Εθνικό Μετσόβιο Πολυτεχνείο
Σχολή Ηλεκτρολόγων Μηχανικών
και Μηχανικών Υπολογιστών
Τομέας Τεχνολογίας Πληροφορικής
και Υπολογιστών

SPLaSh: Κλιμάκωση Προγραμμάτων Κελύφους σε Πλατφόρμες Χωρίς Διακομιστή

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

ΝΙΚΟΛΑΟΣ ΠΑΓΩΝΑΣ

Επιβλέπων : Γεώργιος Γκούμας
Αν. Καθηγητής, Ε.Μ.Π.

Συν-Επιβλέπων : Νίκος Βασιλάκης
Επικ. Καθηγητής, Πανεπιστήμιο Μπράουν

Αθήνα, Ιούνιος 2024

Εθνικό Μετσόβιο Πολυτεχνείο
Σχολή Ηλεκτρολόγων Μηχανικών
και Μηχανικών Υπολογιστών
Τομέας Τεχνολογίας Πληροφορικής
και Υπολογιστών

SPLaSh: Κλιμάκωση Προγραμμάτων Κελύφους σε Πλατφόρμες Χωρίς Διακομιστή

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

ΝΙΚΟΛΑΟΣ ΠΑΓΩΝΑΣ

Επιβλέπων : Γεώργιος Γκούμας
Αν. Καθηγητής, Ε.Μ.Π.

Συν-Επιβλέπων : Νίκος Βασιλάκης
Επικ. Καθηγητής, Πανεπιστήμιο Μπράουν

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 25η Ιουνίου 2024.

Γεώργιος Γκούμας Αν. Καθηγητής, Ε.Μ.Π.	Νικόλαος Παπασπύρου Καθηγητής, Ε.Μ.Π.	Νίκος Βασιλάκης Επικ. Καθηγητής, Πανεπιστήμιο Μπράουν

Αθήνα, Ιούνιος 2024

Νικόλαος Παγώνας

Διπλωματούχος Ηλεκτρολόγος Μηχανικός και Μηχανικός Υπολογιστών Ε.Μ.Π.

Περίληψη

Παρόλο που το κέλυφος Unix χρησιμοποιείται ευρέως σήμερα, αυτή τη στιγμή δεν υπάρχει υποστήριξη για την αυτόματη ανάπτυξη προγραμμάτων κελύφους σε σύγχρονες πλατφόρμες χωρίς διακομιστή—χάνοντας σημαντικά οφέλη όπως η ελαστικότητα, η κλιμακωσιμότητα, και η τιμολόγηση με βάση τη χρήση. Το SPLaSh είναι ένα νέο σύστημα για την αυτόματη κλιμάκωση προγραμμάτων φλοιού σε υποδομές χωρίς διακομιστή, χρησιμοποιώντας ένα συνδυασμό μεταγλώττισης και εκτέλεσης. Το SPLaSh εισάγει μια σειρά νέων εντολών κελύφους για την εκτέλεση χωρίς διακομιστή, εγκαθιστά επικοινωνία ροής δεδομένων μεταξύ συναρτήσεων και παρέχει αυτόματη, πάνω-στην-ώρα ανάπτυξη και κλιμάκωση χωρίς διακομιστή. Αξιολογημένο σε ένα σύνολο πραγματικών προγραμμάτων κελύφους, το SPLaSh προσφέρει ένα εύρος επιταχύνσεων σε σχέση με το Bash (0.75–14.38×, μέσος όρος: 1.75×)—χωρίς να απαιτεί τροποποιήσεις στα αρχικά προγράμματα.

Λέξεις κλειδιά

Unix, Κέλυφος, Υπολογισμός Χωρίς Διακομιστή, Συνάρτηση-ως-Υπηρεσία, Υπολογισμός Νέφους

Ευχαριστίες

Σε αυτό το σημείο θέλω να ευχαριστήσω θερμά τον επιβλέποντά μου, κ. Γεώργιο Γκούμα, για την άψογη συνεργασία, την εμπιστοσύνη που μου έδειξε, καθώς και την πολύτιμη συνδρομή του στα επόμενά μου ακαδημαϊκά βήματα. Επίσης, θέλω να εκφράσω την ευγνωμοσύνη μου στον κ. Νίκο Βασιλάκη, που μου έδωσε την μοναδική ευκαιρία να εκπονήσω μέρος αυτής της διπλωματικής στο Πανεπιστήμιο Μπράουν, και ανέλαβε με ζήλο να με καθοδηγήσει στα πρώτα βήματα της ερευνητικής μου πορείας. Επιπλέον, ευχαριστώ τον Κωνσταντίνο Καλλά, του οποίου οι πολύτιμες συμβουλές—τεχνικές και μη—ήταν καθοριστικές, τόσο εντός, όσο και εκτός του πλαισίου αυτής της εργασίας. Ακόμη, ευχαριστώ τον Yizheng Xie, ο οποίος ήταν πάντα πρόθυμος να συζητήσει και να βοηθήσει έμπρακτα καθ'όλη τη διάρκεια της διπλωματικής, αλλά και τον Haoran Zhang, που συντέλεσε σημαντικά στην υπέρβαση καίριων τεχνικών προκλήσεων. Τέλος, ευχαριστώ τους φίλους μου, που φρόντισαν—ο καθένας με τον τρόπο του—όλα αυτά τα φοιτητικά χρόνια να είναι γεμάτα όμορφες στιγμές, και με βοήθησαν να ξεπεράσω τις όποιες δυσκολίες προέκυπταν.

Πάνω από όλα όμως, θέλω να πω το μεγαλύτερο ευχαριστώ στους γονείς μου, Τάσο και Μίνα, και στην αδερφή μου, Εβίτα, που πάντα πιστεύουν σ' εμένα, με ωθούν συνεχώς να αγωνίζομαι για το καλύτερο, και στηρίζουν κάθε μου προσπάθεια με όλη τους την ψυχή.

Νικόλαος Παγώνας,

Αθήνα, 25η Ιουνίου 2024

Περιεχόμενα

Κατάλογος πινάκων

Κατάλογος σχημάτων

Κατάλογος κωδίκων

Κεφάλαιο 1

Εισαγωγή

Το κέλυφος Unix είναι δημοφιλές και ισχυρό, και έχει μία ευρεία γκάμα εφαρμογών, όπως προγράμματα χτισίματος, συνεχής ανάπτυξη, συνεχής ενσωμάτωση, επεξεργασία δεδομένων, και βιοπληροφορική. Ωστόσο, όσον αφορά τη σύγχρονη κατάσταση του υπολογισμού, το κέλυφος έχει μείνει πίσω. Από τη μία πλευρά, οι υπάρχουσες προσεγγίσεις για την κλιμάκωση των προγραμμάτων κελύφους [34, 40, 54, 82] δεν υποστηρίζουν την ανάπτυξη χωρίς διακομιστή. Από την άλλη πλευρά, τα τρέχοντα πλαίσια ανάπτυξης χωρίς διακομιστή [1, 2, 8, 15, 26, 27, 30, 33, 36, 37, 53, 58, 74, 78, 84] είναι προορισμένα για συγκεκριμένα πεδία. Ενώ ορισμένα συστήματα είναι σχεδιασμένα για ανάπτυξη εφαρμογών χωρίς διακομιστή γενικού σκοπού [17, 18, 87], απαιτούν από τους προγραμματιστές να χρησιμοποιούν μορφές που είναι ειδικές για το εκάστοτε σύστημα και δεν μπορούν να ανταποκριθούν στην εκφραστικότητα και τη δυναμική φύση του κελύφους. Έτσι, το κέλυφος χάνει αυτήν τη στιγμή σημαντικά οφέλη που προσφέρει ο υπολογισμός χωρίς διακομιστή, όπως ελαστικότητα, κλιμακωσιμότητα, και τιμολόγηση με βάση τη χρήση.

Το SPLaSh είναι ένα νέο σύστημα για αυτόματη κλιμάκωση προγραμμάτων κελύφους σε υποδομές χωρίς διακομιστή, χρησιμοποιώντας ένα συνδυασμό μεταγλώττισης και εκτέλεσης. Για να το κάνει αυτό, το SPLaSh παίρνει το αρχικό πρόγραμμα κελύφους ως είσοδο και αναγνωρίζει τμήματα που μπορούν να κατανεμηθούν σε συναρτήσεις. Στη συνέχεια, εισάγει νέες εντολές για λειτουργίες χωρίς διακομιστή—όπως κλήση και επικοινωνία συναρτήσεων, ή αλληλεπίδραση με συστήματα αποθήκευσης νέφους και ουρές μηνυμάτων. Το SPLaSh δημιουργεί επίσης κανάλια επικοινωνίας ροής μεταξύ των συναρτήσεων, επιτρέποντας στις συναρτήσεις να ανακαλύπτουν η μία την άλλη και να ξεπερνούν τους δικτυακούς περιορισμούς του υπολογισμού χωρίς διακομιστή. Τέλος, το SPLaSh προσφέρει υποστήριξη για αυτόματη ανάπτυξη και κλιμάκωση χωρίς διακομιστή με ένα κλικ, επιτρέποντας ένα σύνολο από δυναμικές βελτιστοποιήσεις απόδοσης—χωρίς καμία παρέμβαση του χρήστη. Αξιολογημένο σε ένα σύνολο πραγματικών προγραμμάτων κελύφους, το SPLaSh προσφέρει ένα εύρος επιταχύνσεων σε σχέση με το Bash [20], το καθιερωμένο περιβάλλον σειριακής εκτέλεσης προγραμμάτων κελύφους (0.75–14.38×, μέσος όρος: 1.75×).

Αυτή η διπλωματική εργασία ξεκινά με την ανάλυση των απαραίτητων θεμελιωδών γνώσεων για τον υπολογισμό χωρίς διακομιστή, την ανάπτυξη προγραμμάτων κελύφους Unix, και τις προηγούμενες προσπάθειες κλιμάκωσης προγραμμάτων κελύφους (Κεφάλαιο 2). Στη συνέχεια, παρουσιάζει τις προκλήσεις της εκτέλεσης κελύφους χωρίς διακομιστή μέσω ενός παραδείγματος, και παρέχει μια επισκόπηση του SPLaSh και των συνεισφορών του (Κεφάλαιο 3). Συνεχίζει με την περιγραφή της μεταγλώττισης και εκτέλεσης που προσφέρει το SPLaSh (Κεφάλαια 4 και 5), καθώς και της αξιολόγησης του SPLaSh (Κεφάλαιο 6). Τέλος, παρέχει μια επισκόπηση του προηγούμενου ερευνητικού έργου που σχετίζεται με το SPLaSh (Κεφάλαιο 7), και τελειώνει με μια αναφορά του μελλοντικού έργου και των συμπερασμάτων (Κεφάλαιο 8). Το SPLaSh είναι μέρος του PaSh—ενός έργου με άδεια MIT που φιλοξενείται από το Ίδρυμα Linux. Η έκδοση του SPLaSh που περιγράφεται σε αυτήν τη διπλωματική εργασία είναι διαθέσιμη στο `https://github.com/nikpag/splash`, ενώ η τελευταία έκδοση είναι διαθέσιμη στο `https://github.com/binpash/pash`.

Κεφάλαιο 2

Υπόβαθρο

Αυτό το κεφάλαιο παρέχει αρχικά μια επισκόπηση του υπολογισμού νέφους, των αδυναμιών του, και της εμφάνισης του υπολογισμού χωρίς διακομιστή σε μια προσπάθεια να ξεπεράσει αυτές τις αδυναμίες. Στη συνέχεια, αναλύει λεπτομερώς τον υπολογισμό χωρίς διακομιστή, περιγράφοντας τα οφέλη του, τις προκλήσεις του, και την τρέχουσα κατάσταση της έρευνας πάνω στον υπολογισμό χωρίς διακομιστή. Έπειτα, επικεντρώνεται στο κέλυφος Unix και τις αφαιρέσεις, τα πλεονεκτήματα, και τα μειονεκτήματά του. Τέλος, παρέχει μια επισκόπηση των παρελθοντικών προσπαθειών για τη βελτίωση της επίδοσης του κελύφους, μέσω αυτόματης παραλληλοποίησης και κατανομής.

2.1 Υπολογισμός Νέφους

Ο υπολογισμός νέφους είναι η πραγματοποίηση ενός παλαιού ονείρου του υπολογισμού ως υπηρεσία, όπου οι χρήστες μπορούν να πληρώνουν για τους υπολογιστικούς πόρους με βάση τη χρήση. Παρέχει στους χρήστες πρόσβαση σε ένα κοινόχρηστο σύνολο πόρων—υπολογιστική ισχύ, αποθήκευση, δικτύωση—τους οποίους μπορούν να προμηθευτούν και να απελευθερώσουν γρήγορα, με ελάχιστη προσπάθεια διαχείρισης.

Πλεονεκτήματα Ο υπολογισμός νέφους προσφέρει αρκετά πλεονεκτήματα. Οι χρήστες μπορούν να έχουν πρόσβαση σε σχεδόν απεριόριστους υπολογιστικούς πόρους κατά βούληση, χωρίς αρχική δέσμευση, καθώς πληρώνουν για τη βραχυπρόθεσμη χρήση των υπολογιστικών πόρων, όποτε τους χρειάζονται. Οι πόροι μπορούν να προσφέρονται σε μειωμένο κόστος, επειδή οι πάροχοι υπηρεσιών νέφους έχουν πολλά μεγάλα κέντρα δεδομένων και επωφελούνται από οικονομίες κλίμακας. Η εικονικοποίηση μπορεί να απλοποιήσει τη λειτουργία και να αυξήσει την αξιοποίηση. Οι πάροχοι υπηρεσιών νέφους μπορούν να επιτύχουν υψηλότερη αξιοποίηση του υλικού με την πολυπλεξία φορτίων από διάφορους οργανισμούς.

Αδυναμίες Παρόλο που το νέφος προσφέρει τα περισσότερα από αυτά τα πλεονεκτήματα, δεν παρέχει πλήρως τα τελευταία δύο—απλοποίηση της λειτουργίας και οφέλη από την πολυπλεξία. Όσον αφορά την απλοποίηση της λειτουργίας, το νέφος απαλλάσσει τους χρήστες από τη λειτουργία της φυσικής υποδομής, αλλά τους αφήνει με μια πληθώρα εικονικών πόρων για να διαχειριστούν. Επιπλέον, οι χρήστες πρέπει ακόμα να αντιμετωπίσουν πολλές προκλήσεις κατά την δημιουργία ενός περιβάλλοντος στο νέφος. Συγκεκριμένα, πρέπει (1) να έχουν πλεονασμό μηχανημάτων, ώστε η αποτυχία ενός μηχανήματος να μην απενεργοποιεί την υπηρεσία, (2) να διανέμουν αντίγραφα σε διαφορετικές γεωγραφικές περιοχές, για να διατηρήσουν την υπηρεσία σε περίπτωση καταστροφής, (3) να ισορροπούν το φορτίο και να δρομολογούν τα αιτήματα αποδοτικά, για να εκμεταλλευτούν βέλτιστα τους πόρους, (4) να αυτοματοποιούν την κλιμάκωση ανάλογα με τις αλλαγές στο φορτίο για να αυξήσουν ή να μειώσουν τους πόρους που χρησιμοποιεί το σύστημα, (5) να παρακολουθούν την υπηρεσία για να βεβαιωθούν ότι λειτουργεί καλά, (6) να καταγράφουν μηνύματα που χρειάζονται για αποσφαλμάτωση ή βελτίωση της απόδοσης, (7) να χειρίζονται αναβαθμίσεις του συστήματος, συμπεριλαμβανομένων των ενημερώσεων ασφαλείας, (8) να μεταφέρουν τα μηχανήματά τους σε νέες

εκδόσεις, όταν αυτές γίνονται διαθέσιμες. Όσον αφορά τα οφέλη που προκύπτουν από την πολυπλεξία, αυτά παρατηρούνται κυρίως σε φορτία που επεξεργάζονται σύνολα δεδομένων σε φάσεις, όπως το MapReduce [13] ή ο υπολογισμός υψηλής επίδοσης—τα οποία μπορούν να αξιοποιήσουν πλήρως τους πόρους που εκχωρούν. Δεν παρατηρούνται τόσο πολύ σε υπηρεσίες που διαχειρίζονται κατάσταση (π.χ., κατά τη μεταφορά επιχειρηματικού λογισμικού—όπως συστήματα διαχείρισης βάσεων δεδομένων—στο νέφος).

Επιπλέον, οι γρήγορες και δύσκολες στην πρόβλεψη αλλαγές στη ζήτηση πόρων καθιστούν την πρόβλεψη πόρων μια πρόκληση. Οι χρήστες πρέπει είτε να υπερπρομηθεύονται πόρους με υψηλό κόστος, είτε να υποφέρουν από υψηλή καθυστέρηση όταν η ζήτηση αυξάνεται απότομα. Επειδή μεγάλα φορτία εργασίας μπορεί να προκύψουν οποιαδήποτε στιγμή—από ένα μεγάλο αριθμό ταυτόχρονων αιτημάτων, αιτήματα που χρειάζονται πολλούς υπολογιστικούς πόρους, ή και τα δύο—η ζήτηση πόρων διακυμαίνεται σε γρήγορες, απρόβλεπτες εκρήξεις. Ενώ είναι δυνατόν να χρησιμοποιηθούν μηχανισμοί αυτόματης κλιμάκωσης για να αντιμετωπιστούν οι αλλαγές στη ζήτηση—προσθέτοντας εικονικές μηχανές όταν η ζήτηση αυξάνεται και αφαιρώντας τις όταν είναι αδρανείς—οι νέοι κόμβοι χρειάζονται δεκάδες δευτερόλεπτα ή λεπτά για να γίνουν διαθέσιμοι, με αποτέλεσμα υψηλή καθυστέρηση. Επίσης, οι κόμβοι πρέπει να παραμένουν αδρανείς για κάποιο χρονικό διάστημα μέχρι να απενεργοποιηθούν—με το ακριβές χρονικό διάστημα να εξαρτάται από την εφαρμογή και το φορτίο, και επομένως να είναι δύσκολο να προβλεφθεί—με αποτέλεσμα τυχόν επιπλέον χρεώσεις για αδρανή μηχανήματα.

2.2 Υπολογισμός Χωρίς Διακομιστή

Για να λυθούν τα παραπάνω προβλήματα—και λόγω της ανάγκης για ακόμα μεγαλύτερη ελαστικότητα και χρέωση ανά μικρότερη μονάδα χρόνου—έχει εμφανιστεί ένα νέο παράδειγμα υπολογισμού νέφους που ονομάζεται *υπολογισμός χωρίς διακομιστή*, το οποίο δεν απαιτεί καθόλου διαχείριση υποδομής από τον χρήστη, παρέχοντας ταυτόχρονα ελαστικότητα, κλιμακωσιμότητα, και οικονομία. Ο υπολογισμός χωρίς διακομιστή έχει υιοθετηθεί ευρέως από τους χρήστες νέφους—και η δημοτικότητά του εξακολουθεί να αυξάνεται. Το 2023, το 70% των πελατών της AWS χρησιμοποίησε προσφορές χωρίς διακομιστή, σε σύγκριση με το 50% το 2021. Η κατάσταση είναι παρόμοια για το Νέφος Google (60%, έναντι 20%) και το Microsoft Azure (50%, έναντι 35%) [11, 12].

Επισκόπηση Για να θεωρηθεί μία υπηρεσία ως υπηρεσία χωρίς διακομιστή, πρέπει να κλιμακώνεται αυτόματα χωρίς ρητή δέσμευση μηχανημάτων, και να χρεώνεται με βάση τη χρήση. Ο υπολογισμός χωρίς διακομιστή διαφέρει από προηγούμενες προσεγγίσεις επειδή αποσυνδέει τον υπολογισμό από άλλους πόρους, επιτρέπει στους χρήστες να εκτελούν κώδικα χωρίς να διαχειρίζονται την υποκείμενη υποδομή, και παρέχει τιμολόγηση βάσει της χρήσης πόρων—και όχι της δέσμευσης τους. Αποτελείται από δύο στοιχεία: Συνάρτηση-ως-Υπηρεσία και Backend-ως-Υπηρεσία. Με τη Συνάρτηση-ως-Υπηρεσία, οι χρήστες γράφουν συναρτήσεις που εκτελούνται ως ανταπόκριση σε γεγονότα, και ο πάροχος υπηρεσιών νέφους αναλαμβάνει τις εργασίες διαχείρισης υποδομής—όπως κλιμάκωση, εκχώρηση πόρων, και παρακολούθηση. Το Backend-ως-Υπηρεσία περιλαμβάνει οποιαδήποτε υπηρεσία χωρίς διακομιστή που προορίζεται για συγκεκριμένες εφαρμογές, όπως αποθήκευση, βάσεις δεδομένων ή ανταλλαγή μηνυμάτων. Αυτό αποσυνδέει τον υπολογισμό από άλλους πόρους, επιτρέποντας σε κάθε πόρο να κλιμακώνεται ανεξάρτητα.

Η βασική μονάδα του υπολογισμού χωρίς διακομιστή είναι η συνάρτηση: ο χρήστης γράφει σε μια γλώσσα υψηλού επιπέδου, επιλέγει ένα γεγονός που ενεργοποιεί τη συνάρτηση, και το σύστημα υπολογισμού χωρίς διακομιστή χειρίζεται όλα τα υπόλοιπα: εκχώρηση πόρων, επιλογή μηχανημάτων, φόρτωση, διαχείριση λειτουργικού συστήματος, ισορροπία φορτίου, αυτόματη κλιμάκωση, ανοχή σφαλμάτων, παρακολούθηση, καταγραφή, ενημερώσεις ασφαλείας κλπ. Η πλατφόρμα ξεκινά και σταματά συναρτήσεις αυτόματα, όσο η ζήτηση αυξάνεται και μειώνεται. Με αυτόν τον τρόπο, οι προγραμματιστές μπορούν να επικεντρωθούν στη συγγραφή κώδικα για τις εφαρμογές τους. Η χρέωση για τον υπολογισμό χωρίς διακομιστή γίνεται σε πολύ πιο τακτά χρονικά διαστήματα—παρέχοντας ένα ελά-

χιστο χρονικό διάστημα χρέωσης των 100 ms—ενώ οι άλλες προσφορές νέφους χρεώνουν σε ωριαία βάση. Με αυτόν τον τρόπο, ο πελάτης χρεώνεται μόνο για το χρόνο που η εφαρμογή του εκτελείται πραγματικά—και όχι για τους πόρους που δεσμεύονται εκ των προτέρων για την εκτέλεση του προγράμματός του.

Πλεονεκτήματα Ο υπολογισμός χωρίς διακομιστή προσφέρει οφέλη για όλα τα ενδιαφερόμενα μέρη. Για τους παρόχους νέφους, ο υπολογισμός χωρίς διακομιστή προάγει την επιχειρησιακή ανάπτυξη, αφού καθιστά το νέφος πιο εύκολο στην προγραμματισμό, βοηθώντας να προσελκύσει νέους πελάτες, αλλά και να βοηθήσει τους υπάρχοντες πελάτες να χρησιμοποιούν περισσότερο τις προσφορές του νέφους. Η μικρή διάρκεια εκτέλεσης, το μικρό αποτύπωμα μνήμης, και η έλλειψη κατάστασης ενισχύουν την στατιστική πολυπλεξία, καθώς κάνουν πιο εύκολο στους παρόχους να βρουν αχρησιμοποίητους πόρους για την εκτέλεση συναρτήσεων. Οι χρήστες του νέφους επωφελούνται από αυξημένη παραγωγικότητα, καθώς οι αρχάριοι μπορούν να αναπτύξουν συναρτήσεις χωρίς καμία κατανόηση της υποδομής του νέφους, ενώ οι ειδικοί εξοικονομούν χρόνο ανάπτυξης και παραμένουν εστιασμένοι στα προβλήματα που είναι σημαντικά για τις εφαρμογή τους. Οι χρήστες του νέφους μπορούν επίσης να εξοικονομήσουν χρήματα, καθώς οι συναρτήσεις εκτελούνται μόνο όταν συμβαίνουν γεγονότα, και η τιμολόγηση σε τακτά χρονικά διαστήματα σημαίνει ότι πληρώνουν μόνο για τους πόρους που χρησιμοποιούν αντί για τους πόρους που δεσμεύουν. Οι ερευνητές βρίσκουν ενδιαφέρον στον υπολογισμό χωρίς διακομιστή, επειδή αποτελεί μια νέα υπολογιστική αφαίρεση γενικού σκοπού, που υπόσχεται να γίνει το μέλλον του υπολογισμού νέφους, και επειδή υπάρχουν πολλές ευκαιρίες για την αύξηση της τρέχουσας απόδοσης και την υπέρβαση των παρόντων περιορισμών του.

Υπολογισμός Χωρίς Διακομιστή για Πολύπλοκες Εφαρμογές Ενώ οι πλατφόρμες υπολογισμού χωρίς διακομιστή αρχικά στόχευαν απλές εφαρμογές με μία ή λίγες συναρτήσεις, αυτό το υπολογιστικό μοντέλο έχει αποδειχθεί χρήσιμο για πιο πολύπλοκες εφαρμογές, που αποτελούνται από πολλές συναρτήσεις με περίπλοκες αλληλεπιδράσεις, όπως ανάλυση δεδομένων [33, 36, 37, 38, 53, 58], αριθμητικούς υπολογισμούς [2, 15, 26, 27, 33, 74, 84], κωδικοποίηση βίντεο [1, 17, 18], μηχανική μάθηση [8, 30, 78] και μεταγλώττιση πηγαίου κώδικα [18]. Το 2019, η πλειοψηφία των εφαρμογών χωρίς διακομιστή αποτελούνταν από μία μόνο συνάρτηση, και το 80% είχαν τρεις ή λιγότερες συναρτήσεις [73]. Ωστόσο σήμερα, οι πολύπλοκες εφαρμογές χωρίς διακομιστή δεν είναι πλέον σπάνιες. Μια πρόσφατη μελέτη εφαρμογών ανοικτού κώδικα που χρησιμοποιούν υπολογισμό χωρίς διακομιστή συμπέρανε ότι το 31% των μελετημένων εφαρμογών έχουν δομή ροών [14]. Από το 2019 έως το 2022, η δημοτικότητα των ροών χωρίς διακομιστή που δομούνται ως Κατευθυνόμενοι Ακυκλικοί Γράφοι έχει αυξηθεί κατά 6 φορές στο Microsoft Azure [42]. Η αυξημένη πολυπλοκότητα των εφαρμογών χωρίς διακομιστή μπορεί να αποδοθεί στην ωρίμανση των προσφορών και στην αυξημένη εμπειρία των προγραμματιστών [57].

Προκλήσεις Παρά τα οφέλη του, ο υπολογισμός χωρίς διακομιστή εισάγει ένα σύνολο προκλήσεων.

- Η ανάπτυξη εφαρμογών χωρίς διακομιστή είναι περιοριστική, απαιτώντας από τους προγραμματιστές να γράφουν συναρτήσεις με έναν συγκεκριμένο τρόπο. Για παράδειγμα, ο υπολογισμός χωρίς διακομιστή εγγυάται την εκτέλεση συναρτήσεων τουλάχιστον μία φορά, οπότε οι συναρτήσεις με εξωτερικές επιδράσεις πρέπει να είναι ταυτοδύναμες, για να μπορούν να επανεκτελεστούν με ασφάλεια.
- Οι συναρτήσεις έχουν προσωρινή κατάσταση στη μνήμη και στον δίσκο, οπότε οι προγραμματιστές πρέπει να διασφαλίζουν ότι όλη η μόνιμη κατάσταση αποθηκεύεται σε εξωτερικούς χώρους αποθήκευσης (π.χ., συστήματα αποθήκευσης ή βάσεις δεδομένων που φιλοξενούνται στο νέφος).
- Η πλατφόρμα χωρίς διακομιστή επιβάλλει έναν αυστηρό χρονικό περιορισμό σε όλες τις συναρτήσεις, οπότε οι συναρτήσεις έχουν περιορισμένο χρόνο εκτέλεσης—μερικά λεπτά—πριν τερματίσουν. Αν κάποιος προγραμματιστής χρειάζεται να εκτελέσει έναν μακροσκελή υπολογισμό, πρέπει να τον χωρίσει σε μικρότερες συναρτήσεις.

- Ο υπολογισμός χωρίς διακομιστή υποφέρει από περιορισμένες δυνατότητες δικτύωσης. Οι συναρτήσεις δεν μπορούν να επικοινωνούν απευθείας μεταξύ τους, οπότε οι χρήστες καταφεύγουν σε χειροκίνητες υλοποιήσεις αργής επικοινωνίας βασισμένη στην αποθήκευση ή τη μνήμη. Αυτές οι λύσεις είτε επηρεάζουν την απόδοση, είτε έχουν απαγορευτικό κόστος, είτε εισάγουν μία διαχειριζόμενη από τον χρήστη συνιστώσα—αναιρώντας τον σκοπό του υπολογισμού χωρίς διακομιστή.
- Η ευρυεκπομπή, η συνάθροιση και το ανακάτεμα—μερικά από τα πιο κοινά μοτίβα επικοινωνίας σε κατανεμημένα συστήματα—δεν υποστηρίζονται καλά από τον υπολογισμό χωρίς διακομιστή. Η εκτέλεση που βασίζεται σε γεγονότα καθιστά δύσκολη την εξάρτηση από τα αποτελέσματα πολλαπλών προηγούμενων συναρτήσεων—κάτι που χαρακτηρίζει τα μοτίβα συνάθροισης.
- Καμία από τις υπηρεσίες αποθήκευσης νέφους δεν διαθέτει δυνατότητες ειδοποίησης. Ενώ οι πάροχοι νέφους προσφέρουν αυτόνομες υπηρεσίες ειδοποίησης, αυτές προσθέτουν σημαντική καθυστέρηση—εκατοντάδες χιλιοστά του δευτερολέπτου—και μπορεί να είναι δαπανηρές όταν χρησιμοποιούνται για λεπτομερή συντονισμό.

Ανταλλαγή Δεδομένων Χωρίς Διακομιστή Ο υπολογισμός χωρίς διακομιστή παρέχει μια σειρά επιλογών για ανταλλαγή δεδομένων. Πρώτον, οι άμεσες δικτυακές συνδέσεις μπορούν να προσφέρουν υψηλή απόδοση χωρίς να προκαλούν επιπρόσθετο κόστος για την επικοινωνία. Ωστόσο, οι συναρτήσεις βρίσκονται πίσω από μεταφραστές διευθύνσεων δικτύου και δεν δέχονται εισερχόμενες συνδέσεις. Δεύτερον, η αποθήκευση αντικειμένων παρέχει υψηλή ρυθμαπόδοση, ισχυρή συνέπεια και αξιοπιστία δεδομένων. Ωστόσο, δεν υποστηρίζει εγγραφή των δεδομένων σε ροή, εισάγει υψηλή καθυστέρηση και προκαλεί υψηλό κόστος. Τρίτον, οι βάσεις δεδομένων NoSQL και οι υπηρεσίες ουράς μηνυμάτων προσφέρουν χαμηλή καθυστέρηση και υψηλή ρυθμαπόδοση. Ωστόσο, έχουν χαμηλά όρια μεγέθους και υψηλό κόστος. Τέλος, οι υπηρεσίες ροής απαιτούν από τους χρήστες να δεσμεύουν χωρητικότητα εκ των προτέρων, χρεώνονται ανά ώρα με βάση τον αριθμό των πόρων που δεσμεύονται, και είναι δαπανηρές.

2.3 Το Κέλυφος Unix

Το κέλυφος Unix είναι ένα περιβάλλον—συχνά αλληλεπιδραστικό—για τη σύνθεση προγραμμάτων που γράφονται σε μία πληθώρα γλωσσών προγραμματισμού. Παρέχει μια σειρά χρήσιμων στοιχείων που ονομάζονται εντολές, καθώς και ισχυρά, ανεξάρτητα από τη γλώσσα στοιχεία για τη σύνθεση δομικών στοιχείων—σύμφωνα με τη φιλοσοφία του Unix [55]. Το κέλυφος κατατάσσεται συνεχώς ανάμεσα στις πιο δημοφιλείς γλώσσες προγραμματισμού [22, 50], ενώ το 2021 κατατάχθηκε έκτο στην αύξηση δημοτικότητας—πάνω από γλώσσες με ενεργές κοινότητες, όπως Python και Kotlin [40]. Τα προγράμματα κελύφους είναι κρίσιμα για προγραμματιστές, διαχειριστές συστημάτων και επιστήμονες, και χρησιμοποιούνται στην επεξεργασία δεδομένων, την ενορχήστρωση συστημάτων και τις αυτοματοποιημένες εργασίες, καθώς και σε σύγχρονες πλατφόρμες όπως Docker, Vagrant και Kubernetes.

Το κέλυφος έχει μια σειρά από οφέλη. Πρώτον, είναι συνοπτικό και εκφραστικό. Εκατό γραμμές κώδικα Java που εκτελούν ανάλυση δεδομένων θερμοκρασίας μπορούν να μεταφραστούν σε ένα πρόγραμμα Bash μιας γραμμής [25]. Δεύτερον, το κέλυφος είναι γρήγορο. Υποστηρίζει παράλληλισμό αγωγών χρησιμοποιώντας αγωγούς Unix για τη ροή δεδομένων, και παραλληλισμό εργασίας, μέσω εντολών για εκτέλεση στο παρασκήνιο με τον τελεστή παρασκηνίου (&). Τρίτον, το κέλυφος είναι δυναμικό, έχοντας χαρακτηριστικά όπως αντικατάσταση εντολών και ανάπτυξη μεταβλητών.

Αφαιρέσεις Κελύφους Το κέλυφος του Unix παρέχει μια σειρά χρήσιμων αφαιρέσεων. Πρώτον, οι *ροές δεδομένων του Unix* είναι ακολουθίες από bytes, συνήθως επεξεργαζόμενες γραμμή-γραμμή και συχνά αναφερόμενες με το όνομα ενός αρχείου. Διευκολύνουν τη σύνθεση εντολών, καθώς η έξοδος

```
#!/bin/bash
mkfifo t1 t2
grep "foo" f1 >t1 &
grep "foo" f2 >t2 &
cat t1 t2
```

Κώδικας 2.1: Οκνηρία στο πλαίσιο του κελύφους. Η εντολή `cat` θα ξεκινήσει να καταναλώνει είσοδο από τον αγωγό `t2` μόνο αφού τελειώσει με την ανάγνωση του `t1`, προκαλώντας υποχρησιμοποίηση.

μιας εντολής είναι η είσοδος μιας άλλης. Οι ροές δεδομένων μπορούν να είναι προσωρινοί, ανώνυμοι αγωγοί—που εκφράζονται χρησιμοποιώντας τον χαρακτήρα αγωγού—ή μόνιμοι, επώνυμοι αγωγοί—Unix FIFOs—που δημιουργούνται με την εντολή `mkfifo`. Οι ροές δεδομένων εισάγουν παράλληλη εκτέλεση μεταξύ εντολών, και ο πυρήνας του Unix εξασφαλίζει τη δρομολόγηση, την επικοινωνία, και τον συγχρονισμό, στο παρασκήνιο.

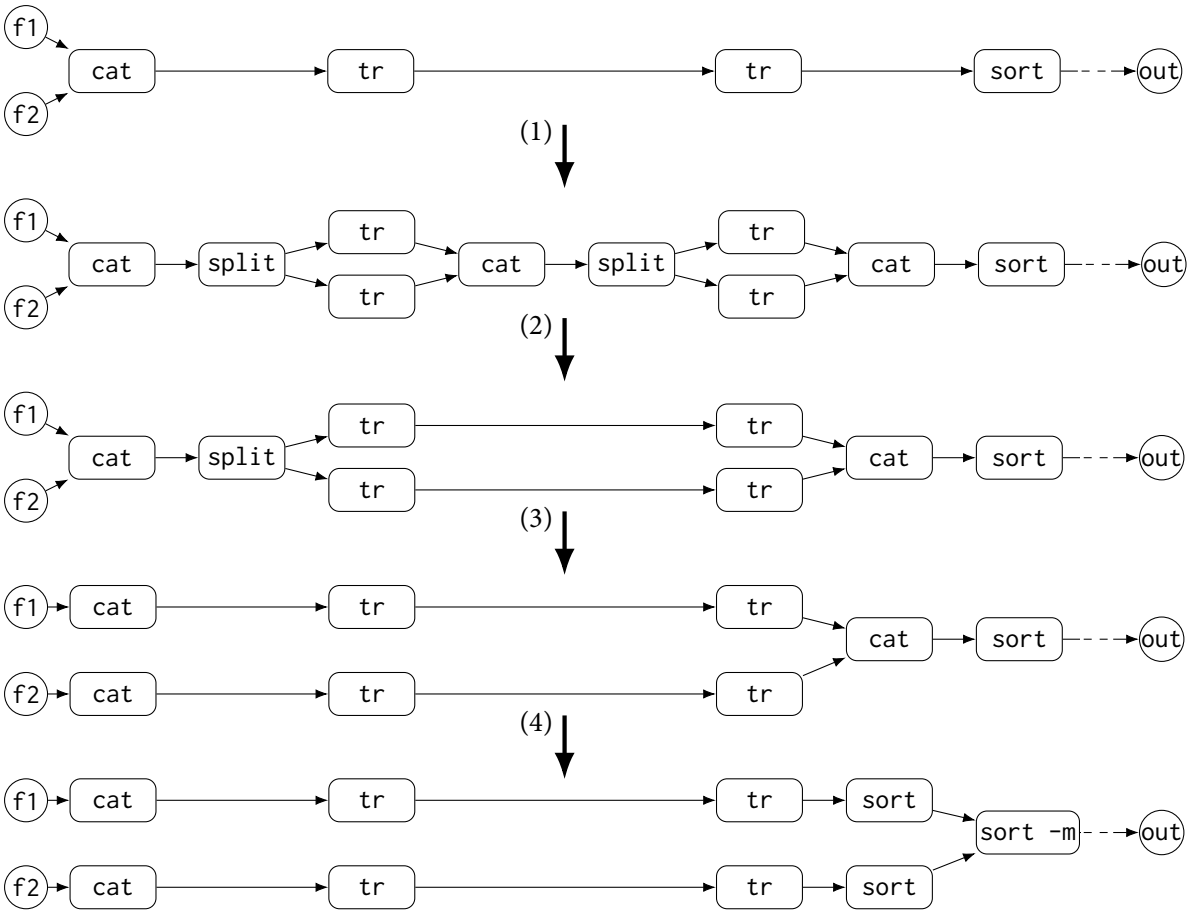
Δεύτερον, οι *εντολές* είναι ανεξάρτητες μονάδες υπολογισμού. Διαβάζουν μία ή περισσότερες ροές εισόδου, και παράγουν μία ή περισσότερες ροές εξόδου. Μία σημαντική διαφορά με άλλες γλώσσες προγραμματισμού που έχουν ένα κλειστό σύνολο εντολών, είναι ότι υπάρχει ένας απεριόριστος αριθμός εντολών Unix, και κάθε εντολή μπορεί να έχει αυθαίρετη συμπεριφορά. Οι εντολές αυτές μπορούν να γραφτούν σε οποιαδήποτε γλώσσα—ή να υπάρχουν ακόμα και σε δυαδική μορφή—κάτι που καθιστά δύσκολη την ανάλυση των χαρακτηριστικών επίδοσής τους. Οι εντολές μπορούν να ρυθμιστούν χρησιμοποιώντας μεταβλητές περιβάλλοντος και σημαίες. Για παράδειγμα, η εντολή `wc` υποστηρίζει τις σημαίες `-l`, `-w`, και `-c`—για την καταμέτρηση του αριθμού των γραμμών, λέξεων, και χαρακτήρων, αντίστοιχα. Οι σημαίες είναι σημαντικές στην ανάλυση της επίδοσης, γιατί μπορούν να αλλάξουν την παραλληλοποιησιμότητα μιας εντολής.[1] Οι εντολές είναι συχνά οκνηρές—καταναλώνουν τις εισόδους τους μόνο όταν είναι έτοιμες να επεξεργαστούν κι άλλα δεδομένα—κάτι που συχνά οδηγεί σε υποεκμετάλλευση του επεξεργαστή (Κώδικας 2.1).

Τέλος, το κέλυφος Unix περιλαμβάνει μία σειρά από τελεστές σύνθεσης. Ο σειριακός τελεστής (`;`) εκτελεί εντολές σειριακά, ο τελεστής παράλληλης σύνθεσης (`&`) εκτελεί εντολές παράλληλα, ο τελεστής αγωγού (`|`) συνδέει την έξοδο μιας εντολής με την είσοδο μιας άλλης, και οι λογικοί τελεστές (`&&`, `||`) εκτελούν εντολές με βάση την επιτυχία ή την αποτυχία προηγούμενων εντολών.

Προκλήσεις Παρά τα οφέλη του, το κέλυφος παρουσιάζει ορισμένες προκλήσεις. Πρώτον, η απόδοση του κέλυφους δεν κλιμακώνεται—και η παραλληλοποίηση των προγραμμάτων κελύφους απαιτεί πολλή χειρωνακτική προσπάθεια. Από τη μία πλευρά, οι προγραμματιστές εντολών υλοποιούν μεμονωμένες εντολές, εργάζονται σε μία μόνο γλώσσα προγραμματισμού, και εκθέτουν τον παραλληλισμό μέσω αυθαίρετων, ειδικών για την εντολή σημαιών. Από την άλλη πλευρά, οι χρήστες του κελύφους συνδυάζουν πολλές εντολές από πολλές γλώσσες στα προγράμματα τους, έχουν περιορισμένες επιλογές για την ενσωμάτωση παραλληλισμού, και εξαρτώνται από εργαλεία όπως το GNU `parallel` [79], το `qsub` [21] και το SLURM [31], ή πρέπει να χρησιμοποιούν τελεστές και εντολές όπως `&` και `wait` [24]. Αυτή η προσέγγιση είναι χρονοβόρα, μη γενική, και επιρρεπής σε σφάλματα.

