ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ

ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Pipelined MapReduce: A Decoupled MapReduce RunTime for Shared Memory Multi-Processors

---

Συγγραφέας:
Κωνσταντίνος Ηλιάκης

Επιβλέπων:
Δημήτριος Σούντρης
Αναπληρωτής Καθηγητής

ΕΡΓΑΣΤΗΡΙΟ ΜΙΚΡΟΥΠΟΛΟΓΙΣΤΩΝ ΚΑΙ ΨΗΦΙΑΚΩΝ ΣΥΣΤΗΜΑΤΩΝ

13 Μαρτίου 2017

ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ

ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ

ΕΡΓΑΣΤΗΡΙΟ ΜΙΚΡΟΥΠΟΛΟΓΙΣΤΩΝ ΚΑΙ ΨΗΦΙΑΚΩΝ ΣΥΣΤΗΜΑΤΩΝ

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Pipelined MapReduce: A Decoupled MapReduce RunTime for Shared Memory Multi-Processors

---

Συγγραφέας:
Κωνσταντίνος Ηλιάκης

Επιβλέπων:
Δημήτριος Σούντρης
Αναπληρωτής Καθηγητής

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 13η Μαρτίου 2017.

(Υπογραφή)	*(Υπογραφή)*	*(Υπογραφή)*
Δημήτριος Σούντρης	Κιαμάλ Πεκμεστζή	Νεκτάριος Κοζύρης
Αναπληρωτής Καθηγητής	Καθηγητής	Καθηγητής

13 Μαρτίου 2017

Πνευματικά Δικαιώματα

Υπογραφή:

_____

Ημερομηνία:

_____

National Technical University of Athens

Abstract

Division of Computer Science
School of Electrical And Computer Engineering

Diploma Thesis

Pipelined MapReduce: A Decoupled MapReduce RunTime for Shared Memory Multi-Processors

by Konstantinos Iliakis

Modern multi-processors embody up to hundreds of cores in a single chip, in an attempt to attain TFlops/sec performance. Many subtle programming frameworks have emerged in order to facilitate the development of parallel, efficient and scalable applications. The MapReduce programming model, after having indisputably, demonstrated its usability and effectiveness in the area of Large-Scale Distributed Systems computation, has been adapted to the needs of shared-memory multi-core and multi-processor systems.

The scope of this thesis is to enhance the existing, traditional MapReduce Architecture, by decoupling Map from Combine into two separate phases. These independent phases are interleaved and executed in parallel. We argue that, interleaving Map and Combine computation, leads to more efficient hardware utilization and competent run-time improvements.

A high-performance, shared queue data structure has been introduced in order to pipeline intermediate data from Map to Combine phase and allow for concurrent execution. Furthermore, an Inter-thread communication aware thread-to-cpu binding policy has been designed to minimize data exchange overhead.

The Pipelined Architecture is evaluated into two inherently diverse multi-core systems and demonstrates execution speedup of up to 5.34X compared to a state-of-the art MapReduce Library, Phoenix++ [1]. Nevertheless, we observe that not all type of workloads profit from our Pipelined Architecture and reason about the application characteristics that define its suitability to our Runtime.

Acknowledgements

This thesis concludes my studies at the school of Electrical and Computer Engineering of the National Technical University of Athens.

I would like to express my profound and sincere thanks to Professor Dimitrios Soudris for giving me the opportunity to carry out my diploma thesis under his supervision. His guidance and advices, alongside with his invaluable research experience provided me with the motivation and inspiration to accomplish this work.

In addition, I would like to thank Dr. Sotirios Xydis, who advised and stood by me throughout the duration of this thesis with accurate observations and suggestions. He was always willing to support me in every difficulty I faced and share his knowledge with me, so I feel grateful for having the opportunity to cooperate with him.

I give my special thanks to Kitsou Aggeliki for her enjoyable companionship and cheering me up whenever I needed it the most. I also thank all my friends who supported me throughout my studies.

Finally, I like to thank my family who always encouraged and assisted me to achieve my goals.

Contents

List of Figures

Κεφάλαιο 1

Εκτεταμένη Περίληψη

1.1 Εισαγωγή

Το MapReduce όπως προτάθηκε από τη Google [2] αποτελεί ένα συναρτησιακό προγραμματιστικό μοντέλο συνοδευόμενο από την σχετική υλοποίηση σε μορφή βιβλιοθήκης. Η πιο δημοφιλής, ανοικτού κώδικα υλοποίηση του ήρθε λίγο αργότερα από την Apache Hadoop [3]. Λόγω της ευχρηστίας του, της κλιμακωσιμότητας του, της αντοχής του σε σφάλματα και την εγγενή υποστήριξη υπολογισμών σε Κατανεμημένα Περιβάλλοντα, το MapReduce είναι πλέον άρρηκτα συνδεδεμένο με την επεξεργασία μεγάλων όγκων δεδομένων (Big Data).

Στην εργασία αυτή θα μελετήσουμε την προσαρμογή του μοντέλου MapReduce σε πολύ-επεξεργαστικά συστήματα μοιραζόμενης μνήμης [4], [5]. Σύγχρονα πολύ-επεξεργαστικά συστήματα υψηλών επιδόσεων ολοκληρώνουν εκατοντάδες [6] ή ακόμα και χιλιάδες [7] υπολογιστικούς πυρήνες σε ένα κύκλωμα. Ως αποτέλεσμα, παρουσιάζουν ομοιότητες με τις συστάδες υπολογιστών (cluster computers).

Το Phoenix [4] και το Metis [5] είναι υπάρχοντες βιβλιοθήκες που μεταφέρουν την ιδέα το MapReduce, σε συστήματα μοιραζόμενης μνήμης. Χρησιμοποιώντας τα απλοποιείται σημαντικά η διαδικασία ανάπτυξης παράλληλων εφαρμογών ενώ ταυτόχρονα, διατηρείται επαρκής κλιμακωσιμότητα και επίδοση.

1.1.1 Επισκόπηση Πρότασης

Αφού μελετήσαμε λεπτομερώς τις σύγχρονες βιβλιοθήκες MapReduce για συστήματα μοιραζόμενης μνήμης, παρατηρούμε ότι η παραδοσιακά προτεινόμενη στη βιβλιογραφία αρχιτεκτονική δεν είναι βέλτιστη. Το βήμα Combine, που εκτελείται στο τέλος κάθε Map εργασίας, μπορεί να προκαλέσει καθυστερήσεις στην εκτέλεση, ειδικά όταν η φάση Map έχει υψηλές απαιτήσεις σε επεξεργαστικούς πόρους και η φάση Combine έχει υψηλές απαιτήσεις σε προσπέλαση μνήμη.

