι πιο φιλικός για την κρυφή μνήμη σε σχέση με ένα δυαδικό σωρό, και γι΄ αυτό το λόγο ο d-ary σωρός είναι πιο γρήγορος στην πράξη, αν και η λειτουργία ExtractMin παρουσιάζει μεγαλύτερο χειρότερο χρόνο εκτέλεσης ($\mathcal{O}(k \cdot log_k n)$).

Υλοποίηση Ο d-ary σωρός υλοποιείται συνήθως χρησιμοποιώντας έναν πίνακα. Για κάθε κόμβο του σωρού που τοποθετείται στη θέση n, ο γονέας του τοποθετείται στη θέση $(n-1)/d$ και οι απόγονοί του τοποθετούνται στις θέσεις $d+1$, ..., $d \cdot k + d$.



Σχήμα 3.2: 3-ary σωρός και η αναπαράσταση του με χρήση πίνακα

Λειτουργίες

- **Extract-Min**

  Η διαδικασία για τη διαγραφή της ρίζας από τον σωρό και την αποκατάσταση των ιδιοτήτων ονομάζεται heapify-down.
  Πολυπλοκότητα: $\mathcal{O}(d \cdot log_d n)$

- **Decrease-Key**

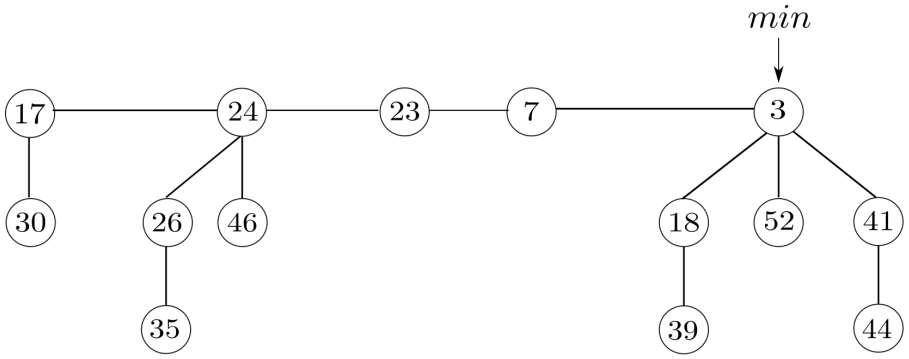
  Η μείωση του κλειδιού ενός κόμβου πραγματοποιείται μειώνοντας την τιμή του κλειδιού και στη συνέχεια πραγματοποιώντας μια λειτουργία heapify-up.
  Πολυπλοκότητα: $\mathcal{O}(log_d n)$

- **Insert**

  Για να προσθέσουμε ένα στοιχείο σε ένα σωρό πρέπει να εκτελέσουμε μια λειτουργία heapify-up.
  Πολυπλοκότητα: $\mathcal{O}(log_d n)$

3.2.2 Σωρός Fibonacci

Ένας σωρός Fibonacci είναι μια συλλογή από δέντρα που ικανοποιούν την ιδιότητα ελάχιστου σωρού [1]. Το κλειδί ενός παιδιού είναι πάντα μεγαλύτερο ή ίσο με το κλειδί του γονέα. Αυτό σημαίνει ότι το ελάχιστο κλειδί είναι πάντα στη ρίζα ενός εκ των δέντρων. Τα δέντρα δεν έχουν καθορισμένο σχήμα και στην ακραία περίπτωση ο σωρός μπορεί να έχει κάθε στοιχείο σε ξεχωριστό δέντρο. Αυτή η ευελιξία επιτρέπει ορισμένες λειτουργίες να εκτελούνται με έναν lazy τρόπο, αναβάλλοντας τις εργασίες για μεταγενέστερες λειτουργίες.



Σχήμα 3.3: Παράδειγμα σωρού Fibonacci

Λειτουργίες

- **Extract-Min**

  Η λειτουργία Extract-Min πραγματοποιείται σε τρεις φάσεις. Πρώτον, επιλέγουμε τη ρίζα που περιέχει το ελάχιστο στοιχείο και την αφαιρούμε. Τα παιδιά της θα γίνουν ρίζες νέων δέντρων. Αν ο αριθμός των παιδιών ήταν d, χρειάζεται χρόνος $\mathcal{O}(d)$ για την επεξεργασία όλων των νέων ριζών και τις πιθανές αυξήσεις κατά $d-1$. Επομένως, ο αποσβεσμενήμενος χρόνος εκτέλεσης αυτής της λειτουργίασ είναι $mathcal O(d) = \mathcal{O}(logn)$.
  Πολυπλοκότητα: $\mathcal{O}(logn)$

- **Decrease-Key**

  Η λειτουργία Δεςρεασε-Κεψ θα μειώσει το κλειδί του κόμβου και σε περίπτωση που η ιδιότητα ελαχίστου παραβιαστεί, ο κόμβος θα αποκοπεί από τον κόμβο-γονέα.
  Πολυπλοκότητα: $\mathcal{O}(k)$

- **Συγχώνευση**

  Η συγχώνευση υλοποιείται απλώς συνδυάζοντας τις λίστες των ριζών δέντρων των δύο σωρών. Αυτό μπορεί να γίνει σε συνεχή χρόνο και το δυναμικό δεν αλλάζει, οδηγώντας πάλι σε σταθερό χρόνο απόσβεσης.
  Πολυπλοκότητα: $\mathcal{O}(1)$

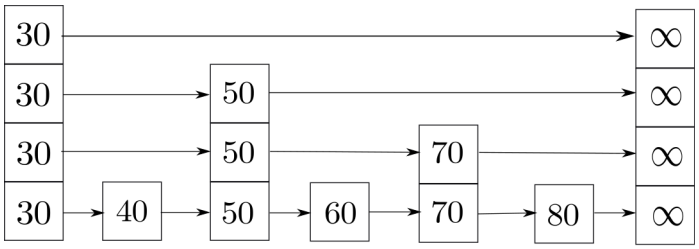
- **Insert**

  Η λειτουργία Insert πραγματοποιείται δημιουργώντας ένα νέο σωρό με ένα στοιχείο και συγχωνεύοντας τον με τα υπόλοιπα δέντρα. Αυτό απαιτεί σταθερό χρόνο και το δυναμικό αυξάνεται κατά ένα, επειδή ο αριθμός των δέντρων αυξάνεται. Συνεπώς, το αποσβεμήμενο κόστος παραμένει σταθερό.
  Πολυπλοκότητα: $\mathcal{O}(1)$

3.2.3 Skip list

Η Skip list είναι μία δομή δεδομένων που επιτρέπει γρήγορες αναζητήσεις σε μία ταξινομημένη ακολουθία στοιχείων (key-value pairs). Η γρήγορη αναζήτηση επιτυγχάνεται διατηρώντας μία συνδεδεμένη ιεραρχία υπακολουθιών, κάθε μία προσπερνάει κάποια στοιχεία. Η Skip list έχει επίπεδα. Το κατώτερο επίπεδο είναι μία συνηθισμένη ταξινομημένη λίστα. Ένα κλειδί στο επίπεδο i εμφανίζεται στο επίπεδο $i+1$ με κάποια σταθερή πιθανότητα p. Έτσι, κάθε στοιχείο της δομής έχει ένα τυχαίο ύψος που αντιπροσωπεύει τα επίπεδα στα οποία εμφανίζεται το κλειδί του στοιχείου. Η *skiplist* έχει ένα μέγιστο ύψος και κάθε φορά που ένα στοιχείο εισάγεται, παίρνει ένα τυχαίο ύψος ανάμεσα σε 1 και στο μέγιστο ύψος της *skiplist*. Το σχήμα 3.4 απεικονίζει ένα παράδειγμα μίας Skip list.

Μία αναζήτηση για ένα στόχο-στοιχείο αρχίζει από την κορυφή της λίστας από το υψηλότερο επίπεδο και προχωράει οριζόντια μέχρι να βρεθεί ένα στοιχείο μεγαλύτερο ή ίσο από το κλειδί του στόχου-στοιχείου. Αν το κλειδί του στοιχείου είναι ίσο με αυτό του στόχου, τότε η λειτουργία επιστρέφει επιτυχώς. Διαφορετικά, επαναλαμβάνεται η ίδια διαδικασία για το επόμενο πιο χαμηλό επίπεδο της Skip list. Ο συνόλικός χρόνος εκτέλεσης της λειτουργίας αναζήτησης είναι ανάλογος με $\mathcal{O}(logn)$.



Σχήμα 3.4: Παράδειγμα Skiplist

Λειτουργίες

- **Insert**

  Τηε ινσερτιον αλγοριτημ φορ σκιπ λιστς υσες ρανδομιζατιον το δεςιδε τηε ηειγητ οφ τηε τοωερ φορ τηε νεω εντρψ.

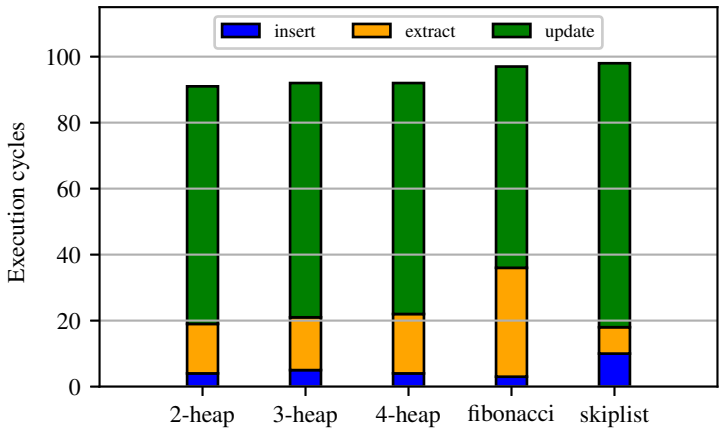
  Ο αλγόριθμος εισαγωγής χρησιμοποιεί τυχαιότητα για να αποφασίσει το ύψος του δέντρου του καινούριου στοιχείου.
  Πολυπλοκότητα: $\mathcal{O}(logn)$

- **Decrease-Key**

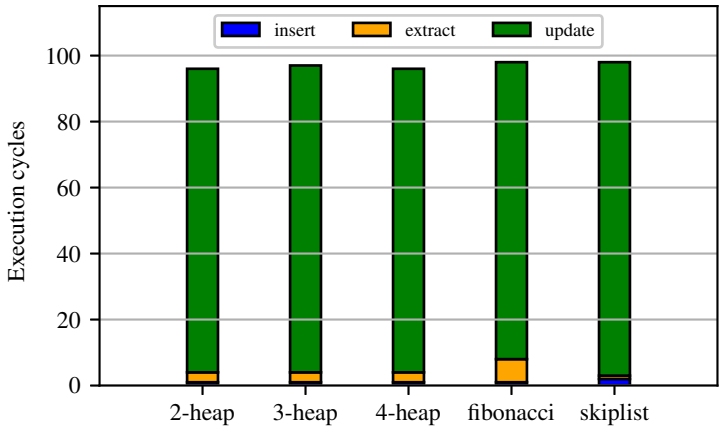
  Για να πραγματοποιήσουμε ένα Decrease-Key διαγράφουμε το στοιχείο και το εισάγουμε εκ νέου με καινούρια προτεραιότητα.
  Πολυπλοκότητα: $\mathcal{O}(logn)$

- **Extract-min**

  Η εξαγωγή ελαχίστου πραγματοποιείται διαγράφοντας τον πρώτο πύργο, ο οποίος διατηρεί το ελάχιστο στοιχείο.
  Πολυπλοκότητα: $\mathcal{O}(1)$



(α′) Γράφος 10Mx50M



(β′) Γράφος 10Mx500M

Σχήμα 3.5: Κατανομή του χρόνου εκτέλεσης κάθε λειτουργίας του αλγορίθμου του Dijkstra για κάθε ουρά προτεραιότητας.

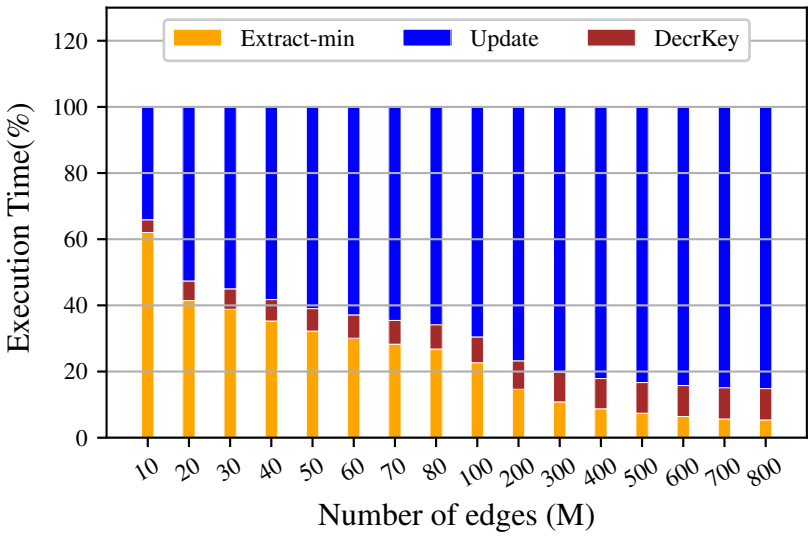
Στο σχήμα 3.5 παρουσιάζουμε τη διανομή των χρόνων εκτέλεσης για κάθε λειτουργία και κάθε ουρά προτεραιότητας, σε 2 γράφους με 10M κόμβους και 50M & 500M ακμές. Όσον αφορά τους 2,3,4 - ary σωρούς, παρατηρούμε παρόμοια συμπεριφορά σε κάθε γράφημα. Στην περίπτωση του σωρού Fibonacci, η λειτουργία Extract-Min είναι πιο δαπανηρή σε σχέση με τις υπόλοιπες ουρές προτεραιότητας, γεγονός όμως το οποίο αναμέναμε. Όσον αφορά τη Skip list, παρότι το κόστος της λειτουργίας Extract-Min

είναι μικρότερο σε σχέση με τις υπόλοιπες, οι λειτουργίες Insert και Update (που αποτελείται από ένα Decrease-Key και μια χαλάρωση) επιβάλλουν αξιοσημείωτο overhead.

Όσον αφορά τον πυκνό γράφο που αποτελείται από 500M ακμές, διαπιστώνουμε ότι η λειτουργία Update διατηρεί το μεγαλύτερο μέρος του χρόνου εκτέλεσης και ότι ο αντίκτυπος κάθε ουράς προτεραιότητας είναι αρκετά μικρό. Αυτό συμβαίνει ακόμα και στην περίπτωση σωρού του Fibonacci, όπου η πολυπλοκότητα του Decrease-Key είναι σταθερή. Λόγω αυτού του συμπεράσματος, θα συμπεριλάβουμε τη λειτουργία Decrease-Key στο profiling και θα προσπαθήσουμε να ανακαλύψουμε το πραγματικό bottleneck του αλγορίθμου το οποίο είναι κρυμμένο στην λειτουργία Update.

3.3 Εκτεταμένο Profiling

Στο σχήμα 3.6 παρουσιάζουμε τη διανομή των χρόνων εκτέλεσης, σε γραφήματα με 10M κόμβους και αυξανόμενο αριθμό ακμών. Η λειτουργία Update περιλαμβάνει τις γραμμές 18,19 του Alg. 2 εξαιρουμένης της γραμμής 17, δηλαδή της λειτουργίας Decrease-Key. Είναι προφανές ότι σε αυτή την περίπτωση, η λειτουργία Update χωρίς το Decrease-Key, καταλαμβάνει το μεγαλυτερο μέρος της εκτέλεσης καθώς τα γραφήματα γίνονται πυκνότερα και ο χρόνος εκτέλεσης της λειτουργίας Insert είναι ελάχιστος σε σχέση με τις υπόλοιπες.



Σχήμα 3.6: Κατανομή των χρόνων εκτέλεσης.

Όπως φαίνεται στο σχήμα 3.6, οι λειτουργίες Extract-Min και Decrease-Key επηρεάζουν το χρόνο εκτέλεσης λιγότερο από το Update, καθώς τα γραφήματα γίνονται πιο πυκνά. Η λειτουργία Update αποτελείται από τέσσερις λειτουργίες ανάγνωσης που προκύπτουν για κάθε ακμή της εξαγόμενης κορυφής.

Οι τέσσερις θέσεις μνήμης που πρέπει να διαβαστούν είναι: η απόσταση d αρραψ στις θέσεις v και u, το κόστος της ακμής $w(u, v)$ και ο πίνακας *previous* στη θέση v . Η ανάγνωση του *distance* και του *previous* πίνακα, του οποίου το μέγεθος είναι ίσο με τον αριθμό των κορυφών (10M), σε τυχαία θέση v, προκαλεί πρόσβαση σε τυχαία θέση μνήμης με αποτέλεσμα αστοχίες στην cache. Με άλλα λόγια, δεν υπάρχει temporal locality μεταξύ των δεδομένων που χρειάζονται για να χαλαρώσουν τις ακμές του u. Στα πυκνά γραφήματα, το φαινόμενο αυτό είναι πιο έντονο, καθώς ο αριθμός των ακμών ανά κορυφή είναι μεγάλος. Καθώς οι μη κανονικές προσβάσεις στη μνήμη

αυξάνονται και το σύνολο δεδομένων δεν χωράει στα χαμηλότερα επίπεδα της ιεραρχίας της κρυφής μνήμης, η απόδοση χειροτερεύει όλο και περισσότερο.

Κεφάλαιο 4

Prefetching

Σε αυτό το κεφάλαιο θα αναλύσουμε τον τρόπο με τον οποίο μπορεί να εφαρμοστεί prefetching στον αλγόριθμο του Dijkstra και θα παρουσιάσουμε δύο σχήματα που χρησιμοποιούν software prefetching για να αντιμετωπίσουν τις μη κανονικές προσβάσεις στη μνήμη.

4.1 Μηχανισμοί Prefetching

Το prefetching είναι ένας μηχανισμός για την μεταφορά δεδομένων σε υψηλότερα επίπεδα στην ιεραρχία μνήμης προς αναμονή για μελλοντική χρήση. Το πρεφετςηινγ μπορεί να γίνει με χρήση υλικού, λογισμικού ή σε συνδυασμό και των δύο. Το prefetching λογισμικού ελέγχεται άμεσα από το πρόγραμμα ή τον μεταγλωττιστή και ως εκ τούτου είναι ευθύνη τους να εκδίδουν τις κατάλληλες εντολές πρεφετςηινγ την κατάλληλη στιγμή. Το prefetching υλικού είναι η εναλλακτική περίπτωση, όπου ένας ελεγκτής υλικού δημιουργεί αιτήσεις πρεφετςηινγ από πληροφορίες που μπορεί να λάβει κατά τη διάρκεια εκτέλεσης (π.χ. διεύθυνση μνήμης και αστοχίες μνήμης cache). Γενικά, οι prefetchers λογισμικού χρησιμοποιούν πληροφορίες οι οποίες παρέχονται από τη μεταγλώττιση και το profiling, ενώ οι prefetchers υλικού χρησιμοποιούν πληροφορίες χρόνου εκτέλεσης. Και τα δύο έχουν τα πλεονεκτήματά τους και και τα δύο μπορούν να είναι πολύ αποτελεσματικά [11].

4.1.1 Prefetching λογισμικού

Το prefetching λογισμικού παρέχει διευκολύνσεις για τον προγραμματιστή / μεταγλωττιστή ώστε να μπορεί να εισαγάγει ρητά prefetch εντολές όποτε εκείνος επιθυμεί. Αυτό μπορεί να γίνει είτε συμπεριλαμβάνοντας μια εντολή φόρτωσης σε ένα σύνολο εντολών μικροεπεξεργαστών ή μέσω ορισμένων καταχωρητών που μπορούν να διαμορφωθούν και να προγραμματιστούν από το λογισμικό. Η τεχνική prefetching λογισμικού μπορεί να γίνει είτε απευθείας από τον προγραμματιστή (π.χ. στον κώδικα C) ή από τον μεταγλωττιστή στη φάση βελτιστοποίησης, καθώς και στον τελικό κωδικό συναρμολόγησης.

Εντολές Prefetching

Η επιλογή της τοποθεσία μιας εντολής prefetch σε σχέση με τις αντίστοιχες εντολές η είναι γνωστή ως prefetch scheduling. Παρόλο που το prefetching λογισμικού μπορεί

να χρησιμοποιήσει περισσότερες πληροφορίες από τον μεταφραστή για το scheduling από hardware τεχνικές, δεν είναι εφικτό μερικές φορές να πραγματοποιηθούν ακριβείς προβλέψεις. Επειδή ο χρόνος εκτέλεσης μεταξύ prefetching και των αντίστοιχων εντολών ενδέχεται να διαφέρουν το ίδιο συμβαίνει και με την καθυστέρηση ανταπόκριση της μνήμης. Εάν η εντολή φόρτωσης πραγματοποιηθεί πολύ νωρίς, η προσωρινή μνήμη μπορεί να αντικαταστήσει το μπλοκ για ένα νέο prefetched block. Τα έγκαιρα prefetches ενδέχεται επίσης να αντικαταστήσουν τα δεδομένα που εξακολουθεί να χρησιμοποιεί η CPU, και αυτό θα προκαλέσει σφάλμα που δεν θα συνέβαινε χωρίς το prefetching, το οποίο ονομάζεται cache pollution.

Εστιάζοντας στις εντολές prefetching, οι εντολές αυτές μπορούν να εισαχθούν με το χέρι αμέσως πριν από τις κλήσεις λειτουργίας, έτσι ώστε να επιτευχθεί ένα πλήρες prefetching και να βεβαιωθούμε ότι είναι διαθέσιμες στην cache πριν αρχίσουν να εκτελούνται. Αυτό μπορεί να είναι πολύ ωφέλιμο για κωδικούς που έχουν πάρα πολλές κλήσεις λειτουργίας ή κλήσεις προς εξωτερικές βιβλιοθήκες. Φυσικά, αυτό απαιτεί ένα εκ των προτέρων profiling της αίτησης, και αναγνώριση της διεύθυνσης και του μεγέθους των blocks καθώς και των διαφορετικών λειτουργιών. Από την άλλη πλευρά, χρησιμοποιώντας ρητά τις εντολές fetch ενδέχεται επίσης να επιφέρουν κάποια performance penalties επειδή το μεγέθους του κώδικα αυξάνεται με την προσθήκη των οδηγιών prefetch. Για αυτό το λόγο, είναι σημαντικό να βελτιστοποιηθεί η θέση και το μέγεθος των εντολών prefetch έτσι ώστε να βεβαιωθούμε ότι επιτυγχάνεται η βέλτιστη απόδοση [21].