Δεύτερον, το κέλυφος είναι υπερβολικά αυθαίρετο, δυναμικό, και δυσνόητο. Είναι υπερβολικά αυθαίρετο, αφού περιέχει μια γλώσσα που μπορεί να χρησιμοποιηθεί για τη σύνθεση αυθαίρετων εντολών, γραμμένων σε αυθαίρετες γλώσσες, και με αυθαίρετες συμπεριφορές. Είναι υπερβολικά δυναμικό, καθώς η εκτέλεσή του εξαρτάται από μια ποικιλία δυναμικών παραγόντων—όπως η κατάσταση του συστήματος αρχείων και οι τιμές των μεταβλητών περιβάλλοντος. Είναι υπερβολικά δυσνόητο, αφού η

[1] Για παράδειγμα, η εντολή `sed s/A/B/g file.txt` είναι ισχυρά παραλληλοποιήσιμη, καθώς επεξεργάζεται κάθε γραμμή ανεξάρτητα και την γράφει στην τυπική έξοδο. Ωστόσο, η προσθήκη της σημαίας `-i` στην εντολή `sed` την κάνει να επεξεργαστεί το αρχείο `file.txt` επιτόπου, κάτι που απαιτεί καταγραφή εσωτερικής κατάστασης.



Σχήμα 2.1: Παράδειγμα μετατροπής γράφου ροής δεδομένων από το PaSh. (1) Παραλληλοποίηση εντολών `tr`; (2) Απαλοιφή περιττού ζεύγους `cat-split`; (3) Παραλληλοποίηση πρώτου `cat`; (4) Χρήση `sort` με συγχώνευση αντί για `cat`.

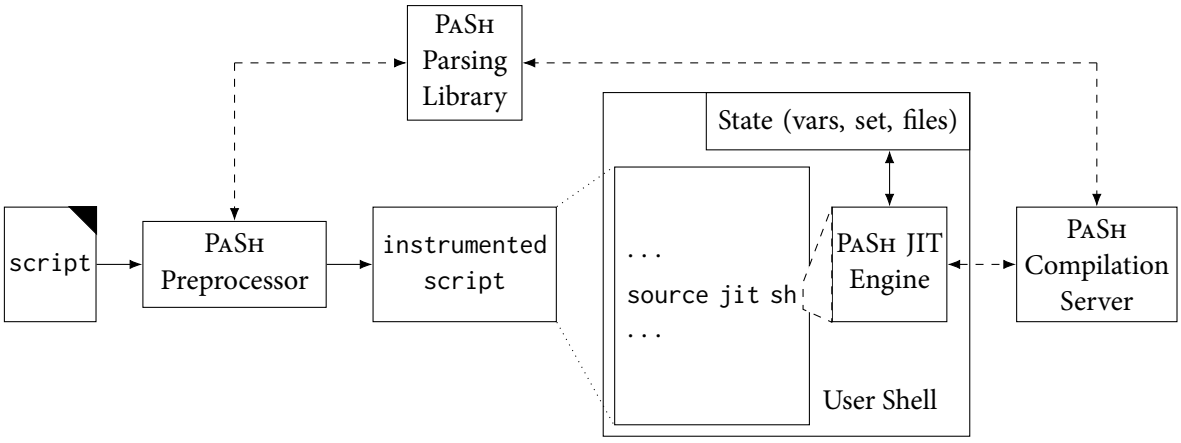
σημασιολογία του κελύφους είναι περίπλοκη—και υπάρχουν πολλές διαφορετικές υλοποιήσεις. Όλοι αυτοί οι παράγοντες εμποδίζουν γενικές προσεγγίσεις στο κέλυφος.

Τρίτον, η ανάπτυξη προγραμμάτων κελύφους είναι επιρρεπής σε σφάλματα, μη-αναμενόμενη, και με πολλές παρενέργειες. Για παράδειγμα, η εντολή `sudo rm -rf $DIR/` θα διαγράψει ολόκληρο το σύστημα αρχείων του χρήστη αν η μεταβλητή `$DIR` είναι κενή—αντί να επιστρέψει σφάλμα κενής μεταβλητής. Το κέλυφος είναι γεμάτο με τέτοιου είδους παγίδες, καθιστώντας την ανάπτυξη εκνευριστική.

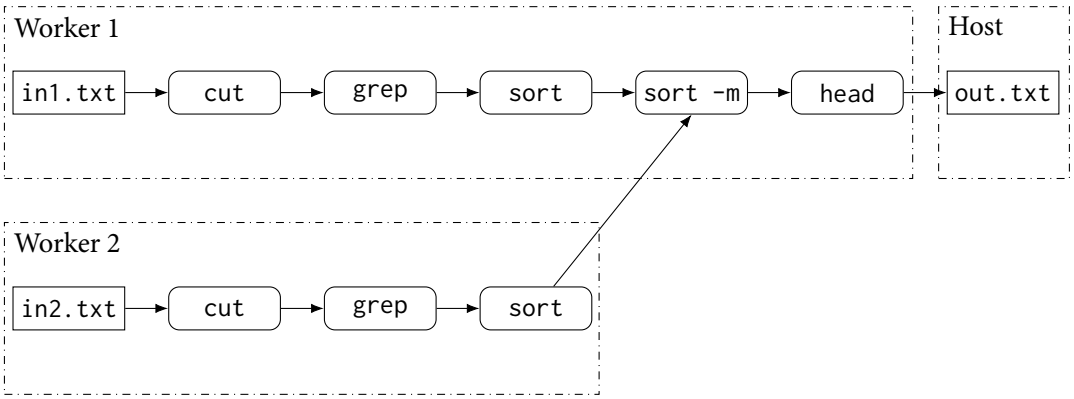
2.4 Αυτόματη Κλιμάκωση Προγραμμάτων Κελύφους

Με σκοπό την επιτάχυνση των προγραμμάτων κελύφους χωρίς τα μειονεκτήματά του, έχουν αναπτυχθεί αρκετά συστήματα για την αυτόματη κλιμάκωση του κελύφους [34, 49, 54, 82]. Από αυτά, το PaSh και το DiSh στοχεύουν στην αυτόματη παραλληλοποίηση και κατανομή προγραμμάτων κελύφους, αντίστοιχα.

PaSh Το PaSh είναι ένα σύστημα που παίρνει ένα πρόγραμμα Unix ως είσοδο και το εκτελεί με παράλληλο τρόπο. Αυτό γίνεται κωδικοποιώντας πληροφορίες παραλληλοποίησης για διάφορους τύπους εντολών σε ένα σύνολο από επισημειώσεις εντολών. Για να αντιμετωπίσει τη δυναμική φύση του κελύφους, το PaSh εκτελείται πάνω-στην-ώρα, εναλλάσσοντας μεταξύ μεταγλώττισης και εκτέλεσης, προκειμένου να συγκεντρώσει τις πιο πρόσφατες πληροφορίες για την κατάσταση του κελύφους.



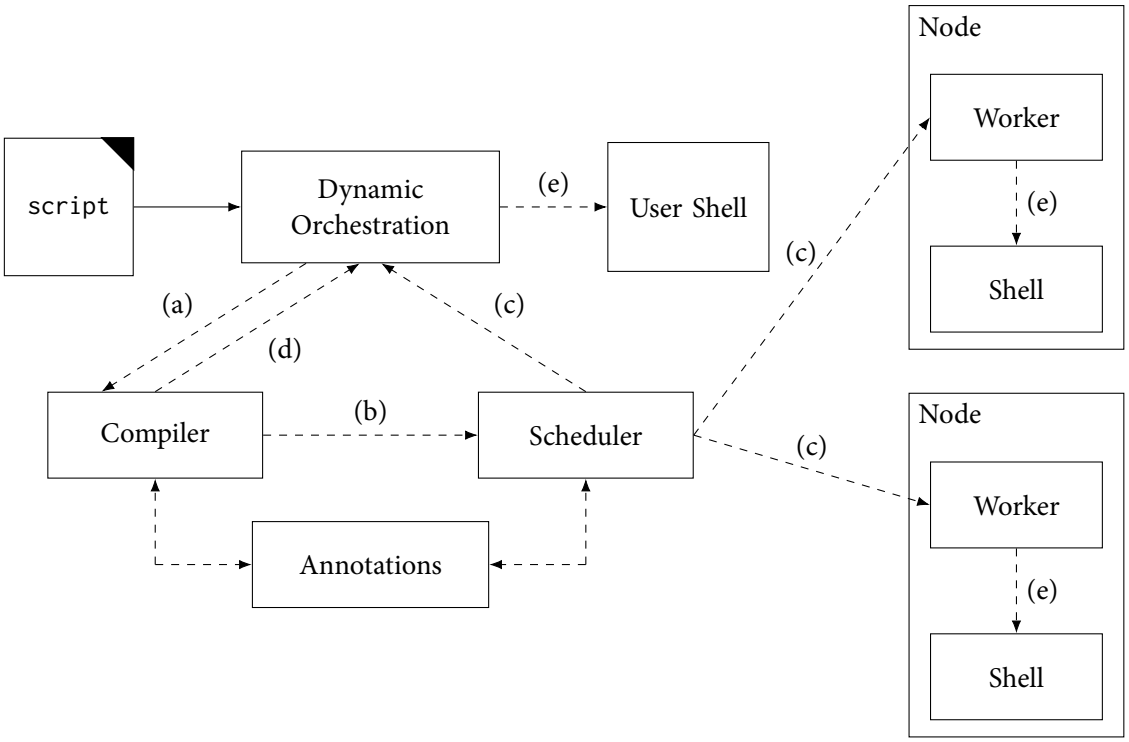
Σχήμα 2.2: Επισκόπηση του PaSh. Το PaSh επεξεργάζεται τα σενάρια με κλήσεις στη μηχανή πάνω-στην-ώρα (PaSh JIT engine), η οποία περνά τα αποσπάσματα προγραμμάτων στον διακομιστή μεταγλώττισης (PaSh compilation server) κατά την εκτέλεση.



Σχήμα 2.3: Υπογράφοι του DiSh. Κάθε μηχάνημα λειτουργεί σε διαφορετικό τμήμα των δεδομένων (`in1.txt` και `in2.txt`), και το τελικό αποτέλεσμα συγκεντρώνεται και αποστέλλεται στο κεντρικό μηχάνημα (host).

Το PaSh αναγνωρίζει δυνητικά παραλληλοποιήσιμες περιοχές του προγράμματος και τις μετατρέπει σε γράφους ροής δεδομένων, όπου κάθε εντολή αντιστοιχεί σε έναν κόμβο και κάθε ροή αντιστοιχεί σε ένα ακμή. Το PaSh εφαρμόζει μετασχηματισμούς παραλληλοποίησης γράφου ροής δεδομένων για να αποκαλύψει την παραλληλία στο γράφο (Σχήμα 2.1)—με τον βαθμό παραλληλίας να είναι δυναμικά ρυθμιζόμενος από τη σημαία `--width` (πλάτος). Το PaSh διασπά εντολές σε συναρτήσεις παράλληλης επεξεργασίας δεδομένων και συναρτήσεις συνάθροισης. Για να το κάνει αυτό, εισάγει ένα σύνολο κατηγοριών παραλληλοποίησης, που εκφράζονται μέσω μιας γλώσσας επισημειώσεων. Το PaSh περιλαμβάνει σημαίες και παραμέτρους εντολών στις επισημειώσεις του—καθώς οι σημαίες και οι παράμετροι μπορούν να αλλάξουν την παραλληλοποιησιμότητα μιας εντολής (Ενότητα 2.3). Αφού εφαρμόσει τις απαραίτητες μετατροπές, το PaSh μετατρέπει κάθε γράφο ροής δεδομένων ξανά πίσω σε μια περιοχή προγράμματος κελύφους. Το PaSh προσθέτει κλασικές εντολές κελύφους Unix (π.χ., `&`, `wait`) σε κάθε περιοχή—προκειμένου να καθοδηγήσει ρητά την παραλληλία—και χρησιμοποιεί κατάλληλα ονομασμένους αγωγούς—προκειμένου να χειριστεί τις εισόδους και εξόδους του κάθε παράλληλου κομματιού δεδομένων, ουσιαστικά ορίζοντας τη δομή του γράφου ροής δεδομένων. Το Σχήμα 2.2 δείχνει μια επισκόπηση του PaSh.

DiSh Το DiSh είναι ένα σύστημα που κατανέμει αυτόματα προγράμματα κελύφους σε πολλαπλά μηχανήματα. Εκτός από λόγους επίδοσης, το DiSh είναι χρήσιμο για μεγάλα σύνολα δεδομένων που μπορεί να μην χωρούν σε ένα μηχάνημα—ή για περιπτώσεις χρήσης που είναι από τη φύση τους κα-



Σχήμα 2.4: Επισκόπηση του DiSh. Βήματα: (a) μεταγλώττιση περιοχής του προγράμματος, (b) δρομολόγηση μεταγλωττισμένου γράφου ροής δεδομένων, (c) αποστολή υπογράφων στα μηχανήματα (nodes), (d) η μεταγλώττιση απέτυχε, επιστροφή στην αρχική έκδοση της περιοχής του προγράμματος, (e) εκτέλεση περιοχής του προγράμματος (μεταγλωττισμένη ή αρχική).

τανεμημένες μεταξύ πολλαπλών υπολογιστών. Όπως και στην περίπτωση του PaSh, το DiSh χρησιμοποιεί επισημειώσεις εντολών, και λειτουργεί πάνω-στην-ώρα. Ο μεταγλωττιστής του DiSh παίρνει ως είσοδο ένα πρόγραμμα, και το μετατρέπει σε γράφο ροής δεδομένων—με βάση πληροφορίες που παίρνει από τις επισημειώσεις εντολών. Επιπρόσθετα από το PaSh, το DiSh χωρίζει τον γράφο ροής δεδομένων σε υπογράφους, για εκτέλεση σε διαφορετικά μηχανήματα (Σχήμα 2.3). Ύστερα από την επεξεργασία του γράφου ροής δεδομένων, το DiSh στέλνει τον γράφο ροής δεδομένων στον δρομολογητή, που προσπαθεί να αντιστοιχίσει τους υπογράφους σε μηχανήματα με βάση την τοπικότητα δεδομένων—για να μειώσει τη μεταφορά δεδομένων μεταξύ μηχανημάτων. Για την επικοινωνία μεταξύ μηχανημάτων, το DiSh εισάγει κανάλια επικοινωνίας αγωγών, για να αντικαταστήσει τους κλασικούς Unix αγωγούς που διασχίζουν τα σύνορα μεταξύ μηχανημάτων—καθώς οι κλασικοί Unix αγωγοί δεν υποστηρίζουν δικτυακή επικοινωνία, και επομένως δεν μπορούν να ξεπεράσουν τα όρια των μηχανημάτων. Το Σχήμα 2.4 δείχνει μια επισκόπηση του DiSh.

Κεφάλαιο 3

Παράδειγμα και Επισκόπηση

Το SPLaSh παίρνει ένα πρόγραμμα κελύφους ως είσοδο, το μετατρέπει αυτόματα για εκτέλεση σε περιβάλλον χωρίς διακομιστή και το εκτελεί στο νέφος. Πρώτα, μετατρέπει το πρόγραμμα κελύφους κατάλληλα για να εισάγει παραλληλία. Στη συνέχεια, χωρίζει το πρόγραμμα σε κομμάτια, ώστε κάθε κομμάτι να εκτελεστεί σε μία διαφορετική συνάρτηση και εισάγει ειδικές εντολές για υπολογισμό χωρίς διακομιστή, όπως κλήση και επικοινωνία συναρτήσεων. Τέλος, αναπτύσσει το τροποποιημένο πρόγραμμα κελύφους στο νέφος, με τρόπο πάνω-στην-ώρα.

3.1 Επιτάχυνση Αντιστοίχισης Κανονικών Εκφράσεων

Το NFA-Regex (Κώδικας 3.1) είναι ένα πρόγραμμα κελύφους που ταιριάζει μία πολύπλοκη κανονική έκφραση σε κάθε γραμμή της εισόδου.[1] Πολύπλοκες κανονικές εκφράσεις όπως αυτή είναι χρήσιμες σε εφαρμογές όπως η αλληλούχιση DNA, αλλά μπορεί να γίνουν υπολογιστικά απαιτητικές [7, 35, 44]. Ευτυχώς, το NFA-Regex έχει πολλές δυνατότητες για επιτάχυνση. Πρώτον, είναι έντονα παραλληλοποιήσιμο—όπως φαίνεται στο Σχήμα 3.1—καθώς οι εντολές `tr` και `grep` λειτουργούν σε μεμονωμένες γραμμές, και το αποτέλεσμα κάθε γραμμής είναι ανεξάρτητο από τις υπόλοιπες γραμμές. Δεύτερον, κάνει έντονη χρήση του επεξεργαστή, καθώς το ταίριασμα της πολύπλοκης κανονικής έκφρασης περιλαμβάνει πολλά πισωγυρίσματα.

Δυστυχώς, τα υπάρχοντα συστήματα για την επιτάχυνση των προγραμμάτων κελύφους δεν μπορούν να αξιοποιήσουν πλήρως τη δυνατότητα του προγράμματος για κλιμάκωση, αφού έχουν έναν στατικό—και συχνά περιορισμένο—αριθμό πόρων στη διάθεσή τους. Το PaSh περιορίζεται στον αριθμό των πυρήνων ενός μηχανήματος, ενώ το DiSh έχει πρόσβαση σε έναν σταθερό αριθμό μηχανημάτων. Ωστόσο, χρησιμοποιώντας υπολογισμό χωρίς διακομιστή, ο χρήστης μπορεί να δημιουργήσει γρήγορα εκατοντάδες συναρτήσεις σε λίγα δευτερόλεπτα—αξιοποιώντας έναν τεράστιο αριθμό μνήμης και υπολογιστικών πόρων—και να τις τερματίσει αυτόματα μόλις ολοκληρωθεί η εκτέλεση.

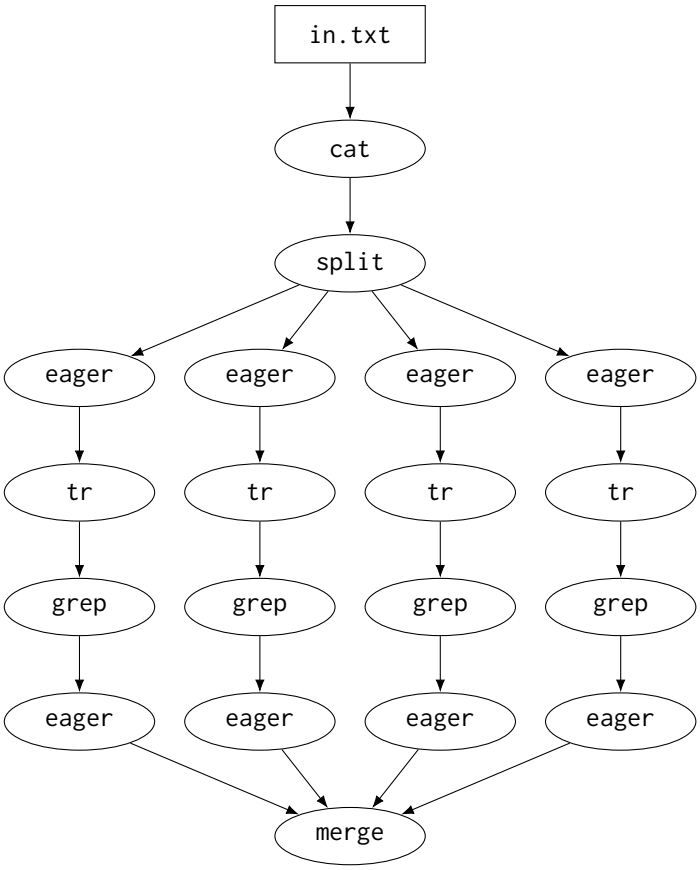
3.2 Κύριες Προκλήσεις

Παρ' όλ' αυτά, η ανάπτυξη εφαρμογών χωρίς διακομιστή είναι πολύ διαφορετική από την ανάπτυξη προγραμμάτων κελύφους, οπότε η εκμετάλλευση του υπολογισμού χωρίς διακομιστή, με ταυ-

```
1 #!/bin/bash
2 cat "in.txt" | tr A-Z a-z | grep '\(.\).*\1\(.\).*\2\(.\).*\3\(.\).*\4'
```

Κώδικας 3.1: Πρόγραμμα NFA-Regex. Αυτό το πρόγραμμα είναι έντονα παραλληλοποιήσιμο, επειδή οι εντολές `tr` και `grep` λειτουργούν ανεξάρτητα σε κάθε γραμμή. Κάνει επίσης έντονη χρήση του επεξεργαστή, λόγω της πολύπλοκης κανονικής έκφρασης.

[1] Η κανονική έκφραση ταιριάζει γραμμές όπου τέσσερεις χαρακτήρες εμφανίζονται δύο φορές, με κάθε χαρακτήρα πιθανώς να ακολουθείται έναν ή περισσότερους χαρακτήρες πριν εμφανιστεί ξανά (π.χ., AqweABrtyBCuioCDpsfD)



Σχήμα 3.1: Ο γράφος ροής δεδομένων που παράγεται από το PaSh για το NFA-Regex (`--width=4`). Οι εντολές `eager` προστίθενται από το PaSh, προκειμένου να αντιμετωπιστεί η οκνηρία του κελύφους (Κώδικας 2.1).

τόχρονη διατήρηση των πλεονεκτημάτων του κελύφους, δημιουργεί ένα σύνολο προκλήσεων:

- Ενώ το κέλυφος είναι έντονα εκφραστικό—υποστηρίζοντας με έναν σχεδόν δηλωτικό τρόπο την επικοινωνία, τον συγχρονισμό, τη δρομολόγηση, και την αλληλεπίδραση με το σύστημα αρχείων—ο υπολογισμός χωρίς διακομιστή επιβάλλει αρκετούς περιορισμούς στην εκφραστικότητα. Για να μεταφράσουν ένα πρόγραμμα φλοιού σε μια εκδοχή που κλιμακώνεται και εκτελείται χωρίς διακομιστή, οι χρήστες πρέπει να εκφράσουν ρητά τα μοτίβα κλήσης και επικοινωνίας των συναρτήσεων, καθώς και την αλληλεπίδραση με την αποθήκη νέφους και τις ουρές μηνυμάτων. Αυτή η χειρωνακτική προσέγγιση είναι χρονοβόρα, επιρρεπής σε σφάλματα, και επιβαρύνει τους προγραμματιστές.

- Ενώ το κέλυφος απολαμβάνει τα οφέλη του παραλληλισμού λόγω της επεξεργασίας ροής δεδομένων, οι συναρτήσεις δεν επιτρέπεται να εγκαθιστούν άμεσες ροές δικτύου μεταξύ τους, επειδή οι συναρτήσεις έχουν περιορισμένες ιδιότητες συνδεσιμότητας δικτύου (Ενότητα 2.2). Τα υπάρχοντα συστήματα για την κλιμάκωση του κελύφους [34, 49] βασίζονται επίσης στην επικοινωνία ροής για να εφαρμόσουν βελτιστοποιήσεις που οδηγούν σε υψηλότερη αξιοποίηση των πόρων.[2]

- Τα προγράμματα κελύφους είναι εγγενώς δυναμικά, ενώ οι συναρτήσεις ρυθμίζονται στατικά εκ των προτέρων. Αυτό σημαίνει ότι η προσπάθεια του υπολογισμού χωρίς διακομιστή να συμβαδίζει με τη δυναμικότητα του κελύφους απαιτεί συνεχή αναδιαμόρφωση των συναρτήσεων κατά τη διάρκεια της εκτέλεσης—πράγμα που εισάγει επιπλέον καθυστερήσεις.

[2] Για παράδειγμα, το PaSh επιτρέπει την αποστολή πακέτων εναλλάξ σε κάθε εντολή, έτσι ώστε όλες οι εντολές σε ένα παράλληλο στάδιο να προμηθεύονται με δεδομένα για επεξεργασία το συντομότερο δυνατόν.

- Ενώ το κέλυφος υποστηρίζεται απρόσκοπτα από το Unix και είναι προεγκατεστημένο σε όλες τις διανομές του, στον υπολογισμό χωρίς διακομιστή το κέλυφος δεν υποστηρίζεται το ίδιο καλά σε σχέση με άλλες γλώσσες υψηλού επιπέδου [66]. Ακόμα και η εκτέλεση *ενός* μόνο προγράμματος κελύφους σε *μια* συνάρτηση απαιτεί τη δημιουργία ενός ειδικού περιβάλλοντος εκτέλεσης [64]—ή την εύρεση λύσεων συγκεκριμένων στην εκάστοτε πλατφόρμα. Αυτή η επιπλέον επιβάρυνση αποθαρρύνει την ανάπτυξη κελύφους σε περιβάλλον χωρίς διακομιστή, καθώς οι χρήστες πρέπει να αναλάβουν τις ίδιες εργασίες που προσπαθούν να αποφύγουν εξαρχής—όπως τη διαχείριση του λειτουργικού συστήματος ή των λεπτομερειών της εκάστοτε πλατφόρμας.

3.3 Επισκόπηση του SPLaSh

Για να αντιμετωπίσει τις προαναφερθείσες προκλήσεις, το SPLaSh:

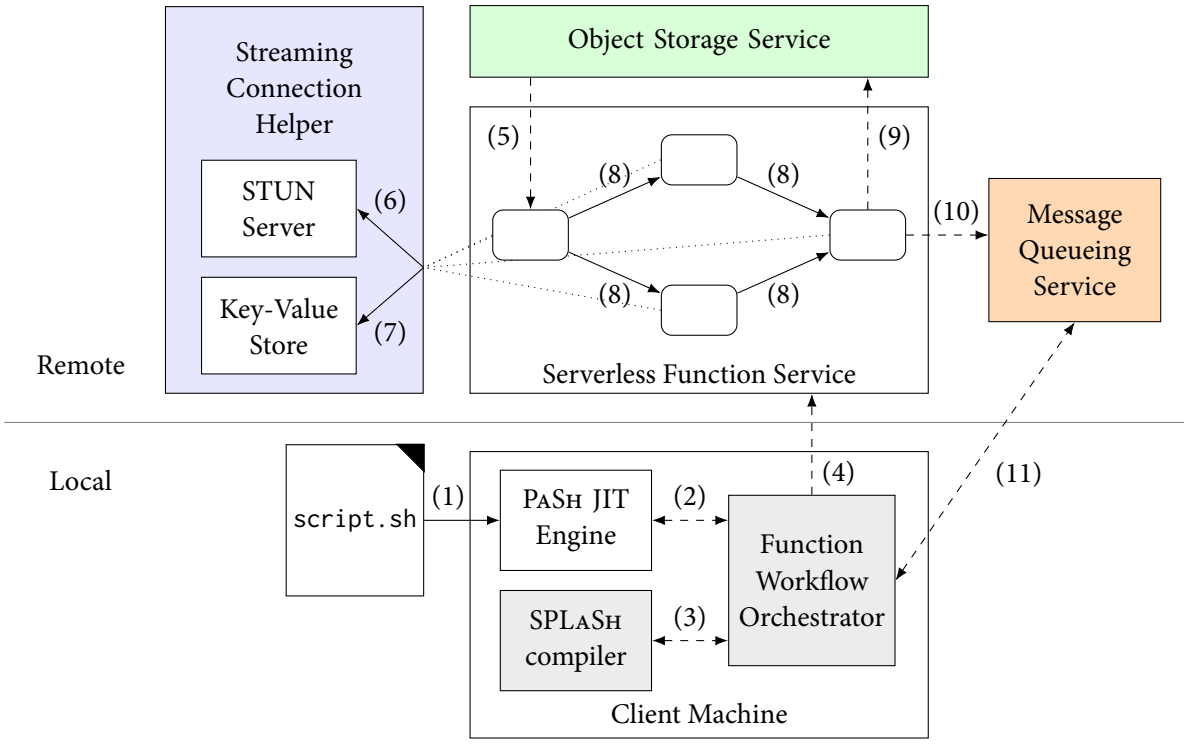
- Αυτοματοποιεί τον εξοπλισμό του αρχικού προγράμματος κελύφους με νέες εντολές για εκτέλεση χωρίς διακομιστή, δηλαδή για την κλήση συναρτήσεων, την επικοινωνία μεταξύ τους, τη διαχείριση αποθήκευσης νέφους, και την αλληλεπίδραση με ουρές μηνυμάτων. Παρέχει επίσης την απαραίτητη υποστήριξη εκτέλεσης για αυτές τις εντολές.

- Ξεπερνά τους περιορισμούς δικτύωσης του υπολογισμού χωρίς διακομιστή, παρέχοντας μηχανισμούς ανακάλυψης και επικοινωνίας που επιτρέπουν στις συναρτήσεις να δημιουργήσουν κανάλια ροής δεδομένων και να εκμεταλλευτούν τα οφέλη της εκτέλεσης ροής.

- Παρέχει ένα σύστημα εκτέλεσης χωρίς διακομιστή για το κέλυφος, που συνδυάζεται με μία μηχανή μεταγλώττισης για υπολογισμό χωρίς διακομιστή που λειτουργεί πάνω-στην-ώρα, προκειμένου να εξυψώσει τη δυναμική φύση του κελύφους και να παρέχει κλιμάκωση του σε περιβάλλον χωρίς διακομιστή, με ένα κλικ.

Το Σχήμα 3.2 δείχνει την επισκόπηση του SPLaSh. Αρχικά, το πρόγραμμα κελύφους τροφοδοτείται στη μηχανή PaSh πάνω-στην-ώρα (PaSh JIT engine), η οποία παράγει έναν γράφο ροής δεδομένων που περιέχει παραλληλία. Ο γράφος ροής δεδομένων περνάει στον συντονιστή ροής συναρτήσεων (function workflow orchestrator), ο οποίος καλεί τον μεταγλωττιστή του SPLaSh (SPLaSh compiler). Ο μεταγλωττιστής του SPLaSh διαιρεί τον γράφο ροής δεδομένων σε υπογράφους, προσθέτει απαραίτητες εντολές και μεταδεδομένα για εκτέλεση χωρίς διακομιστή, και μετατρέπει κάθε υπογράφο πάλι σε πρόγραμμα κελύφους. Στη συνέχεια, περνάει όλα τα προγράμματα κελύφους πίσω στον συντονιστή ροής συναρτήσεων, ο οποίος καλεί την πρώτη συνάρτηση. Μετά από αυτό, κάθε συνάρτηση καλεί τις επόμενες συναρτήσεις.

Για να ανακαλύπτουν οι συναρτήσεις η μία την άλλη και να δημιουργούν συνδέσεις, επικοινωνούν με έναν STUN διακομιστή (STUN server) που παρέχει τις απαραίτητες πληροφορίες διεύθυνσης, τις οποίες οι συναρτήσεις ύστερα γράφουν σε ένα αποθετήριο κλειδιών-τιμών (key-value store), για να είναι ορατές από τις υπόλοιπες συναρτήσεις. Οι συναρτήσεις ερωτούν το αποθετήριο κλειδιών-τιμών για να πάρουν τις πληροφορίες διεύθυνσης των επόμενων συναρτήσεων και δημιουργούν συνδέσεις ροής μαζί τους. Οι συναρτήσεις επεξεργάζονται τις εισόδους τους καθώς έρχονται, και στέλνουν τις εξόδους τους καθώς παράγονται. Η πρώτη συνάρτηση κατεβάζει τις αρχικές εισόδους από την αποθήκη αντικειμένων (object storage service) πριν αρχίσει την επεξεργασία, και η τελευταία συνάρτηση ανεβάζει τις τελικές εξόδους στην αποθήκη αντικειμένων μόλις ολοκληρώσει την επεξεργασία. Επειδή η κλήσεις είναι ασύγχρονες, η τελική συνάρτηση ενημερώνει το μηχάνημα-πελάτη (client machine) για την ολοκλήρωση της εκτέλεσης, μέσω μιας υπηρεσίας ουράς μηνυμάτων (message queueing service). Το μηχάνημα-πελάτης παρακολουθεί την αντίστοιχη ουρά μέχρι να λάβει αυτό το μήνυμα, και μετά επιστρέφει τον έλεγχο στη μηχανή PaSh πάνω-στην-ώρα, και ολόκληρη η διαδικασία επαναλαμβάνεται για το επόμενο μέρος του προγράμματος κελύφους.



Σχήμα 3.2: Επισκόπηση του SPLaSh. Βήματα: (1) ανάγνωση προγράμματος εισόδου, (2) δημιουργία παράλληλου γράφου ροής δεδομένων, (3) διαίρεση του γράφου ροής δεδομένων σε υπογράφους και προσθήκη εντολών για εκτέλεση χωρίς διακομιστή, (4) κλήση της πρώτης συνάρτησης, (5) λήψη αρχικών εισόδων, (6) λήψη ίδιων πληροφοριών διεύθυνσης, (7) εγγραφή ίδιων πληροφοριών διεύθυνσης / ανάγνωση πληροφοριών διεύθυνσης επόμενων συναρτήσεων, (8) κλήση επόμενων συναρτήσεων και αποστολή δεδομένων, (9) μεταφόρτωση τελικών εξόδων, (10) αποστολή μηνύματος ολοκλήρωσης, (11) λήψη μηνύματος ολοκλήρωσης.

Αποτέλεσμα Το SPLaSh μειώνει τον χρόνο εκτέλεσης του NFA-Regex από 10 λεπτά σε 42 δευτερόλεπτα (επιτάχυνση 14.3×)—χωρίς να απαιτεί καμία τροποποίηση στο αρχικό πρόγραμμα.

Κεφάλαιο 4

Μεταγλώττιση

Το SPLaSh παρέχει την υποστήριξη μεταγλώττισης που απαιτείται για την αυτόματη εκτέλεση κελύφους χωρίς διακομιστή. Επεκτείνει τον μεταγλωττιστή του PaSh, διαιρώντας κάθε γράφο ροής δεδομένων σε υπογράφους και αναθέτοντάς τους υπογράφους σε πολλαπλές συναρτήσεις, αλλά και εμπλουτίζοντας κάθε γράφο ροής δεδομένων με ειδικές εντολές για εκτέλεση χωρίς διακομιστή, όπως κλήση συναρτήσεων, επικοινωνία ροής δεδομένων, διαχείριση αποθήκευσης νέφους και αλληλεπίδραση με ουρές μηνυμάτων. Κάθε ενότητα σε αυτό το κεφάλαιο εμπλουτίζει σταδιακά το πρόγραμμα που παράγεται από το PaSh για το NFA-Regex (Κώδικας 4.1) με επιπλέον χαρακτηριστικά για εκτέλεση χωρίς διακομιστή.[1] Τα παρακάτω παραδείγματα χρησιμοποιούν `--width=2` για απλότητα.

```
1  #!/bin/bash
2  mkfifo f{0..10}
3  cat in.txt >f0 &
4  split f0 f1 f2 &
5  eager <f1 >f3 &
6  eager <f2 >f4 &
7  tr A-Z a-z <f3 >f5 &
8  tr A-Z a-z <f4 >f6 &
9  grep '\(.\).*\1\(.\).*\2\(.\).*\3\(.\).*\4' <f5 >f7 &
10 grep '\(.\).*\1\(.\).*\2\(.\).*\3\(.\).*\4' <f6 >f8 &
11 eager <f7 >f9 &
12 eager <f8 >f10 &
13 merge f9 f10 >out.txt &
14 wait
```

Κώδικας 4.1: Παράλληλο πρόγραμμα που παράγεται από το PaSh για το NFA-Regex (`--width=2`). Παρατηρήστε τα ζευγάρια παράλληλων εντολών στις γραμμές 5-6, 7-8, 9-10, και 11-12.

```
1  #!/bin/bash
2  mkfifo f{0..2}
3  cat in.txt >f0 &
4  split f0 f1 f2 &
5  wait
```

Κώδικας 4.2: Πρόγραμμα κελύφους για τον Υπογράφο 1.

[1] Το πρόγραμμα που παράγεται από το PaSh έχει τροποποιηθεί για λόγους απλότητας. Ο Κώδικας A.1 δείχνει την πλήρη, μη τροποποιημένη έκδοση του παραγόμενου προγράμματος.

```
#!/bin/bash
mkfifo f{0..4}
eager <f0 >f1 &
tr A-Z a-z <f1 >f2 &
grep '\(.\).*\1\(.\).*\2\(.\).*\3\(.\).*\4' <f2 >f3 &
eager <f3 >f4 &
wait
```

Κώδικας 4.3: Πρόγραμμα κελύφους για τον Υπογράφο 2.

```
#!/bin/bash
mkfifo f{0..4}
eager <f0 >f1 &
tr A-Z a-z <f1 >f2 &
grep '\(.\).*\1\(.\).*\2\(.\).*\3\(.\).*\4' <f2 >f3 &
eager <f3 >f4 &
wait
```

Κώδικας 4.4: Πρόγραμμα κελύφους για τον Υπογράφο 3.

```
#!/bin/bash
mkfifo f{0..1}
merge f0 f1 >out.txt &
wait
```

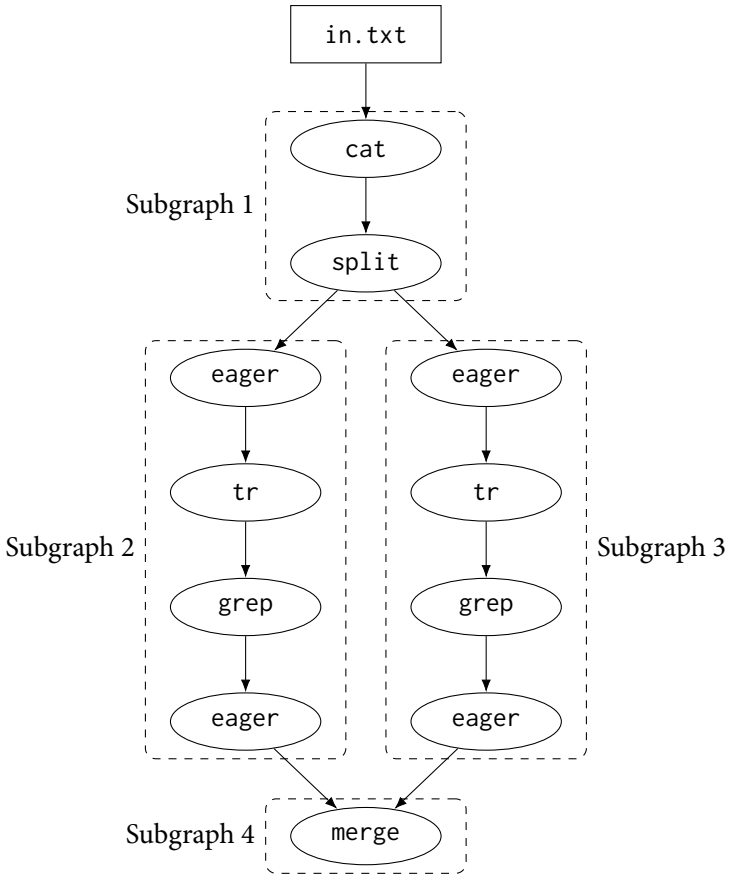
Κώδικας 4.5: Πρόγραμμα κελύφους για τον Υπογράφο 4.

4.1 Ανάθεση Εντολών στις Συναρτήσεις

Στο PaSh, αφού ένα παράλληλο τμήμα κελύφους έχει δημιουργηθεί, ο πυρήνας Unix αναλαμβάνει τη δρομολόγηση στο παρασκήνιο—με βάση τελεστές όπως & και |. Ωστόσο, αυτοί οι τελεστές δεν λειτουργούν για κλιμάκωση σε πλατφόρμες χωρίς διακομιστή, αφού προορίζονται για την περίπτωση ενός μηχανήματος. Το γεγονός αυτό δημιουργεί την ανάγκη ρητού καθορισμού του αριθμού των συναρτήσεων που θα χρησιμοποιηθούν, καθώς και του συνόλου εντολών που θα εκτελέσει κάθε συνάρτηση. Στο ένα άκρο, μία συνάρτηση θα μπορούσε δυνητικά να εκτελέσει το τμήμα κελύφους εξ ολοκλήρου—κάτι το οποίο είναι προφανώς μη κλιμακώσιμο. Στο άλλο άκρο, πολλαπλές συναρτήσεις θα μπορούσαν να εκτελέσουν μόνο μία εντολή η καθεμία, κάτι το οποίο δεν είναι αποδοτικό—καθώς θα εισήγαγε υπερβολική κίνηση δικτύου, καθυστερήσεις εκκίνησης, και κόστη κλήσης.

Το SPLaSh λειτουργεί κάπου ανάμεσα στα δύο άκρα, αποφασίζοντας να χωρίσει τον γράφο στα σημεία όπου εισάγονται παράλληλα στάδια—μέσω των εντολών `split` και `merge`—ενώ ακολουθίες εντολών που δεν διακόπτονται από `split` ή `merge` συγχωνεύονται σε έναν μόνο υπογράφο (Σχήμα 4.1). Αφού το SPLaSh χωρίσει τον γράφο σε υπογράφους, μπορεί να αρχίσει να προσθέτει ειδικές εντολές για την κλήση συναρτήσεων και την επικοινωνία μεταξύ τους. Οι Κώδικες 4.2 έως 4.5 δείχνουν τα προγράμματα κελύφους που παράγονται για κάθε υπογράφο.[2] Το SPLaSh προσθέτει επίσης τις εντολές

[2] Οι προσεκτικοί αναγνώστες μπορεί να παρατηρήσουν ότι ορισμένοι υπογράφοι έχουν "ξεκρέμαστους" αγωγούς (δηλαδή αγωγούς των οποίων η έξοδος δεν πηγαίνει πουθενά, όπως οι αγωγοί `f1`, `f2` στον Υπογράφο 1, ή που δεν λαμβάνουν είσοδο από πουθενά, όπως ο αγωγός `f0` στον Υπογράφο 2). Οι ξεκρέμαστοι αγωγοί υποδηλώνουν την ανάγκη για επικοινωνία μεταξύ συναρτήσεων, η οποία θα συζητηθεί στην Ενότητα 4.3.