Υποστηρίζουμε ότι, διαχωρίζοντας τη φάση Map από τη φάση Combine και επικαλύπτοντας την εκτέλεση τους μέσω μίας τεχνικής Διοχέτευσης δεδομένων,

οδηγούμαστε σε καλύτερη χρησιμοποίηση των υλικών πόρων, μεγαλύτερο βαθμό παραλληλίας και τελικά βελτιωμένο χρόνο εκτέλεσης. Για να διοχετεύσουμε τα δεδομένα μεταξύ των δύο φάσεων, εισάγουμε μία διαμοιραζόμενη ουρά υψηλής απόδοσης. Μέσω μίας βελτιστοποιημένης πολιτικής δεσίματος νημάτων σε Κεντρικές Μονάδες Επεξεργασίας (ΚΜΕ), ελαχιστοποιούμε τις επιπρόσθετες καθυστερήσεις λόγω ανταλλαγής δεδομένων. Ο ευέλικτος σχεδιασμός της ουράς και του μηχανισμού διοχέτευσης δεδομένων μας επιτρέπει εύκολη παραμετροποίηση με βάση τις ανάγκες κάθε εφαρμογής.

Τέλος, συγκρίνουμε την απόδοση της αρχιτεκτονικής μας σε σχέση με την τελευταία έκδοση του Phoenix[4], το Phoenix++[1]. Οι αξιολογήσεις μας παρουσιάζουν ευρεία διαφοροποίηση, με την μεγαλύτερη επιτάχυνση να είναι 5.7X και τη μεγαλύτερη επιβράδυνση να είναι 3.8X. Βασιζόμενοι σε μετρικές που προέρχονται από μετρητές υλικού (hardware counters) [8], [9], επιχειρούμε να χαρακτηρίσουμε την καταλληλότητα των εφαρμογών με την αρχιτεκτονική μας.

1.2 Θεωρητικό Υπόβαθρο

Σε αυτό το κεφάλαιο παραθέτουμε το θεωρητικό υπόβαθρο, πάνω στο οποίο βασίστηκε η υλοποίηση μας καθώς και μία σφαιρική επισκόπηση των μοντέρνων αρχιτεκτονικών MapReduce σε συστήματα μοιραζόμενης μνήμης.

1.2.1 Phoenix++

Το Phoenix++ [1] αποτελεί την τρίτη και τελική έκδοση της βιβλιοθήκης Phoenix. Το Phoenix++ προσφέρει μία πληθώρα από παραμετροποιήσεις, που επιτρέπει στους χρήστες να προσαρμόσουν την βιβλιοθήκη στις ανάγκες της εφαρμογής τους. Πιο συγκεκριμένα η δομή που αποθηκεύει τα ενδιάμεσα ζεύγη κλειδιού-τιμής, εκτίθεται στους χρήστες μέσω των διεπαφής των combiner και container. Επιπρόσθετα, ο προγραμματιστής μπορεί εύκολα να καθορίσει την συνάρτηση δέσμευσης μνήμης που θα χρησιμοποιηθεί καθώς και το αν τα τελικά αποτελέσματα θα είναι ταξινομημένα ή όχι. Καθώς αυτή η ευελιξία αυξάνει τις κλήσεις συναρτήσεων και δίνει λιγότερες ευκαιρίες για βελτιστοποιήσεις του compiler, το Phoenix++ είναι γραμμένο σε C++ και χρησιμοποιεί εκτενώς πρότυπα (templates) για να ενεργοποιήσει την στατική αντικατάσταση κώδικα [10].

Containers

Οι containers έρχονται να αντικαταστήσουν τη δομή των ενδιάμεσων καταχωρητών και μαζί με τους combiners προσφέρουν μία υψηλής επίδοσης δομή αποθήκευσης ζευγών κλειδιού-τιμής για κάθε είδος εφαρμογής. Οι διαθέσιμοι containers

είναι οι εξής:

hash container Πίνακας hash μεταβλητού μεγέθους χρησιμοποιείται όταν κάθε map έργο μπορεί να εκπέμψει οποιοδήποτε κλειδί.

Πίνακας container Όταν κάθε map έργο μπορεί να εκπέμψει οποιοδήποτε κλειδί σε ένα προκαθορισμένο εύρος, ένας στατικός πίνακας για κάθε νήμα (thread) χρησιμοποιείται.

Κοινός πίνακας container Στην περίπτωση που κάθε έργο εκπέμπει ένα μοναδικό κλειδί, ένας κοινός μεταξύ όλων των εργατών πίνακας χρησιμοποιείται.

Combiners

Οι Combiners είναι αντικείμενα που αποθηκεύουν όλες τις τιμές που έχουν εκπεμφθεί και σχετίζονται με το ίδιο κλειδί. Δύο διαφορετικά είδη combiners προσφέρονται:

Καταχωρητής combiner Αυτός ο combiner ουσιαστικά προσφέρει την παραδοσιακή λειτουργία του MapReduce, αποθηκεύοντας όλες τις τιμές που σχετίζονται με το ίδιο κλειδί, σε ένα πίνακα από διακοπτόμενες περιοχές μνήμης.

Συνειρμικός combiner Με αυτόν τον combiner, οι τιμές που σχετίζονται με το ίδιο κλειδί δεν αποθηκεύονται αλλά συνδυάζονται σε μία κοινή τιμή. Με τον τρόπο αυτό μειώνονται δραματικά οι ανάγκες σε μνήμη. Η συνάρτηση που χρησιμοποιείται για να πραγματοποιήσει την συνάθροιση, παρέχεται από τον χρήστη.

Υλοποιώντας τις παραπάνω βελτιστοποιήσεις, το Phoenix++ είναι κατά μέσω όρο 4.7X ταχύτερο από τον προκάτοχο του, το Phoenix Rebirth [11].

Στοχοθετημένες Εφαρμογές

Το Phoenix++, υλοποιεί ένα σύνολο από 8 εφαρμογές, τις οποίες χρησιμοποιούμε ως σημείο αναφοράς κατά την σύγκριση με την υλοποίηση μας. Αποτελούν δημοφιλείς αλγορίθμους από τους τομείς του επιχειρηματικού υπολογισμού (Word Count, Reverse Index, String Match), επιστημονικού υπολογισμού (Matrix Multiply), τεχνητής νοημοσύνης (KMeans, PCA, Linear Regression) και επεξεργασία εικόνας (Histogram).