Irregular access patterns

Το software prefetching αποτελεί μια δελεαστική πρόταση για τα παράτυπα πρότυπα πρόσβασης δεδομένων. Η ιδέα είναι ότι ο προγραμματιστής χρησιμοποιεί δομή δεδομένων και αλγοριθμικές γνώσεις για να εισαγάγει εντολές στο πρόγραμμα με σκοπό να φέρει νωρίς τα απαιτούμενα δεδομένα, βελτιώνοντας έτσι την απόδοση με επικαλυπτόμενες προσβάσεις στη μνήμη. Στο [12] ο prefetching ϛοδε δημιουργείται αυτόματα από τον compiler αντί από την εισαγωγή με μη αυτόματο τρόπο. Ανέπτυξαν έναν νέο αλγόριθμο για την αυτοματοποίηση της εισαγωγής software prefetching για έμμεσες πρόσβασεις στη μνήμη.

Συνοψίζοντας, το software prefetching δίνει στους προγραμματιστές τον έλεγχο και την ευελιξία, και επιτρέπει τη σύνθετη ανάλυση και τη μορφοποίηση των εφαρμογών. Επίσης, δεν απαιτούν σημαντικές τροποποιήσεις υλικού. Αλλά από την άλλη πλευρά, δεν είναι πολύ εύκολο να εκτελέσεις έγκαιρα prefetches. Γι άυτό απαιτείται εκτενές profiling εκ των προτέρων.

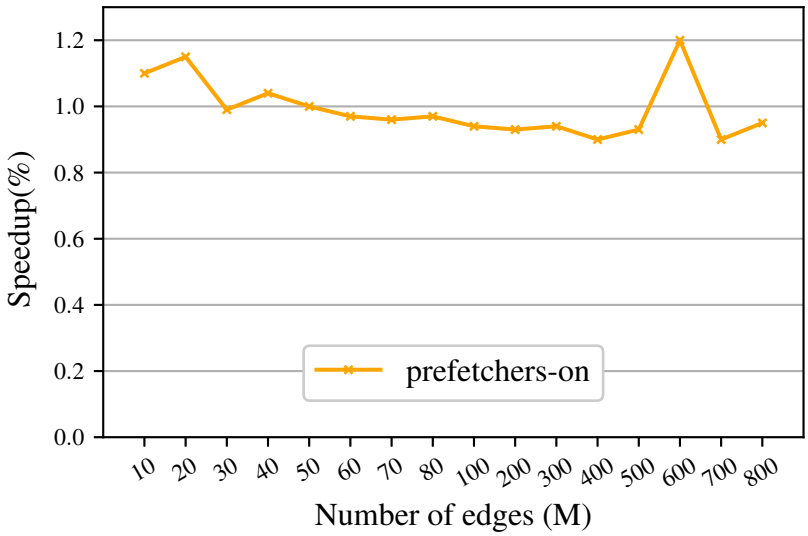
4.2 System Prefetchers

Σε αυτή τη διπλωματική εργασία χρησιμοποιήσαμε έναν Intel Broadwell-EP, του οποίου τα χαρακτηριστικά περιγράφονται στο Κεφάλαιο 5. Υπάρχουν 2 hardware prefetchers που σχετίζονται με την Λ1 δατα ϛαϛηε (επίσης γνωστοί ως DCU) και 2 prefetchers που σχετίζονται με την Λ2 ϛαϛηε. Υπάρχει ένας Model Specific Register (MSR) σε κάθε πυρήνα με διεύθυνση 0x1A4 που μπορεί να χρησιμοποιηθεί για τον έλεγχο αυτών των 4 prefetchers. Τα bits 0-3 σε αυτόν τον καταχωρητή μπορούν να χρησιμοποιηθούν για να ενεργοποιήσουν ή να απενεργοποιήσουν αυτούς τους prefetchers. Στον παρακάτω πίνακα, παρουσιάζουμε τους εν λόγω hardware prefetchers.

Prefetcher	Description
L2 hardware prefetcher	Fetches additional lines of code or data into the L2 cache
L2 adjacent cache line prefetcher	Fetches the cache line that comprises a cache line pair (128 bytes)
DCU prefetcher	Fetches the next cache line into L1-D cache
DCU IP prefetcher	Uses sequential load history (based on Instruction Pointer of previous loads) to determine whether to prefetch additional lines

4.2.1 Επίδραση στην απόδοση

Οι hardware prefetchers του Intel-Broadwell αποτυγχάνουν να φορτώσουν το σωστό κομμάτι δεδομένων σε μη κανονικές προσβάσεις στη μνήμη. Ο εν λόγω επεξεργαστής και πολλά από τα σύγχρονα συστήματα δεν παρέχουν prefetchers για να αντιμετωπίσουν τις μη κανονικές προσβάσεις στη μνήμη. Στην Εικόνα 4.1 παρουσιάζουμε την επιτευχθείσα επιτάχυνση στην περίπτωση που οι hardware prefetchers είναι ενεργοποιημένοι σε σύγκριση με την περίπτωση που είναι απενεργοποιημένοι. Λαμβάνοντας υπόψη γραφήματα στην περιοχή των 40M-500M, παρατηρούμε μια πτώση της απόδοσης, κατά περίπου 5% κατά μέσο όρο. Παρατηρούμε ότι οι prefetchers δεν οδηγούν σε αύξηση της απόδοσης σε αυτόν τον αλγόριθμο ο οποίος χαρακτηρίζεται από μη κανονικές προσβάσεις στη μνήμη. Σε αυτή τη διπλωματική εργασία, θα εφαρμόσουμε software prefetching για να επιταχύνουμε τέτοιες προσβάσεις στη μνήμη αξιοποιώντας τα χαρακτηριστικά του αλγορίθμου.



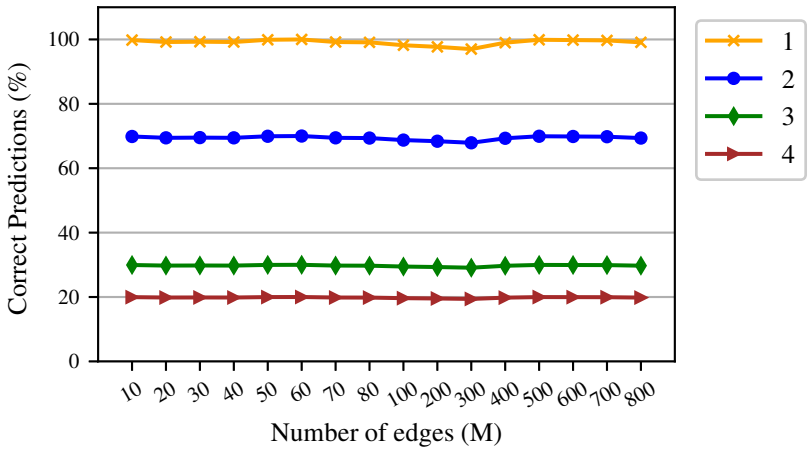
Σχήμα 4.1: Η επίδραση των hardware prefetchers στην απόδοση του αλγορίθμου.

4.3 Χρήση prefetching στον αλγόριθμο του Dijkstra

Ο στόχος μας είναι να αποκρύψουμε την καθυστέρηση εξυπτηρέτησης των αιτήσεων στην μνήμη που προέρχονται από την λειτουργία Υπδατε του αλγορίθμου του Dijkstra. Το prefetching του σωστού τμήματος δεδομένων στηρίζεται στην πρόβλεψη της επικείμενης κορυφής ελάχιστου κλειδιού u. Χρησιμοποιώντας τις πληροφορίες που είναι αποθηκευμένες στην ουρά προτεραιότητας, είμαστε σε θέση να κάνουμε μια αξιόπιστη πρόβλεψη για την επόμενη εξαγόμενη κορυφή.

4.3.1 Πρόβλεψη της επόμενης εξαγόμενης κορυφής

Η εικασία μας είναι ότι μερικά από τα κορυφαία ελάχιστα στοιχεία της ουράς προτεραιότητας μπορεί να είναι η επόμενη εξαγόμενη κορυφή. Το κομμάτι των δεδομένων το οποίο θέλουμε να φορτώσουμε στη μνήμη είναι οι γείτονες της ελάχιστης κορυφής της ουράς προτεραιότητας μετά τη λειτουργία Extract-Min της τρέχουσας φάσης του αλγορίθμου αλλά πριν από την λειτουργία Update. Η εικασία μας είναι ότι υπάρχει μεγάλη πιθανότητα οι n -τοπ κορυφές όπου $n \leq 10$ θα εξάχθούν στη συνέχεια (μετά την τρέχουσα Extract-Min λειτουργία). Στο σχήμα 4.5 παρουσιάζουμε το ποσοστό σωστών προβλέψεων για $n = 1, 2, 3, 4$. Θεωρούμε ότι μια πρόβλεψη είναι σωστή αν το $i - ost$ τοπ στοιχείο έχει εξαχθεί μετά από το πολύ i επαναλήψεις του αλγορίθμου. Παρατηρούμε ότι στην περίπτωση $n = 1$ οι προβλέψεις είναι σχεδόν τέλειες. Χρησιμοποιώντας το $n = 1$, η πιθανότητα να χρησιμοποιηθούν τα δεδομένα, τα οποία θα κάνουμε prefetch, είναι πολύ υψηλότερη.



Σχήμα 4.2: Ποσοστό σωστών προβλέψεων του επόμενου εξαγόμενου ελάχιστου στοιχείου-κόμβου.

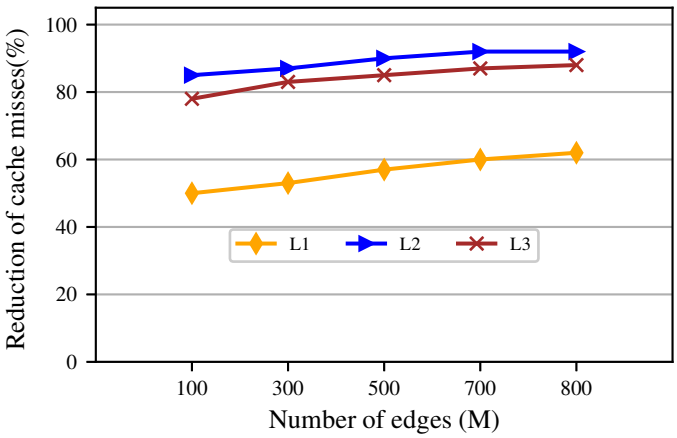
4.3.2 Ιδανικό prefetching

Όπως αναφέρεται στο Κεφάλαιο 2, η ρίζα του δυαδικού σωρού είναι η ελάχιστη κορυφή-κλειδί και θα είναι ο επόμενος εξαγώμενος κόμβος του αλγορίθμου. Μια πρώτη προσέγγιση θα ήταν να φορτώσουμε στη μνήμη τους γείτονες της εξαγόμενης κορυφής λίγο πριν πραγματοποιηθούν οι κατάλληλες χαλαρώσεις. Με αυτόν τον τρόπο, τα δεδομένα που θα απαιτηθούν από τη λειτουργία Update, θα έχουν μεταφερθεί εγκαίρως

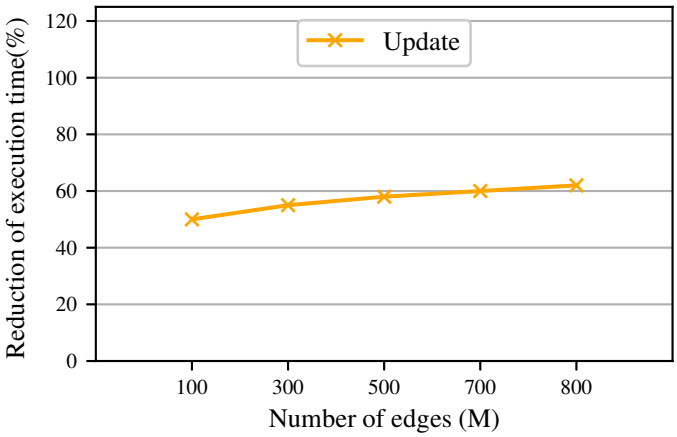
στην ιεραρχία της κρυφής μνήμης. Στο σχήμα 4.3, παρουσιάζουμε τους χρόνους εκτέλεσης των λειτουργιών Update και Extract-Min και επίσης τον συνολικό χρόνο εκτέλεσης. Στο σχήμα 4.4, δείχνουμε το ποσοστό μείωσης των αστοχιών μνήμης της λειτουργίας Υπδατε. Παρατηρούμε ότι η τεχνική πρεφετσηινγ μειώνει τις αστοχίες της L2 cache και της L3 cache κατά περισσότερο από 75 %. Αυτή η μείωση μας προιδεάζει ότι το πρεφετσηινγ μπορεί να βελτιώσει σημαντικά την απόδοση του αλγορίθμου μας. Στον Αλγόριθμο 4 παρουσιάζουμε τον κώδικα για αυτήν την τεχνική.

Algorithm 4 : Optimal prefetching

```
while Q is not empty do
    min ← Extract-Min(Q)
    for all neighbors of min do
        prefetch neighbors' data
    for all edge (u, v) outgoing from u do
        relaxation phase
```



Σχήμα 4.3: Χρόνος εκτέλεσης με χρήση ιδανικού prefetching.



Σχήμα 4.4: Αστοχίες cache με χρήση ιδανικού prefetching.

4.4 Prefetching-Helper-Thread

Αρχικά, υλοποιήσαμε ένα απλό σχήμα Prefetching-Helper-Thread (PHT) παρόμοιο με το σχήμα [13].

Μεθοδολογία

Το σχήμα που προτείνουμε αποτελείται από $n + 1$ νήματα. Το πρώτο εκτελεί συνεχώς τον αλγόριθμο (κύριο) και τα άλλα n λειτουργούν παράλληλα με το κύριο νήμα και κάνουν πρεφετση δεδομένα στην κοινόχρηστη προσωρινής μνήμη, βοηθώντας το κύριο νήμα να αποφύγει τυχόν απώλειες από επερχόμενες μη κανονικές προσβάσεις στη μνήμη. Μια σημαντική επιλογή για αυτό το σχήμα αφορά την επιλογή των γειτόνων που θα γίνουν prefetch από τα βοηθητικά νήματα.

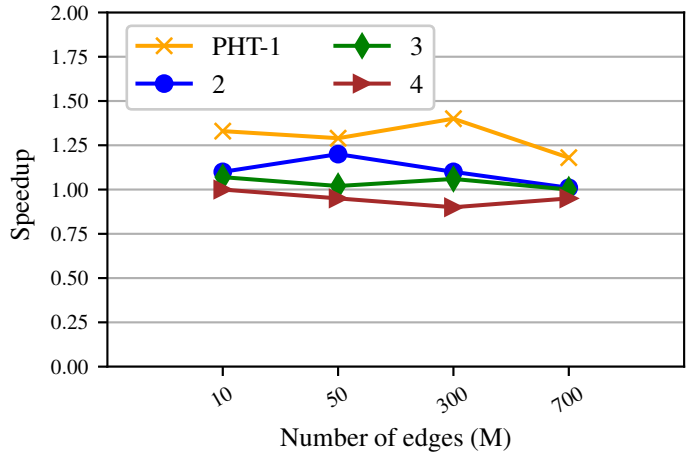
Πιο συγκεκριμένα, στην εφαρμογή μας το κύριο νήμα αρχικοποιεί όλες τις σημαίες που απαιτούνται για συγχρονισμό των δύο νημάτων και των δομών δεδομένων Q, d και *previous*. Κατά τη διάρκεια κάθε γύρου του αλγορίθμου το κύριο νήμα είναι υπεύθυνο για την εκτέλεση του αλγορίθμου ενώ τα βοηθητικά νήματα κάνουν prefetch τα δεδομένα που θα απαιτηθούν κατά τον επόμενο γύρο. Το κύριο νήμα εξάγει την ελάχιστη κορυφή, ενώ τα υπόλοιπα νήματα περιμένουν ειδοποίηση για να ξεκινήσουν να κάνουν prefetch. Μετά τη λειτουργία Extract-Min, το κύριο νήμα σηματοδοτεί τα βοηθητικά νήματα ώστε να διαβάσουν τις n τοπ ελάχιστες κορυφές από την ουρά προτεραιότητας και να αρχίσουν να κάνουν prefetch τους γείτονές τους. Στη συνέχεια, το κύριο νήμα ξεκινά τη λειτουργία ενημέρωσης, η ολοκλήρωση της οποίας διακόπτει τα υπόλοιπα νήματα. Αυτή η διαδικασία συνεχίζεται όσο η Q δεν είναι κενή. Επιλέγοντας $n = 1$, η πιθανότητα να χρησιμοποιηθούν τα δεδομένα τα οποία κάνουμε prefetch είναι αρκετά υψηλή.

4.4.1 Συντονισμός

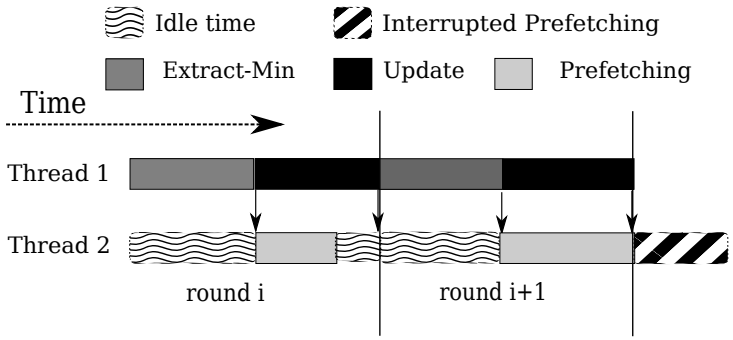
Ο συντονισμός των νημάτων επιτυγχάνεται χρησιμοποιώντας δύο σημαίες συγχρονισμού. Το ένα είναι απαραίτητο για να σηματοδοτήσει τα βοηθητικά νήματα ώστε να αρχίσουν να κάνουν prefetch και το δεύτερο για να τα διακόψουν. Στους αλγορίθμους 5 και 6 παρουσιάζουμε λεπτομερώς τον κώδικα για την περίπτωση $n = 1$.

Στο Σχήμα 4.2 παρουσιάζουμε την επιτάχυνση που επιτυγχάνεται από το PHT όταν γεννιούνται n βοηθητικά νήματα. Είναι προφανές ότι στην περίπτωση όπου $n = 1$ επιτυγχάνουμε τις υψηλότερες επιταχύνσεις. Επιπλέον, υπάρχουν περιπτώσεις, όπως $n = 4$, όπου παρατηρούμε επιβράδυνση. Όταν το βοηθητικό νήμα κάνει prefetch δεδομένα που θα χρειαστούν στο μακρινό μέλλον, η ιεραρχία της κρυφής μνήμης πάσχει cache pollution και σε ορισμένες περιπτώσεις μπορεί να αντικατασταθούν σημαντικά blocks δεδομένων. Βάσει αυτού, η επιλογή $n = 1$ είναι η καλύτερη.

Το PHT σχήμα πάσχει από ένα μεγάλο μειονέκτημα: δεν είμαστε σε θέση να επιλέξουμε πού να κάνουμε πρεφετση τα δεδομένα. Αυτό συμβαίνει διότι το βοηθητικό νήμα κάνει πρεφετση τα δεδομένα στην κρυφή μνήμη που μοιράζεται με το κύριο νήμα. Έτσι, το σχήμα PHT εξαρτάται από την αρχιτεκτονική του συστήματος και στερείται ευελιξίας. Με βάση αυτό το κίνητρο, προτείνουμε το Prefetch-Process-Thread-Alternation σχήμα, το οποίο βασίζεται στο PHT αλλά υπερνικά αυτό το μειονέκτημα.



Σχήμα 4.5: Επιτάχυνση με χρήση PHT όταν γεννιούνται $n \leq 4$ βοηθητικά νήματα



Σχήμα 4.6: Μότιβο εκτέλεσης του PHT σχήματος

Algorithm 5 : Main Thread code for PHT scheme

Initialize $Q, d, previous$
$prefetch \leftarrow 0$
$interrupt \leftarrow 0$
while Q is not empty **do**
  $u \leftarrow$ **Extract-Min**(Q)
  $prefetch \leftarrow 1$
  for all edge (u, v) outgoing from u **do**
    relaxation phase
  $interrupt \leftarrow 1$

Algorithm 6 : Helper Thread code for PHT scheme

while Q is not empty **do**
  while($prefetch$=0);
  $prefetch \leftarrow 0$
  $min \leftarrow$ **ReadMin(Q)**
  for all neighbors of min and while $interrupt$=0 **do**
    prefetch neighbors' data
  while($interrupt$=0);
  $interrupt \leftarrow 0$

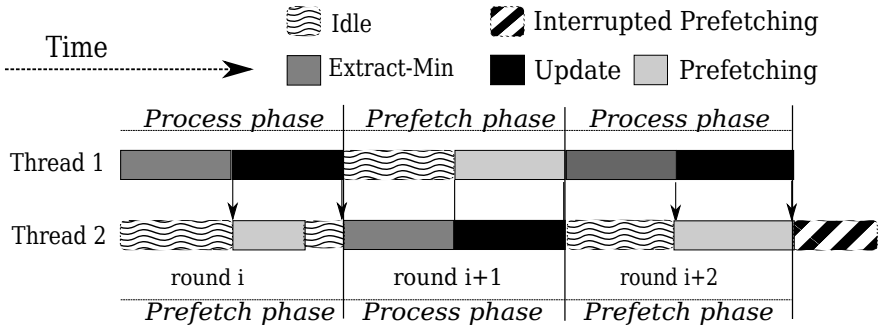
4.5 Prefetch-Process-Thread-Alternation

Σε αυτό το σχήμα παρουσιάζουμε ένα πιο εξελιγμένο σχήμα που ονομάζεται Prefetch-Process-Thread-Alternation (PPTA). Πρόκειται για ένα σχήμα με 2 νήματα τα οποία χρησιμοποιούν software prefetching. Στην πραγματικότητα, προσπαθούμε να κρύψουμε την καθυστέρηση μνήμης χρησιμοποιώντας δύο διαφορετικές φάσεις για κάθε νήμα. Σε σύγκριση με τον αρχικό αλγόριθμο, αναμένουμε σημαντική μείωση των αστοχιών της κρυφής μνήμης, τα οποία δημιουργήθηκαν από τη λειτουργία Update, με αποτέλεσμα τη βελτίωση της απόδοσης.