Σχήμα 4.1: Γράφος ροής δεδομένων μετά τη διαίρεση σε υπογράφους (subgraphs).

```
1 #!/bin/bash
2 mkfifo f{0..2}
3 invoke-function $SCRIPT_ID_2 $SCRIPT_MAP &
4 invoke-function $SCRIPT_ID_3 $SCRIPT_MAP &
5 cat in.txt >f0 &
6 split f0 f1 f2 &
7 wait
```

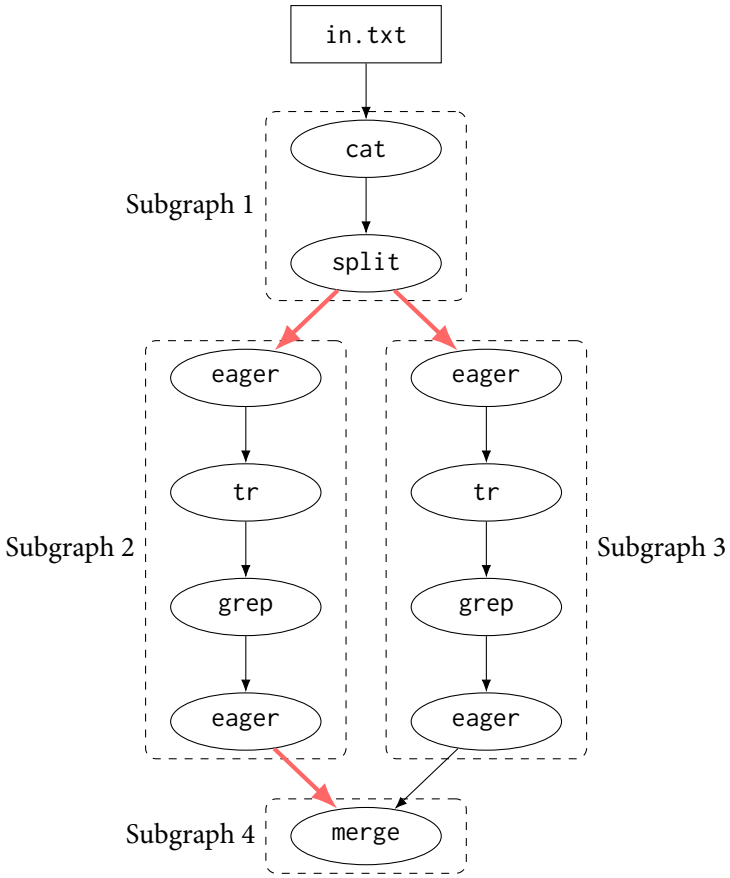
Κώδικας 4.6: Πρόγραμμα κελύφους για τον Υπογράφο 1, μετά την προσθήκη εντολών κλήσης. Ο τελεστής παρασκηνίου (`&`) χρησιμοποιείται για να κληθούν οι συναρτήσεις το συντομότερο δυνατόν.

`mkfifo` και `wait` σε κάθε υπογράφο, αφού τώρα κάθε υπογράφος τρέχει σε ένα διαφορετικό περιβάλλον.[3]

4.2 Εντολές Κλήσης

Στο SPLaSh, μόνο η πρώτη συνάρτηση καλείται από το μηχάνημα-πελάτη. Έπειτα, κάθε συνάρτηση καλεί την επόμενη συνάρτηση (ή τις επόμενες συναρτήσεις) της—η οποία με τη σειρά της καλεί τις επόμενες της συναρτήσεις. Έτσι, το SPLaSh εισάγει μια νέα εντολή (`invoke-function`), η

[3] Για λόγους σαφήνειας και συνέπειας, οι αγωγοί έχουν αριθμηθεί από την αρχή για να ξεκινούν από το μηδέν σε κάθε υπογράφο. Αν και το SPLaSh δεν αριθμεί πραγματικά από την αρχή τους αγωγούς, η αρίθμηση από την αρχή δεν θα προκαλούσε συγκρούσεις ονομάτων, αφού κάθε υπογράφος τρέχει πλέον σε διαφορετική συνάρτηση.



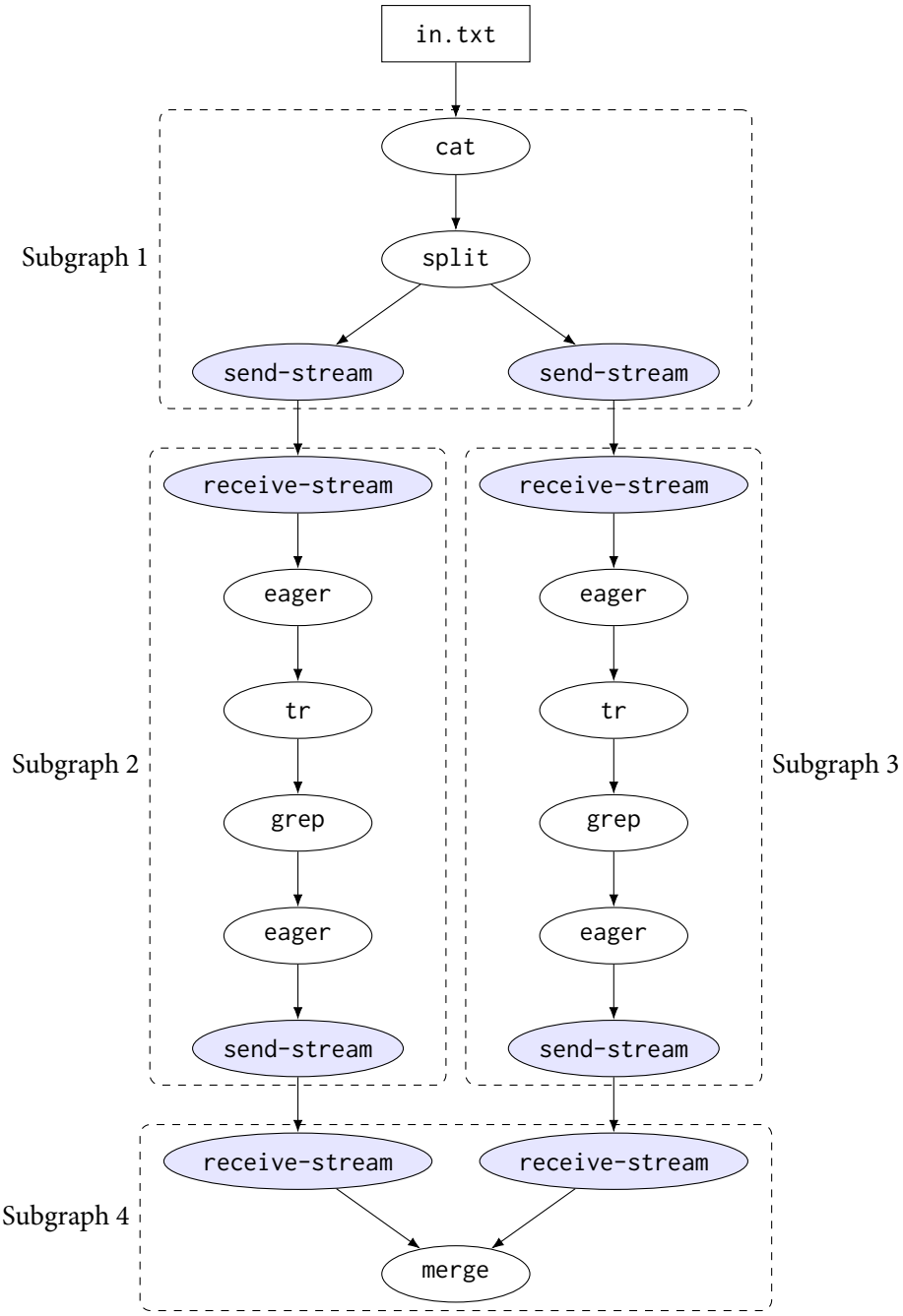
Σχήμα 4.2: Ο γράφος ροής δεδομένων, εμπλουτισμένος με κλήσεις μεταξύ των υπογράφων. Παρατηρήστε ότι ο Υπογράφος 4 πρέπει να κληθεί μόνο από έναν από τους προηγούμενους υπογράφους.

οποία προστίθεται στο πρόγραμμα κελύφους των συναρτήσεων που καλούν άλλες συναρτήσεις. Το Σχήμα 4.2 δείχνει τις κλήσεις που γίνονται στο NFA-Regex. Παρατηρήστε ότι μόνο μία από τις συναρτήσεις που εμπλέκονται σε ένα παράλληλο στάδιο πρέπει να καλέσει τη συνάρτηση που περιέχει την εντολή `merge`.

Το τμήμα προγράμματος κελύφους που πρέπει να εκτελεστεί από κάθε συνάρτηση παρέχεται κατά τη διάρκεια της εκτέλεσης—στο φορτίο κλήσης της συνάρτησης. Αυτό σημαίνει ότι κάθε καλών πρέπει να έχει το πρόγραμμα κελύφους του κληθέντα. Ωστόσο, πρέπει να έχει και το πρόγραμμα κελύφους *του κληθέντα* του κληθέντα—αλλιώς η προηγούμενη ιδιότητα θα έπαυε να ισχύει για τα επόμενα στάδια—και ούτω καθεξής. Για αυτό το λόγο, το SPLASH περιλαμβάνει *όλα* τα τμήματα κελύφους στην μορφή ενός λεξικού που αντιστοιχίζει αναγνωριστικά σε προγράμματα κελύφους (`SCRIPT_MAP`). Ύστερα, κάθε κλήση περιλαμβάνει αυτό το λεξικό στο φορτίο κλήσης, μαζί με το αναγνωριστικό που αντιστοιχεί στο πρόγραμμα του κληθέντα (`SCRIPT_ID`). Η κλήση συναρτήσεων συμβαίνει πρόθυμα—κάθε συνάρτηση καλεί την επόμενη συνάρτηση (ή τις επόμενες συναρτήσεις της) το συντομότερο δυνατόν—ώστε οι κληθέντες να αρχίσουν γρήγορα να λαμβάνουν δεδομένα από τους καλούντες, και να επιτευχθεί υψηλή εκμετάλλευση. Ο Κώδικας 4.6 δείχνει το πρόγραμμα του πρώτου υπογράφου, με τις επιπλέον εντολές κλήσης.

4.3 Εντολές Επικοινωνίας

Η επικοινωνία ροής δεδομένων απαιτεί προσαρμοσμένη υποστήριξη για τον υπολογισμό χωρίς διακομιστή, αφού οι UNIX αγωγοί λειτουργούν στη μνήμη και δεν μπορούν να διασχίσουν τα όρια των συναρτήσεων. Ο μεταγλωττιστής ανιχνεύει πού πρέπει να γίνει αυτή η επικοινωνία, δημιουργεί



Σχήμα 4.3: Ο γράφος ροής δεδομένων, εμπλουτισμένος με κόμβους επικοινωνίας.

εντολές ροής με τα κατάλληλα μεταδεδομένα—ώστε οι συναρτήσεις να μπορούν να βρίσκουν η μία την άλλη—και ενσωματώνει σωστά αυτές τις εντολές στην αρχική ροή δεδομένων. Πιο συγκεκριμένα, το SPLASH εισάγει δύο νέες εντολές (`send-stream` και `receive-stream`), τις οποίες τοποθετεί στα όρια των συναρτήσεων (Σχήμα 4.3). Στην συνάρτηση αποστολής, η εντολή `send-stream` καταναλώνει την έξοδο της τελευταίας εντολής και στέλνει τα δεδομένα μέσω του δικτύου. Στην συνάρτηση λήψης, η εντολή `receive-stream` λαμβάνει τα δεδομένα μέσω του δικτύου και τα παρέχει στην πρώτη εντολή.

Για να ταιριάξει κάθε εντολή `send-stream` με την αντίστοιχη εντολή `receive-stream`—και αντίστροφα—το SPLASH χρησιμοποιεί ένα μοναδικό κλειδί για κάθε ζευγάρι εντολών `send-stream` και `receive-stream` (`COMM_KEY`). Και οι δύο εντολές χρησιμοποιούν αυτό το κλειδί για να βρουν συγκεκριμένες πληροφορίες διεύθυνσης κατά την εκτέλεση—όπως θα περιγραφεί στην Ενότητα 5.3. Οι Κώδικες 4.7 και 4.8 δείχνουν τους πρώτους δύο υπογράφους με τις προσθήκες εντολών επικοινωνίας—παρατηρήστε

```
#!/bin/bash
mkfifo f{0..2}
invoke-function $SCRIPT_ID_2 $SCRIPT_MAP &
invoke-function $SCRIPT_ID_3 $SCRIPT_MAP &
cat in.txt >f0 &
split f0 f1 f2 &
send-stream $COMM_KEY_1_2 <f1 &
send-stream $COMM_KEY_1_3 <f2 &
wait
```

Κώδικας 4.7: Πρόγραμμα για τον Υπογράφο 1, μετά την προσθήκη εντολών επικοινωνίας. Προσέξτε ότι το COMM_KEY_1_2 χρησιμοποιείται επίσης στην εντολή receive-stream, Κώδικας 4.8.

```
#!/bin/bash
mkfifo f{0..4}
invoke-function $SCRIPT_ID_4 $SCRIPT_MAP &
receive-stream $COMM_KEY_1_2 >f0 &
eager <f0 >f1 &
tr A-Z a-z <f1 >f2 &
grep '\(.\).*\1\(.\).*\2\(.\).*\3\(.\).*\4' <f2 >f3 &
eager <f3 >f4 &
send-stream $COMM_KEY_2_4 <f4 &
wait
```

Κώδικας 4.8: Πρόγραμμα για τον Υπογράφο 2, μετά την προσθήκη εντολών επικοινωνίας. Προσέξτε ότι το COMM_KEY_1_2 χρησιμοποιείται επίσης στην εντολή send-stream, Κώδικας 4.7.

τη χρήση του COMM_KEY_1_2 στις εντολές send-stream και receive-stream.

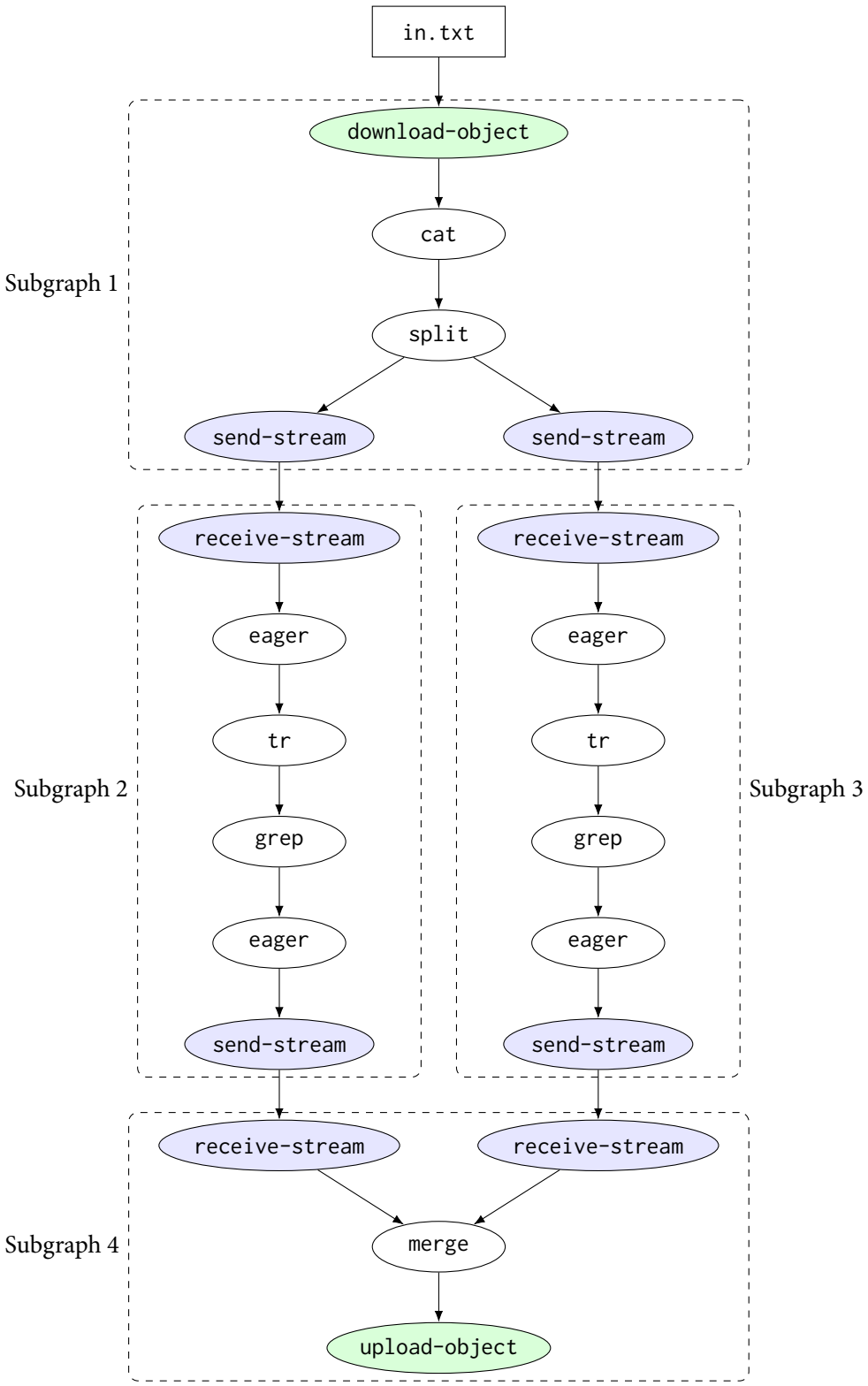
4.4 Εντολές Αποθήκευσης Αντικειμένων

Το SPLASH χρησιμοποιεί αποθήκευση αντικειμένων για την ανάγνωση των αρχικών δεδομένων εισόδου, καθώς και για την εγγραφή των τελικών δεδομένων εξόδου. Για αυτό το λόγο, το SPLASH εισάγει δύο νέες εντολές (download-object και upload-object), που τοποθετούνται πριν από εντολές που διαβάζουν αρχεία, ή μετά από εντολές που γράφουν αρχεία. Και οι δύο εντολές παίρνουν το όνομα του αρχείου ως παράμετρο. Το Σχήμα 4.4 δείχνει πώς φαίνονται αυτοί οι κόμβοι στον γράφο ροής δεδομένων, ενώ οι Κώδικες 4.9 και 4.10 δείχνουν τα προγράμματα του πρώτου και του τελευταίου υπογράφου, μετά την προσθήκη των εντολών αποθήκευσης νέφους. Προσέξτε ότι οι εντολές cat και merge πλέον αλληλεπιδρούν μόνο με επώνυμους αγωγούς—αφού τα ονόματα αρχείων εισόδου και εξόδου χρησιμοποιούνται στις εντολές αποθήκευσης αντικειμένων.

4.5 Εντολές Ουράς Μηνυμάτων

Το SPLASH χρησιμοποιεί ασύγχρονη κλήση συναρτήσεων[4]—οι συναρτήσεις δεν περιμένουν να τελειώσουν οι κληθείσες συναρτήσεις τους—οπότε το μηχάνημα-πελάτης δεν ενημερώνεται αυτόματα όταν η τελευταία συνάρτηση έχει τελειώσει—και έτσι δεν μπορεί να συνεχίσει με ασφάλεια στο

[4] Αυτή η επιλογή θα εξηγηθεί στην Ενότητα 5.2.



Σχήμα 4.4: Ο γράφος ροής δεδομένων, εμπλουτισμένος με κόμβους αποθήκευσης αντικειμένων.

```
#!/bin/bash
mkfifo f{0..3}
invoke-function $SCRIPT_ID_2 $SCRIPT_MAP &
invoke-function $SCRIPT_ID_3 $SCRIPT_MAP &
download-object in.txt >f0 &
cat f0 >f1 &
split f1 f2 f3 &
send-stream $COMM_KEY_1_2 <f2 &
send-stream $COMM_KEY_1_3 <f3 &
wait
```

Κώδικας 4.9: Πρόγραμμα για τον Υπογράφο 1, μετά την προσθήκη εντολών αποθήκευσης αντικειμένων. Προσέξτε ότι η εντολή `cat` πλέον χρησιμοποιεί μόνο Unix αγωγούς.

```
#!/bin/bash
mkfifo f{0..2}
receive-stream $COMM_KEY_2_4 >f0 &
receive-stream $COMM_KEY_3_4 >f1 &
merge f0 f1 >f2 &
upload-object <f2 out.txt &
wait
```

Κώδικας 4.10: Πρόγραμμα για τον Υπογράφο 4, μετά την προσθήκη εντολών αποθήκευσης αντικειμένων. Προσέξτε ότι η εντολή `merge` πλέον χρησιμοποιεί μόνο Unix αγωγούς.

επόμενο μέρος του προγράμματος. Για αυτό το λόγο, το SPLaSh εισάγει ακόμα μία εντολή (`send-notification`), η οποία προστίθεται στο τέλος της τελευταίας συνάρτησης, κάνοντας τη να στείλει μία ειδοποίηση σε μία ουρά μηνυμάτων στην οποία ακούει το μηχάνημα-πελάτης (Κώδικας 4.11). Σημειώστε ότι η εντολή `send-notification` πρέπει να προστεθεί μετά την εντολή `wait`, ώστε να είναι σίγουρο ότι η συνάρτηση έχει τελειώσει όλη την προηγούμενη επεξεργασία. Οι Κώδικες B.1 έως B.4 δείχνουν τα τελικά προγράμματα για κάθε υπογράφο, ύστερα από την προσθήκη όλων των εντολών για εκτέλεση χωρίς διακομιστή—κλήση, επικοινωνία, αποθήκευση, ουρές μηνυμάτων.

```
#!/bin/bash
mkfifo f{0..2}
receive-stream $COMM_KEY_2_4 >f0 &
receive-stream $COMM_KEY_3_4 >f1 &
merge f0 f1 >f2 &
upload-object <f2 out.txt &
wait
send-notification
```

Κώδικας 4.11: Πρόγραμμα για τον Υπογράφο 4, μετά την προσθήκη εντολών ουράς μηνυμάτων. Προσέξτε ότι η εντολή `send-notification` προστίθεται μετά από την εντολή `wait`, ώστε να είναι σίγουρο ότι το υπόλοιπο πρόγραμμα έχει τελειώσει προτού σταλεί η ενημέρωση ολοκλήρωσης.

Κεφάλαιο 5

Εκτέλεση

Το SPLaSh παρέχει επίσης υποστήριξη εκτέλεσης για τις νέες εντολές που εισάγει. Ρυθμίζει τις συναρτήσεις κατάλληλα για εκτέλεση πάνω-στην-ώρα, καλεί τις συναρτήσεις ασύγχρονα, δημιουργεί κανάλια επικοινωνίας με ροή δεδομένων, αλληλεπιδρά με την αποθήκευση νέφους, και αποστέλλει ειδοποιήσεις ολοκλήρωσης στη μηχανή-πελάτη μέσω ουράς μηνυμάτων.

5.1 Εκτέλεση Πάνω-στην-Ώρα

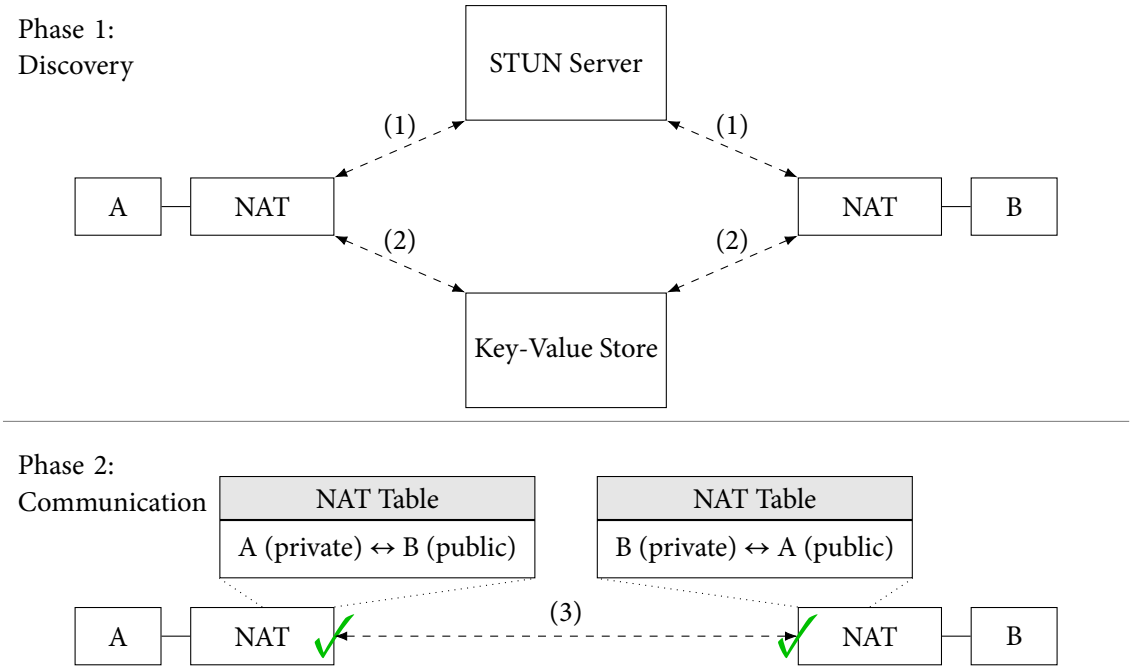
Όπως αναφέρθηκε προηγουμένως, το PaSh χρησιμοποιεί ανάλυση πάνω-στην-ώρα για να εξάγει περισσότερες πληροφορίες από το πρόγραμμα κελύφους, και να αυξήσει την δυνατότητα παραλληλοποίησης. Αυτή η προσέγγιση λειτουργεί καλά σε περιπτώσεις μεμονωμένων μηχανημάτων ή ομάδων μηχανημάτων—όλοι οι πόροι που συμμετέχουν στον υπολογισμό υπάρχουν εκ των προτέρων. Ωστόσο, στην περίπτωση υπολογισμού χωρίς διακομιστή, ο χρήστης πρέπει να ρυθμίσει μία συνάρτηση πριν την καλέσει. Αυτή η φάση ρύθμισης εισάγει κάποια καθυστέρηση, αφού απαιτεί την αποστολή ενός αιτήματος στον πάροχο νέφους, τη μεταφόρτωση των εξαρτήσεων της συνάρτησης, και την αναμονή για απάντηση ότι η συνάρτηση είναι έτοιμη. Αυτή η καθυστέρηση μπορεί να γίνει απαγορευτική αν εισάγεται πριν από κάθε κλήση συνάρτησης, ανάλογα με το μέγεθος της συνάρτησης και την ταχύτητα μεταφόρτωσης του μηχανήματος-πελάτη.

Για αυτό το λόγο, το SPLaSh σχεδιάστηκε έτσι ώστε να απαιτεί μόνο μία ρύθμιση συνάρτησης, που είναι αρκούντως γενική ώστε να εκτελεί διάφορα προγράμματα κελύφους. Αυτό σημαίνει ότι ο χρήστης μπορεί να καθορίσει μόνο μία ρύθμιση συνάρτησης—οποιαδήποτε χρονική στιγμή, πολύ πριν από την εκτέλεση των προγραμμάτων—και να την ενημερώνει μόνο όταν αλλάζουν οι εξαρτήσεις—όπως γίνεται και στην περίπτωση των αντικειμένων λογισμικού, όπου ο κώδικας χρειάζεται ξανά χτίσιμο όταν αλλάξει. Αφού το μόνο που μπορεί να αλλάξει κατά την εκτέλεση είναι το τμήμα του προγράμματος κελύφους που θα εκτελεστεί, το SPLaSh περνάει αυτό το τμήμα ως φορτίο (Ενότητα 4.2). Εκτός από αυτό, η ύπαρξη μίας μοναδικής ρύθμισης συνάρτησης προσφέρει το επιπρόσθετο πλεονέκτημα των *θερμών εκκινήσεων* [67]. Πολλαπλές κλήσεις της ίδιας συνάρτησης προκαλούν την τοποθέτηση του περιβάλλοντος στην κρυφή μνήμη των παρόχων νέφους—οπότε οι επόμενες κλήσεις ξεκινούν πιο γρήγορα.

5.2 Κλήση

Κάθε κλήση συνάρτησης περιλαμβάνει ένα λεξικό τμημάτων προγραμμάτων κελύφους (`SCRIPT_MAP`) και ένα αναγνωριστικό τμήματος προγράμματος κελύφους (`SCRIPT_ID`). Ο χειριστής συνάρτησης χρησιμοποιεί αυτές τις πληροφορίες για να ανακτήσει το κατάλληλο τμήμα προγράμματος κελύφους και να το αποθηκεύσει σε ένα αρχείο—ώστε να μπορεί να το εκτελέσει. Ο χειριστής συνάρτησης αποθηκεύει επιπλέον το `SCRIPT_MAP` σε ένα αρχείο, ώστε το πρόγραμμα κελύφους να μπορέσει αργότερα να το συμπεριλάβει στο φορτίο κλήσης των επόμενων συναρτήσεων (Κώδικας C.2).

Το SPLaSh χρησιμοποιεί ασύγχρονες κλήσεις συναρτήσεων—οι συναρτήσεις δεν περιμένουν να ολοκληρώσουν την εκτέλεση οι κληθείσες συναρτήσεις τους. Αυτό γίνεται έτσι ώστε οι συναρτήσεις



Σχήμα 5.1: Διάτρηση στο πλαίσιο του SPLaSh. Κάθε συνάρτηση (1) λαμβάνει τις *δικές της* πληροφορίες διεύθυνσης, (2) γράφει τις δικές της πληροφορίες διεύθυνσης σε ένα αποθετήριο κλειδιών-τιμών (key-value store), από το οποίο διαβάζει επίσης τις πληροφορίες διεύθυνσης της άλλης συνάρτησης, και (3) προσπαθεί να συνδεθεί με την άλλη συνάρτηση. Δημιουργούνται καταχωρήσεις στους πίνακες μεταφραστών διεύθυνσης δικτύου (NAT tables), οπότε οι συνδέσεις περνάνε επιτυχώς.

να μπορούν να επιστρέψουν αμέσως μετά την ολοκλήρωση της δικής τους εκτέλεσης και να μην χρειάζεται να περιμένουν όλες τις άλλες συναρτήσεις—μειώνοντας έτσι το κόστος και απελευθερώνοντας παράλληλους πόρους. Επίσης, το SPLaSh ρυθμίζει τις συναρτήσεις ώστε να μην προσπαθούν ξανά σε περίπτωση υπέρβασης του χρόνου ολοκλήρωσης, καθώς η αλλεπάλληλη εκτέλεση θα αύξανε σημαντικά το κόστος και θα μείωνε τους παράλληλους πόρους. Ενώ οι παρόχοι νέφους συνήθως θέτουν περιορισμούς στο μέγεθος του φορτίου για ασύγχρονη κλήση, αυτοί οι περιορισμοί κυμαίνονται συνήθως στο εύρος των MB—με τον πιο αυστηρό περιορισμό να είναι 256 kB στην περίπτωση του AWS Lambda [23, 45, 71]. Το μέγεθος του κώδικα κελύφους που παράγει το SPLaSh για κάθε αλυσίδα δεν υπερβαίνει τα όρια αυτά, επομένως αυτοί οι περιορισμοί δεν αποτελούν πρακτικό πρόβλημα.[1]

5.3 Επικοινωνία

Όπως έχει αναφερθεί προηγουμένως, οι συναρτήσεις δεν μπορούν να επιτύχουν άμεση επικοινωνίας ροής δεδομένων μεταξύ τους, και οι εναλλακτικές μέθοδοι επικοινωνίας δεν είναι κατάλληλες για εφαρμογές κελύφους—είναι πολύ δαπανηρές, ενώ δεν υποστηρίζουν αυτόματη κλιμάκωση ή ροή δεδομένων. Για να αντιμετωπίσει αυτήν την πρόκληση, το SPLaSh ξεπερνά τους περιορισμούς δικτύωσης του serverless και δημιουργεί κανάλια άμεσης επικοινωνίας ροής δεδομένων μεταξύ των συναρτήσεων—χωρίς να χρησιμοποιεί επιπλέον υπηρεσίες για ανταλλαγή δεδομένων. Για να εξασφαλίσει αξιόπιστη, διατεταγμένη επικοινωνία—παρόμοια με τους αγωγούς Unix—το SPLaSh χρησιμοποιεί το TCP ως πρωτόκολλο επικοινωνίας.

Για την εγκατάσταση άμεσων καναλιών ροής δεδομένων παρά την παρουσία μεταφραστών διεύθυνσης δικτύου, το SPLaSh χρησιμοποιεί την τεχνική της *διάτρησης* [16]: Έστω ότι τα μηχανήματα A

[1] Σαν σημείο αναφοράς, ολόκληρος ο κώδικας LaTeX που χρειάστηκε για αυτή τη διπλωματική εργασία—κείμενο, πίνακες, και σχήματα—είναι μικρότερος από 256 kB.

και Β θέλουν να εγκαταστήσουν μία σύνδεση μεταξύ τους. Στην κανονική διάτρηση, τα Α και Β συνδέονται σε έναν δημόσια προσβάσιμο ενδιάμεσο διακομιστή, ο οποίος αποκαλύπτει τις πληροφορίες διεύθυνσης του Β στο Α, και τις πληροφίες διεύθυνσης του Α στο Β. Ύστερα, τόσο το Α όσο και το Β εκκινούν μία σύνδεση το ένα με το άλλο, χρησιμοποιώντας αυτές τις νέες πληροφορίες. Αυτό δημιουργεί μία νέα εγγραφή για το Β στον πίνακα μεταφραστή διεύθυνσης δικτύου του Α, και αντίστοιχα μία νέα εγγραφή για το Α στον πίνακα μεταφραστή διεύθυνσης δικτύου του Β, που λειτουργούν σαν "τρύπες", επιτρέποντας τις εισερχόμενες συνδέσεις—εξού και ο όρος "διάτρηση".

Στην πραγματικότητα, το SPLaSh χρησιμοποιεί μια ελαφρώς τροποποιημένη έκδοση της τεχνικής (Σχήμα 5.1). Αντί να χρησιμοποιεί έναν ενδιάμεσο διακομιστή για να λάβει τις πληροφορίες διεύθυνσης της άλλης συνάρτησης, κάθε συνάρτηση ρωτάει έναν διακομιστή STUN (STUN server) [28], ο οποίος επιστρέφει στη συνάρτηση τις *δικές της* πληροφορίες διεύθυνσης. Στη συνέχεια, κάθε συνάρτηση αποθηκεύει τις πληροφορίες διεύθυνσής της σε ένα κοινό αποθετήριο κλειδιών-τιμών, το οποίο μπορούν να ρωτήσουν και οι δύο συναρτήσεις χρησιμοποιώντας ένα κοινό `COMM_KEY`. Έπειτα, η διαδικασία είναι παρόμοια με την κανονική τεχνική της διάτρησης: οι συναρτήσεις εκκινούν συνδέσεις μεταξύ τους, δημιουργούνται καταχωρήσεις στους πίνακες μεταφραστών διεύθυνσης δικτύου και οι συνδέσεις μπορούν να πραγματοποιηθούν επιτυχώς. Να σημειωθεί ότι τόσο ο διακομιστής STUN, όσο και το αποθετήριο κλειδιών-τιμών χρησιμοποιούνται *μόνο* για πληροφορίες διεύθυνσης—*όχι* για ανταλλαγή μεγάλου όγκου δεδομένων—ώστε να μην επηρεάζουν την κλιμακωσιμότητα.

5.4 Αποθήκευση Αντικειμένων

Παρόλο που το SPLaSh χρησιμοποιεί συνδέσεις ροής δεδομένων για την αποστολή ενδιάμεσων δεδομένων, εξακολουθεί να χρειάζεται να διαβάζει από υπάρχοντα δεδομένα και να αποθηκεύει τα αποτελέσματα σε αρχεία (Κώδικες C.5 και C.6). Το SPLaSh επωφελείται περισσότερο από αποθηκευτικούς χώρους αντικειμένων που επιτρέπουν τη λήψη σε ροή δεδομένων, καθώς η επεξεργασία μπορεί να διαδοθεί γρήγορα σε όλη την έκταση του γράφου—επιτυγχάνοντας έτσι υψηλή αξιοποίηση, ακόμα και για μεγάλα αρχεία.

5.5 Ουρά Μηνυμάτων

Επειδή οι κλήσεις είναι ασύγχρονες, το μηχάνημα-πελάτης δεν ενημερώνεται όταν ολοκληρώνεται η τελευταία συνάρτηση. Έτσι, το SPLaSh φροντίζει η τελευταία συνάρτηση να στείλει μια ειδοποίηση σε μια ουρά μηνυμάτων, στην οποία ο πελάτης ακούει. Αφού ο πελάτης λάβει την ειδοποίηση, προχωρά στην επόμενη περιοχή του προγράμματος κελύφους. Στο SPLaSh, ο πελάτης χρησιμοποιεί την τεχνική της *μακράς αναμονής*, για να εκδώσει λιγότερα αιτήματα και να μειώσει το κόστος [62], καθώς ο όγκος των ειδοποιήσεων είναι σχετικά χαμηλός—ένα μήνυμα για κάθε ολοκληρωμένο τμήμα κελύφους.

Κεφάλαιο 6

Αξιολόγηση

Το SPLaSh αξιολογείται στο πόσο γρήγορα εκτελείται σε σχέση με τη σειριακή εκτέλεση κελύφους, καθώς και στο πόσο καλά κλιμακώνει με τον αριθμό των συναρτήσεων.

6.1 Διάταξη

Υλοποίηση Η λειτουργικότητα του SPLaSh είναι χτισμένη πάνω σε αυτή του PaSh. Το SPLaSh χρησιμοποιεί το Serverless Framework [29] για φόρτωση στο AWS, και το boto3 Python AWS SDK [68] για αλληλεπίδραση με τις υπηρεσίες της AWS.[1] Η βιβλιοθήκη διάτρησης είναι υλοποιημένη σε Rust, και φορτώνεται ως επίπεδο AWS Lambda [65]. Το SPLaSh χρησιμοποιεί το AWS Lambda [63] (κλήση συναρτήσεων), το Amazon DynamoDB [59] (αποθετήριο κλειδιών-τιμών), το Amazon S3 [61] (αποθήκη αντικειμένων), και το Amazon SQS [60] (ουρά μηνυμάτων).

Σημείο Αναφοράς Η επίδοση του SPLaSh συγκρίνεται με το GNU Bash [20], το κατεξοχήν σειριακό περιβάλλον εκτέλεσης προγραμμάτων κελύφους. Τόσο το Bash όσο και το SPLaSh εκτελούν κάθε πρόγραμμα κελύφους χωρίς καμία αλλαγή. Αφού αυτή τη στιγμή δεν υπάρχει καμία έτοιμη υποστήριξη για την εκτέλεση προγραμμάτων κελύφους στο AWS Lambda [66], όλα τα πειράματα εκτελούνται στο περιβάλλον εκτέλεσης που προσφέρει το SPLaSh. Για το Bash, μία συνάρτηση εκτελεί ολόκληρο το αρχικό πρόγραμμα φλοιού, ενώ για το SPLaSh, λαμβάνει χώρα όλη η διαδικασία μεταγλώττισης και εκτέλεσης, όπως περιγράφεται στα Κεφάλαια 3 έως 5.

Διάταξη Υλικού Όλες οι εκτελέσεις έχουν τεθεί ώστε (αρχικά) να διαβάζουν και (τελικά) να γράφουν στο Amazon S3. Οι συναρτήσεις έχουν ρυθμιστεί με 10 240 MB μνήμης και 6 εικονικούς επεξεργαστές (vCPU) για το Bash, και με 1769 MB μνήμης και 1 vCPU για το SPLaSh. Αυτά τα μεγέθη μνήμη επιλέγονται για δύο λόγους: πρώτον, το 10 240 MB είναι το μεγαλύτερο μέγεθος μνήμης που είναι διαθέσιμο στο AWS Lambda [72], και δεύτερον, τα 10 240 MB και 1769 MB είναι τα μόνα δύο μεγέθη μνήμης με γνωστή αντιστοιχία μνήμης προς vCPU [70].

6.2 Προγράμματα

Ο Πίνακας 6.1 δείχνει τα σύνολα προγραμμάτων που χρησιμοποιούνται για την αξιολόγηση του SPLaSh.[2] Συνολικά, χρησιμοποιούνται 5 πραγματικά σύνολα προγραμμάτων, που αποτελούνται από 32 προγράμματα κελύφους και 305 γραμμές κώδικα (lines of code, LoC). Τα `Oneliners` και `Unix50` περιέχουν τόσο κλασικά, όσο και πρόσφατα (2019) προγράμματα που κάνουν έντονη χρήση των προεγκαστημένων εντολών του Unix και του Linux. Το `COVID-mts` περιέχει τέσσερα προγράμματα που χρησιμοποιήθηκαν για την ανάλυση πραγματικών δεδομένων τηλεμετρίας από δρομολόγια μέσων μαζικής μεταφοράς στην Αθήνα, κατά τη διάρκεια της πανδημίας του COVID-19.

[1] Να σημειωθεί ότι οι τεχνικές που χρησιμοποιεί το SPLaSh δεν είναι αποκλειστικές στο AWS, αλλά μπορούν να χρησιμοποιηθούν και σε άλλους παρόχους νέφους. Ο Πίνακας D.1 δείχνει τις υπηρεσίες της AWS που χρησιμοποιεί το SPLaSh, καθώς και τις αντίστοιχές τους στο Νέφος Google και στο Microsoft Azure.

[2] Οι Πίνακες E.1 έως E.5 περιέχουν την πλήρη περιγραφή των προγραμμάτων που χρησιμοποιούνται στην αξιολόγηση.

Πίνακας 6.1: Επισκόπηση των προγραμμάτων.