1.2.2 Ανασκόπηση Σχετικών Υλοποιήσεων

Συνοψίζοντας τις αρχιτεκτονικές MapReduce, διαθέσιμες στη βιβλιογραφία, παρατηρούμε ότι το Phoenix αρχικά υιοθέτησε το παραδοσιακό Map, Reduce, Merge σχήμα που προτάθηκε από την Google. Το Metis [5], βασίστηκε στο Phoenix και υλοποίησε ένα υβριδικό καταχωρητή ενδιάμεσων στοιχείων που προσφέρει ικανοποιητική επίδοση σε όλους τους τύπους εφαρμογών. Το Phoenix Rebirth εισήγαγε την έννοια των combiner, οι οποία τελειοποιήθηκε από το Phoenix++. Το MRPhi [12], επιχειρεί να προσαρμόσει στις ανάγκες του Intel®Xeon Phi™ την βιβλιοθήκη MapReduce. Προκειμένου να εκμεταλλευθεί το πλεονέκτημα MIMD hyper-threading, επικαλύπτει τις φάσεις του Map και Reduce και καταγράφει μία μέση βελτίωση 8.5%. Από την άλλη, το Tiled Map-Reduce [13], ακολουθεί την γνωστή από τον κόσμο των μεταγλωττιστών τεχνική loop tiling, και χωρίζει ένα MapReduce έργο σε πολλά μικρότερα τμήματα τα οποία εκτελεί επαναληπτικά. Με τον τρόπο αυτό, μειώνονται αποτελεσματικά οι ανάγκες σε μνήμη και ενεργοποιούνται άλλες βελτιστοποιήσεις, σχετικές με την τοπικότητα των δεδομένων.

Στον κλάδο των μοιραζόμενων ουρών ταυτόχρονης πρόσβασης, ο MCRingBuffer [14] αποφεύγει το φαινόμενο false sharing, μέσω της τεχνικής padding. Επίσης, βελτιώνει περαιτέρω την επίδοση της ουράς χωρίζοντας την σε μικρότερα τμήματα και ανανεώνοντας τις τιμές ελέγχου μόνο όταν ένα ολόκληρο τμήμα έχει επεξεργασθεί. Το Fast Forward [15] ελαχιστοποιεί τις παρεμβολές μεταξύ παραγωγού και καταναλωτή, χρησιμοποιώντας ένα έξυπνο τέχνασμα, το οποίο δένει τις μεταβλητές ελέγχου με τα δεδομένα της ουράς.

1.2.3 Pipelined MapReduce

Έχοντας μελετήσει σε βάθος και κατανοήσει πλήρως τα πλεονεκτήματα, βελτιστοποιήσεις, λεπτομέρειες υλοποίησης καθώς και αναποτελεσματικότητες των πιο εξελιγμένων αρχιτεκτονικών MapReduce για συστήματα μοιραζόμενης μνήμης, παρατηρούμε ότι η κλασσική υλοποίηση είναι υπό-βέλτιστη, λόγω του χρονικού εμποδίου ανάμεσα στη φάση Map και Reduce. Προχωρώντας ένα βήμα παραπέρα, παρατηρούμε με την προσθήκης της φάσης Combine στο τέλος κάθε έργου Map, το μεγαλύτερο βάρος του Reduce έχει μεταφερθεί στο Map στάδιο.

Προτείνουμε μία καινοτόμα αρχιτεκτονική MapReduce, στην οποία το Combine είναι διαχωρισμένο από το Map και αποτελεί ξεχωριστή φάση. Εισαγάγουμε μία ουρά ταυτόχρονης πρόσβασης ώστε να μεταφέρουμε τα δεδομένα από τους Mappers στους Combiners. Αφότου ολοκληρωθεί το έργο map, τα ενδιάμεσα ζεύγη κλειδιών-τιμών, εισάγονται στην ουρά. Από εκεί εξάγονται και καταναλώνονται

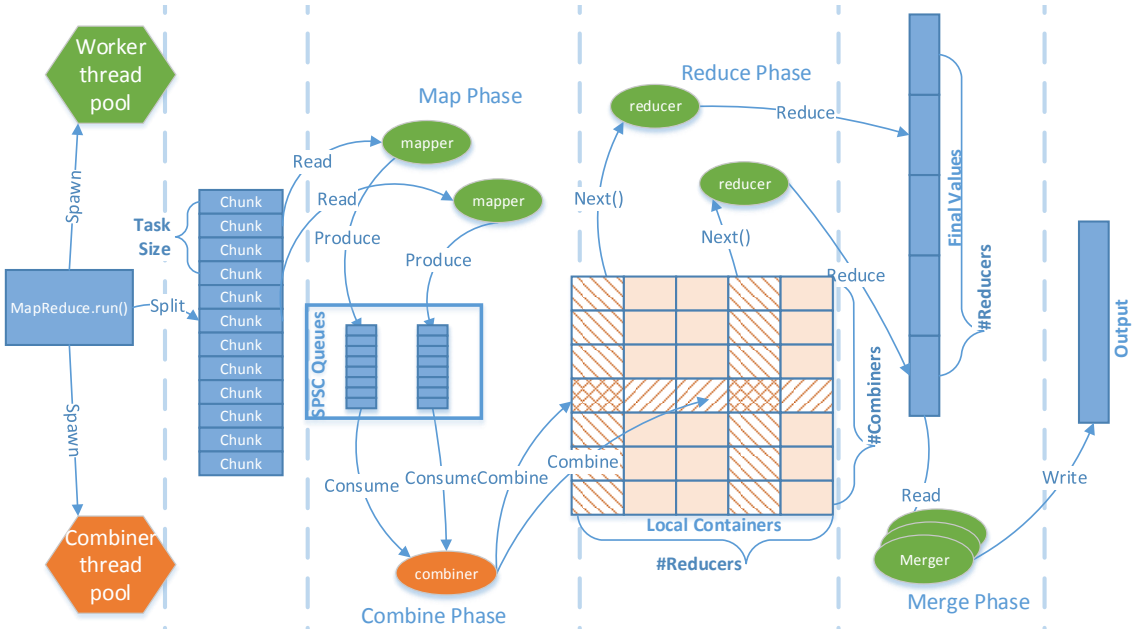
από τους Combiners. Η επικάλυψη των δύο φάσεων οδηγεί σε καλύτερη εκμετάλλευση των υλικών πόρων, μεγαλύτερο βαθμό παραλληλίας και ως αποτέλεσμα βελτιωμένο χρόνο εκτέλεσης.

1.3 Λεπτομέρειες Υλοποίησης

Σκοπός του παρόντος κεφαλαίου είναι να δώσει στον αναγνώστη μία ολοκληρωμένη άποψη της προτεινόμενης υλοποίησης μας. Αρχικά παρουσιάζουμε μία υψηλού επιπέδου περιγραφή της αρχιτεκτονικής μας. Έπειτα εξηγούμε τους λόγους που μας οδήγησαν να επιλέξουμε την συγκεκριμένη υλοποίηση για τις μοιραζόμενες ουρές. Παρουσιάζουμε την πολιτική δεσίματος των νημάτων σε ΚΜΕ που αναπτύξαμε για να ελαχιστοποιήσουμε το κόστος επικοινωνίας των νημάτων και τέλος επιχειρούμε να χαρακτηρίσουμε τις εφαρμογές με βάση την καταλληλότητα τους στην αρχιτεκτονική μας.