4.5.1 Μεθοδολογία

Στην υλοποίησή μας, ένα από τα δύο νήματα αρχίζει με την προετοιμασία όλων των σημαιών που απαιτούνται για το συγχρονισμό και των δύο νημάτων και των δομών δεδομένων Q, d και *previous*. Το σχήμα μας αποτελείται από δύο φάσεις σε κάθε γύρο του αλγορίθμου: τη φάση prefetch και φάση εκτέλεσης του αλγορίθμου. Ένα από τα νήματα είναι υπεύθυνο για την εκτέλεση του αλγορίθμου, ενώ το prefetch νήμα φορτώνει τα δεδομένα που θα απαιτηθούν κατά τον επόμενο γύρο. Ειδικότερα, το νήμα που εκτελεί τον αλγόριθμο εξάγει την ελάχιστη κορυφή ενώ το πρεφετση νήμα περιμένει σήμα για να ξεκινήσει το prefetching. Μετά τη λειτουργία Extract-Min, το νήμα εκτέλεσης ειδοποιεί το prefetch νήμα για να διαβάσει το νέο ελάχιστο κλειδί της ουράς προτεραιότητας και να αρχίσει να φέρνει στην προσωρινή μνήμη το κατάλληλο σύνολο δεδομένων. Στη συνέχεια, το νήμα εκτέλεσης ξεκινά τη λειτουργία Update, η ολοκλήρωση της οποίας σηματοδοτεί το τέλος του γύρου και ενημερώνει το prefetch νήμα ώστε να σταματήσει τη λειτουργία του. Όπως φαίνεται στο σχήμα 4.7, στην αρχή του γύρου $i+1$, τα δύο νήματα αλλάζουν ρόλους, για να εκμεταλλευτούν το γεγονός ότι το prefetch νήμα έφερε στην κρυφή μνήμη τα δεδομένα που χρειάζονται για την επικείμενη Update λειτουργία .

Αυτή η διαδικασία συνεχίζεται εναλλακτικά όσο το Q δεν είναι άδειο. Υπάρχουν δύο περιπτώσεις που πρέπει να εξετάσουμε σχετικά με τη φάση prefetch. Αυτές απεικονίζονται στην εικόνα 4.7. Στους γύρους i και $i+1$, το νήμα 2 έχει ολοκληρώσει τη φάση prefetch πριν το νήμα 1 ολοκληρώσει τη λειτουργία Update. Αντίθετα, κατά τη διάρκεια του κύκλου $i+2$, το νήμα 2 διακόπτεται κατά τη διάρκεια της πρεφετση φάσης και ξεκινά τη λειτουργία Extract-Min χωρίς να έχει προηγουμένως φορτώσει ολόκληρο το σύνολο των δεδομένων που είναι απαραίτητα για τη φάση χαλάρωσης. Αυτό το είδος διακοπής είναι απαραίτητο, δεδομένου ότι αν το νήμα 2 περιμένει για όλα τα κομμάτια των δεδομένων που πρέπει να φορτωθούν, η επερχόμενη λειτουργία Extract-Min



Σχήμα 4.7: Μοτίβο εκτέλεσης του PPTA σχήματος

δεν θα εκτελεστεί μέχρι να ολοκληρωθεί prefetching και επομένως η εκτέλεση του αλγορίθμου θα επιβραδυνθεί. Ο κώδικας και για τα δύο νήματα παρουσιάζεται στους αλγορίθμους 7 & 8.

4.5.2 Συντονισμός

Για να συντονίσουμε τα δύο νήματα χρησιμοποιούμε τέσσερις σημαίες συγχρονισμού. Απαιτούνται δύο σημαίες για να σηματοδοτήσουν την έναρξη της πρεφετση φάσης κάθε νήματος και δύο ακόμη για να τους ενημερώσουν για την ολοκλήρωση της φάσης χαλάρωσης και την έναρξη της λειτουργίας Extract-Min.

Algorithm 7 : Thread 1 code for PPTA scheme

```
Initialize Q, d, previous
prefetch_1 ← 1 extract_1 ← 1
prefetch_2 ← 0 extract_2 ← 0
while Q is not empty do
  while(prefetch_1=0);
  prefetch_1 ← 0
  min ← ReadMin(Q)
  for all neighbors of min and while extract_1=0 do
    prefetch neighbors' data
  while(extract_1=0);
  extract_1 ← 0
  u ← Extract-Min(Q)
  prefetch_2 ← 1
  for all edge (u, v) outgoing from u do
    relaxation phase
  extract_2 ← 1
```

Algorithm 8 : Thread 2 code for the PPTA scheme

```
while Q is not empty do
  while(prefetch_2=0);
  prefetch_2 ← 0
  min ← ReadMin(Q)
  for all neighbors of min and while extract_2=0 do
    prefetch neighbors' data
  while(extract_2=0);
  extract_2 ← 0
  u ← Extract-Min(Q)
  prefetch_1 ← 1
  for all edge (u, v) outgoing from u do
    relaxation phase
  extract_1 ← 1
```

Κεφάλαιο 5

Αξιολόγηση

5.1 Πειραματική αξιολόγηση

5.1.1 Ρυθμίσεις

Για τα πειράματά μας χρησιμοποιήσαμε δύο servers με 2 sockets, εξοπλισμένα με επεξεργαστές AMD Opteron και Intel Broadwell-EP. Τα κύρια χαρακτηριστικά των δύο συστημάτων παρουσιάζονται στον Πίνακα I και οι ιεραρχίες κρυφής μνήμης παρουσιάζονται αντίστοιχα στο σχήμα 5.1. Χρησιμοποιήσαμε τα configurations που παρουσιάζονται παρακάτω:

- **Opteron**:
    1. Νήματα τα οποία τρέχουν σε πυρήνες που μοιράζονται την L2 cache
- **Broadwell-EP**:
    1. Νήματα τα οποία τρέχουν σε πυρήνες που μοιράζονται την L1 cache (SMT)
    2. Νήματα τα οποία τρέχουν σε πυρήνες που μοιράζονται την L3 cache

Και τα δύο συστήματα αναπτύχθηκαν χρησιμοποιώντας C++. Συγκεκριμένα, οι σημαίες συντονισμού που χρησιμοποιήθηκαν και στα δύο σχήματα υλοποιήθηκαν χρησιμοποιώντας ατομικές λειτουργίες της C++ και την acquire & release memory order. Επίσης, απενεργοποιήσαμε τους hardware prefetchers, με βάση τα αποτελέσματα του Κεφάλαιο 4.

Πίνακας 5.1: Ρυθμίσεις συστημάτων

Name	**AMD Opteron**	**Intel Broadwell-EP**
#Cores	4×8	2×22
#Threads	32	88 (SMT)
Core clock	2.4 GHz	2.2 GHz
L1(Data)	16 KB, 4-way, 64B block size	32 KB, 8-way, 64B block size
L2	256 KB, 8-way, 64B block size (**shared per 2 cores**)	2 MB, 16-way, 64B block size (**private**)
L3	16 MB, 128-way, 64B block size (shared per NUMA node)	56 MB, 20-way, 64B block size (shared per die)
Memory	250 GB	64 GB
OS	Debian 8.8	Debian 8.3
Linux Kernel	3.2.0	4.7.0
GCC	4.9.2 with -O3 optimization	4.9.2 with -O3 optimization

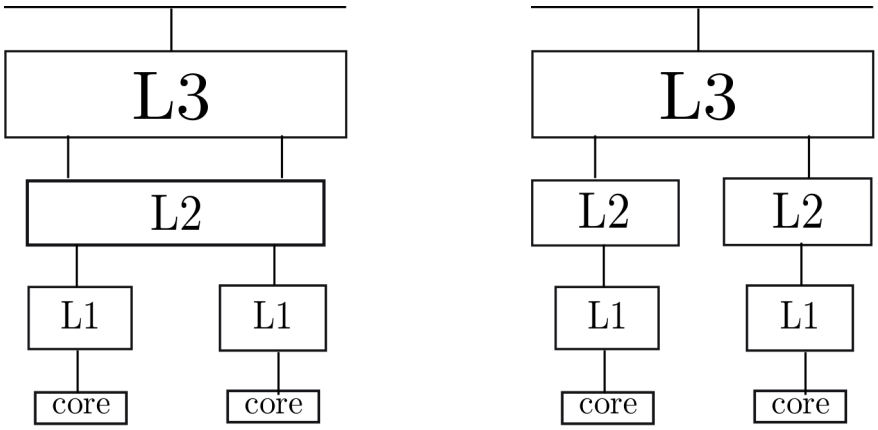
5.1.2 Τύποι γραφημάτων

Για να πειραματιστούμε σε γραφήματα με ποικιλία πυκνότητας και δομής, χρησιμοποιήσαμε την γεννήτρια GTgraph [14]. Κατασκευάσαμε γράφους με 10M κορυφές και διαφορετικό αριθμό ακμών από τις οικογένειες *Random*, *R-MAT* και *SSCA*.

- **Random:** Οι m ακμές κατασκευάζονται επιλέγοντας ένα τυχαίο ζεύγος μεταξύ n κόμβων.
- **R-MAT:** Κατασκευάζεται με τη χρήση του Αναδρομικού Πίνακα (R-MAT).
- **SSCA:** Χρησιμοποιείται στο benchmark DARPA HPCS SSCA.

5.1.3 Αποτελέσματα απόδοσης

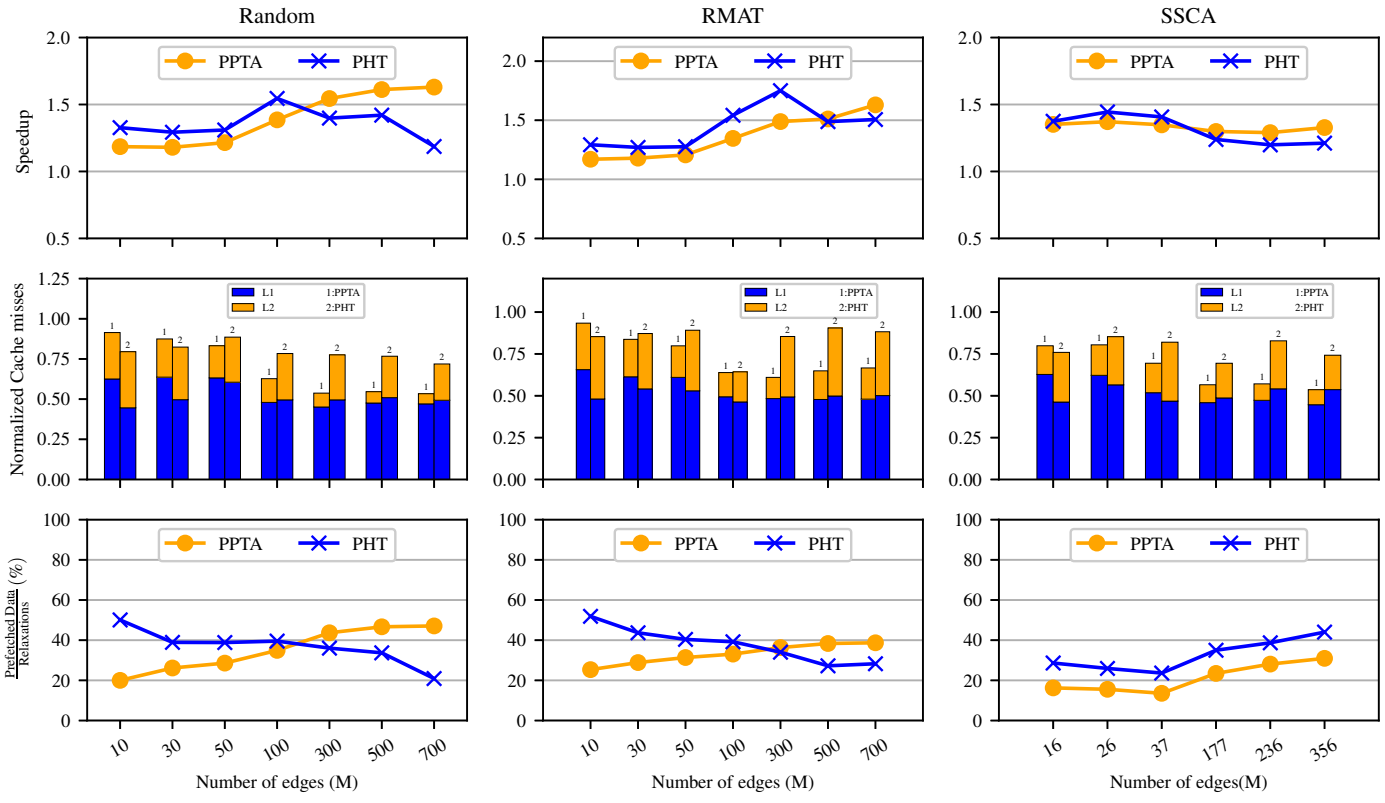
Θα αξιολογήσουμε τόσο το σχήμα PHT όσο και το PPTA. Τα σχήματα 5.2, 5.3 παρουσιάζουν την επιτάχυνση, το άθροισμα των αστοχιών μνήμης cache και την αναλογία των δεδομένων που έγιναν prefetch ανά φάση χαλάρωσης. Τα αποτελέσματα σχετικά με τις αστοχίες της κρυφής μνήμης εξομαλύνθηκαν χρησιμοποιώντας αυτά που μετρήθηκαν από τη σειριακή εκτέλεση και αναφέρονται στη φάση εκτέλεσης και των δύο σχημάτων (κύριο νήμα στο PHT). Στην περίπτωση του Opteron, δεν είχαμε τη δυνατότητα να μετρήσουμε τις αστοχίες μνήμης της L3 cache λόγω έλλειψης κατάλληλων



(α′) Opteron (β′) Broadwell

Σχήμα 5.1: Ιεραρχία κρυφής μνήμης των 2 συστημάτων

εργαλείων παρακολούθησης της απόδοσης. Εστιάζουμε τη μελέτη μας σε μία οικογένεια γραφημάτων, την *Random,* καθώς οι άλλες οικογένειες παρουσιάζουν παρόμοια συμπεριφορά. Τα γραφήματα 5.2, 5.3 παρουσιάζουν τα αποτελέσματα για τον AMD Opteron και τον Intel-Broadwell, αντίστοιχα. Παρακάτω, αναλύουμε τα αποτελέσματα και των δύο σχημάτων και τα συσχετίζουμε με τα χαρακτηριστικά των συστημάτων.

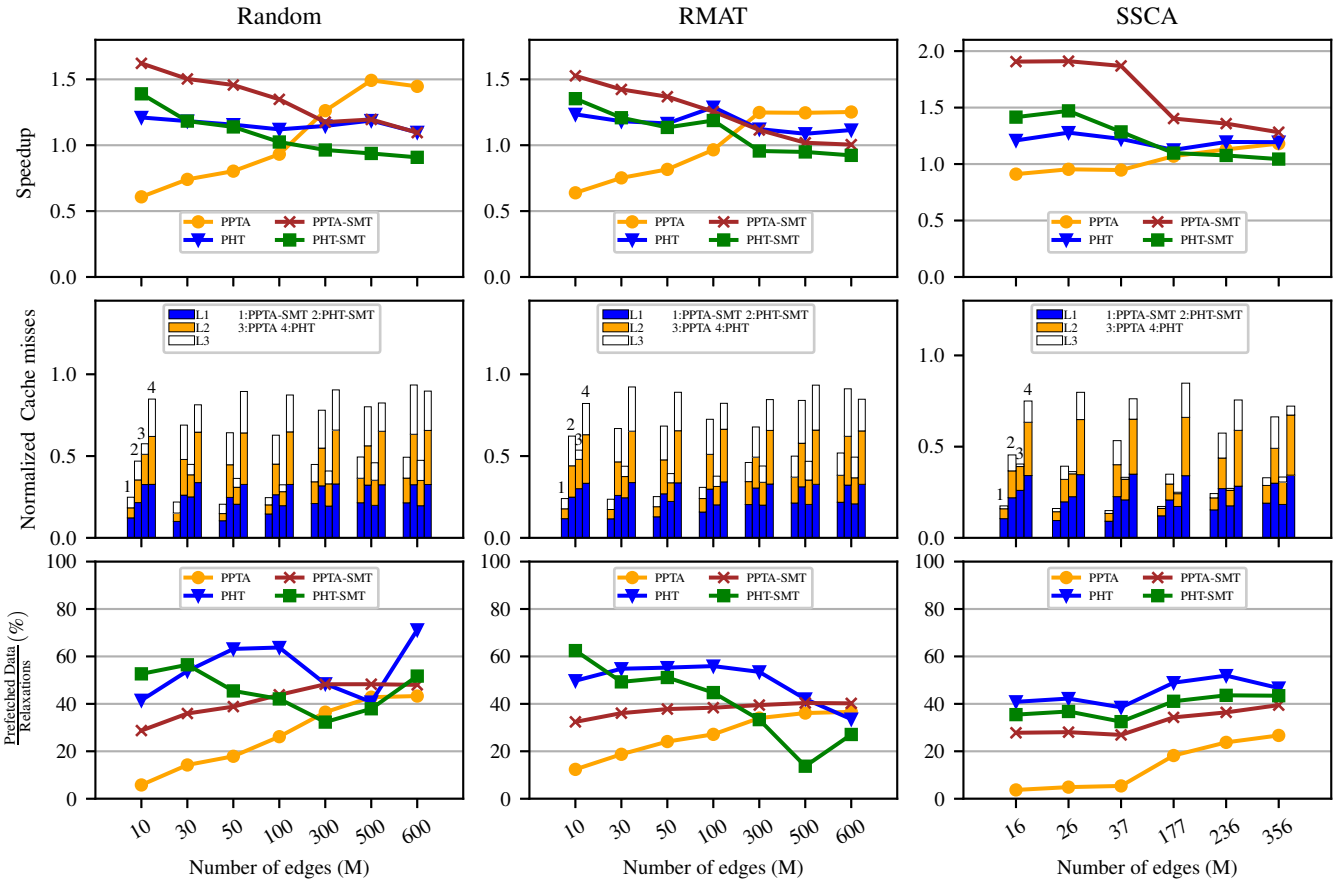


Σχήμα 5.2: Αποτελέσματα AMD-Opteron .

Όσον αφορά το PHT σχήμα, παρόλο που η απόδοση σε αραιά γραφήματα είναι καλύτερη από το PPTA, παρατηρούμε σημαντική επιβράδυνση όσο οι γράφοι γίνονται

πυκνότεροι. Χρησιμοποιώντας το σχήμα PPTA, τα δεδομένα γίνονται prefetch στην κρυφή μνήμη L1. Στην περίπτωση του PHT τα δεδομένα που έγιναν prefetch από το βοηθητικό νήμα μεταφέρονται στην L2 cache (Opteron) ή L3 cache (Broadwell) και η ποινή αστοχίας είναι υψηλότερη σε σύγκριση με το prefetching δεδομένων στην L1.

Παρατηρούμε ότι η επιτάχυνση, όσον αφορά το σχήμα PPTA στον Opteron, είναι ανάλογη με την πυκνότητα του γράφου. Καθώς οι γράφοι γίνονται πιο πυκνοί, περισσότερα δεδομένα γίνονται prefetch, οι αστοχίες κρυφής μνήμης της φάσης εκτέλεσης μειώνονται, και επιτυγχάνεται μεγαλύτερη επιτάχυνση. Το PPTA οδηγεί σε αισθητή μείωση των αστοχιών της cache, με το μέγιστο να είναι 45% όπως φαίνεται στο Σχήμα 5.2, οδηγώντας σε μέγιστη ταχύτητα 1.62 χρησιμοποιώντας το πιο πυκνό γράφο ο οποίος αποτελείται από 700M ακμές .



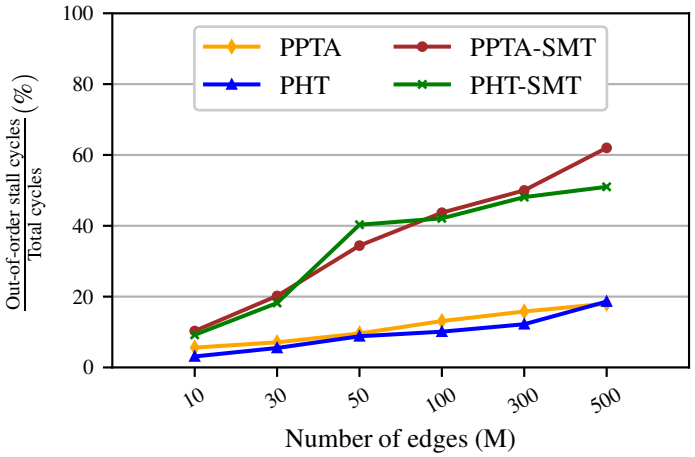
Σχήμα 5.3: Αποτελέσματα Intel-Broadwell-EP. Και για τα δύο συστήματα, οι σειρές παρουσιάζουν την επιτάχυνση, τις αστοχίες μνήμης της φάσης εκτέλεσης και τα δεδομένα που έχουν γίνει prefetch με επιτυχία. Οι στήλες αντιπροσωπεύουν τις τρεις οικογένειες γράφων.

Αυτό το μοτίβο παρατηρείται επίσης στις μετρήσεις του Broadwell, όπου η μέγιστη επιτάχυνση είναι 1.5.

Ωστόσο, υπάρχει μεγάλη διαφορά όσον αφορά τα αραιά γραφήματα. Στην περίπτωση του Broadwell, παρατηρούμε μια σημαντική επιβράδυνση σε αραιά γραφήματα σε αντίθεση με την περίπτωση του Οπτερον όπου υπάρχει μια μικρή επιτάχυνση. Αυτό μπορεί να αποδοθεί στο γεγονός ότι με τη χρήση του PPTA σχήματος εισάγουμε overhead συγχρονισμού και συνάφειας μνήμης. Και τα δύο νήματα επεξεργάζονται συχνά την ουρά προτεραιότητας και τους πίνακες *distance*, *previous* και όταν κάθε ένα από

αυτά προσπαθεί να διαβάσει ή να γράψει σε αυτές τις θέσεις μνήμης, οι μηχανισμοί συνάφειας της κρυφής μνήμης ενεργοποιούν τις cache-to-cache μεταφορές. Όσον αφορά την κοινή ιεραρχία L3 του Broadwell, τα αραιά γραφήματα εκθέτουν το overhead αυτών των μηχανισμών συνάφειας που είναι υψηλότερο σε σύγκριση με την περίπτωση του Opteron, όπου συμβαίνουν μεταφορές δεδομένων μεταξύ των κρυφών μνημών L2.

Συμπεραίνουμε ότι η καλύτερη επιλογή για το PPTA σχήμα είναι να χρησιμοποιούμε πυρήνες που μοιράζονται όσο το δυνατόν περισσότερα επίπεδα της ιεραρχίας της μνήμης. Με αυτόν τον τρόπο, είμαστε σε θέση να μειώσουμε το συνολικό overhead, καθώς τα δεδομένα μετακινούνται για μικρότερες αποστάσεις εντός της ιεραρχίας της κρυφής μνήμης.