	Benchmark Set	Short Label	Scripts	LOC	Input	Source
1	Common Unix One-Liners	`Oneliners`	6	67	1 GB	[4, 5, 55, 80]
2	Bell Labs Unix50	`Unix50`	8	24	9 GB	[6, 39]
3	COVID-19 Mass Transit Analysis	`COVID-mts`	4	38	2.2 GB	[81]
4	Natural Language Processing	`NLP`	13	160	130 books	[9]
5	NOAA Weather Analysis	`NOAA`	1	16	500 MB	[85]

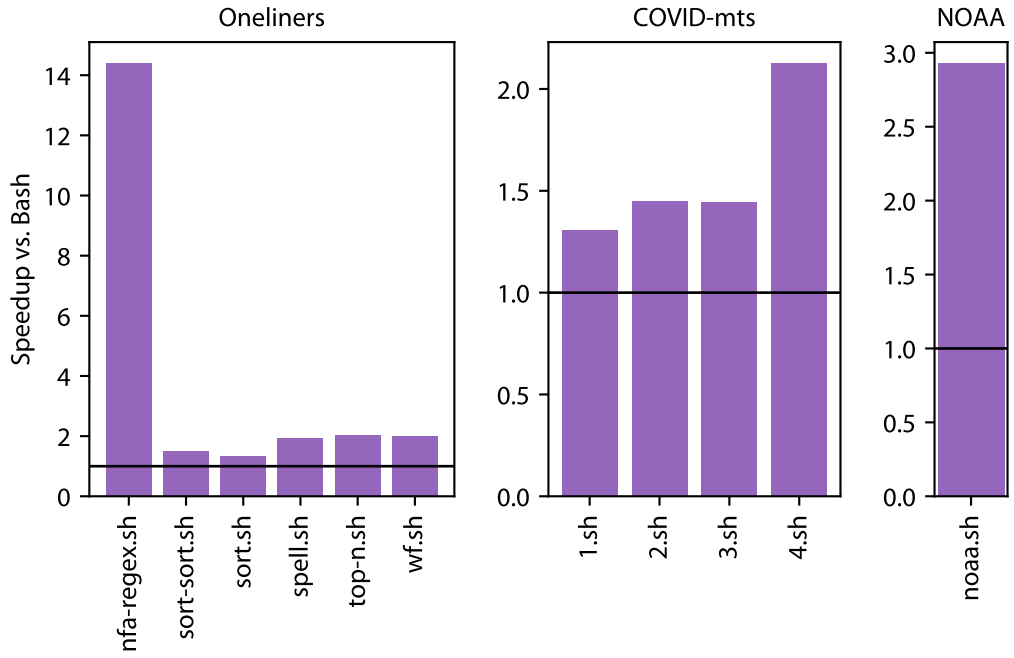
Κοινά Προγράμματα Unix Μίας Γραμμής Το σύνολο προγραμμάτων `Oneliners` περιέχει ένα σύνολο κλασικών προγραμμάτων-σωληνώσεων, που το καθένα αναδεικνύει διαφορετικά χαρακτηριστικά επίδοσης του κελύφους. Το NFA-Regex έχει ως επίκεντρο μία υπολογιστικά απαιτητική κανονική έκφραση. Το Sort ταξινομεί μία μεγάλη είσοδο. Τα Word-Frequency και Top-N-Terms είναι βασισμένα στο κλασικό πρόγραμμα καταμέτρησης λέξεων του McIlroy [5], που χρησιμοποιεί ταξινόμηση, αντί για πινακοποίηση, για να βρει συχνούς όρους σε ένα σύνολο λέξεων. Το Spell, βασισμένο στο πρωτότυπο `spell` που αναπτύχθηκε από τον Johnson [4], είναι ένα ακόμη κλασικό πρόγραμμα Unix: ύστερα από κάποια προεπεξεργασία, κάνει έξυπνη χρήση του `comm` για να βρει λέξεις που δεν ανήκουν στο λεξικό. Τέλος, το Sort-Sort χρησιμοποιεί συνεχόμενες `sort` και `sort -r` εντολές, χωρίς να εισάγει ανάμεσα εντολές που μειώνουν το μέγεθος της εισόδου τους (π.χ., `uniq`).

Bell Labs Unix50 Στον πρόσφατο εορτασμό των 50 χρόνων του Unix, τα Bell Labs δημιούργησαν ένα σύνολο από προκλήσεις που λύνονται με τη χρήση σωληνώσεων Unix. Τα προβλήματα αυτά σχεδιάστηκαν για να αναδείξουν την σπονδυλωτή φιλοσοφία του Unix [55]. Για την αξιολόγηση, χρησιμοποιούνται ανεπίσημες λύσεις που βρέθηκαν στο GitHub [76], που βρίσκονται σε μορφή σωληνώσεων με πολλαπλά στάδια. Κάνουν έντονη χρήση τυπικών εντολών, χρησιμοποιούν μία ποικιλία σημαιών, και φαίνεται να έχουν γραφτεί από μη-ειδικούς, αφού χρησιμοποιούν συχνά υποβέλτιστα κομμάτια που αποκλίνουν ελαφρά από τη φιλοσοφία του Unix.

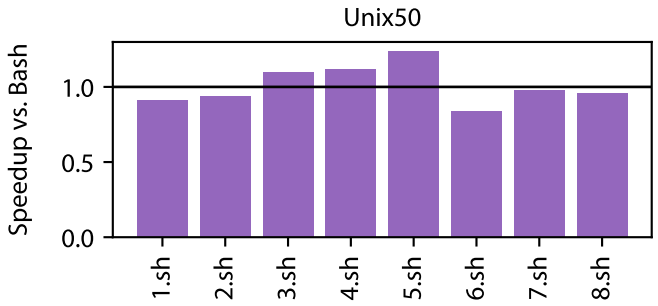
Ανάλυση Μέσων Μαζικής Μεταφοράς στη Διάρκεια του COVID-19 Το σύνολο `COVID-mts` περιέχει τέσσερεις σωληνώσεις που χρησιμοποιήθηκαν για την ανάλυση πραγματικών δεδομένων τηλεμετρίας από δρομολόγια λεωφορείων της Αθήνας, κατά τη διάρκεια του COVID-19 [81]. Οι σωληνώσεις αυτές υπολογίζουν κάποια στατιστικά για το σύστημα μεταφοράς κάθε μέρα, όπως μέσος αριθμός ωρών λειτουργίας, και μέσος αριθμός οχημάτων ανά ημέρα. Αυτές οι σωληνώσεις χρησιμοποιούν κλασικές εντολές του Unix όπως `sed`, `awk`, `sort`, και `uniq`.

Επεξεργασία Φυσικής Γλώσσας Το σύνολο `NLP` περιέχει αρκετά προγράμματα από το "Unix για ποιητές", ένα μάθημα για την ανάπτυξη προγραμμάτων επεξεργασίας φυσικής γλώσσας με χρήση Unix και Linux εργαλείων. Περιέχει εργασίες όπως μέτρηση λέξεων, εύρεση λέξεων με τέσσερα γράμματα, και μέτρηση ακολουθιών από σύμφωνα.

Ανάλυση Καιρού Εθνικού Οργανισμού Ωκεανού και Ατμόσφαιρας των ΗΠΑ Αυτό το πρόγραμμα είναι εμπνευσμένο από το κεντρικό παράδειγμα στο βιβλίο "Hadoop: Ο Καθοριστικός Οδηγός" [85], και εκτελεί ανάλυση δεδομένων θερμοκρασίας, πάνω σε δεδομένα του Εθνικού Οργανισμού Ωκεανού και Ατμόσφαιρας των ΗΠΑ (National Oceanic and Atmospheric Administration, NOAA). Αποτελείται από δύο σωληνώσεις, όπου η καθεμία υπολογίζει την μέγιστη και ελάχιστη θερμοκρασία, αντίστοιχα.



Σχήμα 6.1: Η επιτάχυνση του SPLaSh για τα σύνολα `Oneliners`, `COVID-mts`, και `NOAA` (μεγαλύτερες τιμές είναι καλύτερες). Η τιμή της παραμέτρου `--width` που χρησιμοποιείται σε κάθε πρόγραμμα ορίζεται στο παράρτημα (Πίνακες E.1, E.3, και E.5).

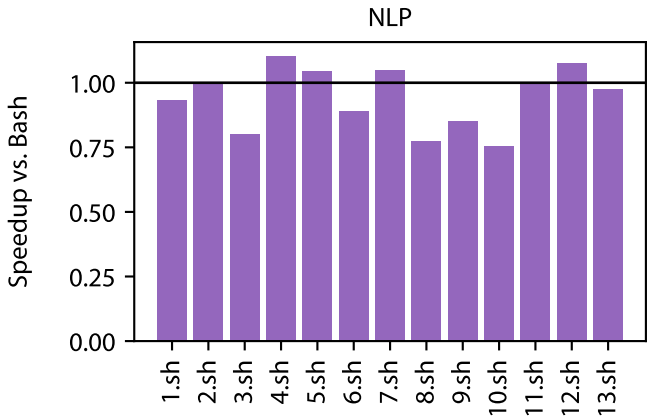


Σχήμα 6.2: Η επιτάχυνση του SPLaSh για το `Unix50` (μεγαλύτερες τιμές είναι καλύτερες). Η τιμή της παραμέτρου `--width` που χρησιμοποιείται σε κάθε πρόγραμμα ορίζεται στο παράρτημα (Πίνακας E.2).

6.3 Επίδοση

Η επιτάχυνση του SPLaSh σε σχέση με το Bash αξιολογείται σε όλα τα σύνολα προγραμμάτων, δοκιμάζοντας διαφορετικές τιμές της παραμέτρου `--width` για κάθε πρόγραμμα, και αναφέροντας τη μεγαλύτερη επιτάχυνση που επετεύχθη. Ο σκοπός είναι να αξιολογηθεί η μέγιστη επίδοση που μπορεί να πετύχει το SPLaSh, και να συγκριθεί με το Bash. Η τιμή της παραμέτρου `--width` που χρησιμοποιείται τελικά για κάθε πρόγραμμα ορίζεται στους Πίνακες E.1 έως E.5.

Oneliners Το Σχήμα 6.1 δείχνει ότι το NFA-Regex έχει τη μεγαλύτερη επιτάχυνση (14.38×), ενώ τα υπόλοιπα προγράμματα έχουν πιο ήπιες επιταχύνσεις, που κυμαίνονται από 1.33× μέχρι 2.05×. Όλα τα προγράμματα εκτός από το NFA-Regex έχουν την εντολή `sort` κάπου στις σωληνώσεις τους, το οποίο



Σχήμα 6.3: Η επιτάχυνση του PaSh για το `NLP` (μεγαλύτερες τιμές είναι καλύτερες). Η τιμή της παραμέτρου `--width` που χρησιμοποιείται σε κάθε πρόγραμμα ορίζεται στο παράρτημα (Πίνακας E.4).

(α) εισάγει αυξημένη μεταφορά δεδομένων μεταξύ των συναρτήσεων[3], και (β) έχει υψηλό αποτύπωμα μνήμης. Έτσι, το PaSh δεν μπορεί να εκμεταλλευτεί πλήρως τους επιπλέον υπολογιστικούς πόρους, και η επιτάχυνση είναι περιορισμένη, ειδικά εφόσον η συνάρτηση Bash έχει 10 240 MB μνήμης.[4] Ειδικά στην περίπτωση των Sort και Sort-Sort, το μέγεθος της εισόδου δεν μειώνεται καθώς προχωράει στα στάδια της σωλήνωσης—κάτι που θα αναδείκνυε τα οφέλη του αυξημένου παραλληλισμού—και για αυτό τον λόγο οι επιταχύνσεις αυτών των δύο προγραμμάτων είναι οι χαμηλότερες.

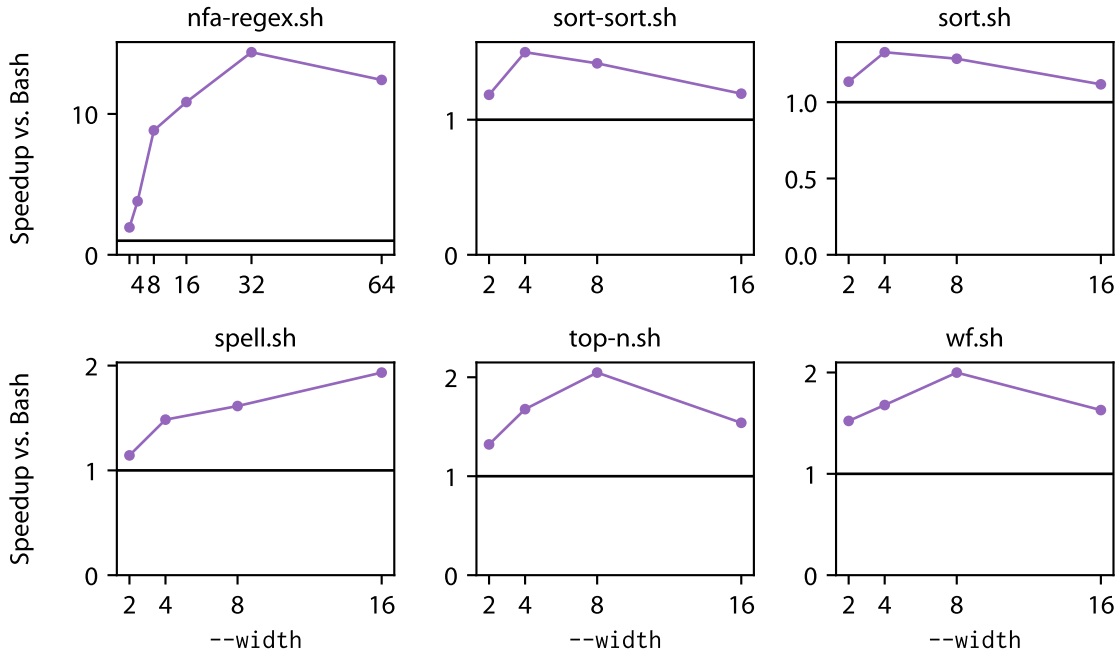
Unix50 Το Σχήμα 6.2 δείχνει ότι το PaSh προσφέρει οριακές επιταχύνσεις στην περίπτωση του `Unix50`—στην πραγματικότητα καθυστερεί 5 από τα 8 προγράμματα. Αυτό συμβαίνει επειδή αυτά τα προγράμματα χρησιμοποιούν σχεδόν αποκλειστικά απλές περιπτώσεις των εντολών `cut`, `tr`, και `grep`, που δεν κάνουν αρκετά έντονη χρήση του επεξεργαστή για να δικαιολογήσουν τον παραπάνω χρόνο που αφιερώνεται στην κατανομή της δουλειάς σε πολλές συναρτήσεις, καθώς και την μεταφορά δεδομένων μεταξύ τους.

COVID-mts Το Σχήμα 6.1 δείχνει ότι το PaSh προσφέρει για το `COVID-mts` παρόμοιες επιταχύνσεις με το `Oneliners`, το οποίο είναι κάπως αναμενόμενο, αφού τα προγράμματα από τα δύο σύνολα μοιάζουν στη δομή τους. Κι εδώ η ύπαρξη της εντολής `sort` περιορίζει την επιτάχυνση που μπορεί να προσφέρει το PaSh.

NLP Το Σχήμα 6.3 δείχνει μία παρόμοια εικόνα με το `Unix50`, με το PaSh να επιταχύνει οριακά κάποια προγράμματα, και να καθυστερεί τα υπόλοιπα. Ωστόσο, εδώ ο λόγος είναι διαφορετικός: κάθε πρόγραμμα στο `NLP` δρα επαναληπτικά πάνω σε μία λίστα βιβλίων, και παρά το γεγονός ότι το PaSh εκτελεί κάθε επανάληψη αυτού του βρόχου με παραλληλία δεδομένων, ο ίδιος ο βρόχος εξελίσσεται *σειριακά*—κάθε επανάληψη του βρόχου πρέπει να τελειώσει πριν ξεκινήσει η επόμενη. Επειδή κάθε βιβλίο από μόνο του είναι μικρό, δεν αξίζει να πληρώσει κανείς το κόστος του στησίματος μίας ολόκληρης ομάδας συναρτήσεων και της επικοινωνίας τους. Μία λύση είναι να τρέχουν ταυτόχρονα πολλές επαναλήψεις του βρόχου, αφού κάθε επανάληψη του βρόχου είναι ανεξάρτητη από τις υπόλοιπες—και έτσι να αλληλοεπικαλύπτονται τα κόστη των σωληνώσεων. Η υποστήριξη αυτής της λειτουργίας στο PaSh είναι εργασία σε εξέλιξη.

[3] Το PaSh, όπως το PaSh, συναθροίζει τα ενδιάμεσα αποτελέσματα της εντολής `sort` με δενδροειδή τρόπο.

[4] Αυτή η επιπλέον μνήμη μπορεί να κάνει προγράμματα που χρησιμοποιούν `sort` να τρέχουν πολύ πιο γρήγορα. Για παράδειγμα, όταν το `spell.sh` εκτελείται σε μία συνάρτηση με μόνο 1769 MB μνήμης, ο χρόνος εκτέλεσης διπλασιάζεται, επειδή η εντολή `sort` αποτελεί τη στενωπό.



Σχήμα 6.4: Η κλιμακωσιμότητα του SPLaSh για το `Oneliners` (μεγαλύτερες τιμές είναι καλύτερες). Η παράμετρος `--width` κυμαίνεται από 2 έως 64, ανάλογα το πρόγραμμα.

NOAA Το Σχήμα 6.1 δείχνει ότι το SPLaSh επιταχύνει το πρόγραμμα ανάλυσης δεδομένων θερμοκρασίας σχεδόν 3×. Το πρόγραμμα αυτό χρησιμοποιεί επίσης την εντολή `sort`, αλλά κάνει ελαφρώς πιο έντονη χρήση του επεξεργαστή από τα υπόλοιπα—η αναλογία των εντολών που χρησιμοποιούν έντονα τον επεξεργαστή είναι λίγο υψηλότερος.

6.4 Κλιμακωσιμότητα

Για την πιο λεπτομερή ανάλυση των χαρακτηριστικών επίδοσης του SPLaSh, η κλιμακωσιμότητα του SPLaSh αξιολογείται στο σύνολο `Oneliners`, μεταβάλλοντας την παράμετρο `--width`[5] από 2 έως 64—ανάλογα το πρόγραμμα—και μετρώντας την επιτάχυνση του SPLaSh σε σχέση με το Bash (Σχήμα 6.4).

Το NFA-Regex κλιμακώνει σχεδόν γραμμικά με τον αριθμό των συναρτήσεων—αναμενόμενο, αφού το πρόγραμμα είναι έντονα παραλληλοποιήσιμο, όπως αναφέρθηκε προηγουμένως—φτάνοντας σε ένα τέλμα στις 64 συναρτήσεις. Αυτό το τέλμα, ωστόσο, προκαλείται από τον χρόνο που χρειάζεται να μεταγλωττιστεί το πρόγραμμα, να κληθούν οι συναρτήσεις, και να εγκαταστήσουν συνδέσεις μεταξύ τους. Αυτή η καθυστέρηση εισάγεται μία φορά, και μπορεί να αποσβεστεί όσο το μέγεθος εισόδου αυξάνεται. Τα Sort και Sort-Sort γρήγορα έχουν φθίνουσες αποδόσεις μετά το `--width=4`, ενώ τα Top-N-Terms και Word-Frequency σταματούν να κλιμακώνουν μετά από την τιμή `--width=8`. Το Spell κλιμακώνει κάπως πιο ευνοϊκά μέχρι το `--width=16`.

Συμπέρασμα Παρόλο που το SPLaSh μπορεί να προσφέρει τους πόρους κλιμακωσιμότητας που απαιτούνται—όπως φαίνεται στην περίπτωση του NFA-Regex—είναι σημαντικό ο χρήστης να λαμβάνει υπόψιν του τη φύση του προγράμματος κελύφους που πρόκειται να εκτελεστεί. Αν ένα πρόγραμμα κελύφους είναι εγγενώς μη-κλιμακώσιμο, δεν υπάρχουν πολλά που μπορεί να κάνει το SPLaSh για

[5] Να σημειωθεί ότι παρόλο που η παράμετρος `--width` καθορίζει εμμέσως τον αριθμό των συναρτήσεων που θα χρησιμοποιηθούν—επιβάλλοντας τον βαθμό του παραλληλισμού δεδομένων—ο αριθμός των συναρτήσεων είναι συνήθως μεγαλύτερος από την τιμή της παραμέτρου `--width`. Για παράδειγμα, ενώ ο γράφος ροής δεδομένων στο Σχήμα 4.1 έχει `--width=2`, το SPLaSh καλεί τέσσερις συναρτήσεις.

να το επιταχύνει. Επομένως, το SPLASH στην παρούσα κατάστασή του προορίζεται για προγράμματα που εκτελούνται για αρκετή ώρα, κάνουν έντονη χρήση του επεξεργαστή, και είναι έντονα παραλληλοποιήσιμα—και δεν είναι μία καθολική λύση για την επιτάχυνση προγραμμάτων κελύφους.

Κεφάλαιο 7

Σχετικό Έργο

Το SPLaSh σχετίζεται με ένα μεγάλο σώμα προηγούμενου έργου σχετικά με την ανάπτυξη προγραμμάτων κελύφους και τον υπολογισμό χωρίς διακομιστή.

7.1 Κλιμάκωση Προγραμμάτων Κελύφους

Εργαλεία που εκθέτουν παραλληλισμό σε σύγχρονα Unix συστήματα, όπως το qsub [21], το SLURM [31], το AMFS [88] και το GNU `parallel` [79], βασίζονται στη ρητή και προσεκτική τροποποίηση των προγραμμάτων από τους χρήστες τους. Επίσης, αρκετά κελύφη προσθέτουν εντολές για μη γραμμικές τοπολογίες αγωγών—μερικές από τις οποίες στοχεύουν στον παραλληλισμό [43, 77, 83]. Ωστόσο, κι εδώ αναμένεται από τους χρήστες να μεταγράψουν χειροκίνητα τα προγράμματα τους για να εκμεταλλευτούν αυτές τις νέες εντολές, χωρίς να διαρραγεί η ορθότητα. Ένα πρόσφατο κατανεμημένο κέλυφος, το POSH [54], μπορεί να χειριστεί ένα υποσύνολο των προγραμμάτων κελύφους χωρίς μεταγραφή, αλλά απευθύνεται σε προγράμματα που κάνουν έντονη χρήση του δικτύου και των συσκεύων εισόδου-εξόδου, περιορίζεται σε υπολογισμούς που έχουν μορφή γράφου ροής δεδομένων και δεν υποστηρίζει αυθαίρετες συμπεριφορές του κελύφους. Επιπλέον, επειδή το POSH είναι μια επανυλοποίηση του κελύφους, δεν έχει ισοδύναμη συμπεριφορά με τα υπάρχοντα κελύφη και, συνεπώς υπάρχει κίνδυνος να χαλάσει την ορθότητα ορισμένων προγραμμάτων.

Άλλα συστήματα για την επιτάχυνση του κελύφους περιλαμβάνουν το hS [40], το PaSh [34, 82] και το DiSh [49]. Το hS εκτελεί εντολές εκτός σειράς, ενώ το PaSh και το DiSh εισάγουν παραλληλία και κατανομή για προγράμματα κελύφους, αντίστοιχα. Από τη μία, τα συστήματα δεν απαιτούν τροποποιήσεις στο αρχικό πρόγραμμα, αλλά από την άλλη δεν υποστηρίζουν αυτόματη κλιμάκωση για υπολογισμό χωρίς διακομιστή. Το SPLaSh βασίζεται στις ιδέες—και την υποδομή—του PaSh και του DiSh, αξιοποιώντας τη βιβλιοθήκη επισημειώσεων εντολών και τη μηχανή μεταγλώττισης πάνω-στην-ώρα.

7.2 Υπολογισμός Χωρίς Διακομιστή για Εφαρμογές Πέραν του Χειρισμού Γεγονότων

Πολλά πρόσφατα άρθρα [1, 2, 8, 15, 26, 27, 30, 33, 36, 37, 53, 58, 74, 78, 84] έχουν εξερευνήσει τη χρήση του υπολογισμού χωρίς διακομιστή για εφαρμογές πέραν του χειρισμού γεγονότων. Ένα βασικό κίνητρο είναι ότι οι συναρτήσεις εκκινούν πολύ πιο γρήγορα από τις εικονικές μηχανές, επιτρέποντας στους χρήστες να εκκινήσουν γρήγορα πολλούς πυρήνες υπολογισμού—χωρίς να χρειάζεται να δεσμεύσουν μηχανήματα για μεγάλο χρονικό διάστημα. Οι πλατφόρμες χωρίς διακομιστή παρέχουν έτσι ένα κλιμακούμενο υποστρώμα υπολογισμού, όπου η ποσότητα των υπολογιστικών πόρων που είναι διαθέσιμοι μπορεί να αλλάξει γρήγορα. Προηγούμενες εργασίες έχουν εκμεταλλευτεί αυτήν την ευελιξία για αριθμητικούς υπολογισμούς [2, 15, 26, 27, 33, 74, 84], μηχανική μάθηση [8, 30, 78], ανάλυση δεδομένων [33, 36, 37, 38, 53, 58], επεξεργασία βίντεο [1] και ταξινόμηση [33, 53]. Ωστόσο, επειδή αυτές οι προσπάθειες στοχεύουν σε συγκεκριμένους τομείς, συχνά είναι υποβέλτιστες για άλλους τύπους υπολογισμού—αν μπορούν να τους εκφράσουν καν.

7.3 Υπολογισμός Χωρίς Διακομιστή για Εφαρμογές Γενικού Σκοπού

Πολλά εργαλεία έχουν αναπτυχθεί για την απλοποίηση του προγραμματισμού χωρίς διακομιστή για εφαρμογές γενικού σκοπού, τόσο από ερευνητικές εργασίες [17, 18, 33, 87], όσο και από παρόχους νέφους [46, 47, 69]. Αν και αυτά τα εργαλεία απλοποιούν την ανάπτυξη εφαρμογών χωρίς διακομιστή, έχουν μερικούς περιορισμούς, καθώς κάθε βήμα της ροής εργασίας δημιουργεί επιπλέον επιβάρυνση στον προγραμματιστή. Για να χρησιμοποιήσει κάποιος αυτά τα εργαλεία, ο προγραμματιστής πρέπει πρώτα να διαχωρίσει τον υπολογισμό σε μικρότερα κομμάτια—τα οποία πρέπει να χωράνε εντός του χρονικού ορίου της συνάρτησης και να πληρούν μια σειρά από περιορισμούς που αφορούν το εκάστοτε εργαλείο—και στη συνέχεια να συνθέσει αυτά τα στοιχεία σε μια εφαρμογή χωρίς διακομιστή χρησιμοποιώντας μία μορφή συγκεκριμένη στο εκάστοτε εργαλείο (π.χ., πεπερασμένες μηχανές καταστάσεων [17, 46, 69], ή γράφους ροής δεδομένων [18]), προσθέτοντας περαιτέρω νοητικό φορτίο. Για παράδειγμα, το Kappa [87] απαιτεί τροποποιήσεις στον κώδικα της εφαρμογής—ο προγραμματιστής πρέπει να εισάγει κλήσεις ελέγχου σε κατάλληλα σημεία του προγράμματος, να επισημάνει κλήσεις που έχουν εξωτερικά ορατές παρενέργειες και να χρησιμοποιεί τις εντολές παραλληλισμού του Kappa αντί για τις αντίστοιχες προεπιλεγμένες εντολές. Επιπλέον, τα περισσότερα από αυτά τα εργαλεία δεν υποστηρίζουν πλήρως δυναμικές συμπεριφορές—όπως αυτές που απαντώνται στα προγράμματα κελύφους. Αντίθετα, η προσέγγιση πάνω-στην-ώρα που χρησιμοποιείται από το SPLASH του επιτρέπει να έχει πάντα τις πιο ενημερωμένες πληροφορίες για την κατάσταση του κελύφους, και να κλιμακώνεται χωρίς να απειλεί την ορθότητα.

7.4 Επικοινωνία και Ενορχήστρωση για Εφαρμογές Χωρίς Διακομιστή

Πολλά εργαλεία υποστηρίζουν επικοινωνία και ενορχήστρωση για εφαρμογές υπολογισμού χωρίς διακομιστή. Από τη μία πλευρά, τα περισσότερα από αυτά στοχεύουν σε συγκεκριμένους τύπους εφαρμογών και μοτίβα εκτέλεσης, όπως ανάλυση δεδομένων και κατανεμημένος συγχρονισμός [3, 37, 38, 41, 48, 52, 53] ή μηχανική μάθηση [8, 32]—και επομένως επικεντρώνονται σε βελτιστοποιήσεις που αφορούν τους συγκεκριμένους τύπους αυτούς. Από την άλλη πλευρά, μερικά από αυτά τα εργαλεία είναι σχεδιασμένα για να χειρίζονται την επικοινωνία για εφαρμογές γενικού σκοπού [17, 18, 33]. Ωστόσο, είτε χρησιμοποιούν αποθήκευση νέφους για ενδιάμεσα δεδομένα—με αποτέλεσμα υψηλή καθυστέρηση και κόστος—είτε χρησιμοποιούν προσαρμοσμένες αποθήκες, κρυφή μνήμη και διακομιστές μηνυμάτων—με αποτέλεσμα να υπονομεύουν τα πλεονεκτήματα του υπολογισμού χωρίς διακομιστή. Οι πάροχοι νέφους προσφέρουν επίσης ορισμένες λύσεις [47, 69], αλλά απαιτούν από τον χρήστη να μεταφράσει χειροκίνητα τα προγράμματά του σε μία λιγότερο εκφραστική μορφή, πριν την εκτέλεση τους. Τέλος, το FMI [10] υποστηρίζει πολλά διαφορετικά περιβάλλοντα επικοινωνίας—αποθήκευση αντικειμένων, αποθήκευση στη μνήμη, επικοινωνία TCP—αλλά και πάλι απαιτεί από τον χρήστη να μεταγράψει χειροκίνητα τα προγράμματά του για να χρησιμοποιήσει τις δικές του διεπαφές.

Κεφάλαιο 8

Συζήτηση

8.1 Μελλοντικό Έργο

Το SPLaSh είναι ένα πρώτο βήμα προς την αξιοποίηση του υπολογισμού χωρίς διακομιστή για την επιτάχυνση και κλιμάκωση των προγραμμάτων κελύφους, όμως υπάρχουν πολλές μελλοντικές κατευθύνσεις για μελλοντικό έργο. Ορισμένες από αυτές παρατίθενται εδώ.

Προγράμματα Μακράς Εκτέλεσης Οι πλατφόρμες χωρίς διακομιστή επιβάλλουν ένα χρονικό όριο εκτέλεσης στις συναρτήσεις—15 λεπτά στην περίπτωση του AWS. Αυτό σημαίνει ότι τα προγράμματα που εκτελούνται για περισσότερο χρόνο από αυτό το όριο δεν υποστηρίζονται—και απαιτούν ειδική προσοχή. Εφόσον ο τερματισμός λόγω υπέρβασης του χρονικού ορίου μπορεί να θεωρηθεί ως αποτυχία, οι ιδέες από την ανοχή σε σφάλματα μπορούν να εφαρμοστούν και σε αυτήν την περίπτωση. Κάτι που πρέπει να ληφθεί υπόψη είναι ότι τα σενάρια που είναι έντονα παραλληλοποιήσιμα και κάνουν έντονη χρήση του επεξεργαστή μπορούν να επωφεληθούν περισσότερο από το SPLaSh, οπότε ακόμα κι αν ένα τέτοιο πρόγραμμα χρειάζεται πολύ χρόνο για να εκτελεστεί σειριακά, μπορεί να επιταχυνθεί σημαντικά όταν χρησιμοποιούνται πολλές συναρτήσεις.

Μία γενικότερη λύση θα ήταν οι συναρτήσεις να έχουν επίγνωση του ορίου εκτέλεσής τους, και να εφαρμόζουν λογική ανακατεύθυνσης δεδομένων. Για παράδειγμα, αν μία συνάρτηση ήδη εκτελείται για το 50% του ορίου εκτέλεσής της, μπορεί να καλεί μία νέα συνάρτηση, και να ενημερώνει την προηγούμενη της συνάρτηση (ή συναρτήσεις) να αρχίσει να στέλνει δεδομένα σε αυτή την καινούργια συνάρτηση—ομοίως, μπορεί να ενημερώνει την επόμενη συνάρτηση (ή συναρτήσεις) να αρχίσει να λαμβάνει δεδομένα από αυτή τη νέα συνάρτηση. Ωστόσο, αυτή η προσέγγιση θα λειτουργούσε μόνο για το υποσύνολο εντολών που δεν διατηρούν ενδιάμεση κατάσταση. Στην περίπτωση των εντολών μαύρων-κουτιών που διατηρούν ενδιάμεση κατάσταση, η αποθήκευση αυτής της κατάστασης δεν είναι τετριμμένη, και το να γίνει αυτό με έναν αρκετά γενικό τρόπο μπορεί να προκαλέσει μεγάλες καθυστερήσεις. Η γενίκευση της ανακατεύθυνσης για όλες τις κατηγορίες εντολών είναι ένα ενδιαφέρον πρόβλημα προς εξερεύνηση.

Μετάφραση Κελύφους για Υπολογισμό Χωρίς Διακομιστή Η μετάφραση εντολών κελύφους όπως `sort`, `tr`, και `grep` για υπολογισμό χωρίς διακομιστή φαίνεται να είναι απλή: οι εντολές αυτές παίρνουν μία είσοδο, την επεξεργάζονται, και παράγουν την αντίστοιχη έξοδο. Ωστόσο, άλλες κατηγορίες εντολών δεν έχουν τόσο ξεκάθαρες μεταφράσεις. Για παράδειγμα, ποια είναι η αναμενόμενη συμπεριφορά όταν ο χρήστης τρέχει τις εντολές `ls` ή `cd`; Πρέπει το `ls` να επιστρέψει τα περιεχόμενα του τοπικού καταλόγου, ή τα περιεχόμενα ενός κουβά στην αποθήκη αντικειμένων νέφους; Πρέπει το `cd` να αλλάξει τον τοπικό κατάλογο εργασίας, ή πρέπει να κάνει όλα τα αιτήματα προς την αποθήκη αντικειμένων να χρησιμοποιούν ένα συγκεκριμένο πρόθεμα—ουσιαστικά, καθιστώντας τις εντολές `(cd newDir; object-get myObj)` και `(object-get newDir/myObj)` ισοδύναμες;

Παρόλο που κάποιος θα μπορούσε απλώς να ορίσει μία καθολική λύση—όλες οι εντολές εφαρμόζονται στο τοπικό περιβάλλον, ή όλες οι εντολές εφαρμόζονται στο περιβάλλον νέφους—αυτή η προσέγγιση θα απέκλειε κάποιες συμπεριφορές, τις οποίες δεν θα μπορούσε να υποστηρίξει απευθείας. Για παράδειγμα, αν το `ls` λειτουργούσε μόνο στο τοπικό περιβάλλον, οι χρήστες που θα ήθελαν να

δουν τα αντικείμενα στην αποθήκη νέφους θα έπρεπε να αλλάξουν χειροκίνητα τα προγράμματά τους για να χρησιμοποιήσουν μία εντολή παρόμοια με το `list-objects` του Amazon S3 στη θέση του `ls`. Μία άλλη ενδιαφέρουσα προσέγγιση είναι η εισαγωγή ενός νέου συνόλου επισημειώσεων εντολών για υπολογισμό χωρίς διακομιστή που μπορεί να ρυθμιστεί εκ των προτέρων. Επίσης, πιθανώς αξίζει να διερευνηθεί αν υπάρχει αξία στο να εισαχθούν επισημειώσεις εντολών που οι ίδιοι οι χρήστες προσθέτουν στα προγράμματά τους, για να ορίσουν αν κάποιες εντολές πρέπει να εκτελεστούν στο τοπικό περιβάλλον ή στο περιβάλλον νέφους, αντίστοιχα.

Ανοχή σε Σφάλματα Το SPLaSh δεν υποστηρίζει αυτή τη στιγμή ανοχή σε σφάλματα. Τα τρέχοντα συστήματα κατανεμημένης επεξεργασίας [13, 19, 86] παρέχουν ανοχή σε σφάλματα είτε επιβάλλοντας στους προγραμματιστές ένα πιο περιοριστικό μοντέλο προγραμματισμού, είτε παρέχοντας μια προκαθορισμένη διεπαφή που επιτρέπει βελτιστοποιήσεις στην ανοχή σε σφάλματα—όπως η δημιουργία σημείων ελέγχου. Αυτό είναι δύσκολο να γίνει στην περίπτωση του κελύφους, επειδή το κέλυφος λειτουργεί ως γλώσσα σύνθεσης για αυθαίρετα στοιχεία που δεν μπορούν να τροποποιηθούν, συνδεδεμένα με αυθαίρετους τρόπους. Για παράδειγμα, η εντολή `wc -l` διατηρεί εσωτερική κατάσταση στη μνήμη, στην οποία δεν μπορεί ο χρήστης να έχει πρόσβαση—τουλάχιστον χωρίς να αλλάξει την ίδια την υλοποίηση του `wc`. Αυτό σημαίνει ότι σε περίπτωση σφάλματος, ουσιαστικά θα ήταν αναγκαστική μια πλήρης επανεκτέλεση, ακόμα κι αν η εκτέλεση της εντολής `wc` είχε ολοκληρωθεί κατά 99%. Ένα άλλο σημείο σημασίας είναι ότι οι πλατφόρμες χωρίς διακομιστή παρέχουν τη δυνατότητα επανεκτέλεσης των συναρτήσεων σε περίπτωση αποτυχίας, αλλά αυτό απαιτεί οι συναρτήσεις να είναι ταυτοδύναμες—ώστε η επανεκτέλεση να μην προκαλέσει ανεπιθύμητες παρενέργειες. Αυτό δεν ισχύει για το κέλυφος, που έχει εγγενώς παρενέργειες, ενώ η επικοινωνία ροής δεδομένων δυσκολεύει ακόμα περισσότερο τα πράγματα.

Ένα πλεονέκτημα που θα μπορούσε να χρησιμοποιηθεί είναι η πληθώρα των πόρων που παρέχουν οι πλατφόρμες χωρίς διακομιστή. Οι χρήστες μπορούν να εκτελούν πολλαπλά αντίγραφα των συναρτήσεων που εκτελούν το ίδιο πρόγραμμα, στα ίδια δεδομένα, την ίδια στιγμή, και να κρατούν το αποτέλεσμα της πρώτης συνάρτησης που ολοκληρώνει την εκτέλεσή της—μειώνοντας επιπλέον τους εργάτες που τερματίζουν καθυστερημένα [52]. Για να αποφευχθεί το υπερβολικό οικονομικό κόστος, ο χρήστης μπορεί να καθορίσει κατηγορίες συναρτήσεων που απαιτούν ειδική μεταχείριση. Για παράδειγμα, οι συναρτήσεις που πραγματοποιούν συνάθροιση μπορούν να θεωρηθούν πιο κρίσιμες—και, συνεπώς, να εκτελούνται σε πολλαπλά αντίγραφα—ενώ οι συναρτήσεις που πραγματοποιούν απλούστερες λειτουργίες μπορούν να εκτελούνται εκ νέου κατά απαίτηση.

Βέλτιστη Εκχώρηση Πόρων και Ρύθμιση Παραμέτρων Το SPLaSh εισάγει ένα σύνολο παραμέτρων που μπορούν να ρυθμιστούν από τον χρήστη—όπως το πλάτος παραλληλοποίησης, η μνήμη κάθε συνάρτησης, ή το μέγεθος πακέτου στη ροή επικοινωνίας. Αυτοί οι παράγοντες μπορούν να επηρεάσουν σημαντικά την επίδοση του προγράμματος με μη προφανή τρόπο, ειδικά επειδή το μέγεθος μνήμης και ο αριθμός των υπολογιστικών πυρήνων συνήθως συνδέονται μεταξύ τους στις πλατφόρμες χωρίς διακομιστή. Επιπλέον, η χειροκίνητη ρύθμιση αυτών των παραμέτρων είναι πολύπλοκη, χρονοβόρα, και αυξάνει το κόστος. Επομένως, θα ήταν χρήσιμο να παρέχεται κάποιος αυτόματος μηχανισμός ρύθμισης που θα προσαρμόζει αυτές τις παραμέτρους με βάση τα χαρακτηριστικά του προγράμματος. Αυτό θα μπορούσε να γίνει είτε πριν την εκτέλεση—μία φορά για κάθε εντολή, παρόμοια με τις επισημειώσεις του PaSh—είτε κατά τη διάρκεια της εκτέλεσης—καταγράφοντας μεταδεδομένα εκτέλεσης σε πραγματικό χρόνο—είτε τόσο πριν όσο και κατά τη διάρκεια.

Υβριδική Εκτέλεση με Βάση το Κόστος Όπως έχει ήδη αναφερθεί, ο παραδοσιακός τρόπος εκτέλεσης των εργασιών σε περιβάλλον χωρίς διακομιστή δεν είναι μια λύση που ταιριάζει σε όλες τις περιπτώσεις—απαιτεί προσεκτική σκέψη για τους τύπους των εργασιών που ο χρήστης θέλει να εκτελέσει, καθώς και τη συχνότητα των εργασιών αυτών. Φυσικά, η εκτέλεση εργασιών σε παραδοσιακές υποδομές νέφους—ή σε φυσική υποδομή—έχει τα δικά της μειονεκτήματα. Μια ενδιαφέρουσα κατεύθυνση είναι να συνδυαστούν τα οφέλη των δύο προσεγγίσεων. Μια λύση είναι να υπάρχει ένα σύνολο

δεσμευμένων μηχανημάτων, και όταν υπάρχει ανάγκη, να γίνεται κλιμάκωση σε πλατφόρμα χωρίς διακομιστή. Αυτή η απόφαση κλιμάκωσης πρέπει να λαμβάνεται γρήγορα, με βάση το κόστος και τον τύπο της εργασίας, για να αποφασιστεί ποια είναι η χρυσή τομή. Υπάρχει ήδη ερευνητικό έργο προς αυτή την κατεύθυνση [51, 56].