1.3.1 Αρχιτεκτονική Pipelined MapReduce

Σε αυτή την παράγραφο περιγράφουμε εκτεταμένα την αρχιτεκτονική του προτεινόμενου, Pipelined MapReduce. Χρησιμοποιούμε το σχήμα 1.1 ως βάση της περιγραφής μας. Κοιτάζοντας στο αριστερότερο τμήμα του σχήματος, βλέπουμε την



Σχήμα 1.1: Αρχιτεκτονική Pipelined MapReduce

κλήση της βιβλιοθήκης του MapReduce μέσω της μεθόδου `MapReduce.run()`. Αρχικά λαμβάνουν χώρα μία σειρά από αρχικοποιήσεις. Μεταβλητές όπως ο συνολικός αριθμός εργατών, το πλήθος των mappers και combiners, η πολιτική

δεσίματος νημάτων σε ΚΜΕ, το πλήθος των map/reduce έργων που θα δημιουργηθούν καθώς και ένα σύνολο παραμέτρων σχετικών με τις ουρές υπολογίζονται και αρχικοποιούνται. Ο προγραμματιστής μπορεί δυναμικά να επέμβει στο πλαίσιο και να το προσαρμόσει στις ανάγκες της εφαρμογής του. Αφού έχουν καθοριστεί όλες οι απαραίτητες μεταβλητές, δύο ομάδες από εργάτες δημιουργούνται. Στο σχήμα 1.1, η χρωματισμένη με πράσινο ομάδα, περιλαμβάνει τους εργάτες γενικού σκοπού που θα εκτελέσουν τις εργασίες του map, reduce και merge. Αντίθετα η χρωματισμένη με πορτοκαλί ομάδα χρησιμοποιείται μόνο κατά το στάδιο του combine και έχει τυπικά λιγότερα ή το πολύ ίσα νήματα με την ομάδα γενικού σκοπού.

Έπειτα ξεκινάει ο διαχωρισμός της εισόδου σε κομμάτια. Αν το αρχείο εισόδου έχει κάποια ιδιαίτερη μορφοποίηση, ο χρήστης πρέπει να παρέχει την συνάρτηση που θα διαχωρίσει την είσοδο σε κομμάτια. Το μέγεθος του έργου καθορίζει τον αριθμό κομματιών εισόδου που θα αποτελούν ένα έργο και είναι υποκείμενο δυναμικής παραμετροποίησης. Ο υπολογισμός του βέλτιστου μεγέθους δεν είναι απλό έργο, καθώς μεγάλα μεγέθη προκαλούν μη ισομερή κατανομή εργασίας μεταξύ των εργατών και πολύ μικρά μεγέθη αυξάνουν το επιπρόσθετο φόρτο εργασίας που εισάγεται από τη βιβλιοθήκη.

Οι φάσεις Map και Combine αρχίζουν αμέσως μετά. Όλα τα map έργα προστίθενται στις ουρές έργων. Οι map εργάτες, επαναληπτικά εξάγουν ένα έργο από την τοπική τους ουρά και εφαρμόζουν την δοσμένη από τον χρήστη map συνάρτηση. Ως αποτέλεσμα, παράγονται ενδιάμεσα ζεύγη κλειδιού-τιμής τα οποία, κατά την παραδοσιακή εκτέλεση MapReduce, θα γίνουν combine από ένα αντικείμενο combine και θα αποθηκευτούν στην τοπική δομή του εργάτη. Αντί αυτού, στην προτεινόμενη έκδοση μας, προκειμένου να επικαλυφτεί η εκτέλεση των map - combine συναρτήσεων, τα ενδιάμεσα δεδομένα διοχετεύονται μέσω ενός συνόλου ουρών Απλού Παραγωγού - Απλού Καταναλωτή (ΑΠΑΚ). Μόλις οι ουρές γεμίσουν, οι combine εργάτες αρχίζουν το έργο τους. Εξάγουν ομάδες από κλειδιά-τιμές, εφαρμόζουν την συνάρτηση combine και τέλος αποθηκεύουν τα αποτελέσματα στον τοπικό τους καταχωρητή. Υποστηρίζουμε ότι επικαλύπτοντας την εκτέλεση του map και combine οδηγούμαστε σε πιο αποτελεσματική εκμετάλλευση των υλικών πόρων και βελτιωμένους χρόνους εκτέλεσης.

Η φάση map θα τελειώσει όταν όλα τα στοιχεία εισόδου έχουν επεξεργαστεί. Μία λογική σημαία θα ειδοποιήσει τους combiners ότι η φάση map τελείωσε. Πριν τερματίσουν, οι combiners θα βεβαιωθούν ότι όλες οι ουρές είναι άδειες εκτελώντας έναν τελικό έλεγχο. Η υπόλοιπη εκτέλεση του MapReduce παραμένει ανεπηρέαστη. Η φάση reduce ακολουθεί, με τα έργα reduce να προστίθενται στις ουρές. Οι εργάτες, ανατίθενται και εκτελούν τα έργα αυτά, δυναμικά. Για κάθε έργο, επαναληπτικά περνούν ένα ζεύγος ενός κλειδιού και όλων των τιμών που

σχετίζονται με το κλειδί αυτό στην συνάρτηση reduce. Εκεί όλες οι τιμές συνδυάζονται σε μία μοναδική τιμή και το αποτέλεσμα αποθηκεύεται στον καταχωρητή των τελικών αποτελεσμάτων. Όταν όλα τα έργα reduce ολοκληρωθούν, ξεκινάει η φάση συγχώνευσης. Η συγχώνευση γίνεται παράλληλα, ακολουθώντας τη δομή ενός δυαδικού δέντρου. Προαιρετικά η συγχώνευση γίνεται με τρόπο ώστε η τελική έξοδος να είναι ταξινομημένη.

Το τελικό αποτέλεσμα επιστρέφεται στον χρήστη. Καθώς αποτελείται από μία λίστα ζευγών κλειδιού-τιμής, μπορεί απευθείας να τροφοδοτήσει ένα επόμενο στιγμιότυπο MapReduce.

1.3.2 Μοιραζόμενες Ουρές Ταυτόχρονης Πρόσβασης

Κεντρικό ρόλο στην απόδοση της αρχιτεκτονικής μας παίζει η δομή ουράς που χρησιμοποιείται για την επικοινωνία mappers - combiners. Στην περίπτωση μας, μόνο ένας παραγωγός εισάγει δεδομένα σε μία ουρά και μόνο ένας καταναλωτής διαβάζει δεδομένα από μία ουρά. Οπότε μία ουρά Απλού-Παραγωγού, Απλού-Καταναλωτή είναι ικανή να καλύψει τις ανάγκες της υλοποίησης μας.