Σχήμα 5.4: Ποσοστό των κύκλων κατα τη διάρκεια των οποίων το νήμα εκτέλεσης παραμένει αδρανές εξαιτίας της συμφόρησης στο pipeline

Χρησιμοποιώντας τον Broadwell-EP, μπορούμε να χρησιμοποιήσουμε ηψπερτηρεαδς που μοιράζονται ολόκληρη την ιεραρχία μνήμης συμπεριλαμβανομένου του υψηλότερου επιπέδου κρυφής μνήμης (L1). Το σχήμα PPTA-SMT παρουσιάζει αξιόλογες επιταχύνσεις σε αραιά γραφήματα. Ωστόσο, καθώς οι γράφοι γίνονται πυκνότεροι, οι επιδόσεις επιδεινώνεται. Αυτό μπορεί να αποδοθεί στο γεγονός ότι καθώς η φάση prefetch γίνεται πιο απαιτητική για τη μνήμη, το hyperthread διατηρεί έναν αυξανόμενο αριθμό πόρων του pipeline για το για κάνει prefetch τα δεδομένα που απαιτούνται από την εφαρμογή μας και στη συνέχεια επιβραδύνει τη λειτουργία ενημέρωσης που εκτελείται από το νήμα διεργασίας.

Στο σχήμα 5.4 μετρήσαμε και για τα δύο σχήματα, το ποσοστό των κύκλων κατα τη διάρκεια των οποίων το νήμα εκτέλεσης παραμένει αδρανές εξαιτίας της συμφόρησης στο pipeline. Παρατηρούμε ότι, με τη χρήση των hyperthreads, και καθώς τα γραφήματα γίνονται πυκνότερα, το κύριο μέρος της εκτέλεσης δαπανάται στην αναμονή για τους πόρους του pipeline. Η συμφόρηση αυξάνεται επίσης λόγω του μεγαλύτερου συνόλου δεδομένων που χρησιμοποιείται. Συγκεκριμένα, στην περίπτωση του γράφου των 500M ακμών *Random*, παρά την παρόμοια μείωση των αστοχιών στη μνήμη που επιτυγχάνουμε με τη χρήση PPTA και PPTA-SMT, η επιτάχυνση είναι μικρότερη στη δεύτερη περίπτωση εξαιτίας του προαναφερθέντος overhead συμφόρησης.

Όσον αφορά την Broadwell, υπάρχει ένα σημαντικό trade-off όσον αφορά τη χρήση του hyperthread. Παρόλο που μειώνουμε το overhead συνάφειας χρησιμοποιώντας μια κοινή ιεραρχία μνήμης και ωφέλουμαστε σε αραιούς γράφους, οι πυκνότεροι αποκαλύπτουν το αυξανόμενο overhead συμφόρησης στο pipeline.

Κεφάλαιο 6

Συμπεράσματα & Μελλοντική έρευνα

Στο πρώτο μέρος της παρούσας διπλωματικής εργασίας, εργαστήκαμε αρχικά σε διάφορες πρακτικές βελτιστοποιήσεις του αλγορίθμου του Dijkstra για να ανακαλύψουμε το σημείο συμφόρησής του (bottleneck). Διερευνήσαμε το χώρο σχεδιασμού των ουρών προτεραιότητας. Γι΄ αυτό, υλοποιήσαμε τρεις διαφορετικές ουρές προτεραιότητας για να ανακαλύψουμε τον αντίκτυπό τους στην απόδοση του αλγορίθμου. Κάθε ουρά επηρεάζει διαφορετικά τις λειτουργίες του αλγορίθμου αλλά η απόδοσή του παραμένει σχεδόν η ίδια.

Επιπλέον, καταγράψαμε τις λειτουργίες του λεπτομερώς, ανακαλύψαμε το σημείο συμφόρησής (Update) και τον χαρακτηρίσαμε έναν memory-bound αλγόριθμο, η απόδοση του οποίου επηρεάζεται αρνητικά από τις μη κανονικές προσβάσεις στη μνήμη. Προκειμένου να μειώσουμε την καθυστέρηση απόκρισης της μνήμης, παρουσιάσαμε δύο σχήματα τα οποία χρησιμοποιούν prefetching λογισμικού για την αντιμετώπιση αυτών των αιτημάτων μνήμης και την επιτάχυνση του αλγορίθμου του Dijkstra. Το σχέδιο Prefetch-Process-Thread-Alternation προέρχεται από την απλούστερη τεχνική Prefetching-Helper-Thread. Από τη μία, το PHT δημιουργεί ένα κύριο νήμα που εκτελεί τον αλγόριθμο, ενώ ένα βοηθητικό νήμα κάνει prefetch τα δεδομένα στην κοινή κρυφή μνήμη. Από την άλλη πλευρά, το PPTA περιλαμβάνει δύο νήματα που εναλλάσσονται μεταξύ μιας φάσης prefetch και μίας φάσης εκτέλεσης, για να αποκρύψουμε την καθυστέρηση που προκαλείται από τις αστοχίες της κρυφής μνήμης.

Συμπεραίνουμε ότι το PPTA εισάγει overhead συνάφειας το οποίο μπορεί να μειωθεί χρησιμοποιώντας πυρήνες που μοιράζονται όσο το δυνατόν περισσότερα επίπεδα της ιεραρχίας της μνήμης. Επιπρόσθετα, το PPTA επιτυγχάνει αυξανόμενα speedups καθώς τα γραφήματα γίνονται πιο πυκνά και ξεπερνάει το PHT. Ο AMD Opteron επιτυγχάνει maximum επιτάχυνση 1.62 για τους πιο πυκνούς γράφους και ο Intel Broadwell-EP φτάνει 1.82 για ένα αραιό γράφο, χρησιμοποιώντας το hyperthread. Λαμβάνοντας υπόψιν τη σειριακή φύση του αλγορίθμου του Dijkstra και τις εγγενείς δυσκολίες παραλληλισμού του, αυτό αποτελεί ένα σημαντικό κέρδος απόδοσης.

Το γεγονός ότι το PPTA σχήμα επιτυγχάνει έναν αυξανόμενο ρυθμό επιτάχυνσης καθώς οι γράφοι γίνονται πιο πυκνοί, είναι ένα σημαντικό αποτέλεσμα, καθώς έχουν προκύψει εξαιρετικά μεγάλης κλίμακας γραφήματα σε διάφορες σύγχρονες εφαρμογές, όπως το γράφημα του κοινωνικού δικτύου του Twitter και του Human Brain Project.

Ως μελλοντική διερεύνηση, σκοπεύουμε να εξετάσουμε τις δυνατότητες του PPTA σχήματος και πώς μπορεί να εφαρμοστεί σε άλλους memory intensive αλγορίθμους

παρόμοιας φύσης. Ένα άλλο σημαντικό ζήτημα είναι το πώς αυτό το σχήμα μπορεί να κλιμακωθεί αποτελεσματικά.

Θα διερευνήσουμε επίσης την ιδέα ενός Near-Data Processing σχήματος που παρέχει εξειδικευμένο υλικό και κατάλληλο λογισμικό API για την αντιμετώπιση memory intensive εφαρμογών, όπως η επεξεργασία γράφων.

Bibliography

[1] M. L. Fredman and R. E. Tarjan, "Fibonacci heaps and their uses in improved network optimization algorithms," *J. ACM*, vol. 34, pp. 596–615, July 1987.

[2] G. S. Brodal, "Worst-case efficient priority queues," in *In proc. 7th ACM-siam symposium on discrete algorithms*, pp. 52–58, 1996.

[3] H. Rihani, P. Sanders, and R. Dementiev, "Brief announcement: Multiqueues: Simple relaxed concurrent priority queues," in *Proceedings of the 27th ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '15, (New York, NY, USA), pp. 80–82, ACM, 2015.

[4] J. Lindén and B. Jonsson, *A Skiplist-Based Concurrent Priority Queue with Minimal Memory Contention*, pp. 206–220. Cham: Springer International Publishing, 2013.

[5] "Skiplist-based concurrent priority queues," in *Proceedings of the 14th International Symposium on Parallel and Distributed Processing*, IPDPS '00, (Washington, DC, USA), pp. 263–, IEEE Computer Society, 2000.

[6] P. Sanders, "Fast priority queues for cached memory," *J. Exp. Algorithmics*, vol. 5, Dec. 2000.

[7] U. Meyer and P. Sanders, "Δ-stepping: A parallelizable shortest path algorithm," *J. Algorithms*, vol. 49, pp. 114–152, Oct. 2003.

[8] N. Anastopoulos, K. Nikas, G. Goumas, and N. Koziris, "Early experiences on accelerating dijkstra's algorithm using transactional memory," in *Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing*, IPDPS '09, (Washington, DC, USA), pp. 1–8, IEEE Computer Society, 2009.

[9] K. Nikas, N. Anastopoulos, G. I. Goumas, and N. Koziris, "Employing transactional memory and helper threads to speedup dijkstra's algorithm," in *ICPP 2009, International Conference on Parallel Processing, Vienna, Austria, 22-25 September 2009*, pp. 388–395, IEEE Computer Society, 2009.

[10] R. E. Tarjan, *Data Structures and Network Algorithms*. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 1983.

[11] S. Mittal, "A survey of recent prefetching techniques for processor caches," *ACM Comput. Surv.*, vol. 49, pp. 35:1–35:35, Aug. 2016.

[12] S. Ainsworth and T. M. Jones, "Software prefetching for indirect memory accesses," in *Proceedings of the 2017 International Symposium on Code Genera tion and Optimization*, CGO '17, (Piscataway, NJ, USA), pp. 305–317, IEEE Press, 2017.

[13] D. Kim, S. S.-w. Liao, P. H. Wang, J. d. Cuvillo, X. Tian, X. Zou, H. Wang, D. Yeung, M. Girkar, and J. P. Shen, "Physical experimentation with prefetching helper threads on intel's hyper-threaded processors," in *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, CGO '04, (Washington, DC, USA), pp. 27–, IEEE Computer Society, 2004.

[14] D. Bader and K. Madduri, ""gtgraph: A suite of synthetic graph generators"," 2006.

Contents

List of Figures

Listings

Chapter 1

Introduction

Optimizing the performance of memory-bound applications is a challenging task and becomes even harder when they do not expose parallelism in a straightforward way. These applications typically operate on large data-sets with limited temporal locality. The cost of intensive data movement and irregular memory accesses is significant since most of these accesses are served by main memory. The so called memory-wall accentuates the need to use techniques that tolerate or reduce memory latency.

1.1 Modern systems

Modern systems have many abilities that reduce memory latency and increase performance:

- Hardware and Software prefetching have been proposed as one of the most important solutions to hide memory latency in systems. Modern systems provide architectular support for software prefetching and a small number of hardware prefetchers.

- Multiple cores to exploit parallelism and scale efficiently.

- High Bandwidth Memory technology achieves higher bandwidth while using less power in a substantially smaller form factor than DDR4 or GDDR5.This is achieved using 3D-stacked memory, including an optional base die with a memory controller, which are interconnected by through-silicon vias (TSV) and microbumps.

- Processing-In-Memory that provides to the main memory module computing capabilities and reduces large data transfers.

- Graphics processing units (GPU) whose highly parallel structure makes them more efficient than general-purpose CPUs for algorithms where the processing of large blocks of data is done in parallel.

1.2 Optimizing memory-bound applications

There are also ways to optimize the applications that are memory-bound:

- Extract parallelism and exploit the total memory bandwidth.
- Design data structures with high throughput.
- Profiling of the application and compiler optimizations.
- Choose the appropriate architectural model to suit the needs of the application.

1.3 Dijkstra's algorithm

Graph applications are used widely in our days. There exist many applications in economy, aeronautics, physics, biology (for analyzing DNA), mathematics and other areas that utilize graph theory to model complex problems. In this thesis, we apply software prefetching to Dijkstra's algorithm since it is a challenging example of memory-bound application whose performance is negatively affected by irregular memory accesses.

Chapter 2

Dijkstra's Algorithm

2.1 Shortest path problem

In graph theory, the shortest path problem is the problem of finding a path between two vertices (or nodes) in a graph such that the sum of the weights of its constituent edges is minimized.

The shortest path problem can be defined for graphs whether undirected, directed, or mixed. For directed graphs the definition of path requires that consecutive vertices be connected by an appropriate directed edge.

Two vertices are adjacent when they are both incident to a common edge. A path in an undirected graph is a sequence of vertices $P = (v_1, v_2, ..., v_n) \epsilon V \times V \times ... \times V$ such that v_i is adjacent to v_{i+1} for $1 \leq i < n$. Such a path P is called a path of length $n - 1$ from v_1 to v_n.

Let $e_{i,j}$ be the edge incident to both v_i and v_j. Given a real-valued weight function $f : E \to \Re$, and an undirected graph G, the shortest path from v to v' is the path $P = (v_1, v_2, ..., v_n)$ (where $v_1 = v$ and $v_n = v'$) that over all possible n minimizes the sum $\sum_{i=1}^{n-1} f(e_{i,i+1})$. When each edge in the graph has unit weight or $f : E \to 1$, this is equivalent to finding the path with fewest edges.

There are some variations of the shortest path problem :

- The single-source shortest path problem, in which we have to find shortest paths from a source vertex v to all other vertices in the graph.
- The single-destination shortest path problem, in which we have to find shortest paths from all vertices in the directed graph to a single destination vertex v. This can be reduced to the single-source shortest path problem by reversing the arcs in the directed graph.
- The all-pairs shortest path problem, in which we have to find shortest paths between every pair of vertices v, v' in the graph.

The most important algorithms for solving this problem are:

- **Dijkstra's algorithm**: solves the single-source shortest path problem.
- **Bellman-Ford algorithm**: solves the single-source problem if edge weights may be negative.

- **A* search algorithm**: solves the single pair shortest path using heuristics to try to speed up the search.
- **Floyd–Warshall algorithm**: solves all pairs shortest paths.
- **Johnson's algorithm**: solves all pairs shortest paths.

2.2 Algorithm's History & Proof of Correctness

In 1959 a three pages long paper entitled A Note on Two Problems in Connexion with Graphs [1] was published in the journal Numerische Mathematik. In this paper Edsger W. Dijkstra - then a twenty-nine-year-old computer scientist - proposed algorithms for the solution of two fundamental graph theoretic problems: the minimum weight spanning tree problem and the shortest path problem. Edsger Wybe Dijkstra is also known for his many essays on programming and received the A. M. Turing Award (widely considered the most prestigious award in computer science) in 1972. Today Dijkstra's Algorithm for the shortest path problem is one of the most celebrated algorithms in computer science (CS) and a very popular algorithm in operations research (OR) . In the literature this algorithm is often described as a greedy algorithm. For example, the book Algorithmics (Brassard and Bratley [1988, pp. 87-92][2]) discusses it in the chapter entitled Greedy Algorithms. The Encyclopedia of Operations Research and Management Science (Gass and Harris [1996, pp. 166-167]) describes it as a "... node labelling greedy algorithm ... " and a greedy algorithm is described as "... a heuristic algorithm that at every step selects the best choice available at that step without regard to future consequences ..." (Gass and Harris [1996, p. 264]).

Although the algorithm is very popular in the OR/MS literature, it is generally regarded as a "computer science method". Apparently this is due to three factors: (a) its inventor was a computer scientist (b) its association with special data structures, and (c) there are competing OR/MS oriented algorithms for the shortest path problem. One of the main reasons for the popularity of Dijkstra's Algorithm is that it is one of the most important and useful algorithms available for generating exact optimal solutions to a large class of shortest path problems. The point being that this class of problems is extremely important theoretically, practically, as well as educationally. The algorithm exists in many variants. Dijkstra's original algorithm finds the shortest path between two nodes, but the most popular variant of this algorithm fixes a single node as the "source" node and finds the shortest paths from the source to all other nodes in the graph, producing a shortest-path tree. Furthermore, in some fields (artificial intelligence) Dijkstra's algorithm or a variant of it is known as uniform-cost search and formulated as an instance of the more general idea of best-first search.

Lemma: When a vertex u is added to S set (visited nodes), then $dist[u] = \delta(s,u)$where $\delta(s,u)$ is the length of the shortest path from the source to vertex u.

Proof: Suppose that the algorithm attempts to add a vertex u to the set S for which $dist[u] \neq \delta(s,u)$. Then, $dist[u] > \delta(s,u)$.

Now consider the shortest path (Fig. 2.1) from source s to vertex u - $s \in S$ and $u \in V \setminus S$. Let (x, y) be the edge taken by the path, where $x \in S$ and $y \in V \setminus S$ (it may be that $x = s$ and/or $y = u$).

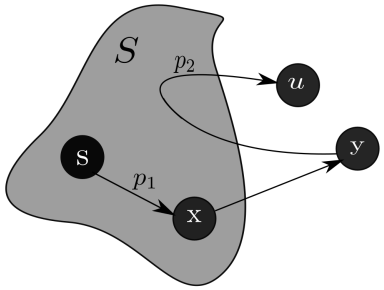


Figure 2.1: Proof of correctness

After doing the relaxation in vertex x we can conclude that

$$dist[y] \leq dist[x] + w[x, y] \tag{2.1}$$

where $w : E \to \Re$ maps edges to real-valued weights.

By hypothesis $x \in S$ so :

$$dist[x] = \delta(s, x) \tag{2.2}$$

But $< s, ..., x, y >$ is a subpath of the shortest path of Fig. 2.1 so :

$$\delta(s, y) = dist[x] + w(x, y) = \delta(s, x) + w(x, y) \tag{2.3}$$

In addition, it is true that y precedes of u in the shortest path (s, u) so

$$\delta(s, y) \leq \delta(s, u) \tag{2.4}$$

and therefore $d[y] = \delta(s, y) \leq \delta(s, u) \leq d[u]$.

The vertex we are going to extract is u so $d[u] \leq d[y]$ since both $y, u \in V \setminus S$.

Using 2.4 it is concluded that $d[y] = \delta(s, y) = \delta(s, u) = d[u]$. This is a contradiction to our hypothesis that $d[u] \neq \delta(s, u)$. By the lemma, it is true that $dist[u] = (s, u)$ when u is added to S for all vertices $u \in V$.

2.2.1 Applications

Dijkstra's algorithm is able to compute the shortest path from a node s to every other node u in a graph with non-negative weights. The range of applications it can be used in is wide. Network routing protocols (RIPM,BGP), VLSI design, Latex typesetting and social networks are some of the applications that this algorithm is used in. Some use cases are also:

- GPS navigation & maps
- Traffic planning
- Robot Motion and Navigation
- Driverless vehicles
- Transportation Networks
- Video Games and Virtual Tours

2.3 Basic Implementation of Dijkstra's Algorithm

2.3.1 Pseudocode & Operations & Data structures

Dijkstra's algorithm computes single source shortest paths (SSSP) for directed graphs with non-negative edges [1]. It is asymptotically the fastest known single-source shortest-path algorithm for graphs with unbounded non-negative weights. The algorithm is based one the observation that any subpath of any shortest path is itself a shortest path (optimal substructure). Specifically, let $G = (V, E)$ be a directed graph with $n = |V|$ vertices, and $w : E \to \Re^+$ weight function assigning non-negative real-valued weights to the edges of G. For each vertex v, the SSSP problem computes $\delta(v)$, the value of the shortest path from a source vertex s to v. For each vertex v, Dijkstra's algorithm maintains a shortest-path estimate (or tentative distance) $d(v)$, which is an upper bound for the actual value of the shortest path from s to v, $\delta(v)$. Initially, $d(v)$ is set to $+\infty$ and through successive edge relaxations it is gradually decreased, converging to $d(v)$. The relaxation of an $edge(v, w)$ sets $d(w)$ to $min\{d(w), d(v) + w(v, w)\}$, which means that the algorithm tests whether it can decrease the weight of the shortest path from s to w by going through v. Finally, this algorithm updates an array called *previous* so that the shortest path to a vertex v can be recursively reconstructed.

The algorithm is presented in more detail in Algorithm 1. It maintains a partition of V into settled, queued and unreached vertices. Settled vertices have $d(v) = \delta(v)$; queued have $d(v) > \delta(v)$ and $d(v)! = \infty$; unreached have $d(v) = \infty$. Initially, only s is queued, $d(s) = 0$ and all other vertices are unreached. In each iteration of the algorithm, the vertex with the smallest shortest-path estimate is selected, its state is permanently changed to settled and all its outgoing edges are relaxed, causing any of its neighbors that were unreached by the source vertex until this point to become queued.

The basic data structure lying at the heart of Dijkstra's algorithm is a priority queue of vertices, keyed by their $d()$ values. The queue maintains all but the settled vertices of the graph and must be a minimum priority queue in order to support the Extract-Min operation.

At each iteration, the vertex with the minimum key is extracted from a min-priority queue (**Extract-Min**) and its state is settled to *visited*. Then, its outgoing edges are relaxed (**Update**) and the keys d of the neighbors, whose $state \neq visited$, are decreased if needed. It is also essential to update the *previous* array and decrease the key of each neighbor-node in the priority queue to retain its minimum property (**Decrease-Key**).

The complexity of the algorithm depends on the kind of the priority queue and therefore the complexity of its operations. It also depends on the data structure used to represent the graph. The Extract-Min's operation complexity is $\mathcal{O}(V)$ since all vertexes need to be extracted from the priority queue. Using a common binary heap results in $\mathcal{O}(VlogV)$ since the extract-min operation of this priority queue is proportional to height of the heap. During the relaxation phase we relax the edges starting from the extracted vertex and update the heap. The so called decrease key operation of the priority queue is performed to all the edges of the input graph and it costs $\mathcal{O}(ElogV)$. Consequently, the running time of the algorithm using a common binary heap is $\mathcal{O}(ElogV + VlogV)$.

Algorithm 1 : Dijkstra's Algorithm

Require: Graph $G(V, E)$, Source vertex S
Ensure: Predecessors array *previous*
Shortest distance array d

```
1: for all v ∈ V[G] do
2:    d[v] ← +∞
3:    previous[v] ← undefined
4: d[s] ← 0                          }
5: S ← empty set                     } Insert Operation
6: Q ← V[G]                          }
7: while Q is not empty do
8:    u ← ExtractMin(Q) // Extract Min operation
9:    S ← S ∪ {u}
10:   for all edge (u, v) outgoing from u do
11:      if v ∉ S then
12:         sum ← d[u] + w(u, v)
13:         if sum < d[v] then
14:            u ← DecreaseKey(Q,v,sum)   }
15:            d[v] ← d[u] + w(u, v)       } Relaxation
16:            previous[v] := u           }
```

2.3.2 Uniform Cost Search

In common presentations of Dijkstra's algorithm, initially all nodes are entered into the priority queue. This is, however, not necessary: the algorithm can start with a priority queue that contains only one item, and insert new items as they are discovered (instead of doing a decrease-key, check whether the key is in the queue; if it is, decrease its key, otherwise insert it). This variant has the same worst-case bounds as the common variant, but maintains a smaller priority queue in practice, speeding up the queue operations. Moreover, not inserting all nodes in a graph makes it possible to extend the algorithm to find the shortest path from a single source to the closest of a set of target nodes on infinite graphs or those too large to represent in memory. The resulting algorithm is called uniform-cost search (UCS) in the artificial intelligence literature and is described using pseudocode in Alg. 1.