Βέλτιστη Μετατροπή του Γράφου με Βάση το Κόστος Αυτή τη στιγμή το SPLaSh διαχωρίζει και κλιμακώνει τον γράφο ροής δεδομένων με βάση στατικούς παράγοντες—όπως το πλάτος της παραλληλοποίησης που ορίζει ο χρήστης, ή το αν ένα στάδιο του υπολογισμού περιέχει παραλληλία. Αυτή η προσέγγιση λειτουργεί σε περιπτώσεις ενός απλού μηχανήματος ή ενός στατικού συνόλου από μηχανήματα, όπου ο αριθμός των νημάτων ή μηχανών είναι γνωστός εκ των προτέρων, και η κλιμάκωση έχει μικρό επιπλέον κόστος, κυρίως συνδεδεμένο με την κατανάλωση ενέργειας. Ωστόσο, στην περίπτωση του υπολογισμού χωρίς διακομιστή, κάθε κλήση συνάρτησης έχει ένα χρηματικό κόστος—το να επεκταθεί κάποιος σε παραπάνω συναρτήσεις από ότι είναι απαραίτητο έχει αρνητικές επιπτώσεις. Για παράδειγμα, αν ένα πρόγραμμα δεν κλιμακώνει καλά μετά τις 32 συναρτήσεις, τότε το να κληθούν 1000 συναρτήσεις θα προκαλούσε ένα αχρείαστο χρηματικό κόστος—και θα μπορούσε επίσης να βλάψει την επίδοση. Επομένως, ο χρήστης θα μπορούσε να ορίζει έναν προϋπολογισμό, και το SPLaSh θα μπορούσε να περιορίσει την χρήση πόρων του σε αυτόν τον προϋπολογισμό, ή—ιδανικά—να βρει μία σχεδόν βέλτιστη κατανομή πόρων για τον προϋπολογισμό αυτό. Κάτι τέτοιο θα μπορούσε να γίνει χρησιμοποιώντας ένα μοντέλο κόστους που προβλέπει το κόστος μίας συγκεκριμένης εκχώρησης πόρων, και να καθοδηγεί τη διαδικασία εκχώρησης πόρων ανάλογα, είτε σε πραγματικό χρόνο είτε εκ των προτέρων [75].

8.2 Συμπεράσματα

Το SPLaSh επιτρέπει την αυτόματη κλιμάκωση των προγραμμάτων κελύφους με τη χρήση του υπολογισμού χωρίς διακομιστή, προκειμένου να αποκομίσει τα οφέλη του. Το πετυχαίνει εισάγοντας παραλληλία στα προγράμματα κελύφους και κατανέμοντας το έργο σε πολλαπλές συναρτήσεις, ενώ διευκολύνει την φόρτωση και την επικοινωνία των συναρτήσεων. Το SPLaSh λειτουργεί με τρόπο πάνω-στην-ώρα και δεν απαιτεί καμία τροποποίηση στα αρχικά προγράμματα. Οι μετρήσεις δείχνουν ότι το SPLaSh είναι αποτελεσματικό για εργασίες που κάνουν έντονη χρήση του επεξεργαστή και είναι έντονα παραλληλοποιήσιμες, παρέχοντας έως και 14.38× επιτάχυνση σε σχέση με την σειριακή εκτέλεση των προγραμμάτων κελύφους.

Βιβλιογραφία

[1] Lixiang Ao κ.ά. "Sprocket: A Serverless Video Processing Framework". Στο: *Proceedings of the ACM Symposium on Cloud Computing*. SoCC '18. Carlsbad, CA, USA: Association for Computing Machinery, 2018, σσ. 263–274. ISBN: 9781450360111. DOI: 10.1145/3267809.3267815. URL: https://doi.org/10.1145/3267809.3267815.

[2] Arda Aytekin και Mikael Johansson. "Harnessing the Power of Serverless Runtimes for Large-Scale Optimization". Στο: *CoRR* abs/1901.03161 (2019). arXiv: 1901.03161. URL: http://arxiv.org/abs/1901.03161.

[3] Daniel Barcelona-Pons κ.ά. "Stateful Serverless Computing with Crucial". Στο: *ACM Trans. Softw. Eng. Methodol.* 31.3 (Μαρ. 2022). ISSN: 1049-331X. DOI: 10.1145/3490386. URL: https://doi.org/10.1145/3490386.

[4] Jon Bentley. "Programming pearls: a spelling checker". Στο: *Commun. ACM* 28.5 (Μάι. 1985), σσ. 456–462. ISSN: 0001-0782. DOI: 10.1145/3532.315102. URL: https://doi.org/10.1145/3532.315102.

[5] Jon Bentley, Don Knuth και Doug McIlroy. "Programming pearls: a literate program". Στο: *Commun. ACM* 29.6 (Ιούν. 1986), σσ. 471–483. ISSN: 0001-0782. DOI: 10.1145/5948.315654. URL: https://doi.org/10.1145/5948.315654.

[6] Pawan Bhandari. *Solutions to unixgame.io*. Accessed: 2020-04-14. 2020.

[7] Ivano Bonesana, Marco Paolieri και Marco D. Santambrogio. "An adaptable FPGA-based system for regular expression matching". Στο: *Proceedings of the Conference on Design, Automation and Test in Europe*. DATE '08. Munich, Germany: Association for Computing Machinery, 2008, σσ. 1262–1267. ISBN: 9783981080131. DOI: 10.1145/1403375.1403681. URL: https://doi.org/10.1145/1403375.1403681.

[8] Joao Carreira κ.ά. "Cirrus: a Serverless Framework for End-to-end ML Workflows". Στο: *Proceedings of the ACM Symposium on Cloud Computing*. SoCC '19. Santa Cruz, CA, USA: Association for Computing Machinery, 2019, σσ. 13–24. ISBN: 9781450369732. DOI: 10.1145/3357223.3362711. URL: https://doi.org/10.1145/3357223.3362711.

[9] Kenneth Ward Church. *Unix for poets*. Notes of a course from the European Summer School on Language and Speech Communication, Corpus Based Methods. 1994.

[10] Marcin Copik κ.ά. "FMI: Fast and Cheap Message Passing for Serverless Functions". Στο: *Proceedings of the 37th International Conference on Supercomputing*. ICS '23. Orlando, FL, USA: Association for Computing Machinery, 2023, σσ. 373–385. ISBN: 9798400700569. DOI: 10.1145/3577193.3593718. URL: https://doi.org/10.1145/3577193.3593718.

[11] Datadog. *The State of Serverless 2021*. Accessed: 2024-06-16. 2021. URL: https://www.datadoghq.com/state-of-serverless-2021/.

[12] Datadog. *The State of Serverless 2023*. Accessed: 2024-06-16. 2023. URL: https://www.datadoghq.com/state-of-serverless/.

[13] Jeffrey Dean και Sanjay Ghemawat. "MapReduce: Simplified Data Processing on Large Clusters". Στο: *6th Symposium on Operating Systems Design & Implementation (OSDI 04)*. San Francisco, CA: USENIX Association, Δεκ. 2004. URL: https://www.usenix.org/conference/osdi-04/mapreduce-simplified-data-processing-large-clusters.

[14] Simon Eismann κ.ά. "The State of Serverless Applications: Collection, Characterization, and Community Consensus". Στο: *IEEE Transactions on Software Engineering* 48.10 (2022), σσ. 4152–4166. DOI: 10.1109/TSE.2021.3113940.

[15] Lang Feng κ.ά. "Exploring Serverless Computing for Neural Network Training". Στο: *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*. 2018, σσ. 334–341. DOI: 10.1109/CLOUD.2018.00049.

[16] Bryan Ford, Pyda Srisuresh και Dan Kegel. *Peer-to-Peer Communication Across Network Address Translators*. 2006. arXiv: cs/0603074 [cs.NI].

[17] Sadjad Fouladi κ.ά. "Encoding, Fast and Slow: Low-Latency Video Processing Using Thousands of Tiny Threads". Στο: *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. Boston, MA: USENIX Association, Μαρ. 2017, σσ. 363–376. ISBN: 978-1-931971-37-9. URL: https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/fouladi.

[18] Sadjad Fouladi κ.ά. "From Laptop to Lambda: Outsourcing Everyday Jobs to Thousands of Transient Functional Containers". Στο: *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. Renton, WA: USENIX Association, Ιούλ. 2019, σσ. 475–488. ISBN: 978-1-939133-03-8. URL: http://www.usenix.org/conference/atc19/presentation/fouladi.

[19] Apache Software Foundation. *Apache Flink*. Accessed: 2024-06-16. 2024. URL: https://flink.apache.org/.

[20] Inc. Free Software Foundation. *GNU Bash*. Accessed: 2024-06-13. 2024. URL: https://www.gnu.org/software/bash/.

[21] W. Gentzsch. "Sun Grid Engine: Towards Creating a Compute Power Grid". Στο: *Cluster Computing and the Grid, IEEE International Symposium on*. Los Alamitos, CA, USA: IEEE Computer Society, Μάι. 2001, σ. 35. DOI: 10.1109/CCGRID.2001.923173. URL: https://doi.ieeecomputersociety.org/10.1109/CCGRID.2001.923173.

[22] GitHub. *The State of the Octoverse: The Top Programming Languages*. Accessed: 2024-06-13. 2022. URL: https://octoverse.github.com/2022/top-programming-languages.

[23] Google. *Resource Limits*. Accessed: 2024-06-13. 2024. URL: https://cloud.google.com/functions/quotas#resource_limits.

[24] Michael Greenberg. *The POSIX Shell Is an Interactive DSL for Concurrency*. URL: https://cs.pomona.edu/~michael/papers/dsldi2018.pdf.

[25] Michael Greenberg, Konstantinos Kallas και Nikos Vasilakis. "Unix shell programming: the next 50 years". Στο: *Proceedings of the Workshop on Hot Topics in Operating Systems*. HotOS '21. Ann Arbor, Michigan: Association for Computing Machinery, 2021, σσ. 104–111. ISBN: 9781450384384. DOI: 10.1145/3458336.3465294. URL: https://doi.org/10.1145/3458336.3465294.

[26] Vipul Gupta κ.ά. "OverSketched Newton: Fast Convex Optimization for Serverless Systems". Στο: *2020 IEEE International Conference on Big Data (Big Data)*. 2020, σσ. 288–297. DOI: 10.1109/BigData50022.2020.9378289.

[27] Vipul Gupta κ.ά. *Serverless Straggler Mitigation using Local Error-Correcting Codes*. 2020. arXiv: 2001.07490 [cs.DC].

[28] IETF. *Session Traversal Utilities for NAT (STUN)*. Accessed: 2024-06-13. 2020. URL: https://www.rfc-editor.org/rfc/rfc8489.

[29] Serverless Inc. *Serverless Framework*. Accessed: 2024-06-13. 2024. URL: https://www.serverless.com/.

[30] Vatche Ishakian, Vinod Muthusamy και Aleksander Slominski. *Serving deep learning models in a serverless platform*. 2018. arXiv: 1710.08460 [cs.DC].

[31] Morris Jette, Andy Yoo και Mark Grondona. "SLURM: Simple linux utility for resource management". Στο: Ιούλ. 2003. ISBN: 978-3-540-20405-3. DOI: 10.1007/10968987_3.

[32] Jiawei Jiang κ.ά. "Towards Demystifying Serverless Machine Learning Training". Στο: *Proceedings of the 2021 International Conference on Management of Data*. SIGMOD '21. Virtual Event, China: Association for Computing Machinery, 2021, σσ. 857–871. ISBN: 9781450383431. DOI: 10.1145/3448016.3459240. URL: https://doi.org/10.1145/3448016.3459240.

[33] Eric Jonas κ.ά. "Occupy the cloud: distributed computing for the 99%". Στο: *Proceedings of the 2017 Symposium on Cloud Computing*. SoCC '17. Santa Clara, California: Association for Computing Machinery, 2017, σσ. 445–451. ISBN: 9781450350280. DOI: 10.1145/3127479.3128601. URL: https://doi.org/10.1145/3127479.3128601.

[34] Konstantinos Kallas κ.ά. "Practically Correct, Just-in-Time Shell Script Parallelization". Στο: *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. Carlsbad, CA: USENIX Association, Ιούλ. 2022, σσ. 769–785. ISBN: 978-1-939133-28-1. URL: https://www.usenix.org/conference/osdi22/presentation/kallas.

[35] Dimitrios Kaloudas, Nikolet Pavlova και Robert Penchovsky. "EBWS: Essential Bioinformatics Web Services for Sequence Analyses". Στο: *IEEE/ACM Transactions on Computational Biology and Bioinformatics* 16.3 (2019), σσ. 942–953. DOI: 10.1109/TCBB.2018.2816645.

[36] Youngbin Kim και Jimmy Lin. *Serverless Data Analytics with Flint*. 2018. arXiv: 1803.06354 [cs.DC].

[37] Ana Klimovic κ.ά. "Pocket: Elastic Ephemeral Storage for Serverless Analytics". Στο: *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. Carlsbad, CA: USENIX Association, Οκτ. 2018, σσ. 427–444. ISBN: 978-1-939133-08-3. URL: https://www.usenix.org/conference/osdi18/presentation/klimovic.

[38] Ana Klimovic κ.ά. "Understanding Ephemeral Storage for Serverless Analytics". Στο: *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. Boston, MA: USENIX Association, Ιούλ. 2018, σσ. 789–794. ISBN: 978-1-939133-01-4. URL: https://www.usenix.org/conference/atc18/presentation/klimovic-serverless.

[39] Nokia Bell Labs. *The unix game—solve puzzles using unix pipes*. Accessed: 2020-03-05. 2019.

[40] Georgios Liargkovas κ.ά. "Executing Shell Scripts in the Wrong Order, Correctly". Στο: *Proceedings of the 19th Workshop on Hot Topics in Operating Systems*. HOTOS '23. Providence, RI, USA: Association for Computing Machinery, 2023, σσ. 103–109. ISBN: 9798400701955. DOI: 10.1145/3593856.3595891. URL: https://doi.org/10.1145/3593856.3595891.

[41] David H. Liu κ.ά. "Doing More with Less: Orchestrating Serverless Applications without an Orchestrator". Στο: *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. Boston, MA: USENIX Association, Απρ. 2023, σσ. 1505–1519. ISBN: 978-1-939133-33-5. URL: https://www.usenix.org/conference/nsdi23/presentation/liu-david.

[42] Ashraf Mahgoub κ.ά. "WISEFUSE: Workload Characterization and DAG Transformation for Serverless Workflows". Στο: *Proc. ACM Meas. Anal. Comput. Syst.* 6.2 (Ιούν. 2022). DOI: 10.1145/3530892. URL: https://doi.org/10.1145/3530892.

[43] Chris McDonald και Trevor I. Dix. "Support for graphs of processes in a command interpreter". Στο: *Software: Practice and Experience* 18.10 (1988), σσ. 1011–1016. DOI: https://doi.org/10.1002/spe.4380181007. eprint: URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.4380181007.

[44] Suejb Memeti και Sabri Pllana. "Analyzing Large-Scale DNA Sequences on Multi-core Architectures". Στο: *2015 IEEE 18th International Conference on Computational Science and Engineering*. 2015, σσ. 208–215. DOI: 10.1109/CSE.2015.25.

[45] Microsoft. *Azure Functions Service Limits*. Accessed: 2024-06-13. 2024. URL: https://learn.microsoft.com/en-us/azure/azure-functions/functions-scale.

[46] Microsoft. *Azure Logic Apps*. Accessed: 2024-06-11. 2024. URL: https://azure.microsoft.com/en-us/products/logic-apps/.

[47] Microsoft. *What are Durable Functions?* Accessed: 2024-06-11. 2024. URL: https://learn.microsoft.com/en-us/azure/azure-functions/durable/durable-functions-overview.

[48] Ingo Müller, Renato Marroquin και Gustavo Alonso. "Lambada: Interactive Data Analytics on Cold Data Using Serverless Cloud Infrastructure". Στο: *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. SIGMOD '20. Portland, OR, USA: Association for Computing Machinery, 2020, σσ. 115–130. ISBN: 9781450367356. DOI: 10.1145/3318464.3389758. URL: https://doi.org/10.1145/3318464.3389758.

[49] Tammam Mustafa κ.ά. "DiSh: Dynamic Shell-Script Distribution". Στο: *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. Boston, MA: USENIX Association, Απρ. 2023, σσ. 341–356. ISBN: 978-1-939133-33-5. URL: https://www.usenix.org/conference/nsdi23/presentation/mustafa.

[50] Stack Overflow. *Stack Overflow Developer Survey 2023*. Accessed: 2024-06-13. 2023. URL: https://survey.stackoverflow.co/2023/#section-most-popular-technologies-programming-scripting-and-markup-languages.

[51] Matthew Perron κ.ά. "Cackle: Analytical Workload Cost and Performance Stability With Elastic Pools". Στο: *Proc. ACM Manag. Data* 1.4 (Δεκ. 2023). DOI: 10.1145/3626720. URL: https://doi.org/10.1145/3626720.

[52] Matthew Perron κ.ά. "Starling: A Scalable Query Engine on Cloud Functions". Στο: *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. SIGMOD '20. Portland, OR, USA: Association for Computing Machinery, 2020, σσ. 131–141. ISBN: 9781450367356. DOI: 10.1145/3318464.3380609. URL: https://doi.org/10.1145/3318464.3380609.

[53] Qifan Pu, Shivaram Venkataraman και Ion Stoica. "Shuffling, Fast and Slow: Scalable Analytics on Serverless Infrastructure". Στο: *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. Boston, MA: USENIX Association, Φεβ. 2019, σσ. 193–206. ISBN: 978-1-931971-49-2. URL: https://www.usenix.org/conference/nsdi19/presentation/pu.

[54] Deepti Raghavan κ.ά. "POSH: A Data-Aware Shell". Στο: *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. USENIX Association, Ιούλ. 2020, σσ. 617–631. ISBN: 978-1-939133-14-4. URL: https://www.usenix.org/conference/atc20/presentation/raghavan.

[55] Dennis M. Ritchie και Ken Thompson. "The UNIX time-sharing system". Στο: *Commun. ACM* 17.7 (Ιούλ. 1974), σσ. 365–375. ISSN: 0001-0782. DOI: 10.1145/361011.361061. URL: https://doi.org/10.1145/361011.361061.

[56] Rohan Basu Roy κ.ά. "Mashup: making serverless computing useful for HPC workflows via hybrid execution". Στο: *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. PPoPP '22. Seoul, Republic of Korea: Association for Computing Machinery, 2022, σσ. 46–60. ISBN: 9781450392044. DOI: 10.1145/3503221.3508407. URL: https://doi.org/10.1145/3503221.3508407.

[57] Ghazal Sadeghian κ.ά. "UnFaaSener: Latency and Cost Aware Offloading of Functions from Serverless Platforms". Στο: *2023 USENIX Annual Technical Conference (USENIX ATC 23)*. Boston, MA: USENIX Association, Ιούλ. 2023, σσ. 879–896. ISBN: 978-1-939133-35-9. URL: https://www.usenix.org/conference/atc23/presentation/sadeghian.

[58] Josep Sampé κ.ά. "Serverless Data Analytics in the IBM Cloud". Στο: *Proceedings of the 19th International Middleware Conference Industry*. Middleware '18. Rennes, France: Association for Computing Machinery, 2018, σσ. 1–8. ISBN: 9781450360166. DOI: 10.1145/3284028.3284029. URL: https://doi.org/10.1145/3284028.3284029.

[59] Amazon Web Services. *Amazon DynamoDB*. Accessed: 2024-06-13. 2024. URL: https://aws.amazon.com/dynamodb/.

[60] Amazon Web Services. *Amazon Simple Queue Service (SQS)*. Accessed: 2024-06-16. 2024. URL: https://aws.amazon.com/sqs/.

[61] Amazon Web Services. *Amazon Simple Storage Service (S3)*. Accessed: 2024-06-16. 2024. URL: https://aws.amazon.com/s3/.

[62] Amazon Web Services. *Amazon SQS Long Polling*. Accessed: 2024-06-13. 2024. URL: https://docs.aws.amazon.com/AWSSimpleQueueService/latest/SQSDeveloperGuide/sqs-short-and-long-polling.html#sqs-long-polling.

[63] Amazon Web Services. *AWS Lambda*. Accessed: 2024-06-16. 2024. URL: https://aws.amazon.com/lambda/.

[64] Amazon Web Services. *AWS Lambda Custom Runtimes*. Accessed: 2024-06-13. 2024. URL: https://docs.aws.amazon.com/lambda/latest/dg/runtimes-custom.html.

[65] Amazon Web Services. *AWS Lambda Layers*. Accessed: 2024-06-13. 2024. URL: https://docs.aws.amazon.com/lambda/latest/dg/chapter-layers.html.

[66] Amazon Web Services. *AWS Lambda Supported Runtimes*. Accessed: 2024-06-13. 2024. URL: https://docs.aws.amazon.com/lambda/latest/dg/lambda-runtimes.html#runtimes-supported.

[67] Amazon Web Services. *AWS Lambda: Cold Starts and Latency*. Accessed: 2024-06-13. 2024. URL: https://docs.aws.amazon.com/lambda/latest/operatorguide/execution-environments.html#cold-start-latency.

[68] Amazon Web Services. *AWS SDK for Python (Boto3)*. Accessed: 2024-06-13. 2024. URL: https://aws.amazon.com/sdk-for-python/.

[69] Amazon Web Services. *AWS Step Functions*. Accessed: 2024-06-11. 2024. URL: https://aws.amazon.com/step-functions/.

[70] Amazon Web Services. *Configure Lambda Function Memory*. Accessed: 2024-06-13. 2024. URL: https://docs.aws.amazon.com/lambda/latest/dg/configuration-memory.html.

[71] Amazon Web Services. *Function Configuration, Deployment, and Execution*. Accessed: 2024-06-13. 2024. URL: https://docs.aws.amazon.com/lambda/latest/dg/gettingstarted-limits.html#function-configuration-deployment-and-execution.

[72] Amazon Web Services. *Memory and Computing Power*. Accessed: 2024-06-13. 2024. URL: https://docs.aws.amazon.com/lambda/latest/operatorguide/computing-power.html.

[73] Mohammad Shahrad κ.ά. "Serverless in the Wild: Characterizing and Optimizing the Serverless Workload at a Large Cloud Provider". Στο: *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. USENIX Association, Ιούλ. 2020, σσ. 205–218. ISBN: 978-1-939133-14-4. URL: https://www.usenix.org/conference/atc20/presentation/shahrad.

[74] Vaishaal Shankar κ.ά. "Serverless linear algebra". Στο: *Proceedings of the 11th ACM Symposium on Cloud Computing*. SoCC '20. Virtual Event, USA: Association for Computing Machinery, 2020, σσ. 281–295. ISBN: 9781450381376. DOI: 10.1145/3419111.3421287. URL: https://doi.org/10.1145/3419111.3421287.

[75] Prasoon Sinha, Kostis Kaffes και Neeraja J. Yadwadkar. "Online Learning for Right-Sizing Serverless Functions". Στο: *Architecture and System Support for Transformer Models (ASSYST @ISCA 2023)*. 2023. URL: https://openreview.net/forum?id=4zdPNY3SDQk.

[76] *Solutions for unixgame.io*. Accessed: 2024-06-13. URL: https://github.com/psinghbh/softsec.github.io/tree/master/ctf/unixgame.io.

[77] Diomidis Spinellis και Marios Fragkoulis. "Extending Unix Pipelines to DAGs". Στο: *IEEE Transactions on Computers* 66.9 (2017), σσ. 1547–1561. DOI: 10.1109/TC.2017.2695447.

[78] Vikram Sreekanti κ.ά. *Optimizing Prediction Serving on Low-Latency Serverless Dataflow*. 2020. arXiv: 2007.05832 [cs.DC].

[79] Ole Tange. "GNU Parallel - The Command-Line Power Tool". Στο: *The USENIX Magazine* 36 (Ιαν. 2011), σσ. 42–47.

[80] Dave Taylor και Brandon Perry. *Wicked Cool Shell Scripts, 2nd Edition*. 2016. URL: https://nostarch.com/wcss2.

[81] Eleftheria Tsaliki και Diomidis Spinellis. *The Real Numbers for Athens Buses (in Greek)*. Accessed: 2024-06-13. 2020. URL: https://insidestory.gr/article/noymera-leoforeia-athinas.

[82] Nikos Vasilakis κ.ά. "PaSh: light-touch data-parallel shell processing". Στο: *Proceedings of the Sixteenth European Conference on Computer Systems*. EuroSys '21. Online Event, United Kingdom: Association for Computing Machinery, 2021, σσ. 49–66. ISBN: 9781450383349. DOI: 10.1145/3447786.3456228. URL: https://doi.org/10.1145/3447786.3456228.

[83] Edward Walker, Weijia Xu και Vinoth Chandar. "Composing and executing parallel data-flow graphs with shell pipes". Στο: *Proceedings of the 4th Workshop on Workflows in Support of Large-Scale Science*. WORKS '09. Portland, Oregon: Association for Computing Machinery, 2009. ISBN: 9781605587172. DOI: 10.1145/1645164.1645175. URL: https://doi.org/10.1145/1645164.1645175.

[84] Sebastian Werner κ.ά. "Serverless Big Data Processing using Matrix Multiplication as Example". Στο: *2018 IEEE International Conference on Big Data (Big Data)*. 2018, σσ. 358–365. DOI: 10.1109/BigData.2018.8622362.

[85] Tom White. *Hadoop: The Definitive Guide*. 2009. URL: https://www.oreilly.com/library/view/hadoop-the-definitive/9780596521974/.

[86] Matei Zaharia κ.ά. "Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing". Στο: *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*. San Jose, CA: USENIX Association, Απρ. 2012, σσ. 15–28. ISBN: 978-931971-92-8. URL: https://www.usenix.org/conference/nsdi12/technical-sessions/presentation/zaharia.

[87] Wen Zhang κ.ά. "Kappa: a programming framework for serverless computing". Στο: *Proceedings of the 11th ACM Symposium on Cloud Computing*. SoCC '20. Virtual Event, USA: Association for Computing Machinery, 2020, σσ. 328–343. ISBN: 9781450381376. DOI: 10.1145/3419111.3421277. URL: https://doi.org/10.1145/3419111.3421277.

[88] Zhao Zhang κ.ά. "Parallelizing the execution of sequential scripts". Στο: *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. SC '13. Denver, Colorado: Association for Computing Machinery, 2013. ISBN: 9781450323789. DOI: 10.1145/2503210.2503222. URL: https://doi.org/10.1145/2503210.2503222.

Παράρτημα Α

Πλήρες Πρόγραμμα Κελύφους που Παράγεται από το PaSh για το NFA-Regex

```
#!/bin/bash
cd "$(dirname "${0}")"
[ -z "${PASH_TOP}" ]
rm_pash_fifos() {
    rm -f "/tmp/Fgq4/f2";
    rm -f "/tmp/Fgq4/f7"
    # ...similarly for the rest of the fifos...
}
mkfifo_pash_fifos() {
    mkfifo "/tmp/Fgq4/f2"
    mkfifo "/tmp/Fgq4/f7"
    # ...similarly for the rest of the fifos...
}
rm_pash_fifos; mkfifo_pash_fifos; pids_to_kill=""
cat "in.txt" >"/tmp/Fgq4/f2" & pids_to_kill="${!} ${pids_to_kill}"
r_split "/tmp/Fgq4/f2" 1000000 "/tmp/Fgq4/f7" "/tmp/Fgq4/f8" &
pids_to_kill="${!} ${pids_to_kill}"
r_wrap bash -c 'tr A-Z a-z' <"/tmp/Fgq4/f13" >"/tmp/Fgq4/f9" &
pids_to_kill="${!} ${pids_to_kill}"
r_wrap bash -c 'tr A-Z a-z' <"/tmp/Fgq4/f14" >"/tmp/Fgq4/f10" &
pids_to_kill="${!} ${pids_to_kill}"
r_wrap bash -c 'grep \(.\).*\1\(.\).*\2\(.\).*\3\(.\).*\4' <"/tmp/Fgq4/f9" >"/tmp/Fgq4/f11" &
pids_to_kill="${!} ${pids_to_kill}"
r_wrap bash -c 'grep \(.\).*\1\(.\).*\2\(.\).*\3\(.\).*\4' <"/tmp/Fgq4/f10" >"/tmp/Fgq4/f12" &
pids_to_kill="${!} ${pids_to_kill}"
dgsh-tee -i "/tmp/Fgq4/f7" -o "/tmp/Fgq4/f13" -b 5M & pids_to_kill="${!} ${pids_to_kill}"
dgsh-tee -i "/tmp/Fgq4/f8" -o "/tmp/Fgq4/f14" -b 5M & pids_to_kill="${!} ${pids_to_kill}"
dgsh-tee -i "/tmp/Fgq4/f11" -o "/tmp/Fgq4/f15" -b 5M & pids_to_kill="${!} ${pids_to_kill}"
dgsh-tee -i "/tmp/Fgq4/f12" -o "/tmp/Fgq4/f16" -b 5M & pids_to_kill="${!} ${pids_to_kill}"
r_merge "/tmp/Fgq4/f15" "/tmp/Fgq4/f16" & pids_to_kill="${!} ${pids_to_kill}"
source wait_for_output_and_sigpipe_rest.sh ${!}; rm_pash_fifos; (exit "${internal_exec_status}")
```

Κώδικας Α.1: Πλήρες πρόγραμμα κελύφους που παράγεται από το PaSh για το NFA-Regex (`--width=2`).

Παράρτημα Β

Τελικά Προγράμματα Κελύφους που Παράγονται από το SPLASH για το NFA-Regex

```
#!/bin/bash
mkfifo f{0..3}
invoke-function $SCRIPT_ID_2 $SCRIPT_MAP &
invoke-function $SCRIPT_ID_3 $SCRIPT_MAP &
download-object in.txt >f0 &
cat f0 >f1 &
split f1 f2 f3 &
send-stream $COMM_KEY_1_2 <f2 &
send-stream $COMM_KEY_1_3 <f3 &
wait
```

Κώδικας Β.1: Τελικό πρόγραμμα κελύφους για τον Υπογράφο 1. Οι εντολές που είναι σχετικές με εκτέλεση χωρίς διακομιστή είναι υπογραμμισμένες.

```
#!/bin/bash
mkfifo f{0..4}
invoke-function $SCRIPT_ID_4 $SCRIPT_MAP &
receive-stream $COMM_KEY_1_2 >f0 &
eager <f0 >f1 &
tr A-Z a-z <f1 >f2 &
grep '\(.\).*\1\(.\).*\2\(.\).*\3\(.\).*\4' <f2 >f3 &
eager <f3 >f4 &
send-stream $COMM_KEY_2_4 <f4 &
wait
```

Κώδικας Β.2: Τελικό πρόγραμμα κελύφους για τον Υπογράφο 2. Οι εντολές που είναι σχετικές με εκτέλεση χωρίς διακομιστή είναι υπογραμμισμένες.

```
#!/bin/bash
mkfifo f{0..4}
receive-stream $COMM_KEY_1_3 >f0 &
eager <f0 >f1 &
tr A-Z a-z <f1 >f2 &
grep '\(.\).*\1\(.\).*\2\(.\).*\3\(.\).*\4' <f2 >f3 &
eager <f3 >f4 &
send-stream $COMM_KEY_3_4 <f4 &
wait
```

Κώδικας Β.3: Τελικό πρόγραμμα κελύφους για τον Υπογράφο 3. Οι εντολές που είναι σχετικές με εκτέλεση χωρίς διακομιστή είναι υπογραμμισμένες.

```
#!/bin/bash
mkfifo f{0..2}
receive-stream $COMM_KEY_2_4 >f0 &
receive-stream $COMM_KEY_3_4 >f1 &
merge f0 f1 >f2 &
upload-object <f2 out.txt &
wait
send-notification
```

Κώδικας Β.4: Τελικό πρόγραμμα κελύφους για τον Υπογράφο 4. Οι εντολές που είναι σχετικές με εκτέλεση χωρίς διακομιστή είναι υπογραμμισμένες.

Παράρτημα C

Κώδικας Python για την Αλληλεπίδραση με τις Υπηρεσίες της AWS

```python
import boto3
import sys
import json

id_, data_path = sys.argv[1:]

with open(data_path, "r") as f:
    data = f.read()

lambda_client = boto3.client("lambda")

lambda_client.invoke(
    FunctionName="lambda",
    InvocationType="Event",
    LogType="None",
    Payload=json.dumps({"id": id_, "data": data}),
)
```

Κώδικας C.1: Κλήση συνάρτησης.

```python
import subprocess
import json

def lambda_handler(event, context):
    data = event["data"]
    id_ = event["id"]
    scripts_dict = json.loads(data)

    with open(f"/tmp/data-{id_}", "w") as f:
        f.write(data)

    with open(f"/tmp/script-{id_}.sh", "w") as f:
        f.write(scripts_dict[id_])

    process = subprocess.run(
        ["/bin/bash", f"/tmp/script-{id_}.sh", f"/tmp/data-{id_}"]
    )
```

Κώδικας C.2: Χειριστής συνάρτησης.

```python
import boto3
import json

sqs_client = boto3.client("sqs")
message_body = {"message": "done", "output_file_id": object_key}

try:
    response = sqs_client.send_message(
        QueueUrl=f"https://sqs.us-east-1.amazonaws.com/{AWS_ACCOUNT_ID}/{QUEUE}",
        MessageBody=json.dumps(message_body),
    )
except Exception as e:
    print(e)
```

Κώδικας C.3: Αποστολή μηνύματος.

```python
import time
import boto3

def wait_msg_done():
    while True:
        sqs = boto3.client("sqs")
        queue_url = f"https://sqs.us-east-1.amazonaws.com/{AWS_ACCOUNT_ID}/{QUEUE}"

        response = sqs.receive_message(
            QueueUrl=queue_url,
            AttributeNames=["SentTimestamp"],
            MaxNumberOfMessages=1,
            MessageAttributeNames=["All"],
            VisibilityTimeout=30,
            WaitTimeSeconds=20,
        )

        try:
            message = response["Messages"][0]
            receipt_handle = message["ReceiptHandle"]

            sqs.delete_message(QueueUrl=queue_url, ReceiptHandle=receipt_handle)

            break
        except:
            time.sleep(1)
```

Κώδικας C.4: Λήψη μηνύματος.

```
import boto3
import sys
import os

BUCKET = os.environ.get("AWS_BUCKET")

object_key, outfile = sys.argv[1:]
batch = 10000

session = boto3.Session()
s3 = session.client("s3")

try:
    response = s3.get_object(Bucket=BUCKET, Key=object_key)
except Exception as e:
    print(e)

with open(outfile, "wb") as f:
    while True:
        x = response["Body"].read(batch)

        if not x:
            break

        f.write(x)
        f.flush()
```

Κώδικας C.5: Λήψη από αποθήκη νέφους.

```python
import boto3
import sys
import json
import os

AWS_ACCOUNT_ID = os.environ.get("AWS_ACCOUNT_ID")
BUCKET = os.environ.get("AWS_BUCKET")
QUEUE = os.environ.get("AWS_QUEUE")

object_key, infile = sys.argv[1:]

session = boto3.Session()
s3 = session.client("s3")

with open(infile, "rb") as file:
    object_data = file.read()

s3.put_object(Bucket=BUCKET, Key=object_key, Body=object_data)

sqs_client = boto3.client("sqs")
message_body = {"message": "done", "output_file_id": object_key}
try:
    response = sqs_client.send_message(
        QueueUrl=f"https://sqs.us-east-1.amazonaws.com/{AWS_ACCOUNT_ID}/{QUEUE}",
        MessageBody=json.dumps(message_body),
    )
except Exception as e:
    print(e)
```

Κώδικας C.6: Μεταφόρτωση σε αποθήκη νέφους.

Παράρτημα D

Ισοδύναμες των AWS Υπηρεσίες σε Άλλους Παρόχους Νέφους

Πίνακας D.1: Ισοδύναμες των AWS υπηρεσίες. Η χρήση ισοδύναμων υπηρεσιών από άλλους παρόχους νέφους μπορεί να προσφέρει ισοδύναμη συμπεριφορά, όμως δεν είναι εγγυημένο ότι θα προσφέρει ισοδύναμη επίδοση.

AWS Service	Google Cloud Equivalent	Microsoft Azure Equivalent
AWS Lambda	Google Cloud Functions	Azure Functions
Amazon S3	Google Cloud Storage	Azure Blob Storage
Amazon DynamoDB	Google Bigtable	Azure Cosmos DB
Amazon SQS	Google Cloud Pub/Sub	Azure Service Bus

Παράρτημα Ε

Πλήρης Χαρακτηρισμός των Προγραμμάτων που Χρησιμοποιήθηκαν στην Αξιολόγηση

Πίνακας Ε.1: Προγράμματα κελύφους που περιλαμβάνονται στο σύνολο `Oneliners`.

	Script	Input	Seq. Time	`--width`	Notes
1	nfa-regex.sh	200 MB	598.74 s	32	Match complex regex over input
2	sort-sort.sh	200 MB	20.11 s	4	Calculate sort twice
3	sort.sh	200 MB	28.74 s	4	Sort input
4	spell.sh	200 MB	62.37 s	16	Calculate mispelled words
5	top-n.sh	100 MB	40.98 s	8	Find top 100 terms
6	wf.sh	100 MB	40.76 s	8	Sort words by frequency

Πίνακας Ε.2: Προγράμματα κελύφους που περιλαμβάνονται στο σύνολο `Unix50`.

	Script	Input	Seq. Time	`--width`	Notes
1	1.sh	1.1 GB	45.32 s	2	Extract the last name
2	2.sh	1.1 GB	42.35 s	4	Get all Unix utilities
3	3.sh	1.1 GB	46.28 s	4	Get lowercase first letter of last names
4	4.sh	1.1 GB	48.26 s	4	Extract hello world
5	5.sh	1.1 GB	52.75 s	2	Extract the word BELL
6	6.sh	1.1 GB	38.14 s	4	Four corners
7	7.sh	1.1 GB	38.75 s	2	List Turing award recipients while working at Bell Labs
8	8.sh	1.1 GB	39.15 s	4	Year Ritchie and Thompson receive the Hamming medal

Πίνακας Ε.3: Προγράμματα κελύφους που περιλαμβάνονται στο σύνολο `COVID-mts`.

	Script	Input	Seq. Time	`--width`	Notes
1	1.sh	200 MB	20.29 s	2	Vehicles on the road per day
2	2.sh	500 MB	43.35 s	2	Days a vehicle is on the road
3	3.sh	500 MB	48.64 s	2	Hours each vehicle is on the road
4	4.sh	1 GB	46.90 s	4	Hours monitored each day

Πίνακας Ε.4: Προγράμματα κελύφους που περιλαμβάνονται στο σύνολο `NLP`.

	Script	Input	Seq. Time	`--width`	Notes
1	1.sh	10 books	19.47 s	2	Count words
2	2.sh	10 books	21.35 s	2	Merge upper and lower counts
3	3.sh	10 books	19.66 s	2	Sort
4	4.sh	10 books	25.32 s	2	Sort words by folding
5	5.sh	10 books	19.66 s	2	Uppercase by token
6	6.sh	10 books	21.27 s	2	Uppercase by type
7	7.sh	10 books	39.81 s	2	Four-letter words
8	8.sh	10 books	19.11 s	4	Words no vowels
9	9.sh	10 books	20.49 s	4	One-syllable words
10	10.sh	10 books	16.64 s	2	Two-syllable words
11	11.sh	10 books	56.10 s	2	Verses with certain instances of the word “light”
12	12.sh	10 books	19.32 s	2	Count consonant sequences
13	13.sh	10 books	21.94 s	2	Vowel sequences greater than 1K

Πίνακας Ε.5: Προγράμματα κελύφους που περιλαμβάνονται στο σύνολο `NOAA`.

	Script	Input	Seq. Time	`--width`	Notes
1	noaa.sh	500 MB	49.98 s	8	Find maximum and minimum temperature

National Technical University of Athens
School of Electrical and Computer Engineering
Department of Computer Science

SPLaSh: Scaling Out Shell Scripts on Serverless Platforms

DIPLOMA THESIS

NIKOLAOS PAGONAS

Supervisor : Georgios Goumas
Associate Professor, NTUA

Co-Supervisor : Nikos Vasilakis
Assistant Professor, Brown University

Athens, June 2024

National Technical University of Athens
School of Electrical and Computer Engineering
Department of Computer Science

SPLaSh: Scaling Out Shell Scripts on Serverless Platforms

DIPLOMA THESIS

NIKOLAOS PAGONAS

Supervisor : Georgios Goumas
Associate Professor, NTUA

Co-Supervisor : Nikos Vasilakis
Assistant Professor, Brown University

Approved by the examining committee on June 25, 2024.

Georgios Goumas Associate Professor, NTUA	Nikolaos Papaspyrou Professor, NTUA	Nikos Vasilakis Assistant Professor, Brown University

Athens, June 2024

Nikolaos Pagonas
Graduate of the School of Electrical and Computer Engineering NTUA

Abstract

Even though the Unix shell is widely used today, no support currently exists for automatically deploying shell scripts on modern serverless platforms—missing out on key benefits such as elasticity, scalability, and pay-as-you-go pricing. SPLaSh is a new system for automatically scaling out shell scripts on serverless infrastructure, using a combination of compilation and runtime support. SPLaSh introduces a range of new shell primitives for serverless execution, establishes pipe-like streaming communication between functions, and provides push-button, just-in-time serverless deployment and scale-out. Evaluated on a set of real-world shell scripts, SPLaSh offers a range of speedups over Bash (0.75–14.38×, average: 1.75×)—without requiring any modifications to the original scripts.

Keywords

Unix, Shell, Serverless, FaaS, Function-as-a-Service, Cloud Computing

Acknowledgements

At this point, I would like to express my warm thanks to my supervisor, Prof. Georgios Goumas, for the trust he showed me, and his invaluable assistance during my diploma thesis—as well as my future academic steps. Also, I want to express my gratitude to Prof. Nikos Vasilakis, for giving me the unique opportunity to work on part of my thesis at Brown University, and enthusiastically guiding me in the early stages of my research journey. Additionally, I would like to thank Konstantinos Kallas, whose valuable advice—both technical and non-technical—was crucial, both within and outside the scope of this work. Furthermore, I thank Yizheng Xie, who was always willing to discuss and provide practical help throughout the thesis, as well as Haoran Zhang, who significantly contributed to overcoming critical technical challenges. Finally, I would like to thank my friends, who ensured—each in their own way—that all these years as a student were filled with beautiful moments, and helped me overcome any difficulties that arose.