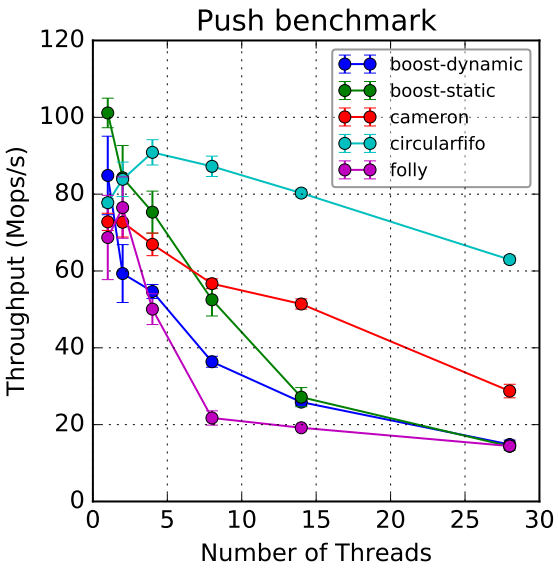
Ο L.Lamport έχει αποδείξει ότι, υπό το μοντέλο της σειριακής συνέπειας (sequential consistency), μία ΑΠΑΚ ουρά μπορεί να υλοποιηθεί χωρίς τη χρήση μηχανισμών συγχρονισμού [16], [17], [18], [19]. Με μικρές προσθήκες στην υλοποίηση του Lamport, μπορεί να αποδειχθεί ότι το ίδιο πόρισμα ισχύει και για λιγότερα ισχυρά μοντέλα συνέπειας [20].

Με βάση τα παραπάνω, καταλήγουμε ότι μία ουρά ταυτόχρονης πρόσβασης, Απλού Παραγωγού - Απλού Καταναλωτή, χωρίς κλειδώματα και αναμονή, κυκλική, στατικού μεγέθους είναι κατάλληλη για την αρχιτεκτονική μας.

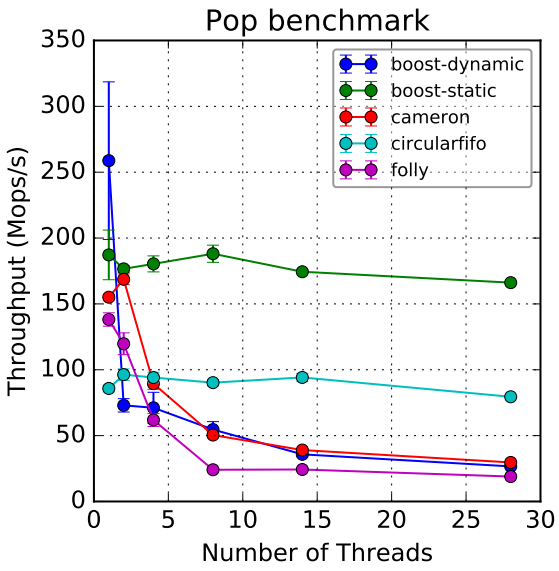
Αξιολόγηση Υλοποιήσεων Ουράς

Το πρόβλημα μας τυχαίνει να είναι ευρέως διαδεδομένο και μελετημένο, οπότε υπάρχουν αρκετές αποδοτικές λύσεις. Προκειμένου να επιλέξουμε την καλύτερη υλοποίηση, συγκεντρώσαμε ένα σύνολο από 5 διαφορετικές δομές ουράς και πραγματοποιήσαμε μία σειρά από 3 τεστ επίδοσης. Το ένα εξετάζει τον ρυθμό εισαγωγής στοιχείων σε άδεια ουρά, το δεύτερο τον ρυθμό εξαγωγής και το τελευταίο εξετάζει την απόδοση της ουράς όταν πραγματοποιούνται ταυτόχρονες εισαγωγές - εξαγωγές στοιχείων.

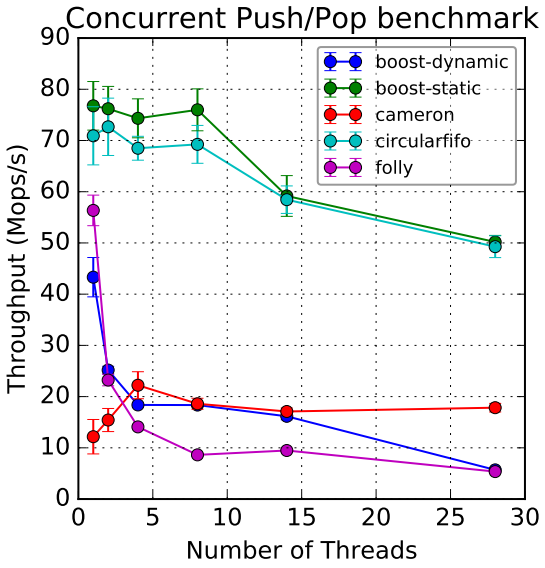
Τα γραφήματα 1.2, αφορούν το αρχιτεκτονικής Haswell πολύ-επεξεργαστικό σύστημα που έχουμε στη διάθεση μας και περιγράφουμε επαρκώς στην παράγραφο 1.4.1. Παρατηρούμε ότι συνολικά η ουρές circularfifo [21] και boost-static [22] προσφέρουν την καλύτερη επίδοση.



(α′) Τεστ εισαγωγής



(β′) Τεστ εξαγωγής

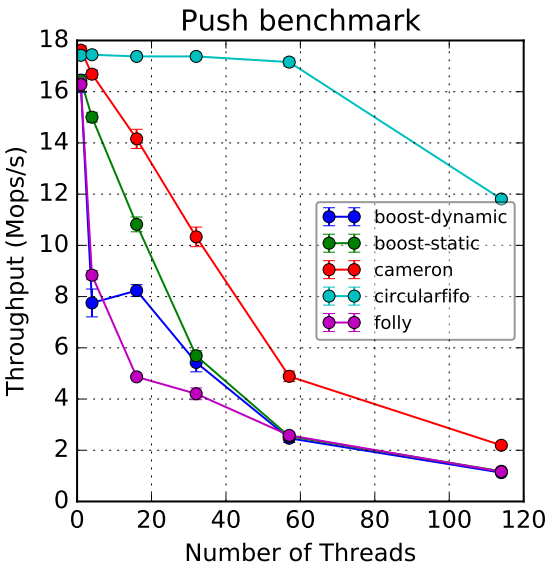


(γ′) Τεστ εισαγωγής-εξαγωγής

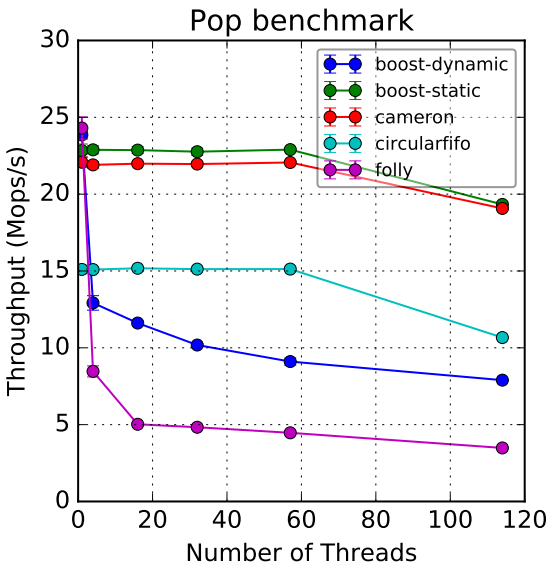
Σχήμα 1.2: Σύγκριση των διαφορετικών ΑΠΑΚ υλοποιήσεων στην πλατφόρμα Intel Haswell.