In Figure 2.2 we present the speedup achieved by using the UCS optimization instead of the default Dijkstra's algorithm. The priority queue we used is a simple binary heap and the graph consists of 10M nodes and 500M edges. The platform used is an Intel-Broadwell-EP whose configuration is described in Chapter 5. We reach a 1.7 speedup since the priority queue remains smaller during the execution of the algorithm. Thus, less time is spent on updating it.

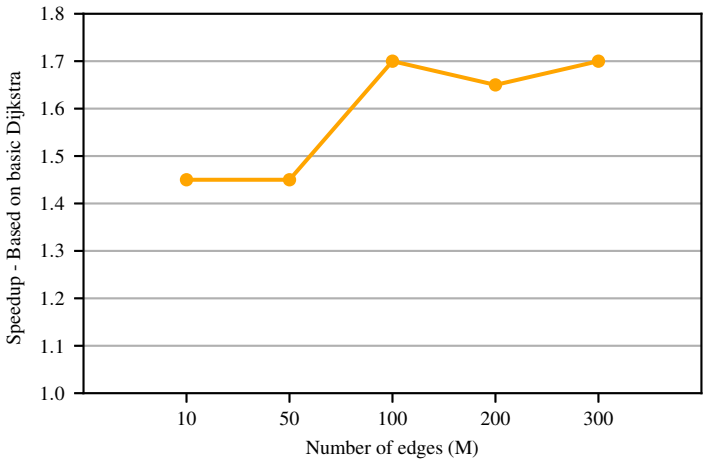


Figure 2.2: We show the speedup reached by using the UCS optimization.

2.3.3 Representation of graph

There are several ways to represent graphs in memory, each one of them having pros and cons. Some important criteria to take into consideration is the memory footprint and how long it takes to determine whether a given edge is in the graph. We are going to present two ways to store the input graph in memory.

Adjacency matrix

Given a graph with $|V|$ vertices, an adjacency matrix is a $|V| \times |V|$ matrix of real or binary values (depending on the type of the graph). In the case of a non-weighted graph, the entry in row i and column j is 1 if and only if the edge (i,j) in the graph. If we need to indicate an edge weight, the entry mentioned before contains a real value w if the edge's (i,j) cost is w. We show the adjacency matrix for a weighted bi-directional graph:

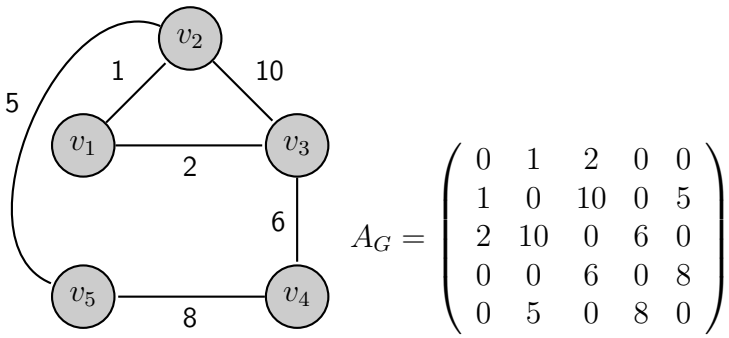


Figure 2.3: Graph and adjacency matrix. Example graph with 5 vertices and 5 edges and its adjacency matrix A_G.

Adjacency Lists

A more space-efficient way to implement a sparsely connected graph is to use an adjacency list. An adjacency list consists of a list of data structures using key-values

Algorithm 2 : Dijkstra with UCS optimization

Require: Graph $G(V, E)$, Source vertex S
Ensure: Predecessors array *previous*
Shortest distance array d

```
1: for all v ∈ V[G] do
2:    d[v] ← +∞
3:    previous[v] ← undefined
4: d[s] ← 0 // Initialization
5: S ← empty set
6: Q ← s
7: while Q is not empty do
8:    u ← Extract-Min(Q) // Extract-Min
9:    S ← S ∪ {u}
10:   for all edge (u, v) outgoing from u do
11:      if v ∉ S then
12:         sum ← d[u] + w(u, v)
13:         if v ∉ Q then
14:            Insert(Q,v,sum)
15:         else
16:            if sum < d[v] then // Update
17:               u ← DecreaseKey(Q,v,sum)
18:               d[v] ← d[u] + w(u, v)
19:               previous[v] ← u
```

where the key is a vertex and the value is the weight of an edge. For each vertex i, we store an array of the vertices adjacent to it. We typically have an array of $|V|$ adjacency lists. For a directed graph, the adjacency list contains a total of $|E|$ elements, one element per directed edge. There are many variations of this basic idea, differing in the details of how the collections are implemented. We are going to present two typical implementations of adjacency lists.

Linked List In Figure 2.4 we show the Adjacency List representing the graph of Figure 2.3. It is created using linked lists.

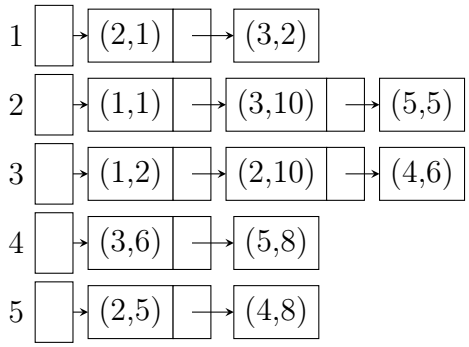


Figure 2.4: Adjacency list implemented using linked lists.

Dynamic Array In Figure 2.5 the Adjacency list presented is created using dynamic arrays.

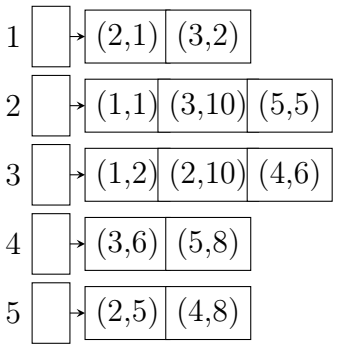


Figure 2.5: Adjacency list implemented using dynamic arrays.

Adjacency Matrix vs Adjacency List

Space

As far as the memory space is concerned, the adjacency matrix consumes $\mathcal{O}(V^2)$ and even in the case of sparse graphs, the consumed space remains the same. On the other hand, the adjacency list uses $\mathcal{O}(|V| + |E|)$ of space. In the worst case, there can be $C(V, 2)$ number of edges in a graph thus consuming $\mathcal{O}(V^2)$ space.

Operations

Besides the space tradeoff, the different data structures also facilitate different operations. Finding all vertices adjacent to a given vertex in an adjacency list is as simple as reading the list, and takes time proportional to the number of neighbors. With an adjacency matrix, an entire row must instead be scanned, which takes a larger amount of time, proportional to the number of vertices in the whole graph. On the other hand, testing whether there is an edge between two given vertices can be determined at once with an adjacency matrix, while requiring time proportional to the minimum degree of the two vertices with the adjacency list.

In this thesis, the graphs we are using are sparse. That is why, we are going to use the adjacency list to represent them. In Figure 2.6 we present the speedup achieved by using a dynamic array data structure instead of a linked list, for a 10M nodes graph with varying number of edges. We also show the reduction of cache misses.

We witness that using a dynamic array instead of a linked list leads to noticeable speedup. Regarding the densest of our graphs, we gain a 6x performance boost which can be attributed to the fact that cache misses are reduced by 50%. Cache misses are reduced since the dynamic array implementation of the adjacency list is more cache-friendly than a single-linked-list. When the algorithm iterates over the neighbors of the extracted vertex and accesses the edge $w(u, i)$ we do not need a pointer chasing to access it, however we can simply use an index to access the dynamic array. This results in transferring a block of data from the memory to the L1 cache instead of bringing just one element. This block consists of continuous elements of the array which will be needed during the next iterations. Therefore, this piece of data will be in L1 cache and cache misses are reduced.

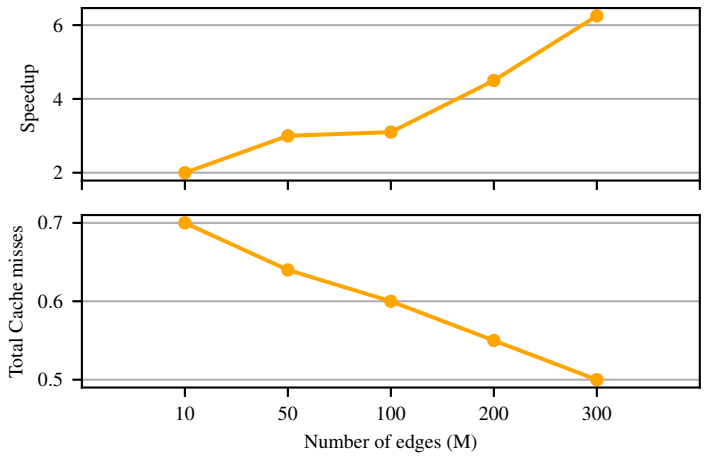


Figure 2.6: Speedup reached by using dynamic arrays instead of linked lists.

2.4 Priority Queues

One of the most important heaps that have been proposed and concern Dijkstra's Algorithm case is the Fibonacci heap which is based on the Binomial heap. This heap has been developped by Fredman et al. [3] and has better amortized running time than many other priority queue data structures including the binary heap and the binomial heap. Although Fibonacci heaps look very efficient, they have the following two drawbacks: They are complicated when it comes to coding them. In addition, they are not as efficient in practice when compared with the theoretically less efficient forms of heaps, since in their simplest version they require storage and manipulation of four pointers per node, compared to the two or three pointers per node needed for other structures.

Furthermore, the Brodal queue is a priority queue with very low worst case time bounds [4]. It is the first heap variant to achieve these bounds without resorting to amortization of operational costs. While having better asymptotic bounds than other priority queue structures, they are, in the words of Brodal himself, "quite complicated" and "not applicable in practice".

Several works have proposed priority queues that reach high throughputs under high contention. In [5] Rihani et al. propose a MultiQueue scheme which manages an array of cp sequential priority queues where p is the number of the systems' threads. Their evaluation indicates that MultiQueues scale really well on single socket systems. However, the semantics of the Delete-Min operation are relaxed and integrating in Dijkstra's algorithm is complex.

Much research has also been conducted towards the use of Skip list based concurrent priority queues rather than heaps [6, 7]. Skip list provides bigger opportunities for extracting parallelism, at the cost of memory footprint and thus it is occasionally used as a priority queue.

Finally, there have been proposed several cache-oblivious priority queues which are used in case of large datasets [8]. A simple approach would be to increase the degree of the underlying heap and ensure that all successors of a node in the heap reside in the same cache line.

2.5 Parallelizing Dijkstra's Algorithm

To implement parallel versions of the algorithm, researchers follow two general strategies.

2.5.1 Outer loop parallelization

The first one attempts to relax the sequential nature of Dijkstra by creating more parallelism in the Extract-Min in the outer loop. This leads to alternative algorithms like $\Delta - stepping$ [9] that enable concurrent extraction of multiple nodes from the unvisited set.

2.5.2 Inner loop parallelization

An intuitive choice for parallelizing Dijkstra's algorithm is to exploit parallelism at the inner loop by relaxing all outgoing edges of vertex u from the heap an then its outgoing edges are assigned (e.g. via cyclic assignment) to parallel threads for relaxation. A number of observations can be made concerning this parallelization scheme. First, the speedup is bounded by the average out-degree of the vertices, i.e. the density of the graph. Clearly, if vertices have a small number of neighbors on average, then the parallel segment of the algorithm (lines 10–16) will consume a small fraction of the total execution time, making the sequential part (line 8, Extract-Min) dominant. The second observation concerns the concurrent accesses to the priority queue by the parallel Decrease-Key operations.

The first, and rather naive, approach to enable parallelization of the relaxation phase, is to use a global mutex to lock the entire heap during each Decrease-Key operation. This constitutes a conservative, coarse-grain synchronization scheme that permits only one Decrease-Key operation at a time and obviously limits concurrency. The alternative, more optimistic approach is to allow multiple sequences of node swaps to execute in parallel as long as they access different parts of the heap. More specifically, instead of using one lock for the entire heap, one can utilize separate locks for each parent-child pair of nodes. Whenever a thread executes a Decrease-Key operation and a node swap is required, it must first acquire the appropriate lock that guards this specific pair of nodes. In this way atomicity is guaranteed and the algorithm can be executed safely in parallel.

In [10] Anastopoulos et al. implemented the aforementioned schemes and obtained a picture of their efficiency, as they evaluated graphs with 10K vertices and 100K edges. The performance of the coarse-grained locking scheme is not good since it is a really conservative scheme and cannot expose enough parallelism. Moreover, the fine-grained locking also fails to outperform the serial execution because of the concurrent accesses to the heap. The results of their evaluation are presented in Figure 2.7. Whenever this kind of concurrency occurs, the threads are serialized, thus limiting the total available parallelism.

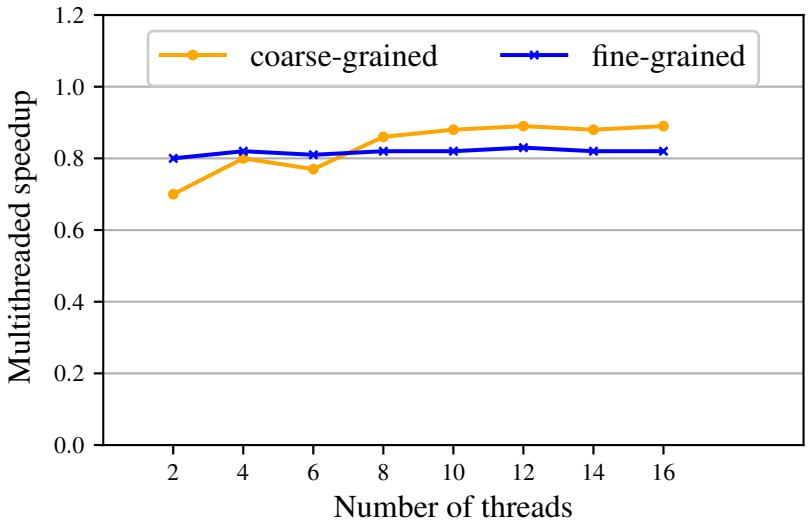


Figure 2.7: Speedups of lock-based parallel versions. The relaxations are performed in parallel and each thread acquires the lock to update the priority queue.

Nikas et al. [11] employed hardware transactional memory and helper threads to extract parallelism. These helper threads offload operations from the main thread in a transparent way by exploiting the following property: the relaxations lead to monotonically decreasing values for the distances of unsettled nodes until each distance reaches its final minimum value. The idea is that parallel threads can serve as helper threads and relax neighbors of nodes belonging to the queued set. Optimistically, the load corresponding to some of these relaxations will be taken off the main thread and accelerate the execution.

Chapter 3

Profiling

3.1 Profiling Dijkstra's algorithm

We profiled the different operations of Dijkstra's Algorithm in a fine-grained way to discover the bottleneck of Dijkstra's algorithm. We used the RDTSCP assembly instruction, that reads the timestamp register, to avoid overhead and errors in benchmarking.

3.1.1 Methodology

RDTSC assembly instruction

Intel and AMD CPUs have a timestamp counter to keep track of every cycle that occurs on the CPU. Starting with the Intel Pentium processor, the devices have included a per-core timestamp register that stores the value of the timestamp counter and that can be accessed by the RDTSC and RDTSCP assembly instructions.

```
// measure cycles of function foo
asm volatile ( "RDTSC\n\t"
      "mov %%edx, %0\n\t"
      "mov %%eax, %1\n\t": "=r" (cycles_high), "=r" (cycles_low));

foo(&variable);

asm volatile ( "RDTSC\n\t"
      "mov %%edx, %0\n\t"
      "mov %%eax, %1\n\t": "=r" (cycles_high1), "=r" (cycles_low1));

start = ( ((uint64_t)cycles_high << 32) | cycles_low );
end = ( ((uint64_t)cycles_high1 << 32) | cycles_low1 );
```

Listing 3.1: Measuring cycles using rdtsc

In listing 3.1, we present the added inline assembly instructions that we use to measure how many clock cycles it takes to call a dummy function. The RDTSC instruction loads the high-order 32 bits of the timestamp register into EDX, and the

low-order 32 bits into EAX. A bitwise OR is performed to reconstruct and store the register value into a local variable. We call the RDTSC assembly instruction to read the timestamp register before calling the foo function. We then call our function and read the timestamp register again (RDTSC) to see how many clock cycles have been elapsed since the first read. The two variables, *Start* and *End* store the timestamp register values at the respective times of the RDTSC calls.

Logically the code above makes sense, but if we try to compile it, we could get segmentation faults or some weird results. This is because we didn't consider a few issues that are related to the "RDTSC" instruction itself:

- **Register Overwriting**

  RDTSC instruction, once called, overwrites the EAX and EDX registers. In the inline assembly that we presented above, we didn't declare any clobbered register. Basically we have to push those register statuses onto the stack before calling RDTSC and popping them afterwards. The practical solution for that is to write the inline assembly as follows:

```
asm volatile ("RDTSC\n\t"
        "mov %%edx, %0\n\t"
        "mov %%eax, %1\n\t": "=r" (cycles_high), "=r" (cycles_low)::
        "%eax", "%edx");
```

  Listing 3.2: Push and pop eax and edx to the stack

- **Out-Of-Order Execution**

  Most modern CPUs support out-of-order execution of the code. The purpose is to optimize the penalties due to the different instruction latencies. Unfortunately this feature does not guarantee that the temporal sequence of the single compiled C instructions will respect the sequence of the instruction themselves as written in the source C file. When we call the RDTSC instruction, we pretend that that instruction will be executed exactly at the beginning and at the end of code being measured (i.e., we don't want to measure compiled code executed outside of the RDTSC calls or executed in between the calls themselves).

  The solution is to call a serializing instruction before calling the RDTSC one. A serializing instruction is an instruction that forces the CPU to complete every preceding instruction of the C code before continuing the program execution. By doing so we guarantee that only the code that is under measurement will be executed in between the RDTSC calls and that no part of that code will be executed outside the calls. A simple choice to avoid out of order execution would be to call CPUID just before both RTDSC calls; this method works but there is a lot of variance (in terms of clock cycles) that is intrinsically associated with the CPUID instruction execution itself. This means that to guarantee serialization of instructions, we lose in terms of measurement resolution when using CPUID.

RDTSCP assembly instruction

The RDTSCP instruction is an assembly instruction that, at the same time, reads the timestamp register and the CPU identifier. The value of the timestamp register is stored into the EDX and EAX registers; the value of the CPU id is stored into the ECX register. What is interesting in this case is the "pseudo" serializing property of RDTSCP. The RDTSCP instruction waits until all previous instructions have been executed before reading the counter. However, subsequent instructions may begin execution before the read operation is performed. This means that this instruction guarantees that everything that is above its call in the source code is executed before the instruction itself is called. It cannot, however, guarantee that the CPU will not execute, before the RDTSCP call, instructions that, in the source code, are placed after the RDTSCP function call itself. If this happens, a contamination caused by instructions in the source code that come after the RDTSCP will occur in the code under measurement.

Improved benchmarking method

```
//start_timer()
asm volatile ("CPUID\n\t"
      "RDTSC\n\t"
      "mov %%edx, %0\n\t"
      "mov %%eax, %1\n\t": "=r" (cycles_high), "=r" (cycles_low)::
      "%rax", "%rbx", "%rcx", "%rdx");

function_to_measure();
//stop_timer()
asm volatile("RDTSCP\n\t"
      "mov %%edx, %0\n\t"
      "mov %%eax, %1\n\t"
      "CPUID\n\t": "=r" (cycles_high1), "=r" (cycles_low1)::
      "%rax", "%rbx", "%rcx", "%rdx");
```

Listing 3.3: Benchmarking method using RDTSCP and CPUID

In the code above, the first CPUID call implements a barrier to avoid out-of order execution of the instructions above and below the RDTSC instruction. Nevertheless, this call does not affect the measurement since it comes before the RDTSC (i.e., before the timestamp register is read). The first RDTSC then reads the timestamp register and the value is stored in memory. Then the code that we want to measure is executed. If the code is a call to a function, it is recommended to declare such function as “inline” so that from an assembly perspective there is no overhead in calling the function itself.

The RDTSCP instruction reads the timestamp register for the second time and guarantees that the execution of all the code we wanted to measure is completed. The two “mov” instructions coming afterwards store the edx and eax registers values into memory. Both instructions are guaranteed to be executed after RDTSC

(i.e., they don't corrupt the measure) since there is a logical dependency between RDTSCP and the register edx and eax (RDTSCP is writing those registers and the CPU is obliged to wait for RDTSCP to be finished before executing the two "mov").

Finally a CPUID call guarantees that a barrier is implemented again so that it is impossible that any instruction coming afterwards is executed before CPUID itself (and logically also before RDTSCP). With this method we avoid to call a CPUID instruction in between the reads of the real-time registers.

Profiling the operations

We used the improved benchmarking method presented above to profile the different operations of Dijkstra's algorithm. We inserted the functions *start_timer*() and *stop_timer*() to measure the cycles that each operation consumes to complete its work. Using the benchmarking method previously mentioned we isolate the execution of each operation from other ones to accurately measure its cycles.

Algorithm 3 : Profiling Dijkstra's algorithm

Require: Graph $G(V, E)$, Source vertex S
Ensure: Predecessors array *previous*
Shortest distance array d

```
1:  for all v ∈ V[G] do
2:      d[v] ← +∞
3:      previous[v] ← undefined
4:  d[s] ← 0 // Initialization
5:  S ← empty set
6:  Q ← s
7:  while Q is not empty do
8:      start_timer()
9:      u ← Extract-Min(Q) // Extract-Min
10:     stop_timer()
11:     S ← S ∪ {u}
12:     for all edge (u, v) outgoing from u do
13:         if v ∉ S then
14:             sum ← d[u] + w(u, v)
15:             if v ∉ Q then
16:                 start_timer()
17:                 Insert(Q,v,sum)
18:                 stop_timer()
19:             else
20:                 start_timer()
21:                 if sum < d[v] then // Update
22:                     start_timer()
23:                     u ← DecreaseKey(Q,v,sum)
24:                     stop_timer()
25:                     d[v] ← d[u] + w(u, v)
26:                     previous[v] ← u
27:                     stop_timer()
```

3.1.2 Profiling results

In Figure 3.1 we present the distribution of the execution times, in graphs with 10M nodes and increasing number of edges. For this experiment, we used a simple binary heap as the priority queue. The Update operation includes lines 18,17,19 of Algorithm 2. It is evident that, the **Update** operation becomes the main part of the execution as graphs become denser and the **Initialization** phase's piece of execution time is minimal compared to Extract-Min.

Both these operations depend on the priority queue that is used in the algorithm. One way to increase the operations' performance is to use priority queues that provide low cost timing complexities and operate fast on large datasets. This way the Decrease-Key operation which is a part of the Update, will be accelerated and performance will be increased.