Most of all, however, I want to say the biggest thank you to my parents, Tasos and Mina, and my sister, Evita, who always believe in me, constantly push me to strive for the best, and support my efforts with all their heart.

Nikolaos Pagonas,

Athens, June 25, 2024

Contents

List of Tables

List of Figures

List of Source Code Listings

Chapter 1

Introduction

The Unix shell is popular and powerful, and is applicable to a wide range of important tasks, such as build scripts, continuous deployment, continuous integration, data processing, and bioinformatics. However, when it comes to keeping up with the modern state of computing, the shell is lagging behind. On the one hand, existing approaches for shell script scale-out [34, 40, 54, 83] do not support serverless shell deployment. On the other, current serverless frameworks [1, 2, 8, 15, 26, 27, 30, 33, 36, 37, 53, 58, 75, 79, 85] are too domain-specific. While some systems are tailored for general-purpose serverless computing [17, 18, 88], they require programmers to use framework-specific formats, and cannot match the shell's expressiveness and dynamism. Thus, the shell is currently missing out on key benefits of serverless, such as elasticity, scalability, and pay-as-you-go-pricing.

SPLaSh is a new system for automatically scaling out shell scripts on serverless infrastructure, using a combination of compilation and runtime support. To do this, SPLaSh takes the original shell script as input, and identifies parts that can be distributed among serverless functions. It then introduces new shell primitives for serverless tasks—such as function invocation and communication, or interaction with cloud storage and messaging queues. SPLaSh also establishes streaming communication channels between functions, by allowing functions to discover each other and overcome the networking limitations of serverless. Finally, SPLaSh offers push-button support for serverless deployment and scale-out in a just-in-time fashion thus allowing for a range of dynamic performance optimizations—all while requiring zero user intervention. Evaluated on a set of real-world shell scripts, SPLaSh offers a range of speedups over Bash [20], the de facto sequential shell script execution environment (0.75–14.38×, average: 1.75×).

This diploma thesis begins by laying down the necessary background on serverless computing, Unix shell scripting, and previous attempts at scaling out shell scripts (Chapter 2). It then outlines the challenges of serverless shell execution through an example, and provides an overview of SPLaSh and its contributions (Chapter 3). It goes on to describe SPLaSh's compilation and runtime support (Chapters 4 and 5), as well as SPLaSh's evaluation (Chapter 6). Finally, it provides an overview of prior work related to SPLaSh (Chapter 7), before concluding (Chapter 8). SPLaSh is part of PaSh—an MIT-licensed project hosted by the Linux Foundation. The version of SPLaSh described in this thesis is available at `https://github.com/nikpag/splash`, while the latest version is available at `https://github.com/binpash/pash`.

Chapter 2

Background

This chapter first gives an overview of cloud computing, its shortcomings, and the emergence of serverless computing in an attempt to overcome them. It then goes over serverless in detail, outlining its benefits, its challenges, and the current state of serverless research. After that, it switches over to the Unix shell and its abstractions, strengths, and weaknesses. Finally, it provides an overview of past efforts in enhancing the shell's performance through automatic parallelization and distribution.

2.1 Cloud Computing

Cloud computing is the realization of a long-held dream of computing as a utility, where users can pay for computing resources on a pay-as-you-go basis. It provides users with access to a pool of shared resources—computing power, storage, networking—that can be quickly provisioned and released with minimal management effort.

Advantages Cloud computing offers several potential advantages. Users can have access to virtually infinite computing resources on demand, with no up-front commitment, since they pay for the use of computing resources on a short-term basis as needed. Resources can come at a reduced cost, because cloud providers have many large data centers and benefit from economies of scale. Virtualization can simplify operation and increase utilization. Cloud providers can achieve higher hardware utilization by multiplexing workloads from different organizations.

Shortcomings While the cloud offers most of these advantages, it does not fully deliver on the last two—simplification of operation and multiplexing benefits. In terms of simplified operation, the cloud relieves users of operating the physical infrastructure, but leaves them with a proliferation of virtual resources to manage. On top of that, users still have to address a multitude of challenges when setting up an environment in the cloud. Specifically, they have to (1) provide redundancy for availability, so that a single machine failure does not take down the service; (2) distribute redundant copies geographically, to preserve the service in case of disaster; (3) load balance and route requests efficiently, to utilize resources; (4) autoscale in response to changes in load to scale up or down the system; (5) monitor to make sure the service is still running well; (6) log messages needed for debugging or performance tuning; (7) handle system upgrades, including security patching; (8) migrate to new instances as they become available. In terms of the benefits the come from multiplexing, these are mostly seen in batch style workloads such as MapReduce [13] or high performance computing—which can fully utilize the instances they allocate. They are not seen as much in stateful services (e.g., when porting enterprise software—like database management systems—to the cloud).

Furthermore, rapid, hard to predict resource demand changes make provisioning a challenge. Users must either over provision at excessive cost, or suffer high latency when demand spikes. Because large workloads may occur at any moment—from a large number of concurrent requests, requests that need a lot of computing resources, or both—resource demands fluctuate in rapid, unpredictable bursts. While it is possible to employ auto-scaling mechanisms to handle changes in demand—by adding virtual machines (VMs) when demand rises, and removing them when they are idle—new nodes take tens of

seconds to minutes to come online, incurring high latency. Also, nodes must remain idle for a period of time until they are taken down—with the exact period of time depending on the application and workload, and thus being hard to predict—leading to possibly extra charges for idle instances.

2.2 Serverless Computing

In order to solve the issues above—and because of a demand for even higher elasticity and more fine-grained billing—a new cloud paradigm called *serverless computing* has emerged, which abstracts away all infrastructure management tasks from the user, while providing elasticity, scalability, and cost-effectiveness. Serverless has been widely adopted by cloud users—and is still increasing in popularity. In 2023, 70% of AWS customers used serverless offerings, compared to 50% in 2021. The case is similar for Google Cloud (60%, up from 20%) and Microsoft Azure (50%, up from 35%) [11, 12].

Overview For a service to be considered serverless, it must scale automatically with no need for explicit provisioning, and be billed based on usage. Serverless differs from previous approaches because it decouples computation from other resources, allows users to execute code without having to manage the underlying infrastructure, and provides pricing based on resource usage—not resource allocation. It consists of two components: Function-as-a-Service (FaaS) and Backend-as-a-Service (BaaS). With FaaS, users write functions that are executed in response to events, and the cloud provider takes care of the infrastructure management tasks—such as scaling, provisioning, and monitoring. BaaS encompasses any application-specific serverless cloud service, such as storage, databases, or messaging. This decouples computation from other resources, allowing each resource to scale independently.

The serverless primitive is the cloud function: the user writes in a high-level language, picks an event that triggers the function, and the serverless system handles everything else: resource allocation, instance selection, deployment, operating system management, load balancing, auto-scaling, fault tolerance, monitoring, logging, security patches, and so on. The platform transparently starts and stops concurrent instances of a serverless function as demand rises and falls. This way, programmers can focus on writing application code. Serverless is charged in a much more fine-grained way—providing a minimum billing increment of 100 ms—while other cloud offerings charge by the hour. This way, the customer is charged for the time their code is actually executing—not for the resources reserved to execute their program.

Benefits Serverless computing provides benefits for several parties. For cloud providers, serverless computing promotes business growth, as making the cloud easier to program helps draw in new customers, and helps existing customers make more use of cloud offerings. Short run time, small memory footprint, and stateless nature improve statistical multiplexing by making it easier for cloud providers to find unused resources on which to run these tasks. Cloud users benefit from increased programming productivity, since novices can deploy functions without any understanding of the cloud infrastructure, while experts save development time and stay focused on problems unique to their application. Cloud users can also save money in many scenarios, since the functions are executed only when events occur, and fine-grained pricing means they pay only for what they use instead of what they reserve. Researchers have been attracted to serverless computing, because it is a new general-purpose compute abstraction that promises to become the future of cloud computing, and because there are many opportunities for boosting the current performance and overcoming its current limitations.

Serverless for Complex Applications While serverless platforms originally targeted simple applications with one or a few functions, this paradigm has increasingly proven useful for more complex applications composed of many functions with rich, stateful interaction patterns, such as data analytics [33, 36, 37, 38, 53, 58], numerical computing [2, 15, 26, 27, 33, 75, 85], video encoding [1, 17, 18], machine learning [8, 30, 79], and source code compilation [18]. In 2019, the majority of serverless applications

were composed of just one function, and 80% had three or fewer functions [74]. Today however, complex serverless workflows are not rare anymore. A recent study of open-source serverless projects has identified 31% of studied applications to have workflow structures [14]. From 2019 to 2022, the popularity of serverless workflows structured as Directed Acyclic Graphs (DAGs) has grown by 6× at Microsoft Azure [42]. The increased complexity of serverless applications can be attributed to the maturation of serverless offerings and the increased proficiency of developers [57].

Challenges Despite its benefits, serverless computing introduces its own set of challenges.

- Serverless development is more restrictive, requiring programmers to write functions in a specific way. For example, serverless guarantees at-least-once execution, so functions with external effects must be idempotent, to ensure that re-execution is safe.
- Functions have transient in-memory and on-disk state, so programmers have to ensure that all persistent state is saved to external storage (e.g., cloud storage or a cloud-hosted database).
- The serverless platform imposes a hard timeout on all functions, so functions have limited runtime—a few minutes—before they are killed. If a programmer needs to perform a lengthier computation, they need to break it up into smaller functions.
- Serverless suffers from restricted network connectivity. Individual functions cannot communicate directly with each other, so users resort to manual implementations of slow, storage-based communication, or in-memory caches. These solutions either hamper performance, have prohibitive cost, or introduce a user-managed component—defeating the purpose of serverless.
- Broadcast, aggregation, and shuffle—some of the most common communication primitives in distributed systems—are not well supported by serverless. Event-driven execution makes depending on the results of multiple previous functions—and therefore fan-in patterns—difficult.
- None of the existing cloud storage services come with notification capabilities. While cloud providers do offer standalone notification services, they add significant latency—hundreds of milliseconds—and can be costly when used for fine grained coordination.

Serverless Data Exchange Serverless has a range of options for data exchange. First, direct network connections could offer high performance without incurring any additional cost for communication. However, functions are hidden behind Network Address Translators (NATs), and do not accept incoming connections. Second, object storage provides high throughput, strong consistency, and data reliability. However, it does not support streaming writes, introduces high latency, and incurs high cost. Third, NoSQL databases and message queueing services offer low latency and high throughput. However, they have low size limits and high costs. Finally, streaming services require users to provision capacity ahead of time, are charged hourly based on the number of resources provisioned, and are costly.

2.3 The Unix Shell

The Unix shell is an environment—often interactive—for composing programs that are written in a plethora of programming languages. It provides a range of useful components called commands, as well as powerful and language-agnostic primitives for composing components—in line with the toolbox philosophy of Unix [55]. The shell has been consistently ranked among the most popular programming languages [22, 50], while in 2021 it ranked sixth for popularity increase—above languages with active communities, such as Python and Kotlin [40]. Shell scripts are critical infrastructure for developers, administrators, and scientists, and are used in data processing, system orchestration, and automation tasks, as well as in modern platforms like Docker, Vagrant, and Kubernetes.

```
#!/bin/bash
mkfifo t1 t2
grep "foo" f1 >t1 &
grep "foo" f2 >t2 &
cat t1 t2
```

Listing 2.1: Laziness in the context of the shell. The `cat` command will start consuming input from `t2` only after it completes reading from `t1`, leading to underutilization.

The shell has a range of benefits. First, it is succinct and expressive. One hundred lines of Java code that perform a temperature analysis task can be translated to a single-line Bash script [25]. Second, the shell is performant. It supports pipeline parallelism by using Unix pipes to stream data, and task parallelism by sending commands to run in the background with the background operator (`&`). Third, the shell is dynamic, having features such as command substitution and variable expansion.

Shell Abstractions The Unix shell features a series of useful abstractions. First, *Unix data streams* are sequences of bytes, commonly processed in a line-by-line fashion, and often referenced using a filename. They facilitate the composition of commands, in the way that the one command's output is another command's output. Data streams can be ephemeral, unnamed pipes—expressed using the pipe character—or persistent, named pipes—Unix FIFOs—created with `mkfifo`. Data streams introduce pipeline parallelism between commands, and the Unix kernel facilitates scheduling, communication, and synchronization behind the scenes.

Second, *commands* are independent computation units. They read one or more input streams, and produce one or more output streams. One major difference with other languages that have a closed set of primitives, is that there is an unlimited number of Unix commands, and each command can have arbitrary behaviors. These commands may be written in any language—or exist in plain binary form—something that makes it hard to reason about their performance characteristics. Commands can often be configured by using environment variables and command flags. For example, the `wc` command supports flags `-l`, `-w`, and `-c`—for counting the number of lines, words, and characters, respectively. Flags are important in performance analysis, because they may change the parallelizability of a command.[1] Commands are often lazy—they consume their inputs only when they are ready to process more—something that often leads to CPU underutilization (Listing 2.1).

Finally, the Unix shell also features a series of composition operators. The sequential operator (`;`) executes commands sequentially, the parallel composition operator (`&`) executes commands in parallel, the pipe operator (`|`) connects the output of one command to the input of another, and the logical operators (`&&`, `||`) execute commands based on the success or failure of previous commands.

Challenges Despite its benefits, the shell poses several challenges. First, shell performance does not scale—and parallelizing shell scripts requires a lot of manual effort. On the one hand, command developers implement individual commands, work in a single programming language, and expose parallelism through ad-hoc, command-specific flags. On the other, shell users combine multiple commands from many languages into their scripts, have limited options for incorporating parallelism, and rely on manual tools such as GNU `parallel` [80], `qsub` [21], and SLURM [31], or have to make use of shell primitives such as `&` and `wait` [24]. This approach is time-consuming, non-generalizable, and error-prone.

Second, the shell is too arbitrary, dynamic, and obscure. It is too arbitrary, since it contains a language that can be used to compose arbitrary commands, written in arbitrary languages and featuring arbitrary behaviors. It is too dynamic, because its execution depends on a variety of dynamic components—such

[1] For example, `sed s/A/B/g file.txt` is highly parallelizable, since it operates on each line independently and writes to standard output. However, passing the `-i` flag to `sed` causes it to edit `file.txt` in-place, which requires keeping internal state.

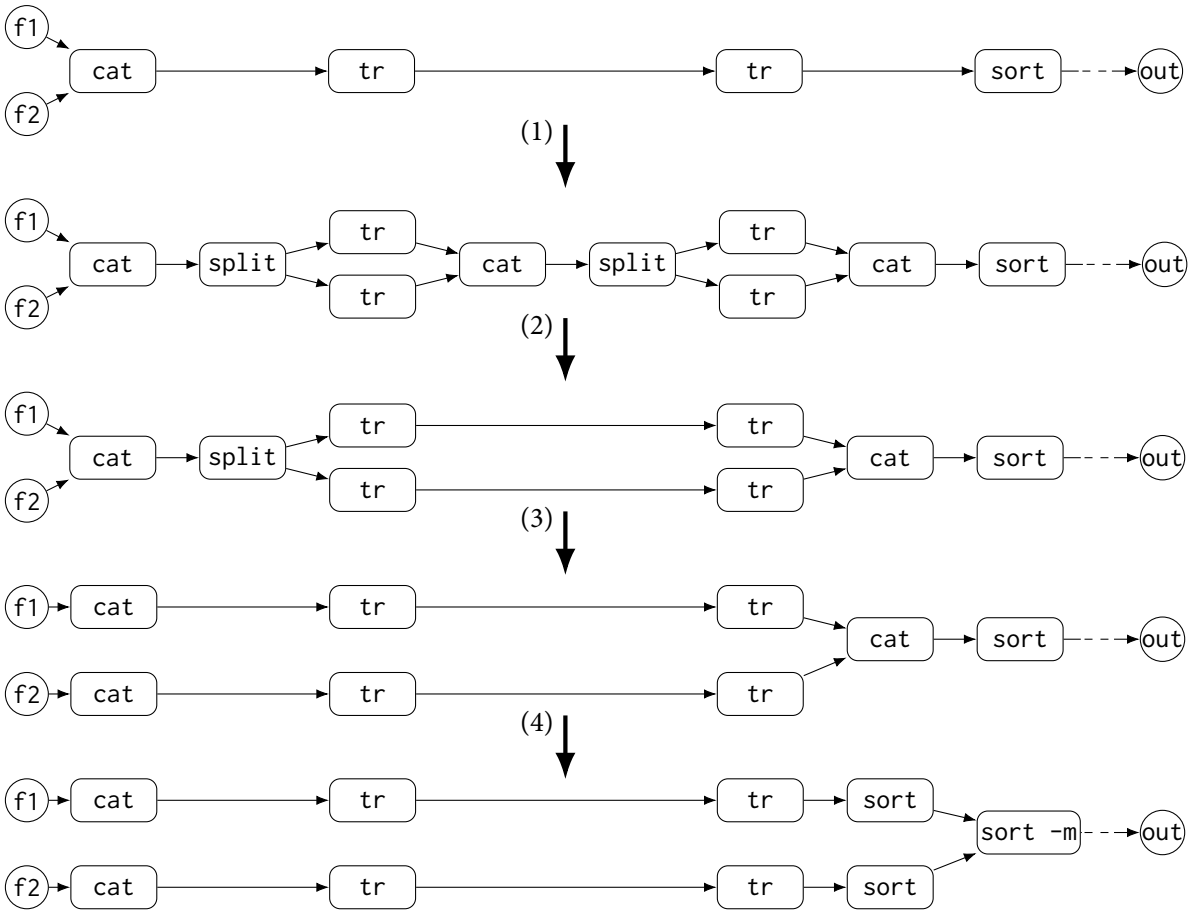


Figure 2.1: PaSh DFG transformation example. (1) Parallelize `tr` commands; (2) Eliminate unnecessary `cat-split` pair; (3) Parallelize first `cat`; (4) Use merging `sort` instead of `cat`.

as the state of the file system and the values of environment variables. It is too obscure, since the shell's semantics are complicated—and lots of different implementations exist. All of these factors prevent principled performance approaches to the shell.

Third, shell development is error-prone, counter-intuitive and heavily side-effectful. For example, the `sudo rm -rf $DIR/` command will delete the user's entire file system if the `$DIR` variable is empty—instead of issuing an "empty variable" error. The shell is inundated with these kinds of pitfalls, making shell development frustrating.

2.4 Automatic Scale-Out of Shell Scripts

In order to accelerate shell scripts without suffering from the shell's shortcomings, several systems have been developed for automatic shell scale-out [34, 49, 54, 83]. Out of these, PaSh and DiSh target automatic shell parallelization and distribution, respectively.

PaSh PaSh is a system that takes a Unix script as input, and executes it in a data-parallel fashion. It does so by encoding parallelizability information for different kinds of commands in a set of command annotations. In order to handle the dynamic nature of the shell, PaSh executes in a just-in-time (JIT) fashion, alternating between compilation and execution, in order to gather the most up-to-date information about the shell's state.

PaSh identifies potentially parallelizable regions of the script, and converts them to dataflow graphs (DFGs), where each command corresponds to a node, and each stream corresponds to an edge. PaSh applies DFG parallelization transformations in order to expose parallelism in the graph (Fig. 2.1)—with

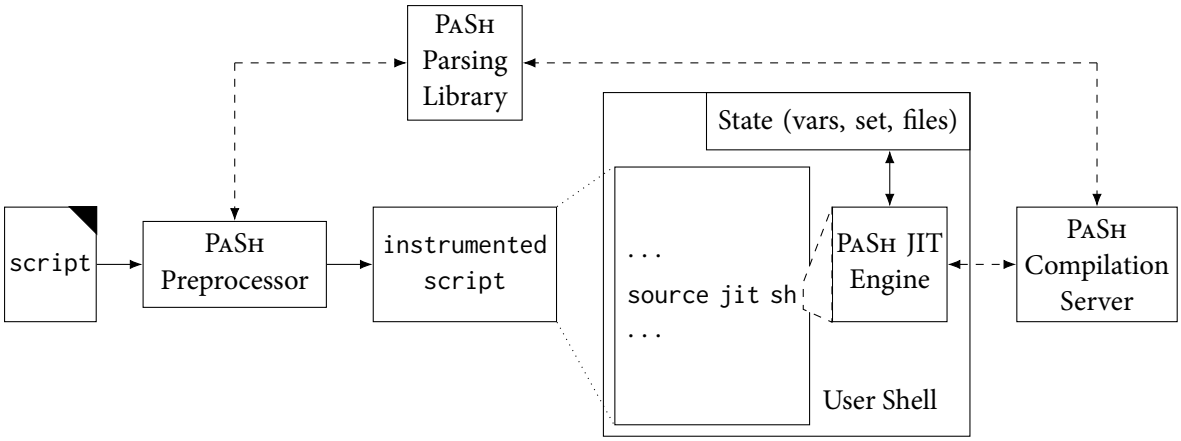


Figure 2.2: PaSh overview. PaSh instruments scripts with calls to the JIT engine, which passes program fragments to the compilation server at runtime.

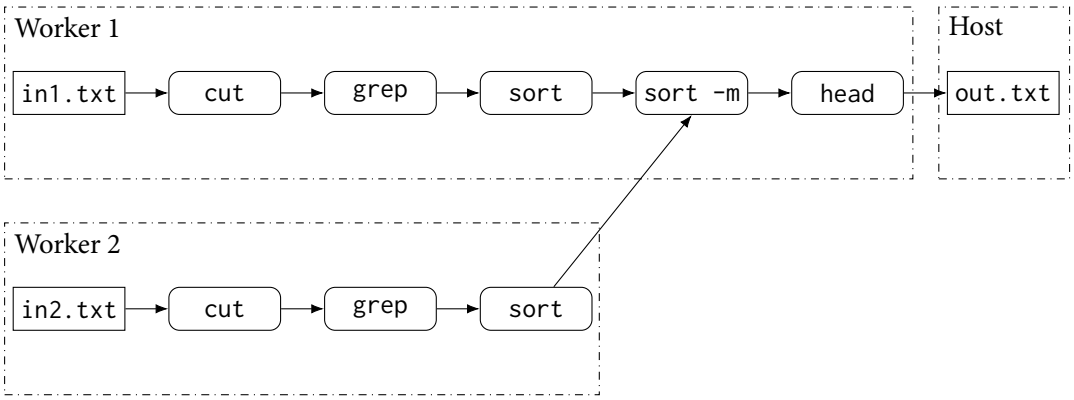


Figure 2.3: DiSh subgraphs. Each worker operates on a different part of the data (`in1.txt` and `in2.txt`), and the final result is aggregated and sent to the host machine.

the degree of parallelism being configurable by the `--width` flag. PaSh decomposes commands into data-parallel functions and aggregation functions. In order to do this, it introduces a set of parallelizability classes, expressible through an annotation language. PaSh includes flags and command arguments in its annotation framework—since these can alter the parallelizability of a command (Section 2.3). After applying the necessary transformations, PaSh converts each DFG back to a shell region. PaSh adds Unix shell primitives (e.g., `&`, `wait`) to each region—in order to drive parallelism explicitly—and uses appropriately named FIFOs—in order to manipulate the input and output streams of each data-parallel instance, essentially laying down the structure of the DFG. Figure 2.2 shows PaSh's overview.

DiSh DiSh is a system that automatically distributes shell scripts to multiple machines. Besides performance reasons, DiSh is useful for large datasets that may not fit on a single computer—or workloads that are naturally distributed across multiple computers. Like PaSh, DiSh uses command annotations, and operates in a just-in-time fashion. The DiSh compiler takes an input script fragment, and transforms it to a DFG—based on information it gathers from command annotations. In addition to PaSh, DiSh splits the DFG into subgraphs, to execute them on different workers (Fig. 2.3). After processing the DFG, DiSh passes it to the scheduler, which tries to map subgraphs to workers based on data locality—to reduce the amount of data transfer between workers. For worker communication, DiSh introduces socket-based Remote FIFO channels, to replace Unix FIFOs that cross worker boundaries—since traditional Unix FIFOs do not support network communication and cannot extend across workers. Figure 2.4 shows DiSh's overview.

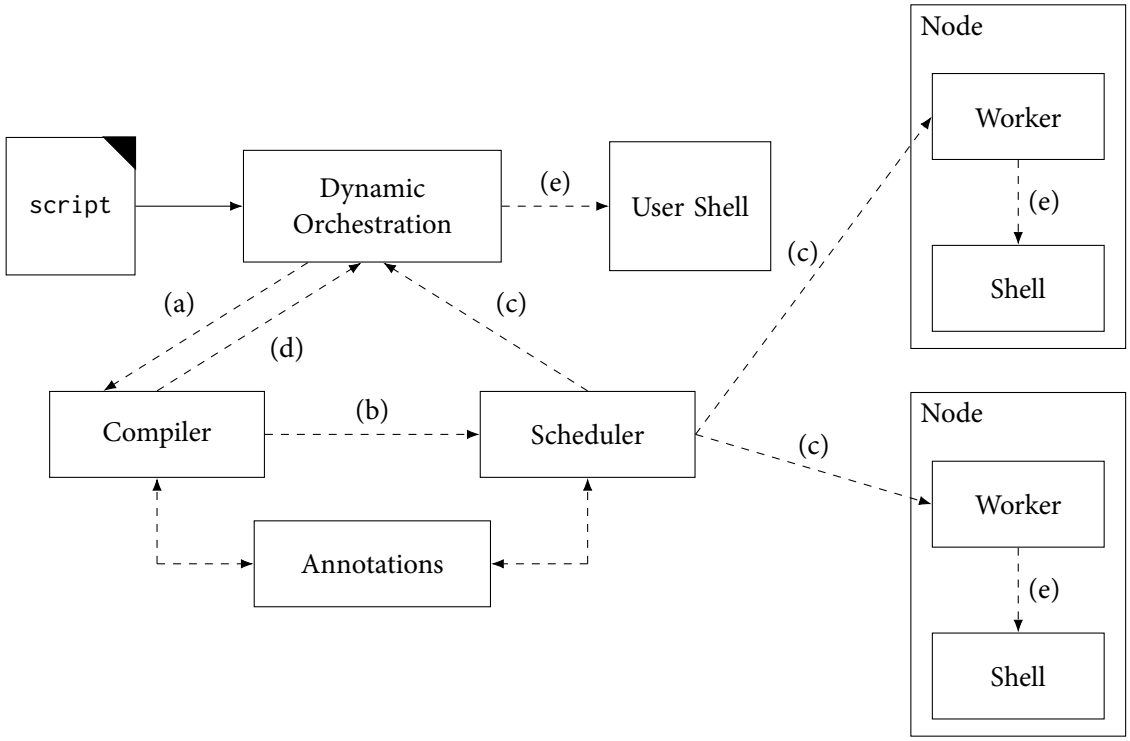


Figure 2.4: DiSh system overview. Steps: (a) compile script region; (b) schedule compiled dataflow; (c) send dataflow subgraphs to workers; (d) compilation failed, fall back to original region; (e) execute script region (compiled or original).

Chapter 3

Example and Overview

SPLASH takes a shell script as input, automatically instruments it for serverless execution, and deploys it on the cloud. First, it transforms the shell script appropriately to introduce parallelism. Then, it splits the script into fragments, in order to execute each fragment on a different function, and introduces custom commands for serverless tasks, such as function invocation and communication. Finally, it deploys the instrumented shell script on the cloud, in a just-in-time fashion.

3.1 Accelerating Regular Expression Matching

NFA-Regex (Listing 3.1) is a shell script that matches a complex regular expression over each line of the input.[1] Complex regular expressions like this are valuable in tasks such as DNA sequencing, but they can be computationally expensive [7, 35, 44]. Fortunately, this script has lots of potential for acceleration. First, it is highly parallelizable—as can be seen in Fig. 3.1—because both the `tr` and `grep` commands operate on individual lines, and the result of each line is independent of the other lines. Second, it is CPU-intensive, because matching the complex regular expression involves a lot of backtracking.

Unfortunately, existing systems for shell script acceleration cannot take full advantage of the script's potential for scale-out, since they have a static—and often limited—number of resources at their disposal. PASH is limited to the number of cores in a single machine, while DISH has access to a fixed number of workers. However, by using serverless, the user can quickly spawn hundreds of functions in a matter of seconds—tapping into a massive pool of memory and computing resources—and automatically tear them down once execution is complete.

3.2 Key Challenges

That being said, serverless development is significantly different from shell development, so taking advantage of serverless without losing the benefits of the shell poses a set of challenges:

- While the shell is highly expressive—facilitating communication, synchronization, scheduling, and filesystem interaction in an almost declarative manner—serverless imposes several restrictions in expressiveness. To translate a shell script to a scaled-out, serverless version, users must explicitly express function invocation and communication patterns, as well as interaction with

```
1 #!/bin/bash
2 cat "in.txt" | tr A-Z a-z | grep '\(.\).*\1\(.\).*\2\(.\).*\3\(.\).*\4'
```

Listing 3.1: NFA-Regex script. This script is highly parallelizable, because `tr` and `grep` operate on each line independently. It is also CPU-intensive, because of the complex regular expression.

[1] The regular expression matches lines where four characters appear twice, with each character possibly followed by one or more characters before reappearing (e.g., AqweABrtyBCuioCDpsfD)

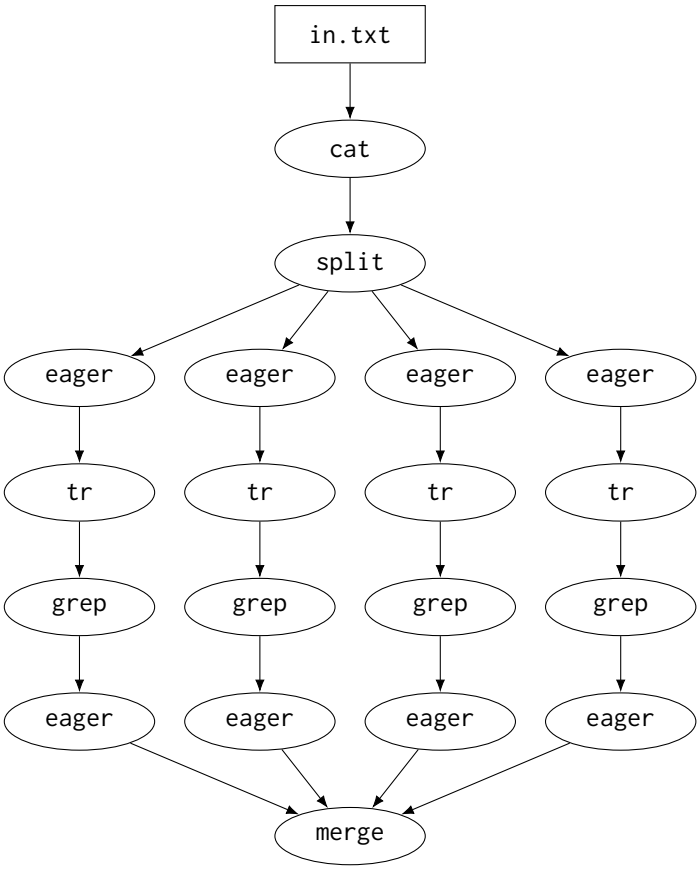


Figure 3.1: DFG produced by PaSh for NFA-Regex (`--width=4`). The `eager` commands are added by PaSh, in order to address the shell's laziness (Listing 2.1).

cloud storage and messaging queues. This manual approach is error-prone, time-consuming, and imposes an additional burden on developers.

- While the shell enjoys the benefits of pipeline parallelism due to streaming, functions are not allowed to directly establish streaming network connections with each other, because serverless functions have limited network connectivity properties (Section 2.2). Existing systems for shell scale-out [34, 49] also rely on streaming to apply optimizations that lead to higher worker utilization.[2]

- Shell scripts are inherently dynamic, while functions are statically configured in advance. This means that keeping up with the shell's dynamism requires continuous function reconfiguration at runtime—which introduces significant performance overhead.

- While the shell is Unix-native and comes preinstalled in all Unix distributions, on serverless it is not treated as favorably out of the box as other high-level languages [66]. Even running a single shell script on a function requires building a custom runtime [64]—or finding platform-specific workarounds. This deployment overhead discourages serverless shell development, since users have to take care of the very tasks they try to avoid by using serverless—such as managing the operating system or worrying about platform-specific minutiae.

[2] For example, PaSh allows sending batches to workers in a round-robin fashion, so that all workers in a parallel stage are provided data to process as soon as possible.

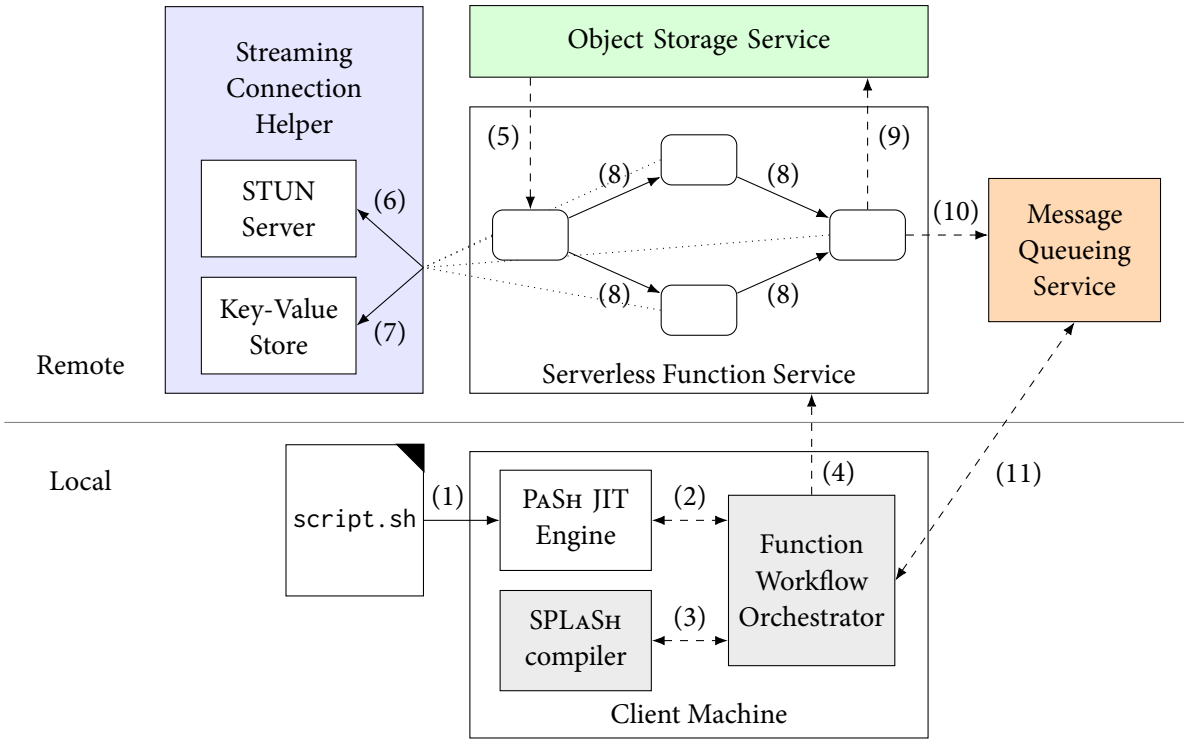


Figure 3.2: SPLaSh overview. Steps: (1) read input script; (2) create parallel DFG; (3) split DFG to subgraphs and add serverless primitives; (4) invoke first function; (5) download initial inputs; (6) get own address information; (7) write own address information / read downstream address information; (8) invoke downstream functions and send data; (9) upload final outputs; (10) send completion message; (11) receive completion message.

3.3 SPLaSh Overview

To address the aforementioned challenges, SPLaSh:

- Automatically instruments the input shell script with new serverless primitives for function invocation, function communication, cloud storage management, and message queue interaction. It also provides the necessary runtime support for these primitives.
- Bypasses the networking restrictions of serverless, by providing discovery and communication mechanisms that allow functions to establish streaming communication channels and exploit the performance benefits of streaming execution.
- Provides a serverless runtime for the shell that goes hand-in-hand with a just-in-time serverless compilation engine, in order to complement the shell's dynamic nature and provide out-of-the-box, push-button serverless shell scale-out.

Figure 3.2 shows SPLaSh's overview. First, the shell script gets fed into the PaSh JIT engine, which produces a DFG with parallelism introduced. The DFG then gets passed to the function workflow orchestrator, which calls SPLaSh's compiler. SPLaSh's compiler splits the DFG into subgraphs, adds the necessary commands and metadata for serverless execution, and transforms each subgraph back to a shell script. It then passes all the shell scripts back to the function workflow orchestrator, which invokes the first function in the workflow. After that, each function invokes its downstream functions.

In order for functions to discover each other and establish connections, they communicate with a STUN server that provides the necessary address information, which they write to a serverless key-value store for other functions to see. Functions query the key-value store to get the address information of

downstream functions, and establish streaming connections with them. Functions process their inputs in a streaming fashion, and send their outputs as they are produced. The first function downloads initial inputs from object storage before it begins processing, and the last function uploads final outputs to object storage once it is done. Because invocation is asynchronous, the final function informs the client machine about completed execution through a message queueing service. The client machine polls the corresponding queue until it receives that message, after which it passes back control to the PaSh JIT engine, and the whole process repeats for the next part of the shell script.

Result SPLaSh drops the execution of NFA-Regex from 10 min to 42 s (14.3× speedup), without requiring any modifications to the original script.

Chapter 4

Compilation

SPLASH provides the compilation support necessary for automatic serverless shell execution. It extends PASH's compiler by splitting each DFG to subgraphs and delegating them to different functions, but also enriching each DFG with serverless-specific primitives for function invocation, streaming communication, cloud storage management, and message queue interaction. Each section in this chapter gradually augments the script produced by PASH for NFA-Regex (Listing 4.1) with additional serverless features.[1] The following examples use `--width=2` for simplicity.

```
1  #!/bin/bash
2  mkfifo f{0..10}
3  cat in.txt >f0 &
4  split f0 f1 f2 &
5  eager <f1 >f3 &
6  eager <f2 >f4 &
7  tr A-Z a-z <f3 >f5 &
8  tr A-Z a-z <f4 >f6 &
9  grep '\(.\).*\1\(.\).*\2\(.\).*\3\(.\).*\4' <f5 >f7 &
10 grep '\(.\).*\1\(.\).*\2\(.\).*\3\(.\).*\4' <f6 >f8 &
11 eager <f7 >f9 &
12 eager <f8 >f10 &
13 merge f9 f10 >out.txt &
14 wait
```

Listing 4.1: Parallel script produced by PASH for NFA-Regex (`--width=2`). Note the pairs of parallel commands at lines 5-6, 7-8, 9-10, and 11-12.

```
1  #!/bin/bash
2  mkfifo f{0..2}
3  cat in.txt >f0 &
4  split f0 f1 f2 &
5  wait
```

Listing 4.2: Shell script for Subgraph 1.

[1] The PASH script has been modified for clarity and simplicity. Listing A.1 shows the full, unmodified version of the resulting script.

```
#!/bin/bash
mkfifo f{0..4}
eager <f0 >f1 &
tr A-Z a-z <f1 >f2 &
grep '\(.\).*\1\(.\).*\2\(.\).*\3\(.\).*\4' <f2 >f3 &
eager <f3 >f4 &
wait
```

Listing 4.3: Shell script for Subgraph 2.

```
#!/bin/bash
mkfifo f{0..4}
eager <f0 >f1 &
tr A-Z a-z <f1 >f2 &
grep '\(.\).*\1\(.\).*\2\(.\).*\3\(.\).*\4' <f2 >f3 &
eager <f3 >f4 &
wait
```

Listing 4.4: Shell script for Subgraph 3.

```
#!/bin/bash
mkfifo f{0..1}
merge f0 f1 >out.txt &
wait
```

Listing 4.5: Shell script for Subgraph 4.

4.1 Assigning Work to Functions

In PaSh, after a parallel shell fragment is created, the Unix kernel facilitates scheduling behind the scenes—based on constructs such as `&` and `|`. However, these constructs do not work for serverless scale-out, since they target single-machine settings. This creates the need to explicitly decide how many functions to use, and which commands to run in each function. On one end of the spectrum, a single function could execute the entire fragment—which is clearly not scalable. On the other, multiple functions could execute only one command each, which is not efficient—since it would introduce excessive network traffic, initialization overheads, and invocation costs.

SPLaSh operates somewhere in between, deciding to split the graph wherever parallel stages are introduced—through the `split` and `merge` commands—while sequences of commands not interrupted by a `split` or a `merge` are combined in a single subgraph (Fig. 4.1). After SPLaSh splits the graph into subgraphs, it can begin adding serverless-specific primitives for function invocation and communication between function boundaries. Listings 4.2 to 4.5 show the shell scripts produced for each subgraph.[2] SPLaSh adds `mkfifo` and `wait` commands to each subgraph, since now each subgraph runs in a different environment.[3]

[2] Careful readers may notice that some subgraphs have "dangling" FIFOs (i.e., FIFOs whose output goes nowhere, such as `f1`, `f2` in Subgraph 1, or that receive their input from nowhere, such as `f0` in Subgraph 2). Dangling FIFOs signify the need for function communication, which will be discussed in Section 4.3.

[3] For clarity and consistency, FIFOs have been renumbered to start from zero in each subgraph. While SPLaSh does not actually renumber FIFOs, renumbering would not cause name collisions, since each subgraph runs on a different function.