Αντίστοιχα, τα σχήματα 1.3 αφορούν την πλατφόρμα Intel® Xeon Phi™ που είχαμε στη διάθεση μας. Εδώ τα αποτελέσματα είναι ελαφρώς διαφορετικά, με τις υλοποιήσεις που ξεχωρίζουν να είναι αυτές των Cameron [23], Boost-static και Circularfifo.

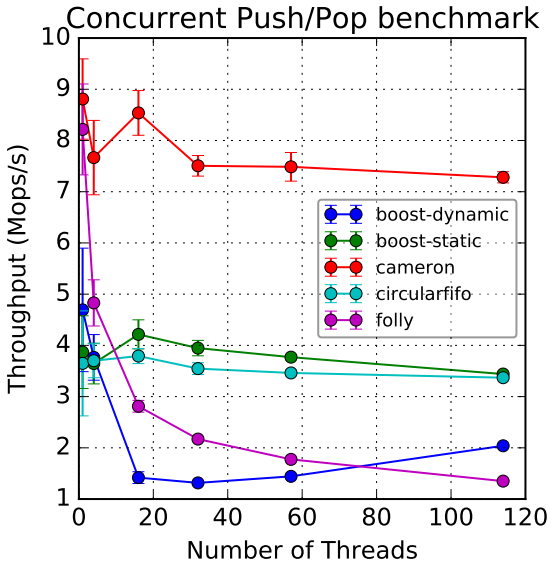
Προκειμένου να επιλέξουμε την υλοποίηση ουράς που θα χρησιμοποιήσουμε, λάβαμε υπόψιν, εκτός από τα παραπάνω τεστ, το API που προσφέρουν. Ενώ όλες σχεδόν οι υλοποιήσεις διαθέτουν ένα αρκετά απλό API, με τις κλασσικές δυνατότητες `push()`, `pop()`, `top()`, η ουρά boost προσφέρει αρκετές πρόσθετες λειτουργικότητες. Μία από αυτές, επιτρέπει στους καταναλωτές να διαβάζουν



(α′) Τεστ Εισαγωγής



(β′) Τεστ Εξαγωγής



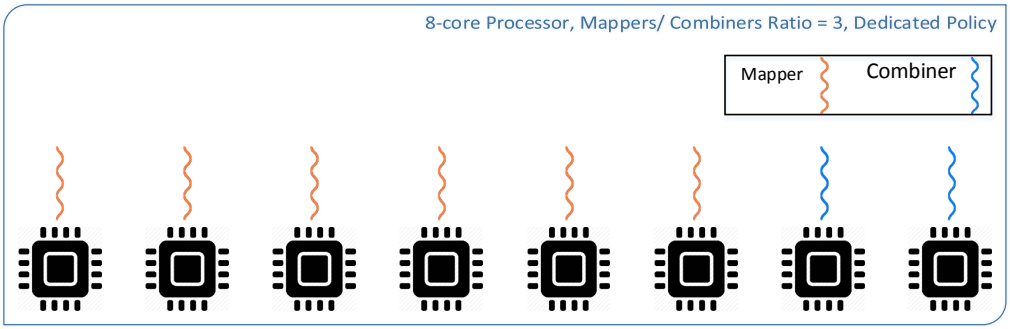
(γ′) Τεστ Εισαγωγής-Εξαγωγής

Σχήμα 1.3: Σύγκριση των διαφορετικών ΑΠΑΚ υλοποιήσεων στην πλατφόρμα Intel Xeon Phi platform.

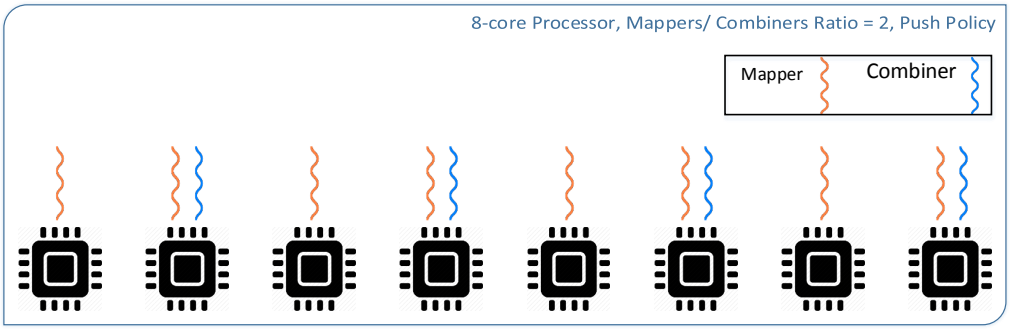
και να επεξεργάζονται στοιχεία σε ομάδες, χωρίς ανάγκη μεταφοράς δεδομένων. Για το λόγο αυτό, επιλέξαμε τελικώς την ουρά Boot με στατική δέσμευση μνήμης.

1.3.3 Μνήμη-ενήμερη Πολιτική Δεσίματος Νημάτων σε ΚΜΕ

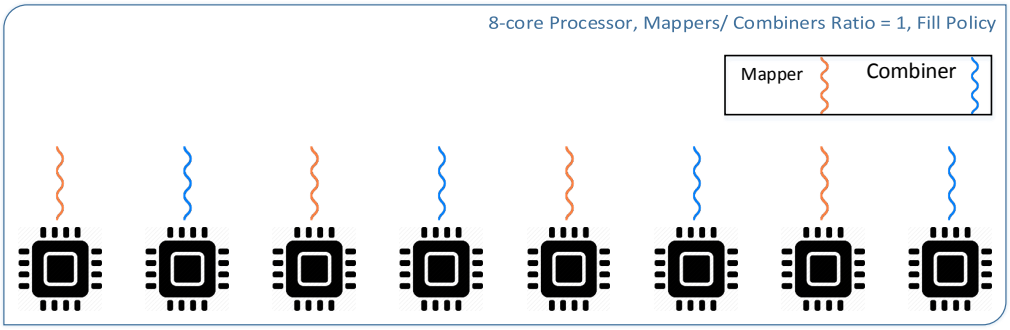
Οι ουρές που εισαγάγαμε, με σκοπό να επικαλύψουμε την εκτέλεση των mappers και των combiners, προσθέτουν πίεση στο σύστημα μνήμης. Χωρίς πρόβλεψη για μία προσεκτική σχεδίαση, το κόστος της επικοινωνίας mappers-combiners θα αντικρούσει κάθε χρονική επιτάχυνση.



(α′) Πολιτική Αφοσίωσης



(β′) Πολιτικής Πίεσης



(γ′) Πολιτική Γεμίσματος

Σχήμα 1.4: Γραφική αναπαράσταση των τριών διαφορετικών πολιτικών δεσίματος νημάτων σε ΚΜΕ.