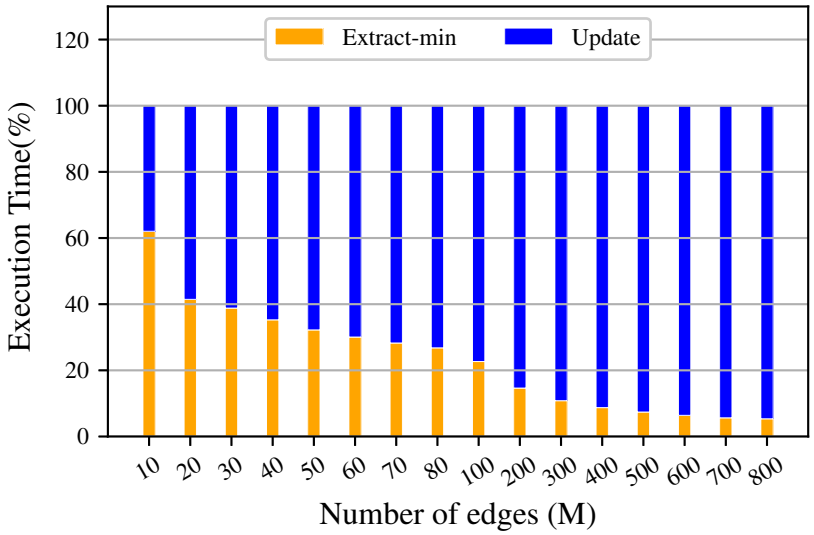


Figure 3.1: Breakdown of execution time percentage. The Update operation dominates in dense graphs.

3.2 Priority Queues

In computer science, a priority queue is an abstract data type which is like a regular queue or stack data structure, but where additionally each element has a "priority" associated with it. In a priority queue, an element with high/low priority is served before an element with low/high priority. If two elements have the same priority, they are served according to their order in the queue.

Many applications require that we process items having keys in order, but not necessarily in full sorted order and not necessarily all at once. Often, we collect a set of items, then process the one with the largest/smallest key, then perhaps collect more items, then process the one with the current largest/smallest key, and so forth. Dijkstra's algorithm is an application following this pattern and that is why we need a priority queue to perform its operations.

A priority queue must at least support the following operations:

- insert: add an element to the queue with an associated priority.
- extract-max/min: remove the element from the queue that has the highest/lowest priority, and return it.

In this section we analyse the operations of three basic priority queues which target different operations of Dijkstra's algorithm. The priority queues that will be examined are the D-ary heap, the Fibonacci heap and the Skip list.

3.2.1 D-ary heap

D-ary heap is a complete d-ary tree filled in left to right manner that every parent node has a higher (or equal value) than all of its descendands [12]. Heaps respecting this ordering are called max-heaps, because the node with the maximal value is on the top of the tree. Analogously min-heap is a heap, in which every parent node has a lower (or equal) value than all of its descendants.Thanks to these properties, d-ary heap behaves as a priority queue. Special case of d-ary heap ($d = 2$) is the binary heap.

D-ary heap allows faster Decrease-Key operation as compared to a binary heap: $\mathcal{O}(log_2 n)$ for binary heap vs $\mathcal{O}(log_k n)$ for d-ary heap. Nevertheless, it causes the complexity of Extract-Min operation to increase to $\mathcal{O}(klog_k n)$ as compared to the complexity of $\mathcal{O}(log_2 n)$ when using a binary heap. This allows D-ary heap to be more efficient in algorithms where decrease priority operations are more frequent than Extract-Min operation. Dijkstra's algorithm is such a case. When operating on a graph with m edges and n vertices, Dijkstra's algorithm for shortest paths uses a min-heap in which there are n Extract-Min operations and as many as m Decrease-Key priority operations. By using a d-ary heap with $d = m/n$, the total times for these two types of operations may be balanced against each other, leading to a total time of $\mathcal{O}(mlog_{m/n} n)$ for the algorithm, an improvement over the $\mathcal{O}(mlogn)$ running time of binary heap versions of these algorithms whenever the number of edges is significantly larger than the number of vertices.

D-ary heap has better memory cache behaviour than a binary heap which allows them to run more quickly in practice, although it has a larger worst case running time of both ExtractMin() operation (being $\mathcal{O}(k \cdot log_k n)$).

Implementation D-ary heap is usually implemented using an array. For every node of the heap placed at index n ,its parent is placed at index $(n-1)/d$ and its descendants are placed at indexes $d \cdot k + 1$, ..., $d \cdot k + d$. It is also useful choosing heap's arity is a power of 2, since we can easily replace multiplications used in the tree traversal by binary shifts.

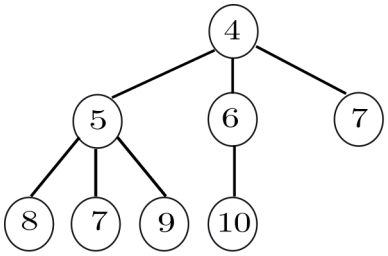


Figure 3.2: 3-ary heap example and its array implementation

Operations

- **Extract-Min**

  The procedure for deleting the root from the heap and restoring the properties is called heapify-down.

  1. Replace the root of the heap with the last element on the last level.
  2. Compare the new root with its children; if they are in the correct order, stop.
  3. If not, swap the element with one of its children and return to the previous step. (Swap with its smaller child in a min-heap and its larger child in a max-heap.)

  Complexity: $\mathcal{O}(d \cdot log_d n)$

- **Decrease-Key**

  Decreasing the key of a node is done by lowering the key's value and then performing a heapify-up operation by following this algorithm:

  1. Decrease the key of the selected node.
  2. Compare this node with its parent; if they are in the correct order, stop.
  3. If not, swap the element with its parent and return to the previous step

  Complexity: $\mathcal{O}(log_d n)$

- **Insert**

  To add an element to a heap we must perform a heapify-up operation by following this algorithm:

  1. Add the element to the bottom level of the heap.
  2. Compare the added element with its parent; if they are in the correct order, stop.
  3. If not, swap the element with its parent and return to the previous step

  Complexity: $\mathcal{O}(log_d n)$

3.2.2 Fibonacci heap

A Fibonacci heap is a collection of trees satisfying the minimum-heap property [3]. The key of a child is always greater than or equal to the key of the parent. This implies that the minimum key is always at the root of one of the trees. The trees do not have a prescribed shape and in the extreme case the heap can have every element in a separate tree. This flexibility allows some operations to be executed in a lazy manner, postponing the work for later operations.

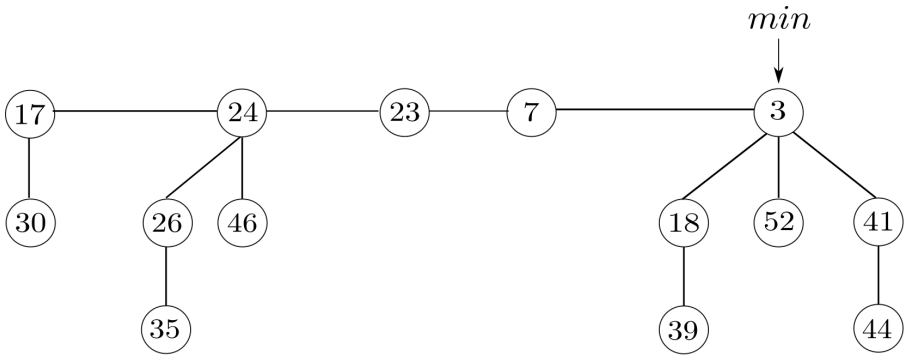


Figure 3.3: Fibonacci heap example

Operations

- **Extract-Min**

  Extract-Min operates in three phases. Firstly, we choose the root containing the minimum element and remove it. Its children will become roots of new trees. If the number of children was d, it takes time $\mathcal{O}(d)$ to process all new roots and the potential increases by $d - 1$. Therefore, the amortized running time of this phase is $\mathcal{O}(d) = \mathcal{O}(logn)$.

  However to complete the Extract-Min operation, we need to update the pointer to the root with the minimum key. Unfortunately there may be up to n roots we need to check. In the second phase decrease the number of roots by successively linking together roots of the same degree. When two roots u and v have the same degree, we make one of them a child of the other so that the one with the smaller key remains the root. Its degree will increase by one. We repeat this until every root has a different degree. To find trees of the same degree efficiently we use an array of length $\mathcal{O}(logn)$ in which we keep a pointer to one root of each degree. When a second root of the same degree is found, they are linked and the array is updated. The actual running time is $\mathcal{O}(logn + m)$ where m is the number of roots at the beginning of the second phase. At the end we will have at most $\mathcal{O}(logn)$ roots (because each one has a different degree).

  Complexity: $\mathcal{O}(logn)$

- **Decrease-Key**

Operation decrease key will find the node, decrease the key and if the heap property becomes violated (the new key is smaller than the key of the parent), the node is cut from its parent. If the parent is not a root, it is marked. If it has been marked already, it is cut as well and its parent is marked. We continue upwards until we reach either the root or an unmarked node. Now we set the minimum pointer to the decreased value if it is the new minimum. In the process we create some number, say k, of new trees. Each of these new trees except possibly the first one was marked originally but as a root it will become unmarked. One node can become marked. Therefore, the number of marked nodes changes by $-(k-1)+1 = -k+2$. Combining these 2 changes, the potential changes by $2(-k+2)+k = -k+4$. The actual time to perform the cutting was O(k), therefore (again with a sufficiently large choice of c) the amortized running time is constant.

Complexity: $\mathcal{O}(k)$

- **Insert**

  The insert operation works by creating a new heap with one element and then performing a merge. This takes constant time, and the potential increases by one, because the number of trees is increased. Thus, the amortized cost is still constant.

  Complexity: $\mathcal{O}(1)$

- **Merge**

  Merge is implemented simply by concatenating the lists of tree roots of the two heaps. This can be done in constant time and the potential does not change, leading again to constant amortized time.

  Complexity: $\mathcal{O}(1)$

3.2.3 Skip list

A Skip list is a data structure that allows fast search within an ordered sequence of elements [13]. Fast search is made possible by maintaining a linked hierarchy of subsequences, with each successive subsequence skipping over fewer elements than the previous one. This structure is built in layers. The bottom layer is an ordinary ordered linked list. Each higher layer acts as an "express lane" for the lists below, where an element in layer i appears in layer i+1 with some fixed probability p (two commonly used values for p are 1/2 or 1/4). On average, each element appears in $1/(1-p)$ lists, and the tallest element (usually a special head element at the front of the skip list) in all the lists. The skip list contains $log_{1/p}n$ lists.

A lookup, called SkipSearch(k), of a target element begins at the head element in the top list, and proceeds horizontally until the current element is greater than or equal to the target. If the current element is equal to the target, it has been found. If the current element is greater than the target, or the search reaches the end of the linked list, the procedure is repeated after returning to the previous element and dropping down vertically to the next lower list. The expected number of steps

in each linked list is at most $1/p$, which can be counted by tracing the search path backwards from the target until reaching an element that appears in the next higher list or reaching the beginning of the current list. Therefore, the total expected cost of a search is $(log_{1/p}n)/p$, which is $\mathcal{O}(logn)$, when p is a constant. By choosing different values of p, it is possible to trade search costs against storage costs.

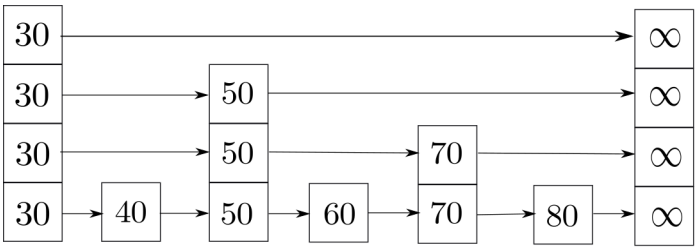


Figure 3.4: Skip list example

Operations

- **Insert**

  The insertion algorithm for skip lists uses randomization to decide the height of the tower for the new entry. We begin the insertion of a new entry (k, v) by performing a SkipSearch(k) operation as described in the previous paragraph. This gives us the position p of the bottom-level entry with the largest key less than or equal to k. We then insert (k, v) immediately after position p. After inserting the new entry at the bottom level, we "flip" a coin. If the flip comes up tails, then we stop here. In the case of heads , we backtrack to the previous level and insert (k,v) in this level at the appropriate position. We again flip a coin; if it comes up heads, we go to the next higher level and repeat. Thus, we continue to insert the new entry (k,v) in lists until we finally get a flip that comes up tails.

  Complexity: $\mathcal{O}(logn)$

- **Delete**

  The removal algorithm for a Skip list is quite simple. That is, to perform a remove(k) operation, we begin by executing SkipSearch(k). If the position p stores an entry with key different from k, **null** is returned. Otherwise, we remove p and all the positions above p, which are easily accessed by using above operations to climb up the tower of this entry.

  Complexity: $\mathcal{O}(logn)$

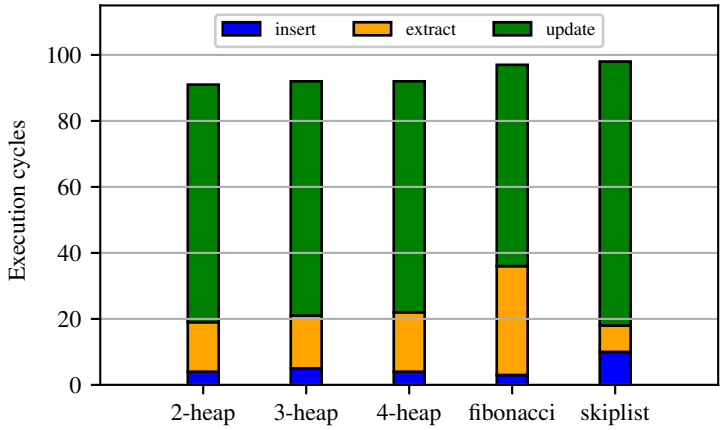
- **Decrease-Key**
  In order to perform a Decrease-Key operation we begin by deleting the element and then by reinserting the element with the its new priority.

  Complexity: $\mathcal{O}(logn)$

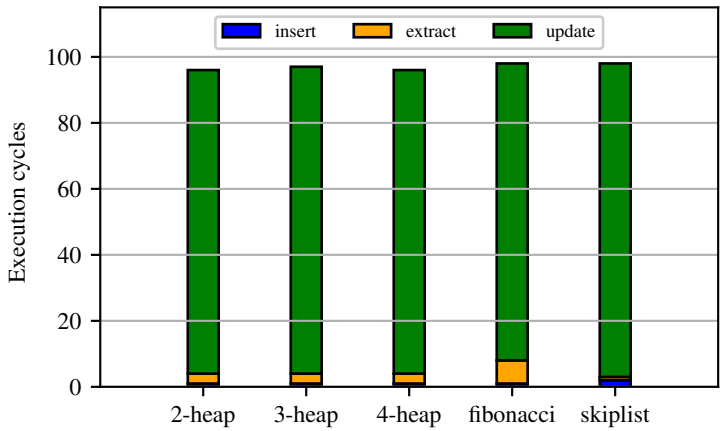
- **Extract-Min**

  Extracting the minimum element from the skiplist is performed by removing the tower of the first element which holds the minimum key.

  Complexity: $\mathcal{O}(1)$



(a) 10Mx50M graph



(b) 10Mx500M graph

Figure 3.5: Breakdown of execution times of each operation for each priority queue.

In Figure 3.5 we present the distribution of the execution times for each operation and each priority queue, in graphs with 10M nodes 50M & 500M edges. Regarding the 2,3,4 - ary heaps we witness similar behaviour in every graph. In Fibonacci heap's case the Extract-Min operation is more costly than every other priority queue, which is expected. As far as the Skip list is concerned, although the Extract-Min's cost is smaller compared to the other ones, Insert and Update (which consists of a Decrease-Key and a relaxation) impose noticeable overhead.

In the dense graph consisting of 500M edges, we observe that the Update operation reserves the biggest part of the execution time and the impact of each priority queue is minor to performance. This occurs even in Fibonacci's heap case whose Decrease-Key's timing complexity is constant. Due to this conclusion, we will include the Decrease-Key operation in our profiling and attempt to discover the true bottleneck of the algorithm which is hidden under the Update operation.

3.3 Extended Profiling

In Figure 3.6 we present the distribution of the execution times, in graphs with 10M nodes and increasing number of edges. The Update operation includes lines 18,19 of Alg. 2 excluding line 17, i.e. the Decrease-Key operation. It is evident that in this case, the **Update** operation without the Decrease-Key, becomes the main part of the execution as graphs become denser and the **Insert** operation's piece of execution time is minimal compared to the other ones.

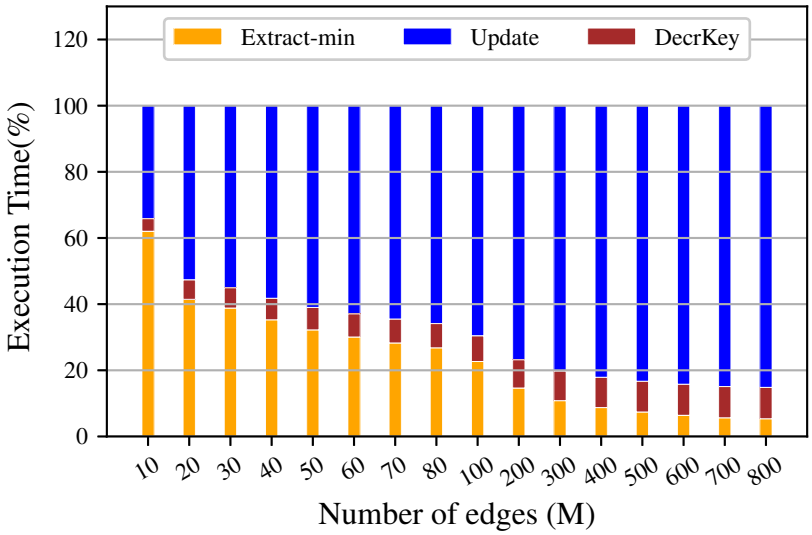


Figure 3.6: Breakdown of execution time percentage. The Update operation dominates in dense graphs.

We observe that the **Update** operation dominates as graphs become denser.

The Update operation consists of four read operations that occur for every edge of the extracted vertex: (i) the *distance* array at indexes v and (ii) u, (iii) the cost of the edge $w(u, v)$, and (iv) the *previous* array at index v. Reading the *distance* and the *previous* array, whose size is equal to the number of vertices (10M), at a random index v, generates irregular memory accesses resulting in cache misses. In dense graphs, this phenomenon is more intense since the number of edges per vertex is large. Therefore, more relaxations are needed and random memory accesses are increased.

On the other hand, the percentage of the **Extract-min** operation becomes less important as graphs become denser, while that of the **Decrease-Key** operation remains constant and significantly smaller. Both operations consist of an upward traversal of the binary heap that involves a small number of hops, since the height of the heap is logarithmic to the number of vertices. In addition, the Decrease-Key operation is performed only if an edge relaxation is necessary. Finally, the percentage of the **Initialization** phase is negligible.

Chapter 4

Prefetching

Memory latency has become increasingly important as the gap between processors speeds and memory speeds grows [14]. Many methods have been proposed to overcome this disparity, such as caching [15], prefetching [16], multi-threading [17], and out of order execution. These techniques fall broadly into two categories: those that reduce latency, and those that tolerate latency. Techniques for reducing latency include caching data and making the best use of those caches through locality optimizations. Techniques for tolerating latency include buffering, pipelining, prefetching, and multithreading. In this chapter, we will analyse how software prefetching can be applied to Dijkstra's algorithm and present two schemes that employ software prefetching to deal with irregular memory accesses.

4.1 Background on Prefetching

Prefetching is a mechanism to speculatively move data to closer to the CPU in anticipation of future use for this block. Prefetching can be performed in hardware, software, or a combination of both. Software prefetching is directly controlled by the program or the compiler and therefore it is their responsibility to issue proper prefetch requests at the right time. Hardware prefetching is the alternative case, where a hardware controller generates prefetch requests based on information it obtains at run-time (e.g., memory references and cache miss addresses). Generally, software prefetchers use compile-time and profiling information while hardware prefetchers use run-time information. Both have their advantages and both can be very effective [16]. Prefetching reduces the cache miss rate because it eliminates the demand fetching of cache lines in the cache hierarchy [18]. It is also called a latency hiding technique because it attempts to hide the long-latency transfers from lower levels to higher levels of the memory hierarchy behind periods of time during which the processor is busy executing other instructions.

A central aspect of all cache prefetching techniques is their ability to detect and predict particular memory reference patterns. Prefetching must be done accurately and early enough to reduce/eliminate both miss rate and miss latency. There are four basic questions which need to be answered:

- **What addresses to prefetch:** Prefetching useless data wastes resources and consumes memory bandwidth. Prediction can be based on past access patterns or by using the compilers' knowledge of data structures. Nevertheless, a prefetching algorithm determines what to prefetch.

- **When to initiate a prefetch request:** If prefetching is done too early then prefetched data might not be used before they are evicted. On the other hand if prefetching is done too late, it might not hide the whole memory latency. This is defined by the timeliness of the prefetcher. Prefetcher can be made more timely by making it more aggressive (try to stay far ahead of the processors access stream (hardware) or moving the prefetch instructions earlier in the code (software) [16].

- **Where to place the prefetched data:** Prefetched data can be placed inside the cache or in a separate prefetch buffer. If it is placed inside the cache, a simple design is required, however cache pollution is an issue. If a separate prefetch buffer is designed, the data in the caches will be protected from prefetches and thus cache pollution is avoided. However, this design is more complex and costly. It is important to research on how to place the prefetch buffer, when to access it (parallel vs. serial with cache), when to move the data from the prefetch buffer to the caches, and how to size the prefetch buffer.

- **How to prefetch:** Prefetching can be performed in hardware, software, or as a cooperation of both. Also, it can rely on statically profiling the application and analyzing its patterns, or it can be performed dynamically.

4.1.1 Hardware Prefetching

Hardware prefetching is typically performed by employing a dedicated hardware mechanism in the processor which monitors the stream of instructions or data being requested by the executing program, predicts the next elements that the program might need based on this stream, and prefetches them into the processor's cache.

Hardware monitors the memory access pattern of the running program and triesto predict which data are going to be accessed by the program and prefetches them. Then it memorizes the patterns/strides of the application and in order to generate prefetch addresses in an automated way. There are few different variants of how this can be done.

Sequential Prefetching

Sequential prefetching can take the advantage of spatial locality by prefetching consecutive smaller cache blocks, without introducing some of the problems that exist with large blocks.