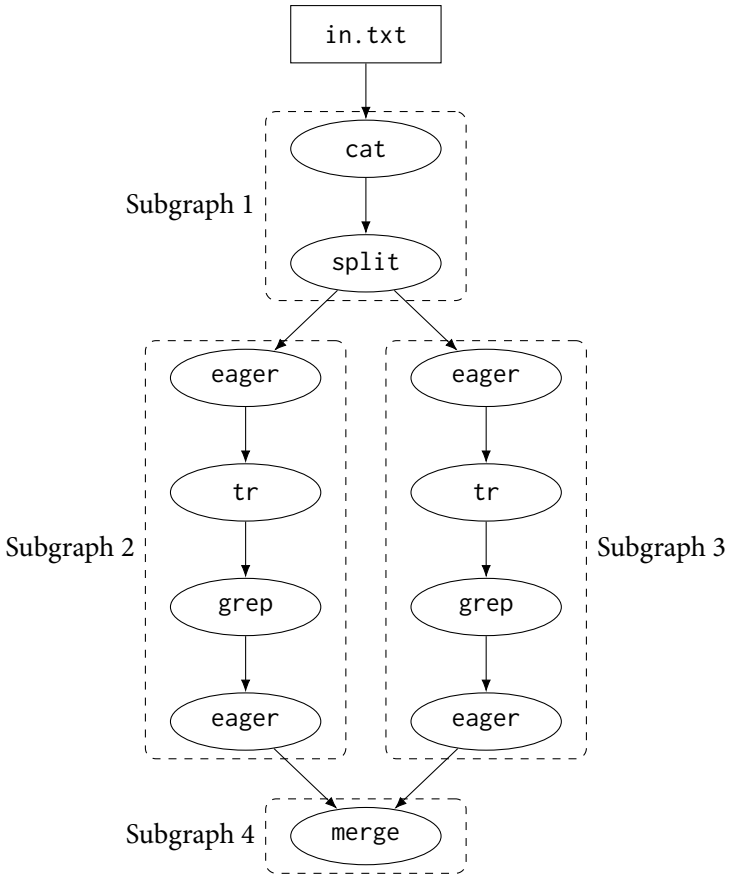


Figure 4.1: DFG after subgraph splitting.

```
1 #!/bin/bash
2 mkfifo f{0..2}
3 invoke-function $SCRIPT_ID_2 $SCRIPT_MAP &
4 invoke-function $SCRIPT_ID_3 $SCRIPT_MAP &
5 cat in.txt >f0 &
6 split f0 f1 f2 &
7 wait
```

Listing 4.6: Script for Subgraph 1 after invocation commands are added. The background operator (`&`) is used to invoke functions as soon as possible.

4.2 Invocation Commands

In SPLASH, only the first function in a pipeline gets invoked from the client machine. After that, each function invokes its next function(s)—which in turn invoke their next functions. Thus, SPLASH introduces a new command (`invoke-function`), which is added in the shell script of functions that invoke other functions. Figure 4.2 shows the invocations that take place in the NFA-Regex script. Note that only one of the functions involved in a parallel stage must invoke the function containing `merge`.

The shell fragment to be executed by each function is provided at runtime—in the function's invocation payload. This means that each caller must have the shell fragment of their callee. However, they must also have the shell fragment of the *callee's* callee—otherwise the previous property would break for the next stages—and so on. For this reason, SPLASH includes *all* shell fragments in the form of a map that matches identifiers to scripts (`SCRIPT_MAP`). Then, each invocation includes this map in the pay-

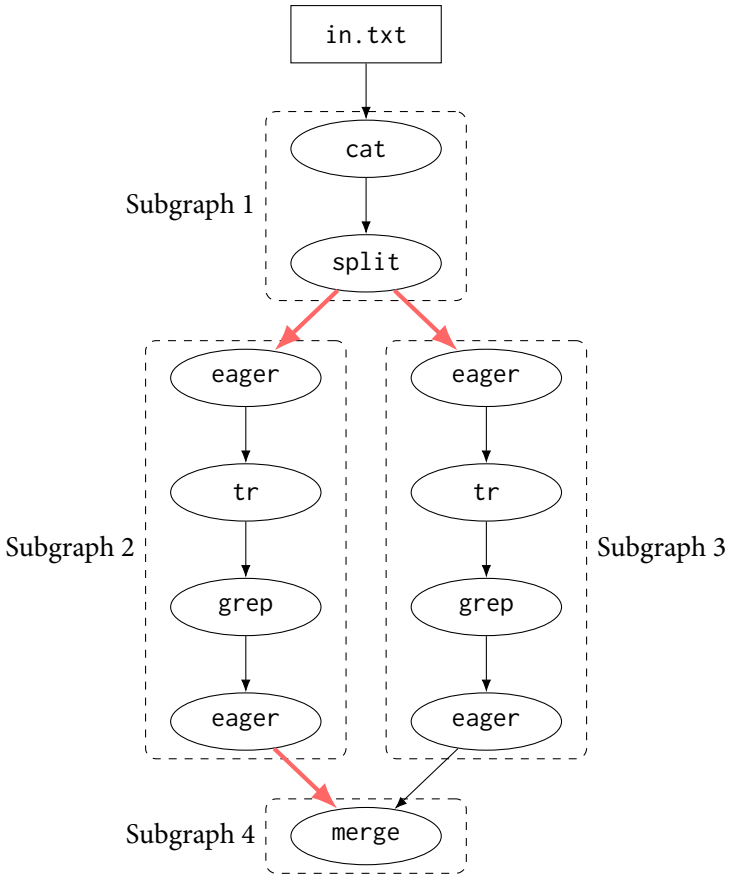


Figure 4.2: DFG enriched with invocation between subgraphs. Note that Subgraph 4 needs to be invoked by only one of the previous subgraphs.

load, along with the identifier that corresponds to the callee's script (`SCRIPT_ID`). Function invocation happens eagerly; each function invokes the next function(s) as soon as possible, so callees can quickly start receiving data from their callers, achieving high utilization. Listing 4.6 shows the script of the first subgraph, with the invocation commands added.

4.3 Communication Commands

Streaming communication requires custom support for serverless, since Unix FIFOs operate in memory and cannot cross function boundaries. The compiler detects where this communication must happen, creates streaming commands with the appropriate metadata—so functions can find each other—and correctly interposes these commands into the original dataflow. More specifically, PLaSh introduces two new commands (`send-stream` and `receive-stream`), which it places at function boundaries (Fig. 4.3). On the sender function, the `send-stream` command consumes the last command's output and sends the data over the network. On the receiver function, the `receive-stream` command receives the data over the network and feeds it to the first command.

In order for each `send-stream` to match its corresponding `receive-stream` and vice versa, PLaSh uses a unique key for each pair of `send-stream` and `receive-stream` commands (`COMM_KEY`). Both these commands use this key to find certain address information at runtime—as will be described in Section 5.3. Listings 4.7 and 4.8 show the first two subgraphs with communication commands added—note the matching `COMM_KEY_1_2` in the `send-stream` and `receive-stream` commands.

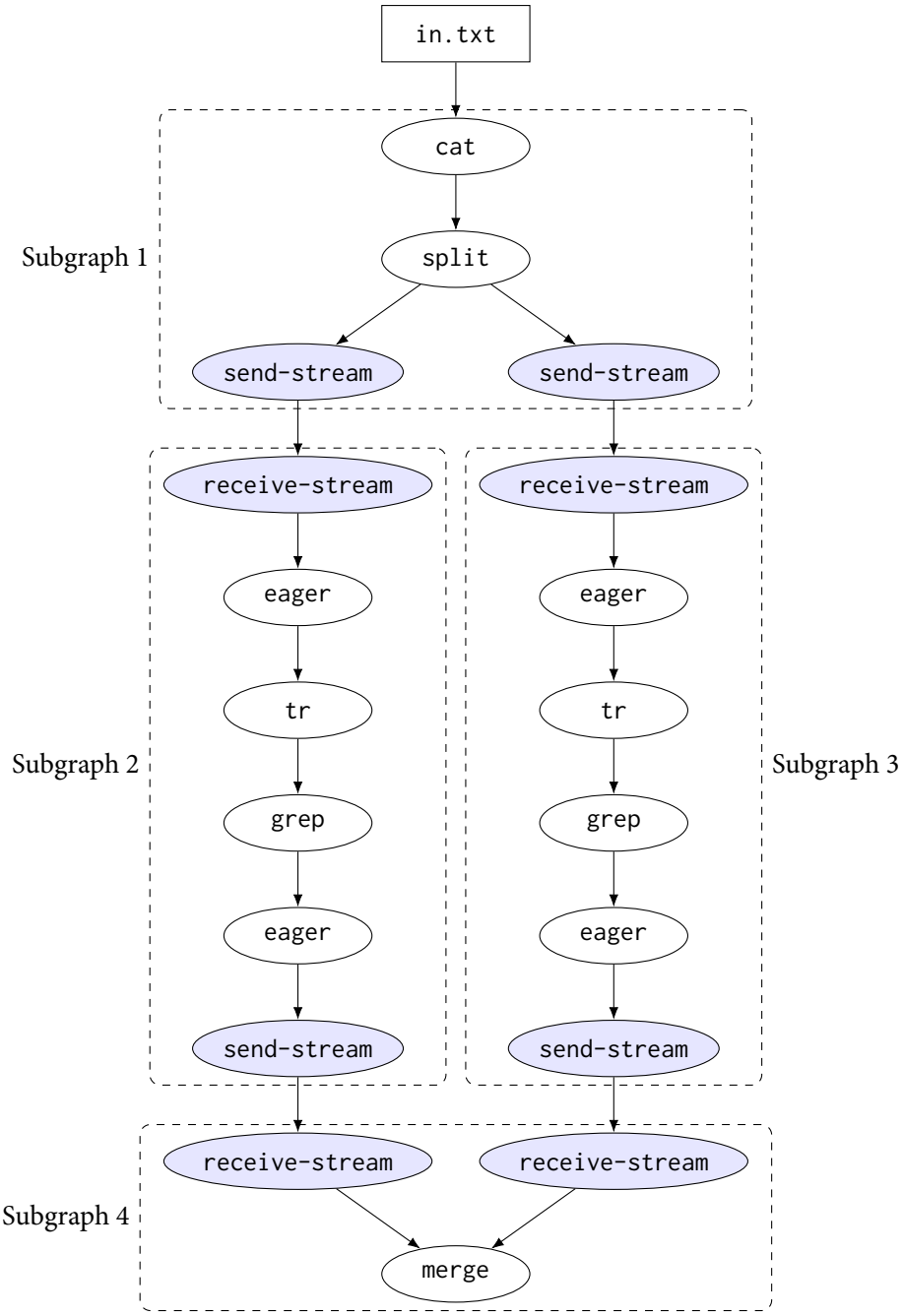


Figure 4.3: DFG enriched with communication nodes.

4.4 Object Storage Commands

SPLASH uses object storage for reading initial input data, as well as writing final output data. For this reason, SPLASH introduces two new commands (`download-object` and `upload-object`), which are placed before commands that read from files, or after commands that write to files. Both commands take the name of the target file as a parameter. Figure 4.4 shows how these nodes look in the DFG, while Listings 4.9 and 4.10 show the scripts of the first and last subgraphs with object storage commands added. Note how the `cat` and `merge` commands now interact exclusively with named FIFOs—since the input and output files are now specified in the object storage commands.

```
#!/bin/bash
mkfifo f{0..2}
invoke-function $SCRIPT_ID_2 $SCRIPT_MAP &
invoke-function $SCRIPT_ID_3 $SCRIPT_MAP &
cat in.txt >f0 &
split f0 f1 f2 &
send-stream $COMM_KEY_1_2 <f1 &
send-stream $COMM_KEY_1_3 <f2 &
wait
```

Listing 4.7: Script for Subgraph 1 after communication commands are added. Note that `COMM_KEY_1_2` is also used in the `receive-stream` command of Listing 4.8.

```
#!/bin/bash
mkfifo f{0..4}
invoke-function $SCRIPT_ID_4 $SCRIPT_MAP &
receive-stream $COMM_KEY_1_2 >f0 &
eager <f0 >f1 &
tr A-Z a-z <f1 >f2 &
grep '\(.\).*\1\(.\).*\2\(.\).*\3\(.\).*\4' <f2 >f3 &
eager <f3 >f4 &
send-stream $COMM_KEY_2_4 <f4 &
wait
```

Listing 4.8: Script for Subgraph 2 after communication commands are added. Note that `COMM_KEY_1_2` is also used in the `send-stream` command of Listing 4.7.

```
#!/bin/bash
mkfifo f{0..3}
invoke-function $SCRIPT_ID_2 $SCRIPT_MAP &
invoke-function $SCRIPT_ID_3 $SCRIPT_MAP &
download-object in.txt >f0 &
cat f0 >f1 &
split f1 f2 f3 &
send-stream $COMM_KEY_1_2 <f2 &
send-stream $COMM_KEY_1_3 <f3 &
wait
```

Listing 4.9: Script for Subgraph 1 after object storage commands are added. Note that the `cat` command now only uses Unix FIFOs.

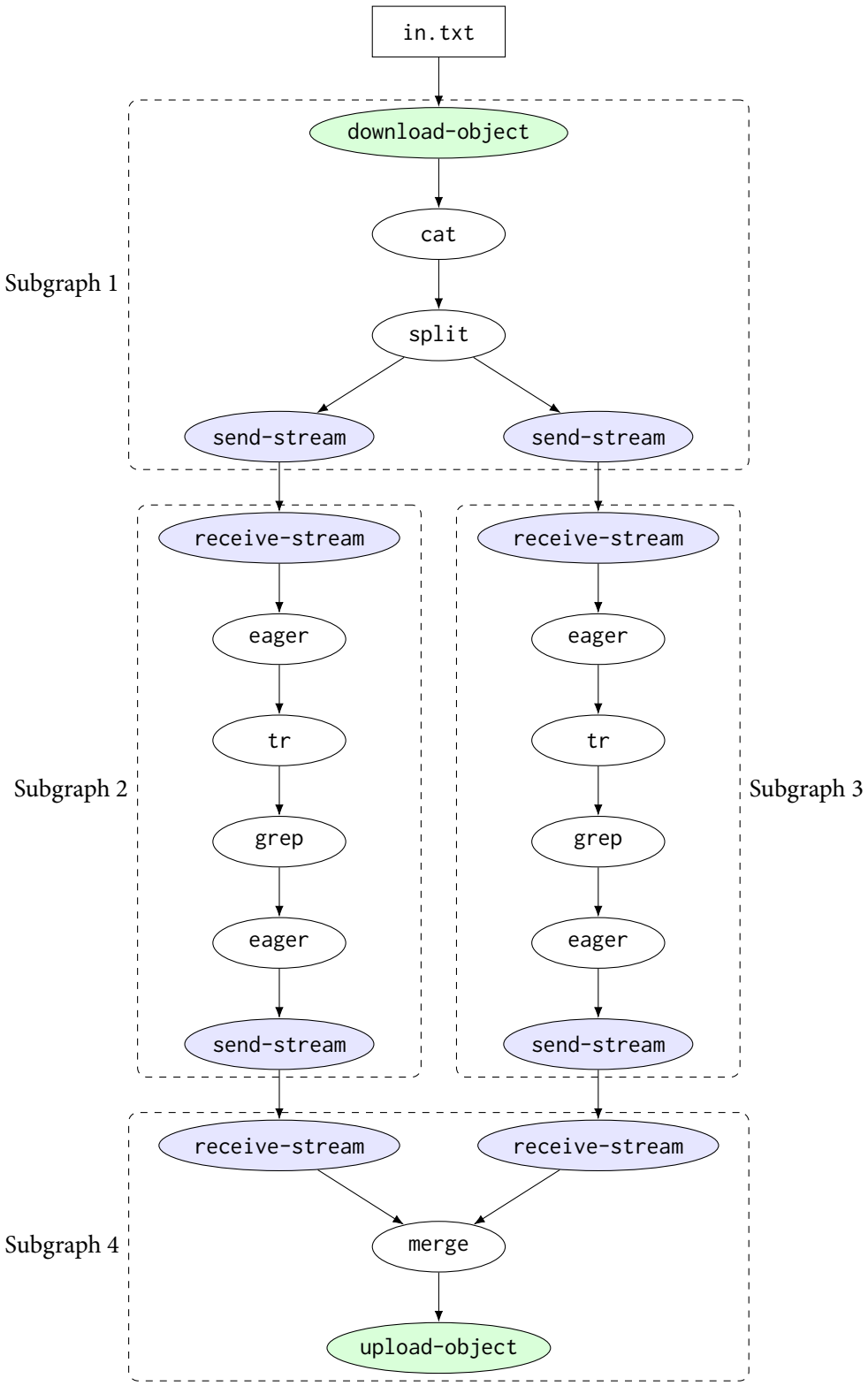


Figure 4.4: DFG enriched with object storage nodes.

```
#!/bin/bash
mkfifo f{0..2}
receive-stream $COMM_KEY_2_4 >f0 &
receive-stream $COMM_KEY_3_4 >f1 &
merge f0 f1 >f2 &
upload-object <f2 out.txt &
wait
```

Listing 4.10: Script for Subgraph 4 after object storage commands are added. Note that the `merge` command now only uses UNIX FIFOs.

```
#!/bin/bash
mkfifo f{0..2}
receive-stream $COMM_KEY_2_4 >f0 &
receive-stream $COMM_KEY_3_4 >f1 &
merge f0 f1 >f2 &
upload-object <f2 out.txt &
wait
send-notification
```

Listing 4.11: Script for Subgraph 4 after message queueing commands are added. Note that the `send-notification` is added after the `wait` command, to make sure the rest of the script has finished before signaling completion.

4.5 Message Queueing Commands

SPLASH uses asynchronous function invocation[4]—each function does not wait for the invoked function to finish execution before returning—so the client machine does not get automatically notified when the final function has finished—so it cannot safely move on to the next part of the script. For this reason, SPLASH introduces another command (`send-notification`), which gets added to the end of the last function, causing it to send a notification to a message queue that the client machine is listening on (Listing 4.11). Note that the `send-notification` has to be added after the `wait` command, to make sure the function has finished all previous work. Listings B.1 to B.4 show the final scripts for each subgraph, after all serverless commands have been added—invocation, communication, storage, and messaging.

[4] This choice will be justified in Section 5.2.

Chapter 5

Runtime

SPLASH also provides runtime support for the new serverless primitives it introduces. It configures functions appropriately for just-in-time deployment, invokes functions asynchronously, establishes streaming communication channels, interacts with cloud storage, and sends completion notifications to the client machine through message queues.

5.1 Just-in-Time Deployment

As has been mentioned before, PASH instruments just-in-time analysis to extract more information from the shell script, and increase the potential for parallelization. This approach works when deployed in a single machine or a cluster—all the resources involved in the computation exist beforehand. However, in the serverless case, the user must configure a function before invoking it. This configuration phase introduces a performance overhead, since it requires initiating a request to the cloud provider, uploading the function's dependencies, and waiting for a response that the function is ready. This overhead can become prohibitive if introduced before every function invocation, depending on the function's size and the client machine's upload speed.

For this reason, SPLASH is designed to require only one function configuration that is general enough for executing different shell scripts. This means that the user can specify only one function configuration—at any point, well before the script's execution—and only update it when dependencies change—similarly to how they would rebuild a software artifact after changing its code. Since the only thing that can change at runtime is the shell fragment that the function will execute, SPLASH passes that fragment as payload (Section 4.2). Apart from that, having only one function configuration offers the additional benefit of *warm starts* [67]; multiple invocations of the same function cause the environment to be cached—so subsequent invocations start faster.

5.2 Invocation

Each function invocation payload includes a script fragment map (`SCRIPT_MAP`) and a script fragment identifier (`SCRIPT_ID`). The function handler uses this information to retrieve the appropriate shell script fragment and save it to a file—so it can then execute it. The function handler also saves `SCRIPT_MAP` to a file, so the shell script can later include it in the invocation payload of the next functions (Listing C.2).

SPLASH uses asynchronous function invocation—functions do not wait for their callees to finish execution (Listing C.1). This is done so functions can return immediately after finishing their own execution and not have to wait for all other functions down the pipeline—thus reducing monetary cost and freeing up concurrency resources [68]. Also, SPLASH configures functions not to retry on timeout, since cascading re-execution would heavily increase monetary cost and reduce concurrency resources. While cloud providers often place more restrictive limits on payload size for asynchronous invocation, these limits are typically in the range of megabytes—with the most restrictive being 256 kB for AWS Lambda [23, 45, 72]. These limits surpass the size of shell code produced by SPLASH for each pipeline, and thus are not a practical concern.[1]

[1] To put things into perspective, *all* the LaTeX code needed for this thesis—text, tables, and figures—is less than 256 kB.

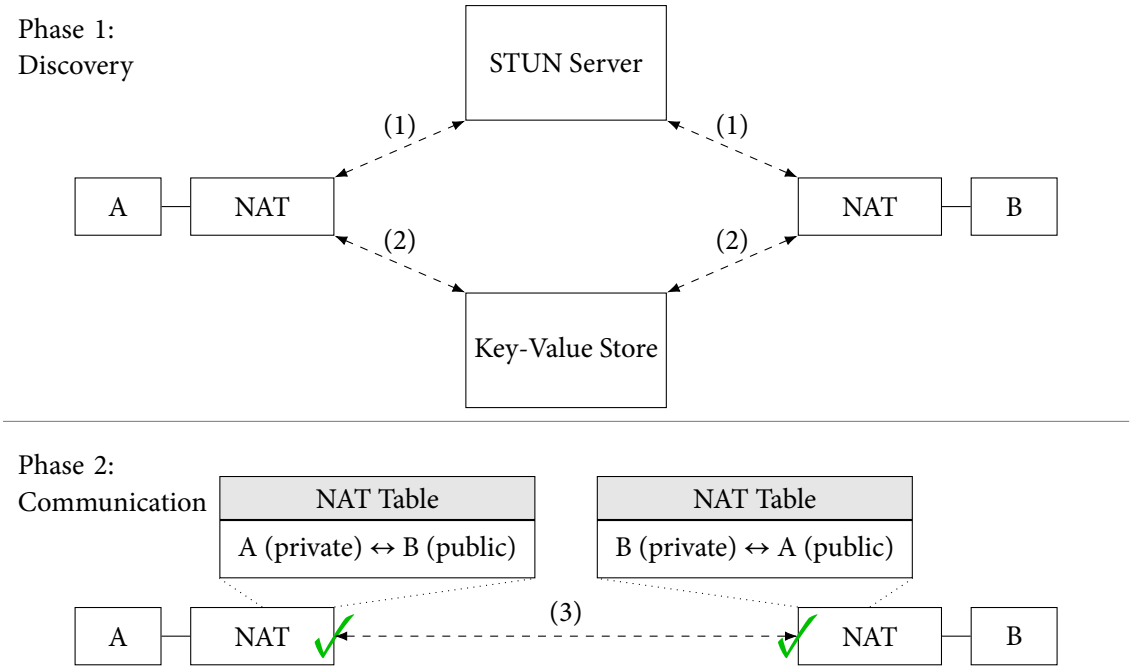


Figure 5.1: SPLASH hole punching. Each function (1) gets its *own* address information; (2) writes its own address information to a key-value store, from which it also reads the other function's address information; (3) initiates a connection to the other function. NAT tables are created, and connections go through.

5.3 Communication

As has been mentioned before, functions cannot achieve direct, streaming communication with each other, and alternative methods of communication are not well-suited for shell workloads—they are either too costly, do not autoscale, or do not support streaming. To overcome this challenge, SPLASH surpasses the networking limitations of serverless and establishes direct, streaming communication channels between functions—all while not using extra services for data exchange. In order to ensure reliable, ordered communication similar to UNIX FIFOs, SPLASH uses TCP as its communication protocol.

To establish direct streaming connections despite the presence of NATs, SPLASH employs a technique called *hole punching* [16]: Suppose machines A and B want to establish a connection with each other. In regular hole punching, A and B connect to a publicly reachable relay server, which then reveals B's address information to A, and A's address information to B. Then, both A and B initiate a connection to each other with this new information. This creates a new entry about B in A's NAT table, and an entry about A in B's NAT table—the "holes"—thus allowing incoming connections.

In reality, SPLASH uses a slightly modified version of the technique (Fig. 5.1). Instead of using a relay server to get the other function's information, each function queries a STUN server [28], which returns to the function its *own* public address information. Then, each function saves its address information to a common key-value store, which both functions can query based on a common `COMM_KEY`. After that, the process is similar to regular hole punching; functions initiate connections to each other, NAT table entries get created, and connections can come through. It should be noted that both the STUN server and the key-value store are used only for address information—*not* heavyweight data exchange—so that they do not affect scalability.

5.4 Object Storage

Even though SPLASH uses streaming connections for intermediate communication data, it still needs to read from pre-existing data, and persist results to files (Listings C.5 and C.6). SPLASH benefits the most

from object stores that allow streaming downloads, since processing can propagate quickly through the pipeline—thus achieving high utilization, even for large files.

5.5 Message Queueing

Because invocations are asynchronous, the client machine does not get notified when the last function has finished. Thus, SPLASH causes the last function to send a notification to a message queue that the client is listening on. After the client receives the notification, it moves on to the next region of the shell script (Listings C.3 and C.4). In SPLASH, the client uses *long polling* while waiting for each notification, to issue fewer requests and reduce cost [62] because notification volume is relatively low—one message per finished pipeline.

Chapter 6

Evaluation

SPLaSh is evaluated on how fast it performs compared to the sequential shell execution, and how well it scales with the number of functions.

6.1 Setup

Implementation SPLaSh's functionality is built on top of PaSh. SPLaSh uses the Serverless Framework [29] for deployment on AWS, and the boto3 Python AWS SDK [69] for interacting with AWS services.[1] The hole punching library is implemented in Rust, and is deployed as an AWS Lambda layer [65]. SPLaSh uses AWS Lambda [63] (serverless functions), Amazon DynamoDB [59] (key-value store), Amazon S3 [61] (object storage), and Amazon SQS [60] (message queue).

Baseline SPLaSh's performance is compared against GNU Bash [20], the de facto sequential shell script execution environment. Both Bash and SPLaSh execute every shell script completely unmodified. Since there is currently no out-of-the-box support for running shell scripts on AWS Lambda [66], all experiments are executed on SPLaSh's runtime. For Bash, one function executes the original script in its entirety, while for SPLaSh, the entire compilation and execution process takes place, as described in Chapters 3 to 5.

Hardware Setup All executions are set to (initially) read from and (finally) write to Amazon S3. Functions are configured with 10 240 MB of memory and 6 vCPUs for Bash, and 1769 MB of memory and 1 vCPU for SPLaSh. These memory sizes are chosen for two reasons; first, 10 240 MB is the highest memory size available for AWS Lambda [73]; and second, 10 240 MB and 1769 MB are the only two memory sizes with a documented memory-to-vCPU correspondence [71].

Table 6.1: Benchmarks overview.

	Benchmark Set	Short Label	Scripts	LOC	Input	Source
1	Common Unix One-Liners	`Oneliners`	6	67	1 GB	[4, 5, 55, 81]
2	Bell Labs Unix50	`Unix50`	8	24	9 GB	[6, 39]
3	COVID-19 Mass Transit Analysis	`COVID-mts`	4	38	2.2 GB	[82]
4	Natural Language Processing	`NLP`	13	160	130 books	[9]
5	NOAA Weather Analysis	`NOAA`	1	16	500 MB	[86]

6.2 Benchmarks

Table 6.1 shows the benchmark sets used to evaluate SPLaSh.[2] In total, 5 real-world benchmark sets are used, comprising 32 shell scripts and 305 lines of code (LOC). `Oneliners` and `Unix50` contain classic and recent (circa 2019) scripts that make heavy use of Unix and Linux built-in commands. `COVID-mts` contains four scripts used to analyze real telemetry data from mass transit schedules during the COVID-19 response in Athens.

Common Unix One-Liners The `Oneliners` benchmark set contains a range of classic pipelines, each highlighting different performance characteristics of the shell. NFA-Regex is centered around an expensive NFA-based backtracking expression. The Sort script sorts a large input. Word-Frequency and Top-N-Terms are based on McIlroy's classic word counting program [5]; they use sorting, rather than tabulation, to identify high-frequency terms in a corpus. Spell, based on the original `spell` developed by Johnson [4], is another Unix classic: after some preprocessing, it makes clever use of `comm` to report words that do not appear in a dictionary. Finally, Sort-Sort uses consecutive `sort` and `sort -r` commands, without interleaving them with commands that condense their input size (e.g., `uniq`).

Bell Labs Unix50 In a recent celebration of Unix's 50-year legacy, Bell Labs created a set of challenges solvable by Unix pipelines. The problems were designed to highlight Unix's modular philosophy [55]. For the evaluation, unofficial solutions found on GitHub [77] are used, expressed as single pipelines with multiple stages. They make extensive use of standard commands, use a variety of flags, and appear to be written by non-experts, since they often use sub-optimal or non-Unix-y constructs.

COVID-19 Mass Transit Analysis The `COVID-mts` set contains four pipelines that were used to analyze real telemetry data from bus schedules during the COVID-19 response in Athens [82]. The pipelines compute several statistics on the transit system per day, such as average serving hours per day and average number of vehicles per day. These pipelines use typical Unix staples such as `sed`, `awk`, `sort`, and `uniq`.

Natural Language Processing The `NLP` set contains several scripts from Unix-for-poets, a tutorial for developing programs for natural-language processing out of Unix and Linux utilities. It includes tasks such as word counting, finding four-letter words, and counting sequences of consonants.

NOAA Weather Analysis This program is inspired by the central example in "Hadoop: The Definitive Guide" [86], and performs temperature analysis on data from the National Oceanic and Atmospheric Administration (NOAA). It consists of two pipelines, each calculating the maximum and minimum temperature, respectively.

6.3 Performance

SPLaSh's speedup with respect to Bash is evaluated on all the benchmark sets, by trying different `--width` values for each script, and reporting on the best speedup achieved. The goal is to evaluate the maximum performance SPLaSh can achieve, and compare it with Bash. The `--width` that is finally used for each script is specified in Tables E.1 to E.5.

Oneliners Figure 6.1 shows that NFA-Regex has the largest speedup (14.38×), while the rest of the scripts have more modest speedups, ranging from 1.33× to 2.05×. All scripts except for NFA-Regex have the `sort` command somewhere in the pipeline, which (i) introduces increased data exchange between

[1] Note that the techniques used by SPLaSh are not unique to AWS, and can be used with other cloud providers as well. Table D.1 shows the AWS services used by SPLaSh and their equivalents in Google Cloud and Microsoft Azure.

[2] Tables E.1 to E.5 show the full specification of shell scripts used in the evaluation.

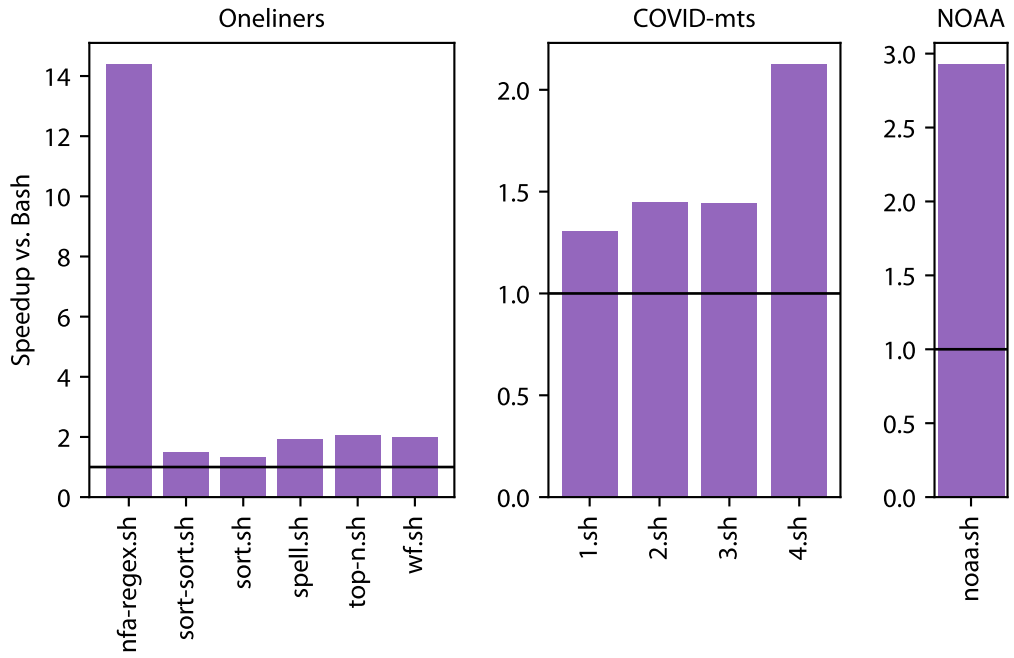


Figure 6.1: SPLASH's speedup for `Oneliners`, `COVID-mts`, and `NOAA` (higher is better). The `--width` used for each script is specified in Tables E.1, E.3 and E.5.

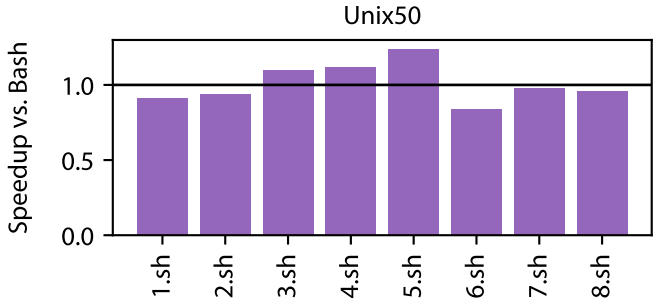


Figure 6.2: SPLASH's speedup for `Unix50` (higher is better). The `--width` used for each script is specified in Table E.2.

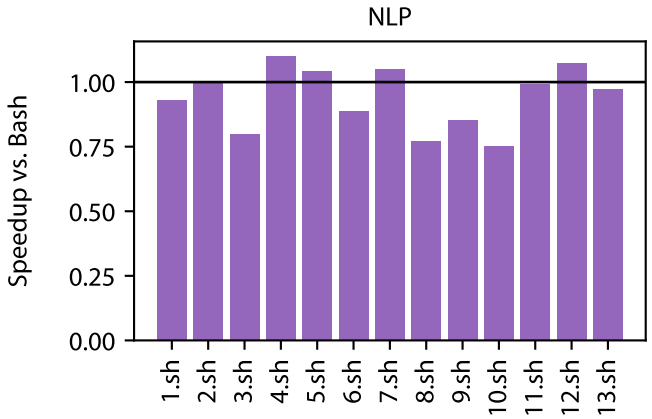


Figure 6.3: SPLASH's speedup for `NLP` (higher is better). The `--width` used for each script is specified in Table E.4.

functions[3], and (ii) has a high memory footprint. Thus, SPLASH is not able to take full advantage of the extra CPU resources, and the speedup is limited, especially since the Bash function has 10 240 MB of memory.[4] Especially in the case of Sort and Sort-Sort, the size of the input does not get reduced through the pipeline—which would highlight the benefits of increased parallelism—and this is why the speedups of these two scripts are the lowest.

Unix50 Figure 6.2 shows that SPLASH offers marginal speedups in the case of `Unix50`—it actually slows down 5 out of 8 scripts. This is because these scripts almost exclusively use simple instances of the `cut`, `tr`, and `grep` commands, which are not computationally intensive enough to justify the overhead of splitting the workload to functions and transferring data between them.

COVID-mts Figure 6.1 shows that SPLASH offers similar speedups for `COVID-mts` as it does for `Oneliners`, which is somewhat expected, since the scripts from the two sets are similar in structure. Again, the existence of `sort` restricts the amount of speedup that SPLASH can offer.

NLP Figure 6.3 paints a similar picture to `Unix50`, with SPLASH marginally speeding up some scripts, while slowing down others. However, the reason here is different; each script in `NLP` loops over a range of books, and while SPLASH executes each iteration of the loop in a data-parallel manner, the loop itself progresses in a *sequential* manner—each iteration of the loop must finish before the next iteration can continue. Since each book by itself is small, it is not worth it to pay a once-per-book overhead of instrumenting an entire serverless workflow and exchanging data. A solution is to run multiple iterations of the loop in parallel, since they are independent—thus overlapping the overhead of these serverless workflows. Adding this support for SPLASH is work in progress.

NOAA Figure 6.1 shows that SPLASH speeds up the temperature analysis script by almost 3×. This script also has a `sort` command, but is slightly more computationally intensive than the others—the proportion of CPU-intensive commands is slightly higher.

6.4 Scalability

To provide a more detailed insight into SPLASH's performance characteristics, SPLASH's scalability is evaluated on the `Oneliners` set, by varying the `--width` parameter[5] from 2 to 64—depending on the script—and measuring SPLASH's speedup with respect to Bash (Fig. 6.4).

NFA-Regex scales almost linearly with the number of functions—which is expected, since it is highly parallelizable, as mentioned before—hitting a plateau at 64 functions. This plateau, however, is caused by the time needed to compile the script, invoke functions, and make connections between them. This is a once-off overhead, and can be amortized as the input size gets larger. Sort and Sort-Sort quickly show diminishing returns after `--width=4`, while Top-N-Terms and Word-Frequency stop scaling after `--width=8`. Spell scales somewhat more favorably up to `--width=16`.

Conclusion While SPLASH is able to provide the scalability resources needed—as can be seen in the case of NFA-Regex—it is important to consider the nature of the shell script to be executed. If a shell script is innately non-scalable, there is not much SPLASH can do to speed it up. Thus, SPLASH in its current state is targeted toward long-running, CPU-intensive, highly parallelizable shell scripts, and is not a one-size-fits-all solution for shell script acceleration.

[3] SPLASH, like PASH, aggregates the results of `sort` in a tree-like manner.

[4] This extra memory can make scripts that use `sort` much faster. For example, when `spell.sh` is executed on a single function with only 1769 MB of memory, the execution time is doubled, because `sort` becomes a bottleneck.

[5] Note that while `--width` implicitly specifies how many functions will be used—by dictating the degree of data parallelism—the number of functions is typically larger than the value of the `--width` parameter. For example, while the DFG in Fig. 4.1 has a `--width=2`, SPLASH invokes four functions.

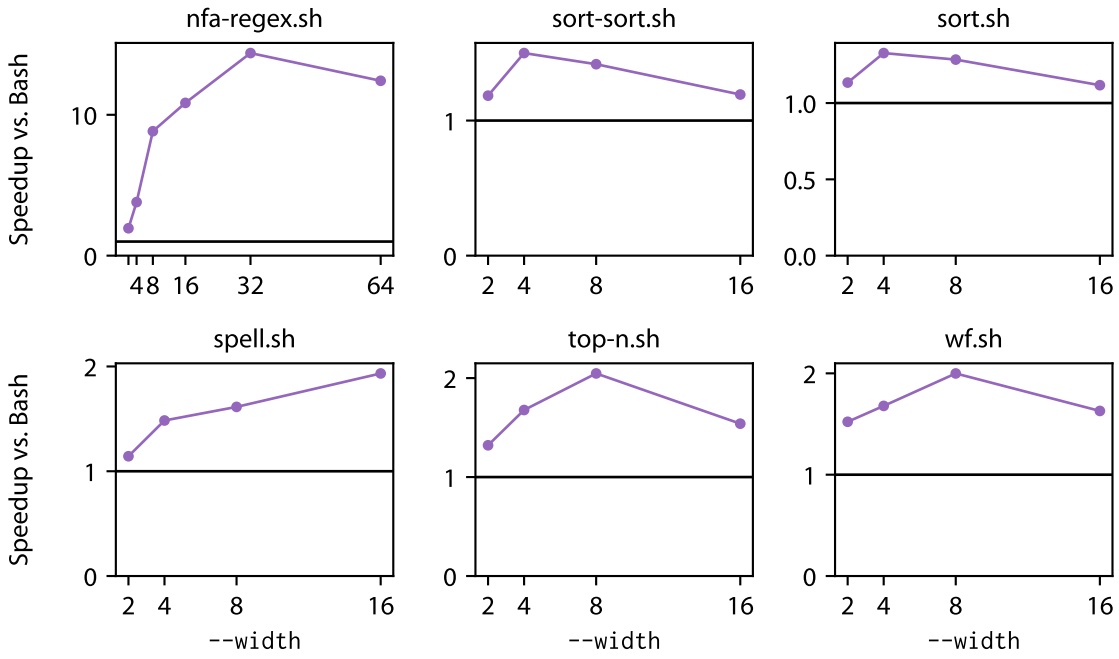


Figure 6.4: SPLASH's scalability for `Oneliners` (higher is better). The `--width` parameter varies from 2 to 64, depending on the script.

Chapter 7

Related Work

SPLaSh is related to a large body of prior work on shell scripting and serverless computing.

7.1 Shell Script Scale-Out

Tools exposing parallelism on modern Unixes such as `qsub` [21], SLURM [31], AMFS [89], and GNU `parallel` [80] are predicated upon explicit and careful orchestration from their users. Similarly, several shells add primitives for non-linear pipe topologies—some of which target parallelism [43, 78, 84]. Here too, however, users are expected to manually rewrite scripts to exploit these new primitives without jeopardizing correctness. A recent distributed shell named POSH [54] can handle a subset of shell scripts without rewriting, but is targeted towards networked, IO-intensive shell scripts, is limited to dataflow-only computations, and does not support arbitrary shell behaviours. In addition, since POSH is a shell reimplementation, it is not behaviorally equivalent with existing shells and thus risks breaking ported scripts.

Other systems for accelerating the shell include hS [40], PaSh [34, 83], and DiSh [49]. hS runs commands speculatively out of order, while PaSh and DiSh introduce parallelism and distribution to shell scripts, respectively. These systems require no modifications to the original shell, but they do not support automatic serverless deployment. SPLaSh builds on the insights—and infrastructure—of PaSh and DiSh, by taking advantage of their command annotation library and JIT compilation engine.

7.2 Serverless Beyond Event-Handling

Many recent papers [1, 2, 8, 15, 26, 27, 30, 33, 36, 37, 53, 58, 75, 79, 85] have explored using serverless for tasks beyond event handling. A main motivation is that functions boot much faster than VMs, allowing tenants to quickly launch many compute cores—without provisioning a long-running cluster. Serverless platforms thus provide a scalable computation substrate where the amount of computational resources available to a task can be rapidly altered. Prior work has exploited this flexibility for numerical computation [2, 15, 26, 27, 33, 75, 85], machine learning [8, 30, 79], data analytics [33, 36, 37, 38, 53, 58], video processing [1], and sorting [33, 53]. However, since these efforts target specific domains, they are often suboptimal for other types of computation—if they can express them at all.

7.3 General-Purpose Serverless Computing

Several frameworks have been developed for simplifying general-purpose serverless computing, both from academia [17, 18, 33, 88] and cloud providers [46, 47, 70]. While these frameworks simplify serverless development, they share a few limitations, as each step of the workflow creates programming burden. To use these frameworks, the programmer must first partition the computation into small components—each of which must fit within the function time limit and meet a range of framework-specific restrictions—and then compose these components into a serverless application using a framework-specific format (e.g., finite state machines [17, 46, 70] or data-flow graphs [18]), adding further cognitive overhead. For example, Kappa [88] requires modifications to application code—the programmer

must insert checkpoint calls at appropriate points in the program, mark calls that have externally visible side-effects, and use Kappa's concurrency primitives instead of native ones. Additionally, most of these frameworks do not support fully dynamic behaviors—such as those seen in shell scripts. On the contrary, SPLASH's just-in-time approach allows it to always have the most up-to-date information on the shell's state, and scale out without jeopardizing correctness.

7.4 Serverless Communication and Orchestration

Several frameworks support communication and orchestration for serverless workflows. On the one hand, most of them target specific workloads and execution patterns, such as data analytics and distributed synchronization [3, 37, 38, 41, 48, 52, 53] or machine learning [8, 32]—and thus focus on domain-specific optimizations. On the other, some of these frameworks are designed to handle communication for general-purpose tasks [17, 18, 33]. However, they either use cloud storage for intermediate data—thus incurring high latency and cost—or use custom stores, in-memory caches, and messaging servers—thus compromising the advantages of serverless. Cloud providers also offer some solutions [47, 70], but they require the user to manually translate their programs to a less expressive format, ahead of time. Finally, FMI [10] supports many different communication backends—object storage, in-memory storage, TCP communication—but again requires the user to manually rewrite their programs to use its API.

Chapter 8

Discussion

8.1 Future Work

SPLASH is only a first step in leveraging serverless for shell script acceleration and scale-out, and lots of opportunities exist for future work. Some of them are listed here.

Long Running Scripts Serverless platforms impose an execution timeout on functions—15 min in the case of AWS. This means that scripts that run for longer than that are not supported by default—and thus require special care. Since timeouts could be viewed as failures, the insights from fault tolerance can apply in this case as well. Something to keep in mind is that highly parallelizable, CPU-intensive scripts can benefit the most from SPLASH, and even if such a script needs a long time to run in a sequential manner, it can be drastically sped up when a large number of functions is employed.

A more general solution would be to make functions aware of their timeout, and employ some data re-routing logic. For example, if a function has been running for 50% of its timeout, it can invoke a new function, and notify its upstream function(s) to start sending data to that new function—similarly, it can notify its downstream function(s) to start receiving data from that new function. However, this approach would work only for the subset of commands that do not track intermediate state. In the case of stateful black-box commands, saving this state is not trivial, and doing so in a general enough way could incur high overheads. Generalizing re-routing for all classes of commands is an interesting problem to explore.

Serverless Mappings for the Shell Porting shell commands such as `sort`, `tr`, and `grep` to serverless appears to be simple; these commands take an input, process it, and produce the corresponding output. However, other classes of commands do not have such clear serverless mappings. For example, what is the expected behavior when the user runs `ls` or `cd`? Should `ls` return the contents of the local directory, or the contents of a bucket in serverless object storage? Should `cd` change the local working directory, or should it cause all object storage requests to use a certain prefix—i.e., making `(cd newDir; object-get myObj)` equivalent to `(object-get newDir/myObj)`?

While one could simply define a one-size-fits-all solution for all commands—everything has local semantics, or everything has serverless semantics—this approach would exclude some behaviors, which it would not support out of the box. For example, if `ls` had only local semantics, users that wanted to list objects in cloud storage would have to manually modify their scripts to instead use a command similar to Amazon S3's `list-objects`. Another approach is to add a new set of serverless command annotations that can be configured beforehand. It is also perhaps worth investigating if there is value in introducing annotations that users place in their scripts, to specify that some commands should run in a local or serverless context, respectively.

Fault Tolerance SPLASH currently does not support fault tolerance. Current distributed processing frameworks [13, 19, 87] provide fault tolerance by either confining developers to a more restrictive programming model, or providing a pre-defined API that allows for fault tolerance optimizations—like checkpointing. This is difficult to do in the case of the shell, because the shell acts as a composition

language for arbitrary components that cannot be tampered with, composed in arbitrary ways. For example, commands like `wc -l` track internal state in memory, and it is not possible to have access to that state—without changing the source code of `wc`. This means that in case of a fault, complete re-execution, would be required, even if the aggregation of the `wc` command was 99% complete. Another point is that serverless platforms provide the option to re-execute functions in case of failure, but this requires that the functions be idempotent—so that re-execution does not cause any unwanted side effects. This is not the case for the shell, since it is inherently stateful, with streaming making matters even worse.

An advantage that could possibly be exploited is the abundance of resources that serverless platforms provide. Users could execute multiple instances of functions running the same script, on the same data, at the same time, and keep the result of the first function that completes execution—additionally reducing stragglers [52]. In order to avoid excessive monetary cost, the user could specify classes of functions that require such special treatment. For example, functions performing aggregation could be deemed as more valuable—and thus executed in multiple instances—while functions that perform map operations can be re-executed as needed.

Optimal Resource Allocation and Parameter Tuning SPLASH introduces a set of parameters that are tunable by the user—such as the parallelization width, the memory of each function, or the batch size in streaming communication. These factors can greatly influence the performance of the script in non-obvious ways, especially since memory and CPU resources are often coupled in serverless offerings. Furthermore, it is complex, time-consuming, and cost-inducing to tune these parameters by hand. Thus, it would be beneficial to provide some automatic tuning mechanism that would adjust these parameters based on the script's characteristics. This could be done in an offline manner—once for each command, similar to PASH annotations—in an online manner—by logging execution metadata in real time—or a combination of both.

Cost-Aware Hybrid Execution As has already been mentioned, the serverless paradigm is not a one-size-fits-all solution—it requires careful thought about the types of workloads that a user wants to execute, and how frequent these workloads are. Of course, running workloads on managed clusters on the cloud or on physical infrastructure has its own set of shortcomings. An interesting direction is how to get the best of both worlds. One solution is to have a set of reserved instances, and when the need arises, to burst into serverless. That decision has to be made quickly, in a cost- and workload-aware manner, to decide what the sweet spot is. Some work has already been done on that end [51, 56].

Cost-Aware Optimal Graph Transformation Currently, SPLASH splits and scales out the dataflow graph based on static factors—such as the parallelization width defined by the user, or whether a dataflow segment involves a parallel stage. This approach works in single-machine or static cluster settings, where the number of threads or nodes is known in advance, and scaling out has small monetary cost implications mostly connected to energy consumption. However, in the serverless case, every function invocation has a monetary cost attached to it—expanding to more functions than necessary has negative effects. For example, if a script does not scale well beyond 32 functions, then invoking 1000 functions would incur an unnecessary monetary cost—and possibly hamper performance. Thus, the user could specify a budget, and SPLASH could try to limit its resource usage to that budget, or—more ideally—find a near-optimal resource allocation for that budget. This could be done by using an cost model that predicts the cost of a certain resource allocation, and guide the resource allocation process accordingly, in an online or offline manner [76].

8.2 Conclusion

SPLASH enables push-button serverless scale-out for the shell, in order to reap the benefits of serverless. It does so by introducing parallelism to shell scripts and distributing work to multiple functions, while facilitating function deployment and communication. SPLASH operates in a just in time manner, and

does not require any modifications to the original scripts. SPLASH's evaluation shows it to be effective for CPU-intensive, highly parallelizable workloads, providing up to 14.38× speedup over sequential shell execution.

Bibliography

[1] Lixiang Ao et al. "Sprocket: A Serverless Video Processing Framework." In: *Proceedings of the ACM Symposium on Cloud Computing*. SoCC '18. Carlsbad, CA, USA: Association for Computing Machinery, 2018, pp. 263–274. ISBN: 9781450360111. DOI: 10.1145/3267809.3267815. URL: https://doi.org/10.1145/3267809.3267815.

[2] Arda Aytekin and Mikael Johansson. "Harnessing the Power of Serverless Runtimes for Large-Scale Optimization." In: *CoRR* abs/1901.03161 (2019). arXiv: 1901.03161. URL: http://arxiv.org/abs/1901.03161.

[3] Daniel Barcelona-Pons et al. "Stateful Serverless Computing with Crucial." In: *ACM Trans. Softw. Eng. Methodol.* 31.3 (Mar. 2022). ISSN: 1049-331X. DOI: 10.1145/3490386. URL: https://doi.org/10.1145/3490386.

[4] Jon Bentley. "Programming pearls: a spelling checker." In: *Commun. ACM* 28.5 (May 1985), pp. 456–462. ISSN: 0001-0782. DOI: 10.1145/3532.315102. URL: https://doi.org/10.1145/3532.315102.

[5] Jon Bentley, Don Knuth, and Doug McIlroy. "Programming pearls: a literate program." In: *Commun. ACM* 29.6 (June 1986), pp. 471–483. ISSN: 0001-0782. DOI: 10.1145/5948.315654. URL: https://doi.org/10.1145/5948.315654.

[6] Pawan Bhandari. *Solutions to unixgame.io*. Accessed: 2020-04-14. 2020.

[7] Ivano Bonesana, Marco Paolieri, and Marco D. Santambrogio. "An adaptable FPGA-based system for regular expression matching." In: *Proceedings of the Conference on Design, Automation and Test in Europe*. DATE '08. Munich, Germany: Association for Computing Machinery, 2008, pp. 1262–1267. ISBN: 9783981080131. DOI: 10.1145/1403375.1403681. URL: https://doi.org/10.1145/1403375.1403681.

[8] Joao Carreira et al. "Cirrus: a Serverless Framework for End-to-end ML Workflows." In: *Proceedings of the ACM Symposium on Cloud Computing*. SoCC '19. Santa Cruz, CA, USA: Association for Computing Machinery, 2019, pp. 13–24. ISBN: 9781450369732. DOI: 10.1145/3357223.3362711. URL: https://doi.org/10.1145/3357223.3362711.

[9] Kenneth Ward Church. *Unix for poets*. Notes of a course from the European Summer School on Language and Speech Communication, Corpus Based Methods. 1994.

[10] Marcin Copik et al. "FMI: Fast and Cheap Message Passing for Serverless Functions." In: *Proceedings of the 37th International Conference on Supercomputing*. ICS '23. Orlando, FL, USA: Association for Computing Machinery, 2023, pp. 373–385. ISBN: 9798400700569. DOI: 10.1145/3577193.3593718. URL: https://doi.org/10.1145/3577193.3593718.

[11] Datadog. *The State of Serverless 2021*. Accessed: 2024-06-16. 2021. URL: https://www.datadoghq.com/state-of-serverless-2021/.

[12] Datadog. *The State of Serverless 2023*. Accessed: 2024-06-16. 2023. URL: https://www.datadoghq.com/state-of-serverless/.

[13] Jeffrey Dean and Sanjay Ghemawat. "MapReduce: Simplified Data Processing on Large Clusters." In: *6th Symposium on Operating Systems Design & Implementation (OSDI 04)*. San Francisco, CA: USENIX Association, Dec. 2004. URL: https://www.usenix.org/conference/osdi-04/mapreduce-simplified-data-processing-large-clusters.

[14] Simon Eismann et al. "The State of Serverless Applications: Collection, Characterization, and Community Consensus." In: *IEEE Transactions on Software Engineering* 48.10 (2022), pp. 4152–4166. DOI: 10.1109/TSE.2021.3113940.

[15] Lang Feng et al. "Exploring Serverless Computing for Neural Network Training." In: *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*. 2018, pp. 334–341. DOI: 10.1109/CLOUD.2018.00049.

[16] Bryan Ford, Pyda Srisuresh, and Dan Kegel. *Peer-to-Peer Communication Across Network Address Translators*. 2006. arXiv: cs/0603074 [cs.NI].

[17] Sadjad Fouladi et al. "Encoding, Fast and Slow: Low-Latency Video Processing Using Thousands of Tiny Threads." In: *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. Boston, MA: USENIX Association, Mar. 2017, pp. 363–376. ISBN: 978-1-931971-37-9. URL: https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/fouladi.

[18] Sadjad Fouladi et al. "From Laptop to Lambda: Outsourcing Everyday Jobs to Thousands of Transient Functional Containers." In: *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. Renton, WA: USENIX Association, July 2019, pp. 475–488. ISBN: 978-1-939133-03-8. URL: http://www.usenix.org/conference/atc19/presentation/fouladi.

[19] Apache Software Foundation. *Apache Flink*. Accessed: 2024-06-16. 2024. URL: https://flink.apache.org/.

[20] Inc. Free Software Foundation. *GNU Bash*. Accessed: 2024-06-13. 2024. URL: https://www.gnu.org/software/bash/.

[21] W. Gentzsch. "Sun Grid Engine: Towards Creating a Compute Power Grid." In: *Cluster Computing and the Grid, IEEE International Symposium on*. Los Alamitos, CA, USA: IEEE Computer Society, May 2001, p. 35. DOI: 10.1109/CCGRID.2001.923173. URL: https://doi.ieeecomputersociety.org/10.1109/CCGRID.2001.923173.

[22] GitHub. *The State of the Octoverse: The Top Programming Languages*. Accessed: 2024-06-13. 2022. URL: https://octoverse.github.com/2022/top-programming-languages.

[23] Google. *Resource Limits*. Accessed: 2024-06-13. 2024. URL: https://cloud.google.com/functions/quotas#resource_limits.

[24] Michael Greenberg. *The POSIX Shell Is an Interactive DSL for Concurrency*. URL: https://cs.pomona.edu/~michael/papers/dsldi2018.pdf.

[25] Michael Greenberg, Konstantinos Kallas, and Nikos Vasilakis. "Unix shell programming: the next 50 years." In: *Proceedings of the Workshop on Hot Topics in Operating Systems*. HotOS '21. Ann Arbor, Michigan: Association for Computing Machinery, 2021, pp. 104–111. ISBN: 9781450384384. DOI: 10.1145/3458336.3465294. URL: https://doi.org/10.1145/3458336.3465294.

[26] Vipul Gupta et al. "OverSketched Newton: Fast Convex Optimization for Serverless Systems." In: *2020 IEEE International Conference on Big Data (Big Data)*. 2020, pp. 288–297. DOI: 10.1109/BigData50022.2020.9378289.

[27] Vipul Gupta et al. *Serverless Straggler Mitigation using Local Error-Correcting Codes*. 2020. arXiv: 2001.07490 [cs.DC].

[28] IETF. *Session Traversal Utilities for NAT (STUN)*. Accessed: 2024-06-13. 2020. URL: https://www.rfc-editor.org/rfc/rfc8489.

[29] Serverless Inc. *Serverless Framework*. Accessed: 2024-06-13. 2024. URL: https://www.serverless.com/.

[30] Vatche Ishakian, Vinod Muthusamy, and Aleksander Slominski. *Serving deep learning models in a serverless platform*. 2018. arXiv: 1710.08460 [cs.DC].

[31] Morris Jette, Andy Yoo, and Mark Grondona. "SLURM: Simple linux utility for resource management." In: July 2003. ISBN: 978-3-540-20405-3. DOI: 10.1007/10968987_3.

[32] Jiawei Jiang et al. "Towards Demystifying Serverless Machine Learning Training." In: *Proceedings of the 2021 International Conference on Management of Data*. SIGMOD '21. Virtual Event, China: Association for Computing Machinery, 2021, pp. 857–871. ISBN: 9781450383431. DOI: 10.1145/3448016.3459240. URL: https://doi.org/10.1145/3448016.3459240.

[33] Eric Jonas et al. "Occupy the cloud: distributed computing for the 99%." In: *Proceedings of the 2017 Symposium on Cloud Computing*. SoCC '17. Santa Clara, California: Association for Computing Machinery, 2017, pp. 445–451. ISBN: 9781450350280. DOI: 10.1145/3127479.3128601. URL: https://doi.org/10.1145/3127479.3128601.

[34] Konstantinos Kallas et al. "Practically Correct, Just-in-Time Shell Script Parallelization." In: *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. Carlsbad, CA: USENIX Association, July 2022, pp. 769–785. ISBN: 978-1-939133-28-1. URL: https://www.usenix.org/conference/osdi22/presentation/kallas.

[35] Dimitrios Kaloudas, Nikolet Pavlova, and Robert Penchovsky. "EBWS: Essential Bioinformatics Web Services for Sequence Analyses." In: *IEEE/ACM Transactions on Computational Biology and Bioinformatics* 16.3 (2019), pp. 942–953. DOI: 10.1109/TCBB.2018.2816645.

[36] Youngbin Kim and Jimmy Lin. *Serverless Data Analytics with Flint*. 2018. arXiv: 1803.06354 [cs.DC].

[37] Ana Klimovic et al. "Pocket: Elastic Ephemeral Storage for Serverless Analytics." In: *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. Carlsbad, CA: USENIX Association, Oct. 2018, pp. 427–444. ISBN: 978-1-939133-08-3. URL: https://www.usenix.org/conference/osdi18/presentation/klimovic.

[38] Ana Klimovic et al. "Understanding Ephemeral Storage for Serverless Analytics." In: *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. Boston, MA: USENIX Association, July 2018, pp. 789–794. ISBN: 978-1-939133-01-4. URL: https://www.usenix.org/conference/atc18/presentation/klimovic-serverless.

[39] Nokia Bell Labs. *The unix game—solve puzzles using unix pipes*. Accessed: 2020-03-05. 2019.

[40] Georgios Liargkovas et al. "Executing Shell Scripts in the Wrong Order, Correctly." In: *Proceedings of the 19th Workshop on Hot Topics in Operating Systems*. HOTOS '23. Providence, RI, USA: Association for Computing Machinery, 2023, pp. 103–109. ISBN: 9798400701955. DOI: 10.1145/3593856.3595891. URL: https://doi.org/10.1145/3593856.3595891.

[41] David H. Liu et al. "Doing More with Less: Orchestrating Serverless Applications without an Orchestrator." In: *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. Boston, MA: USENIX Association, Apr. 2023, pp. 1505–1519. ISBN: 978-1-939133-33-5. URL: https://www.usenix.org/conference/nsdi23/presentation/liu-david.

[42] Ashraf Mahgoub et al. "WISEFUSE: Workload Characterization and DAG Transformation for Serverless Workflows." In: *Proc. ACM Meas. Anal. Comput. Syst.* 6.2 (June 2022). DOI: 10.1145/3530892. URL: https://doi.org/10.1145/3530892.

[43] Chris McDonald and Trevor I. Dix. "Support for graphs of processes in a command interpreter." In: *Software: Practice and Experience* 18.10 (1988), pp. 1011–1016. DOI: https://doi.org/10.1002/spe.4380181007. eprint: URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.4380181007.

[44] Suejb Memeti and Sabri Pllana. "Analyzing Large-Scale DNA Sequences on Multi-core Architectures." In: *2015 IEEE 18th International Conference on Computational Science and Engineering*. 2015, pp. 208–215. DOI: 10.1109/CSE.2015.25.

[45] Microsoft. *Azure Functions Service Limits*. Accessed: 2024-06-13. 2024. URL: https://learn.microsoft.com/en-us/azure/azure-functions/functions-scale.

[46] Microsoft. *Azure Logic Apps*. Accessed: 2024-06-11. 2024. URL: https://azure.microsoft.com/en-us/products/logic-apps/.

[47] Microsoft. *What are Durable Functions?* Accessed: 2024-06-11. 2024. URL: https://learn.microsoft.com/en-us/azure/azure-functions/durable/durable-functions-overview.

[48] Ingo Müller, Renato Marroquin, and Gustavo Alonso. "Lambada: Interactive Data Analytics on Cold Data Using Serverless Cloud Infrastructure." In: *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. SIGMOD '20. Portland, OR, USA: Association for Computing Machinery, 2020, pp. 115–130. ISBN: 9781450367356. DOI: 10.1145/3318464.3389758. URL: https://doi.org/10.1145/3318464.3389758.

[49] Tammam Mustafa et al. "DiSh: Dynamic Shell-Script Distribution." In: *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. Boston, MA: USENIX Association, Apr. 2023, pp. 341–356. ISBN: 978-1-939133-33-5. URL: https://www.usenix.org/conference/nsdi23/presentation/mustafa.

[50] Stack Overflow. *Stack Overflow Developer Survey 2023*. Accessed: 2024-06-13. 2023. URL: https://survey.stackoverflow.co/2023/#section-most-popular-technologies-programming-scripting-and-markup-languages.

[51] Matthew Perron et al. "Cackle: Analytical Workload Cost and Performance Stability With Elastic Pools." In: *Proc. ACM Manag. Data* 1.4 (Dec. 2023). DOI: 10.1145/3626720. URL: https://doi.org/10.1145/3626720.

[52] Matthew Perron et al. "Starling: A Scalable Query Engine on Cloud Functions." In: *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. SIGMOD '20. Portland, OR, USA: Association for Computing Machinery, 2020, pp. 131–141. ISBN: 9781450367356. DOI: 10.1145/3318464.3380609. URL: https://doi.org/10.1145/3318464.3380609.

[53] Qifan Pu, Shivaram Venkataraman, and Ion Stoica. "Shuffling, Fast and Slow: Scalable Analytics on Serverless Infrastructure." In: *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. Boston, MA: USENIX Association, Feb. 2019, pp. 193–206. ISBN: 978-1-931971-49-2. URL: https://www.usenix.org/conference/nsdi19/presentation/pu.

[54] Deepti Raghavan et al. "POSH: A Data-Aware Shell." In: *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. USENIX Association, July 2020, pp. 617–631. ISBN: 978-1-939133-14-4. URL: https://www.usenix.org/conference/atc20/presentation/raghavan.

[55] Dennis M. Ritchie and Ken Thompson. "The UNIX time-sharing system." In: *Commun. ACM* 17.7 (July 1974), pp. 365–375. ISSN: 0001-0782. DOI: 10.1145/361011.361061. URL: https://doi.org/10.1145/361011.361061.

[56] Rohan Basu Roy et al. "Mashup: making serverless computing useful for HPC workflows via hybrid execution." In: *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. PPoPP '22. Seoul, Republic of Korea: Association for Computing Machinery, 2022, pp. 46–60. ISBN: 9781450392044. DOI: 10.1145/3503221.3508407. URL: https://doi.org/10.1145/3503221.3508407.

[57] Ghazal Sadeghian et al. "UnFaaSener: Latency and Cost Aware Offloading of Functions from Serverless Platforms." In: *2023 USENIX Annual Technical Conference (USENIX ATC 23)*. Boston, MA: USENIX Association, July 2023, pp. 879–896. ISBN: 978-1-939133-35-9. URL: https://www.usenix.org/conference/atc23/presentation/sadeghian.

[58] Josep Sampé et al. "Serverless Data Analytics in the IBM Cloud." In: *Proceedings of the 19th International Middleware Conference Industry*. Middleware '18. Rennes, France: Association for Computing Machinery, 2018, pp. 1–8. ISBN: 9781450360166. DOI: 10.1145/3284028.3284029. URL: https://doi.org/10.1145/3284028.3284029.

[59] Amazon Web Services. *Amazon DynamoDB*. Accessed: 2024-06-13. 2024. URL: https://aws.amazon.com/dynamodb/.

[60] Amazon Web Services. *Amazon Simple Queue Service (SQS)*. Accessed: 2024-06-16. 2024. URL: https://aws.amazon.com/sqs/.

[61] Amazon Web Services. *Amazon Simple Storage Service (S3)*. Accessed: 2024-06-16. 2024. URL: https://aws.amazon.com/s3/.

[62] Amazon Web Services. *Amazon SQS Long Polling*. Accessed: 2024-06-13. 2024. URL: https://docs.aws.amazon.com/AWSSimpleQueueService/latest/SQSDeveloperGuide/sqs-short-and-long-polling.html#sqs-long-polling.

[63] Amazon Web Services. *AWS Lambda*. Accessed: 2024-06-16. 2024. URL: https://aws.amazon.com/lambda/.

[64] Amazon Web Services. *AWS Lambda Custom Runtimes*. Accessed: 2024-06-13. 2024. URL: https://docs.aws.amazon.com/lambda/latest/dg/runtimes-custom.html.

[65] Amazon Web Services. *AWS Lambda Layers*. Accessed: 2024-06-13. 2024. URL: https://docs.aws.amazon.com/lambda/latest/dg/chapter-layers.html.

[66] Amazon Web Services. *AWS Lambda Supported Runtimes*. Accessed: 2024-06-13. 2024. URL: https://docs.aws.amazon.com/lambda/latest/dg/lambda-runtimes.html#runtimes-supported.

[67] Amazon Web Services. *AWS Lambda: Cold Starts and Latency*. Accessed: 2024-06-13. 2024. URL: https://docs.aws.amazon.com/lambda/latest/operatorguide/execution-environments.html#cold-start-latency.

[68] Amazon Web Services. *AWS Lambda: Lambda Functions Calling Lambda Functions*. Accessed: 2024-06-13. 2024. URL: https://docs.aws.amazon.com/lambda/latest/operatorguide/functions-calling-functions.html.

[69] Amazon Web Services. *AWS SDK for Python (Boto3)*. Accessed: 2024-06-13. 2024. URL: https://aws.amazon.com/sdk-for-python/.

[70] Amazon Web Services. *AWS Step Functions*. Accessed: 2024-06-11. 2024. URL: https://aws.amazon.com/step-functions/.

[71] Amazon Web Services. *Configure Lambda Function Memory*. Accessed: 2024-06-13. 2024. URL: https://docs.aws.amazon.com/lambda/latest/dg/configuration-memory.html.

[72] Amazon Web Services. *Function Configuration, Deployment, and Execution*. Accessed: 2024-06-13. 2024. URL: https://docs.aws.amazon.com/lambda/latest/dg/gettingstarted-limits.html#function-configuration-deployment-and-execution.

[73] Amazon Web Services. *Memory and Computing Power*. Accessed: 2024-06-13. 2024. URL: https://docs.aws.amazon.com/lambda/latest/operatorguide/computing-power.html.

[74] Mohammad Shahrad et al. "Serverless in the Wild: Characterizing and Optimizing the Serverless Workload at a Large Cloud Provider." In: *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. USENIX Association, July 2020, pp. 205–218. ISBN: 978-1-939133-14-4. URL: https://www.usenix.org/conference/atc20/presentation/shahrad.

[75] Vaishaal Shankar et al. "Serverless linear algebra." In: *Proceedings of the 11th ACM Symposium on Cloud Computing*. SoCC '20. Virtual Event, USA: Association for Computing Machinery, 2020, pp. 281–295. ISBN: 9781450381376. DOI: 10.1145/3419111.3421287. URL: https://doi.org/10.1145/3419111.3421287.

[76] Prasoon Sinha, Kostis Kaffes, and Neeraja J. Yadwadkar. "Online Learning for Right-Sizing Serverless Functions." In: *Architecture and System Support for Transformer Models (ASSYST @ISCA 2023)*. 2023. URL: https://openreview.net/forum?id=4zdPNY3SDQk.

[77] *Solutions for unixgame.io*. Accessed: 2024-06-13. URL: https://github.com/psinghbh/softsec.github.io/tree/master/ctf/unixgame.io.

[78] Diomidis Spinellis and Marios Fragkoulis. "Extending Unix Pipelines to DAGs." In: *IEEE Transactions on Computers* 66.9 (2017), pp. 1547–1561. DOI: 10.1109/TC.2017.2695447.

[79] Vikram Sreekanti et al. *Optimizing Prediction Serving on Low-Latency Serverless Dataflow*. 2020. arXiv: 2007.05832 [cs.DC].

[80] Ole Tange. "GNU Parallel - The Command-Line Power Tool." In: *The USENIX Magazine* 36 (Jan. 2011), pp. 42–47.

[81] Dave Taylor and Brandon Perry. *Wicked Cool Shell Scripts, 2nd Edition*. 2016. URL: https://nostarch.com/wcss2.

[82] Eleftheria Tsaliki and Diomidis Spinellis. *The Real Numbers for Athens Buses (in Greek)*. Accessed: 2024-06-13. 2020. URL: https://insidestory.gr/article/noymera-leoforeia-athinas.

[83] Nikos Vasilakis et al. "PaSh: light-touch data-parallel shell processing." In: *Proceedings of the Sixteenth European Conference on Computer Systems*. EuroSys '21. Online Event, United Kingdom: Association for Computing Machinery, 2021, pp. 49–66. ISBN: 9781450383349. DOI: 10.1145/3447786.3456228. URL: https://doi.org/10.1145/3447786.3456228.

[84] Edward Walker, Weijia Xu, and Vinoth Chandar. "Composing and executing parallel data-flow graphs with shell pipes." In: *Proceedings of the 4th Workshop on Workflows in Support of Large-Scale Science*. WORKS '09. Portland, Oregon: Association for Computing Machinery, 2009. ISBN: 9781605587172. DOI: 10.1145/1645164.1645175. URL: https://doi.org/10.1145/1645164.1645175.

[85] Sebastian Werner et al. "Serverless Big Data Processing using Matrix Multiplication as Example." In: *2018 IEEE International Conference on Big Data (Big Data)*. 2018, pp. 358–365. DOI: 10.1109/BigData.2018.8622362.

[86] Tom White. *Hadoop: The Definitive Guide*. 2009. URL: https://www.oreilly.com/library/view/hadoop-the-definitive/9780596521974/.

[87] Matei Zaharia et al. "Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing." In: *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*. San Jose, CA: USENIX Association, Apr. 2012, pp. 15–28. ISBN: 978-931971-92-8. URL: https://www.usenix.org/conference/nsdi12/technical-sessions/presentation/zaharia.

[88] Wen Zhang et al. "Kappa: a programming framework for serverless computing." In: *Proceedings of the 11th ACM Symposium on Cloud Computing*. SoCC '20. Virtual Event, USA: Association for Computing Machinery, 2020, pp. 328–343. ISBN: 9781450381376. DOI: 10.1145/3419111.3421277. URL: https://doi.org/10.1145/3419111.3421277.

[89] Zhao Zhang et al. "Parallelizing the execution of sequential scripts." In: *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. SC '13. Denver, Colorado: Association for Computing Machinery, 2013. ISBN: 9781450323789. DOI: 10.1145/2503210.2503222. URL: https://doi.org/10.1145/2503210.2503222.

Appendix A

Full Shell Script Produced by PaSh for NFA-Regex

```
#!/bin/bash
cd "$(dirname "${0}")"
[ -z "${PASH_TOP}" ]
rm_pash_fifos() {
    rm -f "/tmp/Fgq4/f2";
    rm -f "/tmp/Fgq4/f7"
    # ...similarly for the rest of the fifos...
}
mkfifo_pash_fifos() {
    mkfifo "/tmp/Fgq4/f2"
    mkfifo "/tmp/Fgq4/f7"
    # ...similarly for the rest of the fifos...
}
rm_pash_fifos; mkfifo_pash_fifos; pids_to_kill=""
cat "in.txt" >"/tmp/Fgq4/f2" & pids_to_kill="${!} ${pids_to_kill}"
r_split "/tmp/Fgq4/f2" 1000000 "/tmp/Fgq4/f7" "/tmp/Fgq4/f8" &
pids_to_kill="${!} ${pids_to_kill}"
r_wrap bash -c 'tr A-Z a-z' <"/tmp/Fgq4/f13" >"/tmp/Fgq4/f9" &
pids_to_kill="${!} ${pids_to_kill}"
r_wrap bash -c 'tr A-Z a-z' <"/tmp/Fgq4/f14" >"/tmp/Fgq4/f10" &
pids_to_kill="${!} ${pids_to_kill}"
r_wrap bash -c 'grep \(.\).*\1\(.\).*\2\(.\).*\3\(.\).*\4' <"/tmp/Fgq4/f9" >"/tmp/Fgq4/f11" &
pids_to_kill="${!} ${pids_to_kill}"
r_wrap bash -c 'grep \(.\).*\1\(.\).*\2\(.\).*\3\(.\).*\4' <"/tmp/Fgq4/f10" >"/tmp/Fgq4/f12" &
pids_to_kill="${!} ${pids_to_kill}"
dgsh-tee -i "/tmp/Fgq4/f7" -o "/tmp/Fgq4/f13" -b 5M & pids_to_kill="${!} ${pids_to_kill}"
dgsh-tee -i "/tmp/Fgq4/f8" -o "/tmp/Fgq4/f14" -b 5M & pids_to_kill="${!} ${pids_to_kill}"
dgsh-tee -i "/tmp/Fgq4/f11" -o "/tmp/Fgq4/f15" -b 5M & pids_to_kill="${!} ${pids_to_kill}"
dgsh-tee -i "/tmp/Fgq4/f12" -o "/tmp/Fgq4/f16" -b 5M & pids_to_kill="${!} ${pids_to_kill}"
r_merge "/tmp/Fgq4/f15" "/tmp/Fgq4/f16" & pids_to_kill="${!} ${pids_to_kill}"
source wait_for_output_and_sigpipe_rest.sh ${!}; rm_pash_fifos; (exit "${internal_exec_status}")
```

Listing A.1: Full script produced by PaSh for NFA-Regex (`--width=2`).

Appendix B

Final Shell Scripts Produced by SPLASH for NFA-Regex

```
#!/bin/bash
mkfifo f{0..3}
invoke-function $SCRIPT_ID_2 $SCRIPT_MAP &
invoke-function $SCRIPT_ID_3 $SCRIPT_MAP &
download-object in.txt >f0 &
cat f0 >f1 &
split f1 f2 f3 &
send-stream $COMM_KEY_1_2 <f2 &
send-stream $COMM_KEY_1_3 <f3 &
wait
```

Listing B.1: Final script for Subgraph 1. Serverless-specific commands are highlighted.

```
#!/bin/bash
mkfifo f{0..4}
invoke-function $SCRIPT_ID_4 $SCRIPT_MAP &
receive-stream $COMM_KEY_1_2 >f0 &
eager <f0 >f1 &
tr A-Z a-z <f1 >f2 &
grep '\(.\).*\1\(.\).*\2\(.\).*\3\(.\).*\4' <f2 >f3 &
eager <f3 >f4 &
send-stream $COMM_KEY_2_4 <f4 &
wait
```

Listing B.2: Final script for Subgraph 2. Serverless-specific commands are highlighted.

```
#!/bin/bash
mkfifo f{0..4}
receive-stream $COMM_KEY_1_3 >f0 &
eager <f0 >f1 &
tr A-Z a-z <f1 >f2 &
grep '\(.\).*\1\(.\).*\2\(.\).*\3\(.\).*\4' <f2 >f3 &
eager <f3 >f4 &
send-stream $COMM_KEY_3_4 <f4 &
wait
```

Listing B.3: Final script for Subgraph 3. Serverless-specific commands are highlighted.

```
#!/bin/bash
mkfifo f{0..2}
receive-stream $COMM_KEY_2_4 >f0 &
receive-stream $COMM_KEY_3_4 >f1 &
merge f0 f1 >f2 &
upload-object <f2 out.txt &
wait
send-notification
```

Listing B.4: Final script for Subgraph 4. Serverless-specific commands are highlighted.

Appendix C

Python Code for Interacting With AWS

```
import boto3
import sys
import json

id_, data_path = sys.argv[1:]

with open(data_path, "r") as f:
    data = f.read()

lambda_client = boto3.client("lambda")

lambda_client.invoke(
    FunctionName="lambda",
    InvocationType="Event",
    LogType="None",
    Payload=json.dumps({"id": id_, "data": data}),
)
```

Listing C.1: Function invocation.

```python
import subprocess
import json

def lambda_handler(event, context):
    data = event["data"]
    id_ = event["id"]
    scripts_dict = json.loads(data)

    with open(f"/tmp/data-{id_}", "w") as f:
        f.write(data)

    with open(f"/tmp/script-{id_}.sh", "w") as f:
        f.write(scripts_dict[id_])

    process = subprocess.run(
        ["/bin/bash", f"/tmp/script-{id_}.sh", f"/tmp/data-{id_}"]
    )
```

Listing C.2: Function handler.

```python
import boto3
import json

sqs_client = boto3.client("sqs")
message_body = {"message": "done", "output_file_id": object_key}

try:
    response = sqs_client.send_message(
        QueueUrl=f"https://sqs.us-east-1.amazonaws.com/{AWS_ACCOUNT_ID}/{QUEUE}",
        MessageBody=json.dumps(message_body),
    )
except Exception as e:
    print(e)
```

Listing C.3: Message send.

```python
import time
import boto3

def wait_msg_done():
    while True:
        sqs = boto3.client("sqs")
        queue_url = f"https://sqs.us-east-1.amazonaws.com/{AWS_ACCOUNT_ID}/{QUEUE}"

        response = sqs.receive_message(
            QueueUrl=queue_url,
            AttributeNames=["SentTimestamp"],
            MaxNumberOfMessages=1,
            MessageAttributeNames=["All"],
            VisibilityTimeout=30,
            WaitTimeSeconds=20,
        )

        try:
            message = response["Messages"][0]
            receipt_handle = message["ReceiptHandle"]

            sqs.delete_message(QueueUrl=queue_url, ReceiptHandle=receipt_handle)

            break
        except:
            time.sleep(1)
```

Listing C.4: Message receive.

```python
import boto3
import sys
import os

BUCKET = os.environ.get("AWS_BUCKET")

object_key, outfile = sys.argv[1:]
batch = 10000

session = boto3.Session()
s3 = session.client("s3")

try:
    response = s3.get_object(Bucket=BUCKET, Key=object_key)
except Exception as e:
    print(e)

with open(outfile, "wb") as f:
    while True:
        x = response["Body"].read(batch)

        if not x:
            break

        f.write(x)
        f.flush()
```

Listing C.5: Object storage download.

```python
import boto3
import sys
import json
import os

AWS_ACCOUNT_ID = os.environ.get("AWS_ACCOUNT_ID")
BUCKET = os.environ.get("AWS_BUCKET")
QUEUE = os.environ.get("AWS_QUEUE")

object_key, infile = sys.argv[1:]

session = boto3.Session()
s3 = session.client("s3")

with open(infile, "rb") as file:
    object_data = file.read()

s3.put_object(Bucket=BUCKET, Key=object_key, Body=object_data)

sqs_client = boto3.client("sqs")
message_body = {"message": "done", "output_file_id": object_key}
try:
    response = sqs_client.send_message(
        QueueUrl=f"https://sqs.us-east-1.amazonaws.com/{AWS_ACCOUNT_ID}/{QUEUE}",
        MessageBody=json.dumps(message_body),
    )
except Exception as e:
    print(e)
```

Listing C.6: Object storage upload.

Appendix D

AWS Equivalent Services in Other Cloud Providers

Table D.1: AWS equivalent services. Substituting equivalent services from other cloud providers should provide equivalent behavior, but is not guaranteed to provide equivalent performance.

AWS Service	Google Cloud Equivalent	Microsoft Azure Equivalent
AWS Lambda	Google Cloud Functions	Azure Functions
Amazon S3	Google Cloud Storage	Azure Blob Storage
Amazon DynamoDB	Google Bigtable	Azure Cosmos DB
Amazon SQS	Google Cloud Pub/Sub	Azure Service Bus

Appendix E

Full Specification of Shell Scripts Used in Evaluation

Table E.1: Scripts included in the `Oneliners` benchmark set. For each script, the value of `--width` used in the evaluation is shown.

	Script	Input	Seq. Time	`--width`	Notes
1	nfa-regex.sh	200 MB	598.74 s	32	Match complex regex over input
2	sort-sort.sh	200 MB	20.11 s	4	Calculate sort twice
3	sort.sh	200 MB	28.74 s	4	Sort input
4	spell.sh	200 MB	62.37 s	16	Calculate mispelled words
5	top-n.sh	100 MB	40.98 s	8	Find top 100 terms
6	wf.sh	100 MB	40.76 s	8	Sort words by frequency

Table E.2: Scripts included in the `Unix50` benchmark set. For each script, the value of `--width` used in the evaluation is shown.

	Script	Input	Seq. Time	`--width`	Notes
1	1.sh	1.1 GB	45.32 s	2	Extract the last name
2	2.sh	1.1 GB	42.35 s	4	Get all Unix utilities
3	3.sh	1.1 GB	46.28 s	4	Get lowercase first letter of last names
4	4.sh	1.1 GB	48.26 s	4	Extract hello world
5	5.sh	1.1 GB	52.75 s	2	Extract the word BELL
6	6.sh	1.1 GB	38.14 s	4	Four corners
7	7.sh	1.1 GB	38.75 s	2	List Turing award recipients while working at Bell Labs
8	8.sh	1.1 GB	39.15 s	4	Year Ritchie and Thompson receive the Hamming medal

Table E.3: Scripts included in the `COVID-mts` benchmark set. For each script, the value of `--width` used in the evaluation is shown.

	Script	Input	Seq. Time	`--width`	Notes
1	1.sh	200 MB	20.29 s	2	Vehicles on the road per day
2	2.sh	500 MB	43.35 s	2	Days a vehicle is on the road
3	3.sh	500 MB	48.64 s	2	Hours each vehicle is on the road
4	4.sh	1 GB	46.90 s	4	Hours monitored each day

Table E.4: Scripts included in the `NLP` benchmark set. For each script, the value of `--width` used in the evaluation is shown.

	Script	Input	Seq. Time	`--width`	Notes
1	1.sh	10 books	19.47 s	2	Count words
2	2.sh	10 books	21.35 s	2	Merge upper and lower counts
3	3.sh	10 books	19.66 s	2	Sort
4	4.sh	10 books	25.32 s	2	Sort words by folding
5	5.sh	10 books	19.66 s	2	Uppercase by token
6	6.sh	10 books	21.27 s	2	Uppercase by type
7	7.sh	10 books	39.81 s	2	Four-letter words
8	8.sh	10 books	19.11 s	4	Words no vowels
9	9.sh	10 books	20.49 s	4	One-syllable words
10	10.sh	10 books	16.64 s	2	Two-syllable words
11	11.sh	10 books	56.10 s	2	Verses with certain instances of the word “light”
12	12.sh	10 books	19.32 s	2	Count consonant sequences
13	13.sh	10 books	21.94 s	2	Vowel sequences greater than 1K

Table E.5: Scripts included in the `NOAA` benchmark set. For each script, the value of `--width` used in the evaluation is shown.

	Script	Input	Seq. Time	`--width`	Notes
1	noaa.sh	500 MB	49.98 s	8	Find maximum and minimum temperature