Next-line prefetching (one-block look-ahead) is the simplest form of hardware prefetching [19]. In this scheme, when a cache line is fetched, a prefetch for the next sequential line is also initiated. One way to do this is called the prefetch-on-miss algorithm [20], in which the prefetch of block b+1 is initiated whenever an access for block b results in a cache miss. If b+1 is already cached, no memory access is initiated. Next-N-line prefetch schemes extend this basic concept by prefetching the next N sequential lines following the one currently being fetched by the processor [21]. The benefits of prefetching the next N-lines include, increase the timeliness of the prefetches, and the ability to cover short non-sequential transfers (where the target falls within the N-line "prefetch-ahead" distance). In adddition, this method

is simple to implement and there is no need for sophisticated pattern detection. This scheme works well for sequential/streaming access patterns.

A **stream prefetcher** looks for streams where a sequence of consecutive cache lines are accessed by the program. When such a stream is found the processor starts prefetching the cache lines ahead of the program's accesses [22]. Again, this method is simple to implement and it can be designed as an extension to the basic next-line prefetchers.

A **stride prefetcher** looks for instructions that make accesses with regular strides, that do not necessarily have to be to consecutive cache lines. When such an instruction is detected the prefetcher tries to prefetch the cache lines ahead of the access of the processor [23]. It should be noted that stream prefetching can be considered as a special case of stride prefetching (with stride of 1). Also, strides $>$ 1 does not apply to instruction caches and is only useful for data.

Non-sequential Prefetching

In many applications cache misses occur because of transitions to distant lines, especially when the application is composed of small functions or there are frequent changes in control flow. There are some kinds of prefetchers specifically targeted at non sequential misses. Target-line prefetching, for example, tries to address next-line prefetchings inability to correctly prefetch non sequential cache lines. It uses a target prefetch table maintained in hardware to supply the address of the next line to prefetch when the current line is accessed [24].

Moreover, hybrid schemes can be built by combining different prefetching techniques. For example, in a combination of next line and target prefetching both a target line and next line can be prefetched, offering double protection against a cache line miss [24]. Finally, a combination of hardware and software prefetching mechanisms can be implemented to benefit from both of their capabilities.

Indirect Prefetching

Sequential hardware prefetching does not work for irregular access patterns, as seen in linked data structures, and also in indirect memory accesses, where the addresses loaded are based on indices stored in arrays ; $A[B[i]]$. The Indirect Memory prefetcher is an example that deals with this kind of memory accesses [25]. It combines an irregular pattern detector used to track irregular patterns and a conventional stream buffer system to track indices.

4.1.2 Software Prefetching

Software prefetching provides facilities for the programmers/compilers to explicitly issue prefetch requests whenever they want. This can be done either by including a fetch instruction in a microprocessors instruction set, or through some registers configurable and programmable by software. Software prefetching can be either done directly by the programmer (e.g. in the C code), or by the compiler in the optimization phase, and on the final assembly code.

Prefetch Instructions

Prefetch instructions can be manually inserted in the code and prefetch data in the caches before they are needed by the application. On the other hand, using explicit fetch instructions may also bring some performance penalties since the addition of the prefetch instructions result in increased code size. For this reason, it is important to optimize the location and size of the prefetch commands to make sure the optimal performance is achieved [21].

Irregular access patterns

Software prefetching is alo a tempting proposition for irregular data access patterns. The idea is that the programmer uses data structure and algorithmic knowledge to insert instructions into the program to bring the required data early, thus improving performance by overlapping memory accesses. In [26] prefetching code is autogenerated by the compiler instead of manually inserting prefetch instructions. They developed a novel algorithm to automate the insertion of software prefetches for indirect memory accesses into programs. Within the compiler, they find loads that reference loop induction variables and use a depth-first search algorithm to identify the set of instructions which need to be duplicated to load in data for future requests.

4.2 System Prefetchers

In this thesis we are using an Intel Broadwell-EP , whose characteristics are described in Chapter 5 . There are 2 prefetchers associated with L1-data cache (also known as DCU) and 2 prefetchers associated with L2 cache. There is a Model Specific Register (MSR) on every core with address of 0x1A4 that can be used to control these 4 prefetchers. Bits 0-3 in this register can be used to either enable or disable these prefetchers. In the table below, we present these prefetchers.

Prefetcher	Description
L2 hardware prefetcher	Fetches additional lines of code or data into the L2 cache
L2 adjacent cache line prefetcher	Fetches the cache line that comprises a cache line pair (128 bytes)
DCU prefetcher	Fetches the next cache line into L1-D cache
DCU IP prefetcher	Uses sequential load history (based on Instruction Pointer of previous loads) to determine whether to prefetch additional lines

4.2.1 Impact on performance

The hardware prefetchers of Intel's Broadwell-EP fail to prefetch the correct piece of data in irregular cases, such as Dijkstra's algorithm. Intel's Broadwell-EP and many of the modern systems provide no hardware prefetchers to deal with irregular memory accesses. In Fig. 4.1 we present the achieved speedup when hardware prefetchers are enabled compared to when they are disabled. Given graphs in range of 40M-500M, we notice a drop in performance, at about 5% on average. We observe that prefetchers provide no performance boost to this irregular memory accesses algorithm. In this thesis, we will apply software prefetching to accelerate such memory accesses by exploiting the characteristics of the algorithm.

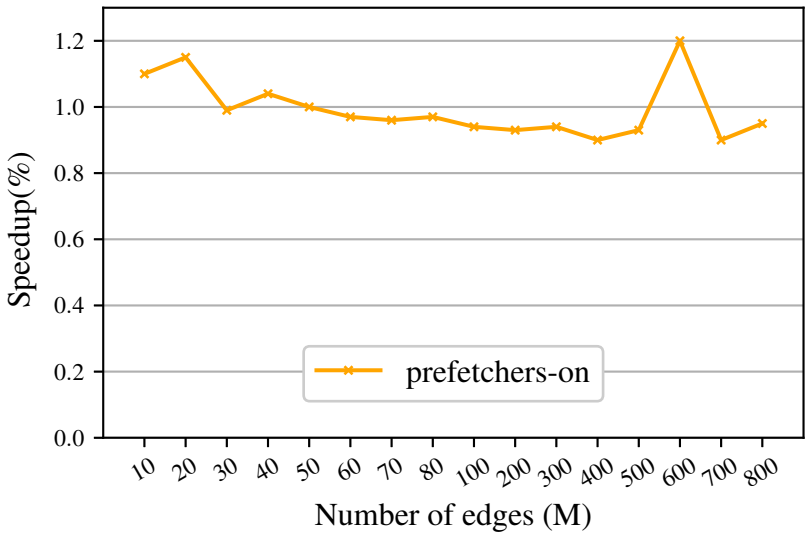


Figure 4.1: The effect of enabling hardware prefetchers on Intel Broadwell-EP.

4.3 Prefetching for Dijkstra's Algorithm

Our goal is to hide the latency of cache misses that originate from the Update operation through software prefetching. The idea is to predict irregular memory accesses, so that the soon-to-be-used data is loaded from the main memory to the cache hierarchy before it is accessed by the application [26, 27].

4.3.1 Speculative selection of the Next-Extracted vertex

Prefetching the correct chunk of data in the Update operation relies on the prediction of the forthcoming extracted minimum-key vertex u. We observe that the information stored in the priority queue provides a reliable insight about the next extracted vertex. Hence, the piece of data we target for prefetching are the neighbors of the minimum element of the priority queue, once the Extract-min operation of the current phase has finished but before the Update Operation is performed. Our key observation is that there is a high probability this vertex will be the one extracted next. We quantify this potential for correct prefetching by counting a prediction as correct when the i_{th} top vertex of the priority queue is extracted at least after i rounds. Fig. 4.5 presents the ratio of correct predictions for $i = \{1, 2, 3, 4\}$.

We observe that the prediction for $i = 1$ is almost perfect, and consequently the probability that the prefetched data will be actually used by the application is much higher.

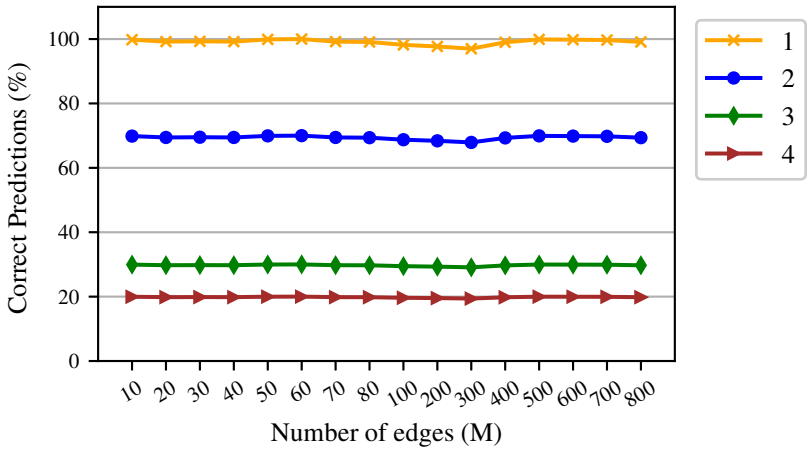


Figure 4.2: Correct Predictions (%) of the forthcoming Extracted minimum vertex. The prediction is 99.1% accurate on average.

4.3.2 Optimal prefetching

As mentioned in Chapter 2, the binary heap's root is the minimum key-vertex and will be the next extracted node of the algorithm. A first approach would be to prefetch the neighbors of the extracted vertex just before performing the appropriate relaxations. This way, during the Update operation, the data which will be required are going to be transferred in the cache hierarchy on time. In Figure 4.3, we present the reduction of execution time of the Update operation. In Figure 4.4, we show the ratio at which cache misses of the Update operation have been reduced. We witness that the Update's operation execution time and the cache misses have noticeably been reduced - up to 75% on average. The reduced execution time of the Update operation is nearly optimal since we prefetch the correct piece of data, at the correct moment. In Algorithm 4 we present the code for this technique.

Algorithm 4 : Optimal prefetching

```
while Q is not empty do
    min ← Extract-Min(Q)
    for all neighbors of min do
        prefetch neighbors' data
    for all edge (u, v) outgoing from u do
        relaxation phase
```

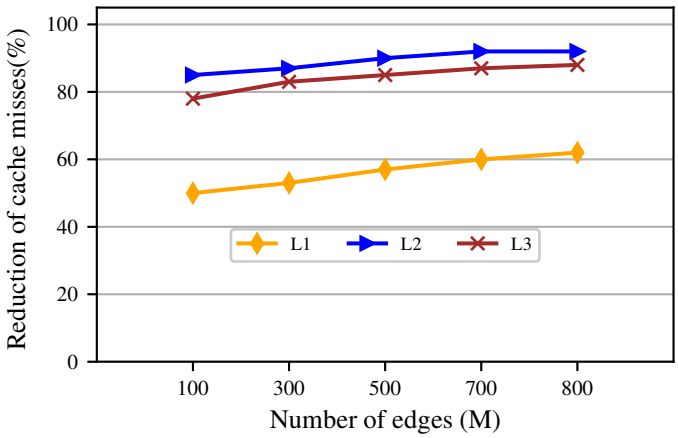


Figure 4.3: Normalized execution time of optimal prefetching

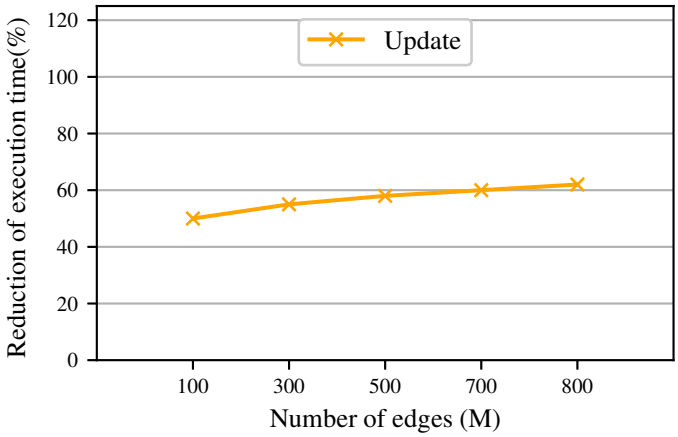


Figure 4.4: Normalized Update's cache misses of optimal prefetching

4.4 Prefetching-Helper-Thread scheme

To begin with, we implemented a simple Prefetching-Helper-Thread (PHT) scheme similar to [28].

Helper Threading Mechanism

Helper threading is an optimization technique used in non traditional parallelism to accelerate a program and provide performance speedups. Helper threads are "assist" threads that perform certain critical computations on behalf of a main thread in order to help the main ("master") thread and reduce its tasks. Typically, this optimization has been exploited either to prefetch future data accesses or to precompute the outcome of blocks of code that would otherwise be executed by the main thread. Finally, to adapt to the dynamic behavior, helper threads need to be activated and synchronized frequently. Thus, a low overhead thread synchronization mechanism is needed.

Methodology

The scheme we propose consists of $n + 1$ threads. The main thread constantly executes the algorithm and n helper threads work in parallel with the main one

and aggresively prefetch data to the nearest shared cache memory, helping it avoid misses from upcoming irregular memory accesses.

More specifically, in our implementation the main thread initializes all the flags needed for synchronization of both threads and the data structures Q, d and *previous*. During each round of the algorithm the main thread is responsible for the execution of the algorithm while the helper threads prefetch the data that will be required during the next round. The main thread extracts the minimum vertex while the helper threads spin-wait to start prefetching. After the Extract-Min operation the main thread signals the helpers to read the n top minimum key-vertices from the priority queue and start prefetching their neighbors. Then, the main thread begins the Update operation whose completion interrupts the prefetching threads. This process continues while Q is not empty.

4.4.1 Coordination

Coordinating the threads is achieved by using two synchronization flags. One is needed to signal the helper threads start prefetching and the second one to interrupt them. In Algorithms 5 and 6 we present the code for the case of $n = 2$ in detail.

In Figure 4.2 we present the speedup achieved by the PHT when $n + 1$ helper threads are spawned. The n helper threads prefetch the n top minimum vertices that are stored in the priority queue. It is evident that in case of $n = 1$ we achieve the highest speedups. In additions, there are cases, such as $n = 4$, where we notice slowdowns. When the helper thread constantly prefetches data that will be needed in the far future, the cache hierarchy suffers from pollution and in some cases important blocks may be evicted. Based on this, $n = 1$ will always be selected as the tentative next extracted vertex.

The PHT scheme suffers from a major drawback: we are not able to choose where to prefetch data. This happens since the helper thread prefetches data at the cache that is shared with the main thread. Thus, the PHT scheme depends on the system's architecture and lacks flexibility. Based on this motivation, we came up with the Prefetch-Process-Thread-Alternation scheme which is based on the PHT but overcomes this disadvantage.

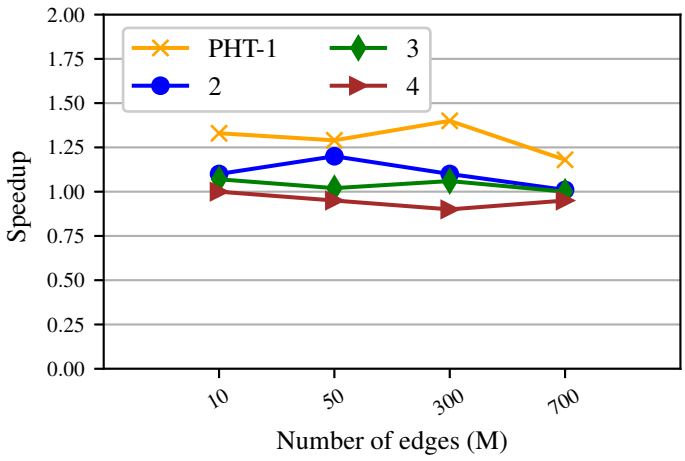


Figure 4.5: Speedup of PHT when $n \leq 4$ helper threads are spawned

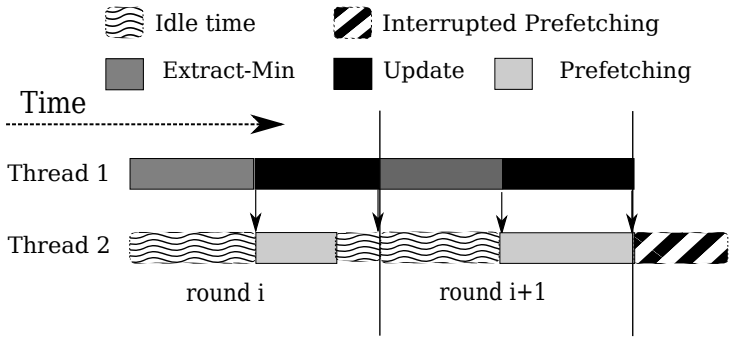


Figure 4.6: Execution pattern of PHT scheme

Algorithm 5 : Main Thread code for PHT scheme

```
Initialize Q, d, previous
prefetch ← 0
interrupt ← 0
while Q is not empty do
  u ← Extract-Min(Q)
  prefetch ← 1
  for all edge (u, v) outgoing from u do
    relaxation phase
  interrupt ← 1
```

Algorithm 6 : Helper Thread code for PHT scheme

```
while Q is not empty do
  while(prefetch=0);
  prefetch ← 0
  min ← ReadMin(Q)
  for all neighbors of min and while interrupt=0 do
    prefetch neighbors' data
  while(interrupt=0);
  interrupt ← 0
```

4.5 Prefetch-Process-Thread-Alternation

In this scheme we present a more sophisticated scheme called Prefetch-Process-Thread-Alternation (PPTA). It isa two-thread scheme that employs software prefetching. However, this technique derives from the idea of moving computation near data. In fact, we try to hide cache misses using two different phases for each thread. Compared to the original algorithm, we expect a significant reduction of cache misses, created by the Update operation, that results in improved performance.

4.5.1 Methodology

In PPTA, one of the two threads starts by initializing all the flags needed for synchronization as well as the data structures Q, d, and *previous*. Our scheme consists of two phases in each round of the algorithm: (i) the prefetch phase and (ii) the process phase. One of the threads is responsible for executing the algorithm (process phase), while the other thread prefetches the data that will be required during the next round (prefetch phase). More specifically, the process-thread, i.e., the thread that is currently in the process phase, extracts the minimum vertex while the prefetch-thread, i.e., the thread that is currently in the prefetch phase, is waiting for the prefetch signal. Once the Extract-Min operation finishes, the process-thread notifies the prefetch-thread to read the new minimum key of the priority queue and start prefetching the neighbors of the speculatively selected minimum key-vertex. Then, the process-thread begins the Update operation and, upon completion, signals the end of the round and informs the prefetch-thread to stop prefetching. As shown in Fig. 4.7, at the start of the next round the two threads change roles; hence, the new process-thread finds in its L1 cache the data that were just prefetched in the previous round.

This execution model continues alternately until Q is empty and all distances have settled. There are two cases we need to examine regarding the prefetch phase. They are also illustrated in Fig. 4.7. In rounds i and $i + 1$, the prefetch-thread has completed prefetching before the process-thread completes the process phase. In contrast, in round $i + 2$, thread 2, i.e., the current prefetch-thread, is notified to alternate to the process-phase before prefetching the whole set of data. This notification is necessary because if thread 2 waits for all data to be prefetched, then the upcoming process phase will be delayed and therefore the application's performance will decrease.

The code of PPTA for both threads is presented in Alg. 2 and Alg. 3.

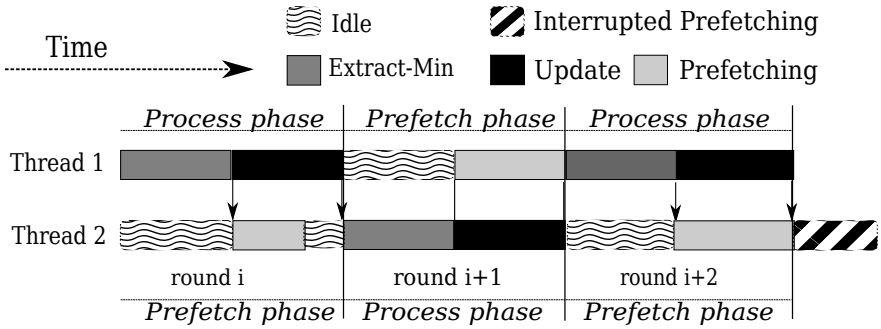


Figure 4.7: Execution pattern of the PPTA scheme

4.5.2 Coordination

To coordinate the execution of the two threads, we use four synchronization flags. Two flags are needed to signal the beginning of each thread's prefetch phase and two more to notify the completion of the relaxation phase and the beginning of the Extract-Min operation.

Algorithm 7 : Thread 1 code for PPTA scheme

```
Initialize Q, d, previous
prefetch_1 ← 1 extract_1 ← 1
prefetch_2 ← 0 extract_2 ← 0
while Q is not empty do
  while(prefetch_1=0);
  prefetch_1 ← 0
  min ← ReadMin(Q)
  for all neighbors of min and while extract_1=0 do
    prefetch neighbors' data
  while(extract_1=0);
  extract_1 ← 0
  u ← Extract-Min(Q)
  prefetch_2 ← 1
  for all edge (u, v) outgoing from u do
    relaxation phase
  extract_2 ← 1
```

Algorithm 8 : Thread 2 code for the PPTA scheme

```
while Q is not empty do
  while(prefetch_2=0);
  prefetch_2 ← 0
  min ← ReadMin(Q)
  for all neighbors of min and while extract_2=0 do
    prefetch neighbors' data
  while(extract_2=0);
  extract_2 ← 0
  u ← Extract-Min(Q)
  prefetch_1 ← 1
  for all edge (u, v) outgoing from u do
    relaxation phase
  extract_1 ← 1
```

Chapter 5

Evaluation

5.1 Experimental Evaluation

5.1.1 Experimental setup

For our experiments we used two systems, equipped with AMD's Opteron and Intel's Broadwell-EP processors. The characteristics of the systems are shown in Table I and their cache hierarchies are presented in Fig. 5.1. We used the following configurations: (i) in the AMD system, the threads are pinned to separate cores that share the L2 cache, (ii) in the Intel system, the threads are either pinned to separate cores that share the L3 cache or (iii) to the same core that share the L1 cache when simultaneous multithreading (SMT) is employed.

We have implemented PPTA and PHT in C++. The coordination flags used in both schemes are implemented with atomic operations and the acquire/release memory order. All types of cycles and cache misses have been measured using specific model specific registers for each system. We disabled the hardware prefetchers based on the results in Chapter 4. Finally, to experiment on a variety of graphs, we have constructed graphs with 10M vertices and varying number of edges from the *Random*, *R-MAT*, and *SSCA* families, using GTgraph [29].

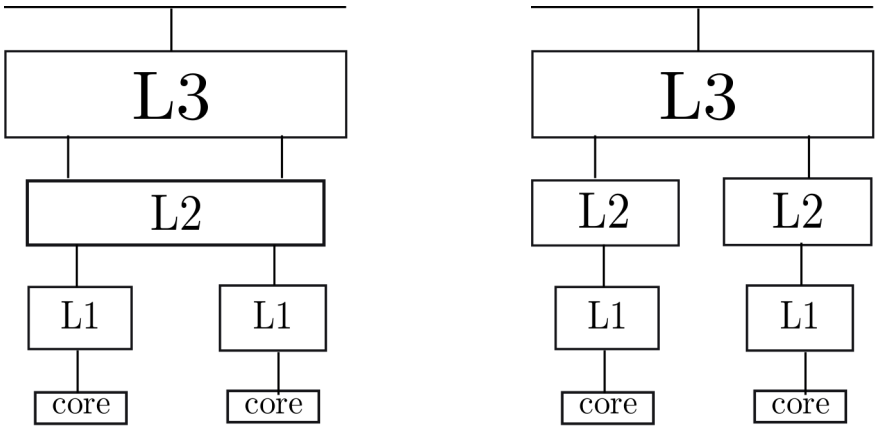
5.1.2 Acquire & Release Memory Order

- **acquire:** A load operation with this memory order performs the acquire operation on the affected memory location: no reads or writes in the current thread can be reordered before this load. All writes in other threads that release the same atomic variable are visible in the current thread.

- **release:** A store operation with this memory order performs the release operation: no reads or writes in the current thread can be reordered after this store. All writes in the current thread are visible in other threads that acquire the same atomic variable and writes that carry a dependency into the atomic variable become visible in other threads that consume the same atomic.

Table 5.1: Systems' Configurations

Name	**AMD Opteron**	**Intel Broadwell-EP**
#Cores	4x8	2x22
#Threads	32	88 (SMT)
Core clock	2.4 GHz	2.2 GHz
L1(Data)	16 KB, 4-way, 64B block size	32 KB, 8-way, 64B block size
L2	256 KB, 8-way, 64B block size (**shared per 2 cores**)	2 MB, 16-way, 64B block size (**private**)
L3	16 MB, 128-way, 64B block size (shared per NUMA node)	56 MB, 20-way, 64B block size (shared per die)
Memory	250 GB	64 GB
OS	Debian 8.8	Debian 8.3
Linux Kernel	3.2.0	4.7.0
GCC	4.9.2 with -O3 optimization	4.9.2 with -O3 optimization



(a) Opteron (b) Broadwell

Figure 5.1: Cache hierarchies of both platforms. Opterons' neighbor nodes share the L2 cache and Broadwells' the L3.

5.1.3 Reference graphs

In order to experiment on graphs with variety of density and structure, we used the GTgraph generator [29]. We constructed graphs with 10M vertices and varying number of edges from the *Random*, *R-MAT* and *SSCA families*.

- **Random:** Their m edges are constructed choosing a random pair among n vertices.
- **R-MAT:** Constructed using the Recursive Matrix (R-MAT) graph model.
- **SSCA:** Used in the DARPA HPCS SSCA graph analysis benchmark.

5.1.4 Performance Results

We evaluate the proposed PPTA approach and the simple PHT scheme. Figures 5.2,5.3 present speedup, sum of data cache misses, and the ratio of prefetched data per chunk of relaxations, on the AMD and Intel systems. The results are normalized to the serial execution. The cache misses results refer only to the process phase of both threads in PPTA or to the main thread in PHT. Note that the AMD system lacks support for measuring the L3 data misses. We focus our discussion on the *Random* graph family, as the other families exhibit similar behavior.

Comparison with PHT. PHT improves the performance of the Dijkstra's algorithm by up to 1.75×. However, the improvement of PHT decreases as graphs become denser. Compared to PPTA, we observe that the performance of PHT is better only for sparse graphs. This happens because in PPTA the process-phase thread finds its data prefetched in the L1 cache, while in PHT the main thread finds the prefetched data in the L2 (AMD) or L3 (Intel) cache.

PPTA on dense graphs. We observe that PPTA improves significantly the performance of Dijkstra's algorithm. The obtained speedup is proportional to the density of the graph, and increases for denser graphs. This occurs because as graphs

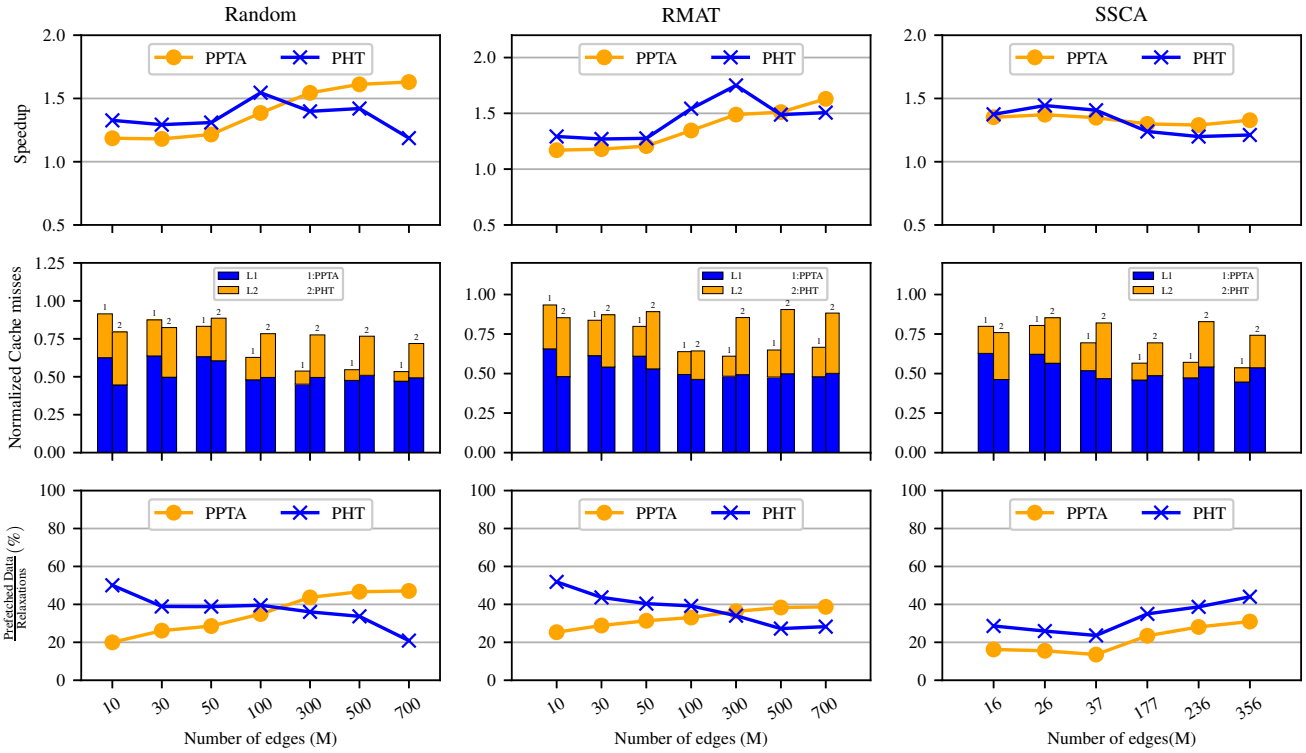


Figure 5.2: AMD-Opteron results.

become denser, PPTA prefetches more data. Therefore, the process-phase thread experiences less cache misses, and achieves greater speedup. Indeed, PPTA reduces the cache misses by up to 45% on the AMD system (Fig. 5.2), and achieves a maximum speedup of 1.62× on the densest graph of 700M edges. This pattern occurs also on the Intel system, where maximum speedup is 1.5×.

PPTA on sparse graphs. For PPTA with sparse graphs, i.e., graphs with up to 100M edges, we still observe some speedup on the AMD system but not as high as for PPTA with dense graphs, while on the Intel system we observe slowdown compared to the baseline. This behaviour occurs because the PPTA scheme introduces synchronization and coherence overhead. Both threads frequently update the priority queue and the *distance* and *previous* arrays. When each thread reads or writes on these memory locations, the cache coherence mechanisms trigger cache-to-cache transfers. In the case of the Intel system where the two threads share the L3 cache, sparse graphs magnify the coherence overhead compared to the AMD system where the two threads share the L2 cache. Hence, we conclude that the most beneficial option for the PPTA scheme is to employ cores that share as many levels of the cache hierarchy as possible; this way, the coherence overhead is reduced.

PPTA with SMT. Previously we focused on the performance of PPTA when the two threads run on two separate cores. In this part, we focus on PPTA-SMT, i.e., when the two threads run on the same core using the available SMT support. Note that only our Intel system provides support for SMT.

The PPTA-SMT scheme shows appreciable speedups in sparse graphs, by up to 1.61×. The threads in the PPTA-SMT configuration share the whole memory hierarchy including the L1 cache, reducing the coherence overhead and resulting in better performance for sparse graphs compared to PPTA. However, as graphs become denser, the speedup for PPTA-SMT decreases. To understand this be-

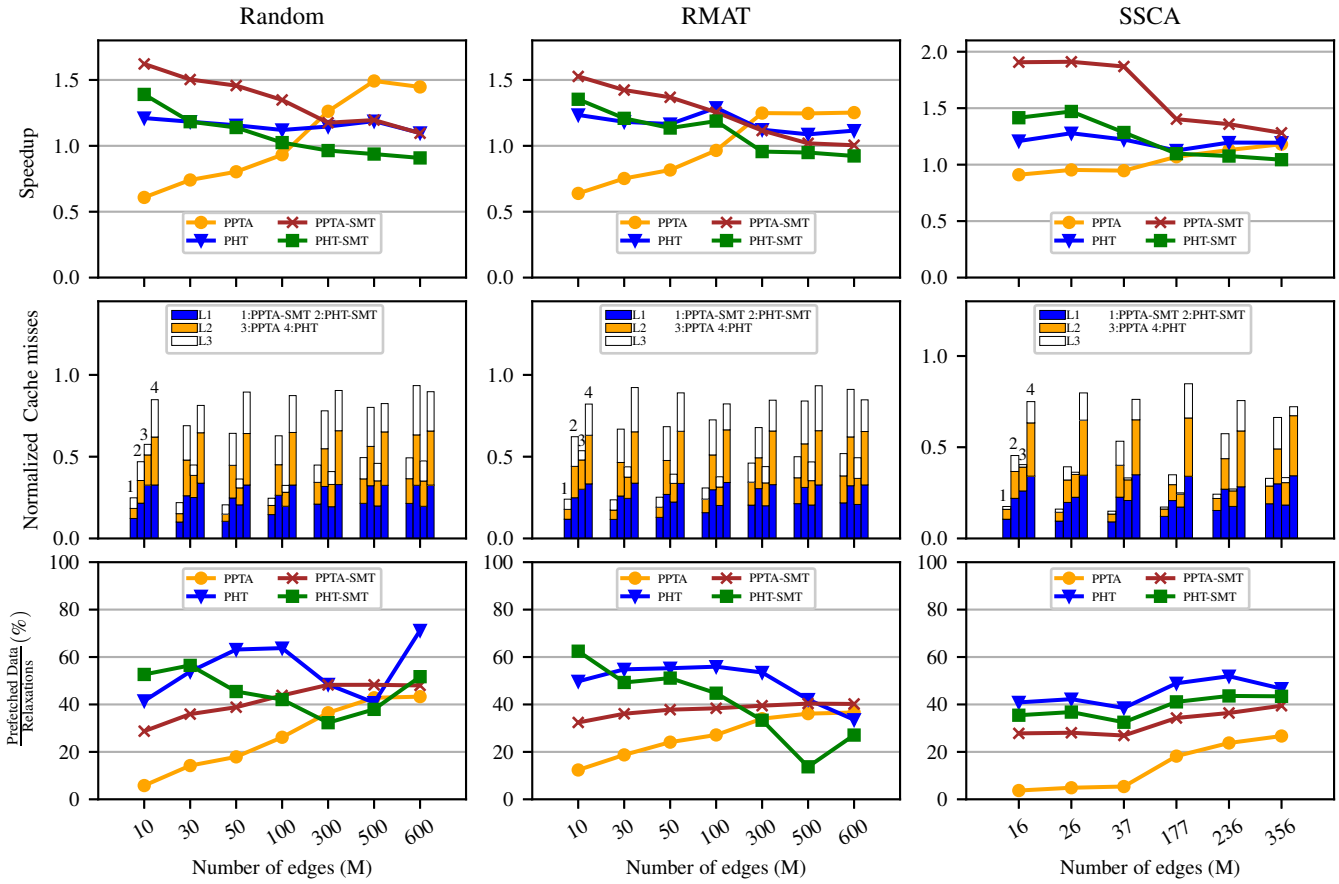


Figure 5.3: Intel-Broadell-EP results. For both systems, rows present speedup, cache misses of the process phase, and ratio of the successfully prefetched data. Columns represent the three graph families.

haviour, we measure the percentage of cycles that the threads remain idle during the process-phase due to pipeline contention, for both PPTA and PHT schemes with and without SMT (Fig. 5.4). We observe that by employing SMT on denser graphs, a significant part of the execution time is spent waiting to reserve out-of-order resources. Because the prefetch phase becomes more memory-intensive for denser graphs, the prefetch-phase thread consumes the shared pipeline resources for longer time to prefetch data and subsequently slows down the process-phase thread. For example, in the case of the 500M-edges *Random* graph, PPTA and PPTA-SMT reduce similarly the number of cache misses; however, the speedup is smaller in the second case because of the resource contention overhead.

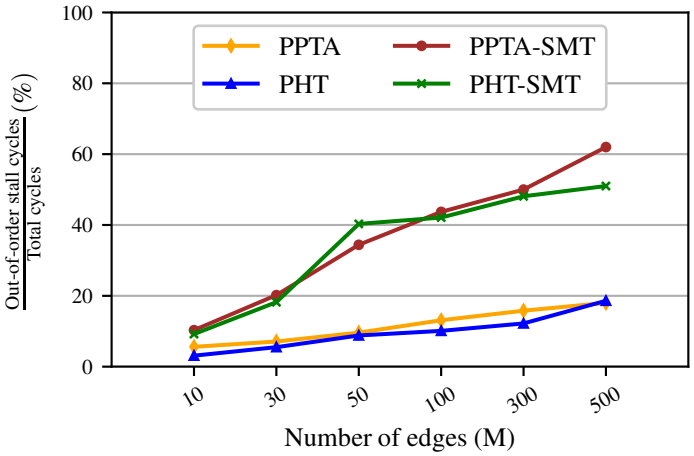


Figure 5.4: Portion of process phase core is stalled due to pipeline's resource contention, as reported by performance counter(Event A2H, Umask 01H)

Summary. Our results show that PPTA improves performance as graphs become denser on both platforms, and surpasses the PHT scheme for most configurations. In the AMD system, PPTA improves performance by exploiting the shared L2 cache and reducing the coherence overhead. However, in the Intel system, PPTA suffers from the coherence overhead, particularly for sparse graphs, since data traffic is greater due to the shared L3 cache. Finally, regarding the use of SMT with PPTA, there is a trade-off: for sparse graphs, the threads compete less for the pipeline resources resulting in noticeable speedups, whereas for dense graphs the increased pipeline contention may lead to slowdowns.

Chapter 6

Conclusion & Future Work

In the first part of this thesis, we worked on several practical optimizations on Dijkstra's algorihm to discover its bottleneck. We explored the design space of priority queues. Thus, we implemented three different priority queues to discover their impact on the algorithm's performance. Each queue effects differently the operations of the algorithm but performance remains almost the same.

Moreover, we profiled its operations in a fine-grained way, discovered its bottleneck(Update) and characterized it aa a memory-bound algorithm whose performance is negatively affected by irregular memory accesses. In order to reduce memory latency, we introduced two schemes that employ software prefetching to deal with this kind of memory requests and speed up Dijkstra's algorithm. We came up with the Prefetch-Process-Thread-Alternation scheme which derives from a much simpler Prefetching-Helper-Thread technique. On the one hand PHT spawns a main thread that executes the algorithm while a helper thread aggresively prefetches data to the nearest shared cache memory. On the other hand, PPTA involves two threads that alternately switch between a prefetching and a process phase, to hide the memory latency caused by cache misses.

We deduce that PPTA introduces coherency overhead which can be diminished by employing cores that share as many levels of the cache hierarchy as possible. Moreover, PPTA achieves increasing speedups as graphs become denser and surpasses PHT. AMD Opteron achieves speedups up to 1.62 for the densest of our graphs and Intel Broadwell-EP reaches 1.82 for a sparse graph, with the hyperthread in use. Considering the serial nature of Dijkstra's algorithm and the inherent difficulties in its parallelization, this is a significant performance gain.

The fact that the PPTA scheme achieves an increasing rate of speedups as graphs become denser, is a key result since extremely large scale graphs have emerged in various modern applications, such as, the graph of the Twitter social network and the neuronal network of the Human Brain Project.

As future work, we intend to explore the potential of the PPTA scheme and how it can be applied to other memory intensive algorithms of similar nature. Another important issue is how these schemes can scale effectively beyond two threads.

We will also investigate the idea of a Near-Data Processing scheme which provides specialized hardware and appropriate software API to deal with memory-intensive applications such as graph processing.

Bibliography

[1] E. W. Dijkstra, "A note on two problems in connexion with graphs," *Numer. Math.*, vol. 1, pp. 269–271, Dec. 1959.

[2] G. Brassard and P. Bratley, *Fundamentals of Algorithmics*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1996.

[3] M. L. Fredman and R. E. Tarjan, "Fibonacci heaps and their uses in improved network optimization algorithms," *J. ACM*, vol. 34, pp. 596–615, July 1987.

[4] G. S. Brodal, "Worst-case efficient priority queues," in *In proc. 7th ACM-siam symposium on discrete algorithms*, pp. 52–58, 1996.

[5] H. Rihani, P. Sanders, and R. Dementiev, "Brief announcement: Multiqueues: Simple relaxed concurrent priority queues," in *Proceedings of the 27th ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '15, (New York, NY, USA), pp. 80–82, ACM, 2015.

[6] J. Lindén and B. Jonsson, *A Skiplist-Based Concurrent Priority Queue with Minimal Memory Contention*, pp. 206–220. Cham: Springer International Publishing, 2013.

[7] "Skiplist-based concurrent priority queues," in *Proceedings of the 14th International Symposium on Parallel and Distributed Processing*, IPDPS '00, (Washington, DC, USA), pp. 263–, IEEE Computer Society, 2000.

[8] P. Sanders, "Fast priority queues for cached memory," *J. Exp. Algorithmics*, vol. 5, Dec. 2000.

[9] U. Meyer and P. Sanders, "Δ-stepping: A parallelizable shortest path algorithm," *J. Algorithms*, vol. 49, pp. 114–152, Oct. 2003.

[10] N. Anastopoulos, K. Nikas, G. Goumas, and N. Koziris, "Early experiences on accelerating dijkstra's algorithm using transactional memory," in *Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing*, IPDPS '09, (Washington, DC, USA), pp. 1–8, IEEE Computer Society, 2009.

[11] K. Nikas, N. Anastopoulos, G. I. Goumas, and N. Koziris, "Employing transactional memory and helper threads to speedup dijkstra's algorithm," in *ICPP 2009, International Conference on Parallel Processing, Vienna, Austria, 22-25 September 2009*, pp. 388–395, IEEE Computer Society, 2009.

[12] R. E. Tarjan, *Data Structures and Network Algorithms*. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 1983.

[13] W. Pugh, "Skip lists: A probabilistic alternative to balanced trees," *Commun. ACM*, vol. 33, pp. 668–676, June 1990.

[14] W. A. Wulf and S. A. McKee, "Hitting the memory wall: Implications of the obvious," *SIGARCH Comput. Archit. News*, vol. 23, pp. 20–24, Mar. 1995.

[15] J.-M. Kuntz, *Performance evaluation of cache memories in tightly coupled multiprocessor systems*, pp. 733–750. Berlin, Heidelberg: Springer Berlin Heidelberg, 1992.

[16] S. Mittal, "A survey of recent prefetching techniques for processor caches," *ACM Comput. Surv.*, vol. 49, pp. 35:1–35:35, Aug. 2016.

[17] T. Ungerer, B. Robič, and J. Šilc, "A survey of processors with explicit multithreading," *ACM Comput. Surv.*, vol. 35, pp. 29–63, Mar. 2003.

[18] J. L. Hennessy and D. A. Patterson, *Computer Architecture, Fifth Edition: A Quantitative Approach*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 5th ed., 2011.

[19] J. E. Smith and W.-C. Hsu, "Prefetching in supercomputer instruction caches," in *Proceedings of the 1992 ACM/IEEE Conference on Supercomputing*, Supercomputing '92, (Los Alamitos, CA, USA), pp. 588–597, IEEE Computer Society Press, 1992.

[20] R. Bianchini, "A preliminary evaluation of cache-miss-initiated prefetching techniques in scalable multiprocessors," 1994.

[21] A. J. Smith, "Sequential program prefetching in memory hierarchies," *Computer*, vol. 11, pp. 7–21, Dec. 1978.

[22] C. Zhang and S. A. McKee, "Hardware-only stream prefetching and dynamic access ordering," in *Proceedings of the 14th International Conference on Supercomputing*, ICS '00, (New York, NY, USA), pp. 167–175, ACM, 2000.

[23] J. W. C. Fu, J. H. Patel, and B. L. Janssens, "Stride directed prefetching in scalar processors," in *Proceedings of the 25th Annual International Symposium on Microarchitecture*, MICRO 25, (Los Alamitos, CA, USA), pp. 102–110, IEEE Computer Society Press, 1992.

[24] J. Pierce and T. Mudge, "Wrong-path instruction prefetching," in *Proceedings of the 29th Annual ACM/IEEE International Symposium on Microarchitecture*, MICRO 29, (Washington, DC, USA), pp. 165–175, IEEE Computer Society, 1996.

[25] X. Yu, C. J. Hughes, N. Satish, and S. Devadas, "Imp: Indirect memory prefetcher," in *Proceedings of the 48th International Symposium on Microarchitecture*, MICRO-48, (New York, NY, USA), pp. 178–190, ACM, 2015.

[26] S. Ainsworth and T. M. Jones, "Software prefetching for indirect memory accesses," in *Proceedings of the 2017 International Symposium on Code Genera tion and Optimization*, CGO '17, (Piscataway, NJ, USA), pp. 305–317, IEEE Press, 2017.

[27] M. Kamruzzaman, S. Swanson, and D. M. Tullsen, "Inter-core prefetching for multicore processors using migrating helper threads," in *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVI, (New York, NY, USA), pp. 393–404, ACM, 2011.

[28] D. Kim, S. S.-w. Liao, P. H. Wang, J. d. Cuvillo, X. Tian, X. Zou, H. Wang, D. Yeung, M. Girkar, and J. P. Shen, "Physical experimentation with prefetching helper threads on intel's hyper-threaded processors," in *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, CGO '04, (Washington, DC, USA), pp. 27–, IEEE Computer Society, 2004.

[29] D. Bader and K. Madduri, "gtgraph: A suite of synthetic graph generators"," 2006.