Για να ελαχιστοποιήσουμε το κόστος της επικοινωνίας, είναι σημαντικό να τοποθετήσουμε σε κοντινούς υπολογιστικούς πυρήνες τα νήματα που συνεργάζονται. Το πραγματοποιούμε σε τρία βήματα. Αρχικά, ανάλογα με τον λόγο πλήθους mappers / combiners, αντιστοιχίζουμε στατικά έναν combiner σε ένα ή περισσότερους mappers. Αυτή η αντιστοίχιση θα παραμείνει ανέπαφη καθ όλη την εκτέλεση της εφαρμογής. Έπειτα, διαλέγουμε την πιο κατάλληλη πολιτική δεσίματος νημάτων σε ΚΜΕ, από τις διαθέσιμες πολιτικές που υλοποιήσαμε και παρουσιάζονται στο σχήμα 1.4. Τέλος, μέσω της κλήσης συστήματος `sched_setaffinity()` τα νήματα δένονται στους κατάλληλους, γειτονικούς, λογικούς υπολογιστικούς πυρήνες.

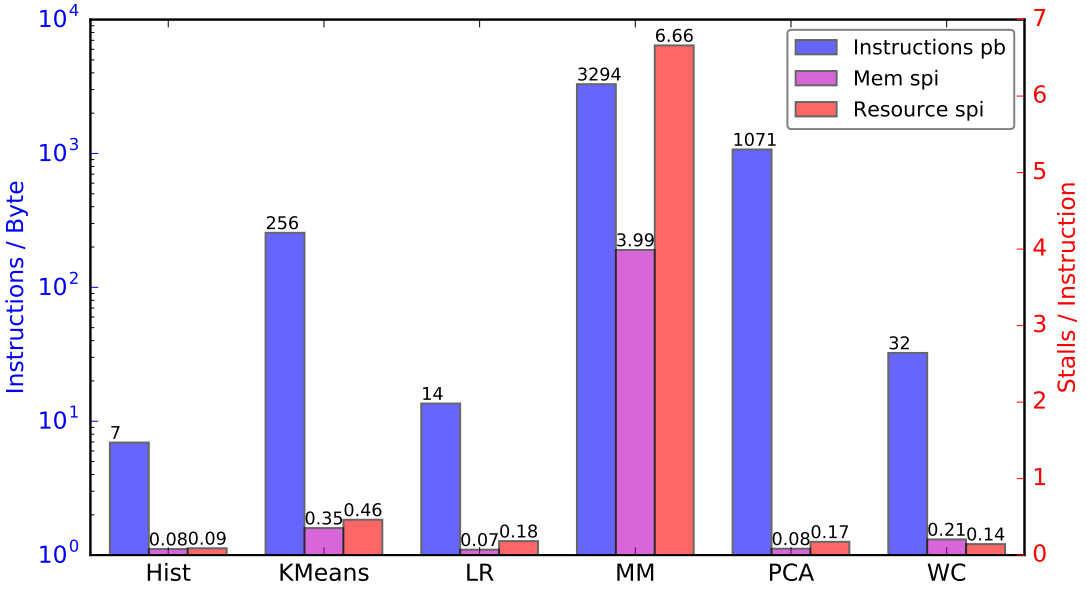
Μία σύντομη περιγραφή των διαθέσιμων πολιτικών ακολουθεί, ενώ η σύγκριση τους γίνεται στην παράγραφο 1.4.2.

Πολιτική Αφιέρωσης Αυτή είναι η αρχική και πιο απλή πολιτική. Δεν λαμβάνει υπόψη της το μοτίβο επικοινωνίας μεταξύ mappers – combiners.

Πολιτική Πίεσης Μία εναλλακτική πολιτική, που χρησιμοποιεί περισσότερα νήματα απ΄ότι λογικοί πυρήνες. Λόγω του busy wait που πραγματοποιούν οι mappers σε περίπτωση γεμάτης ουράς, η πολιτική αυτή δεν αποδίδει βέλτιστα.

Πολιτική Γεμίσματος Σκοπός της πολιτικής αυτής είναι να αντιμετωπίσει την αναποτελεσματικότητα της πολιτικής Αφιέρωσης. Το επιτυγχάνει αναμειγνύοντας mappers και αντίστοιχους combiners με τρόπο που ελαχιστοποιεί τη μεταξύ τους απόσταση.

1.3.4 Χαρακτηρισμός Εφαρμογών



Σχήμα 1.5: Εντολές ανά byte, ΄Αεργοι κύκλοι λόγω πόρων ανά εντολή και ΄Αεργοι κύκλοι λόγω μνήμης ανά εντολή στο Phoenix++ με τους προκαθορισμένους Containers.

Στην παράγραφο αυτή επιχειρούμε να εξηγήσουμε τη συμπεριφορά των διαφόρων εφαρμογών στην αρχιτεκτονική μας. Πιο συγκεκριμένα, με τη βοήθεια της πλατφόρμας VTune [24], διαβάζουμε κάποιους μετρητές υλικού και υπολογίζουμε ένα σύνολο τριών απλών μετρικών. Οι μετρικές αυτές μας βοηθούν να χαρακτηρίσουμε την καταλληλότητα μίας εφαρμογής στην αρχιτεκτονική μας. Οι μετρικές

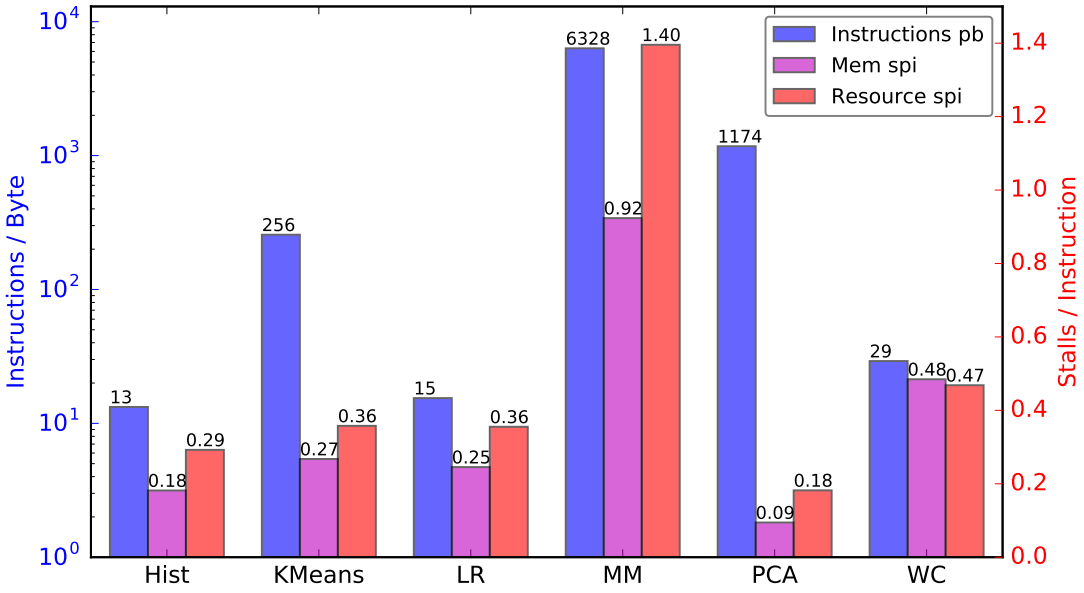
που επιλέξαμε ορίζονται ως:

$$IPB = \frac{Instructions}{Input\_Bytes} \tag{1.1}$$

$$MSPI = \frac{Memory\_Stalls}{Instructions} \tag{1.2}$$

$$RSPI = \frac{Resource\_Stalls}{Instructions} \tag{1.3}$$

Το IPB μετράει την πολυπλοκότητα μίας εφαρμογής. Εφαρμογές με πολύ χαμηλή τιμή IPB, δεν είναι καλοί υποψήφιοι για την υλοποίηση μας, καθώς το επιπρόσθετο κόστος λόγω διαχείρισης των ουρών δεν μπορεί να δικαιολογηθεί. Από την άλλη πλευρά, από μόνο του το IPB δεν μπορεί να χαρακτηρίσει την καταλληλότητα. Για αυτό εισάγουμε το MSPI και το RSPI. Οι μετρικές αυτές ποσοτικοποιούν την συχνότητα με την οποία μία εφαρμογή εκτελεί άεργους κύκλους λόγω αναμονής του συστήματος μνήμης η λόγω κάποιου μη διαθέσιμου πόρου. Βασιζόμενοι σε αυτές τις μετρικές, καταλήγουμε ότι εφαρμογές με αρκετή πολυπλοκότητα, που αντιμετωπίζουν συχνά άεργους κύκλους, ωφελούνται από την αρχιτεκτονική μας. Στο σχήμα 1.5 βλέπουμε τις προαναφερθείσες για τις διαθέσιμες εφαρμο-



Σχήμα 1.6: Εντολές ανά byte, Άεργοι κύκλοι λόγω πόρων ανά εντολή και Άεργοι κύκλοι λόγω μνήμης ανά εντολή στο Phoenix++ με Hash Containers.

γές, όταν χρησιμοποιούνται οι προκαθορισμένοι ενδιάμεσοι καταχωρητές. Για όλες τις εφαρμογές, η προκαθορισμένη δομή ενδιάμεσου καταχωρητή είναι ένας απλός πίνακας, εκτός από το Word Count, όπου χρησιμοποιείται πίνακας Hash.

Παρατηρούμε ότι ο αλγόριθμος KMeans και Matrix Multiply έχουν αρκετή πολυπλοκότητα και άεργους κύκλους, οπότε είναι ιδανικοί υποψήφιοι. Αντίθετα, οι αλγόριθμοι Histogram και Linear Regression παρουσιάζουν ένα πολύ απλό φόρτο εργασίας και δείχνουν ιδιαίτερα ακατάλληλες για την αρχιτεκτονική μας. Για τις περιπτώσεις του Word Count και PCA είναι δύσκολο να αποφανθούμε.

Προκειμένου να αυξήσουμε την πολυπλοκότητα των διαφόρων εφαρμογών, στο σχήμα 1.6 εναλλάσσουμε τον τύπο των δομών αποθήκευσης των ενδιάμεσων δεδομένων από απλό πίνακα, σε πίνακα hash. Όπως παρατηρούμε, το αποτέλεσμα είναι να αυξηθούν όλες οι μετρικές που υποδεικνύουν επιπρόσθετη πολυπλοκότητα. Με βάση τη νέα εικόνα, παρατηρούμε ότι όλοι οι αλγόριθμοι, εκτός του PCA, φαίνονται κατάλληλοι υποψήφιοι για την αρχιτεκτονική μας. Για την περίπτωση του PCA, συνεχίζουμε να δυσκολευόμαστε να αποφανθούμε, καθώς ενώ το IPB είναι αρκετά υψηλό, οι τιμές των MSPI και RSPI είναι πολύ χαμηλές.

1.4 Αποτελέσματα

Στο τμήμα αυτό αξιολογούμε την απόδοση των διαφορετικών χαρακτηριστικών της αρχιτεκτονικής μας. Στο τέλος, συγκρίνουμε την υλοποίηση μας με το Phoenix++ [1].

1.4.1 Πειραματική Πλατφόρμα

Προς ενίσχυση των αποτελεσμάτων μας, όλα τα πειράματα μας εκτελέστηκαν σε δύο πλατφόρμες. Η πρώτη είναι ένας ισχυρός πολύ-επεξεργαστής μικρο-αρχιτεκτονικής Hασωελλ, και ο δεύτερος είναι ένας συν-επεξεργαστής Intel Xeon Phi.

Intel ® Haswell

Haswell [25] είναι η εμπορική ονομασία της 4ης γενιάς μικρο-αρχιτεκτονικής της Intel. Αποτελεί τον διάδοχο της Intel Ivy Bridge [26] αρχιτεκτονικής. Διαθέτει τρία επίπεδα κρυφής μνήμης, με το μέγεθος της L1 να είναι 32KB ανά πυρήνα, της L2 256KB ανά πυρήνα και της L3 2.5MB ανά πυρήνα. Η πλατφόρμα που χρησιμοποιήσαμε για τα πειράματα μας αποτελούταν από δύο Intel Xeon E5-2683 v3 στα 2GHz με 14 πυρήνες και δύο νήματα ανά πυρήνα. Η τεχνολογία Turbo boost ήταν απενεργοποιημένη στις μετρήσεις για πιο σταθερά αποτελέσματα.

Intel ® Xeon Phi™

Ο συνεπεξεργαστής Xeon Phi είναι ένα μία νέα οικογένεια προϊόντων που ανακοινώθηκαν στο τέλος του 2012. Έχει εδραιωθεί στην κατηγορία των επιταχυντών

υλικού και συναγωνίζεται, μεταξύ άλλων, την σειρά Tesla της NVIDIA. Περιλαμβάνει μέχρι και 61 πυρήνες τύπου Intel Many Integrated Core (MIC) σε κάθε πλατφόρμα.

Ένας από το λόγους της επιτυχίας του έγκεινται στο ευέλικτο προγραμματιστικό του μοντέλο. Υποστηρίζει δύο τρόπους εκτέλεσης, την τοπική και την απομακρυσμένη. Κατά την τοπική εκτέλεση, το εκτελέσιμο μεταγλωττίζεται στον κύριο επεξεργαστή, μεταφέρεται στον συν-επεξεργαστή και εκτελείται τοπικά. Ο κώδικας σε C, C++ και Fortran δεν χρειάζεται καμία αλλαγή προκειμένου να εκτελεστεί στον Xeon Phi. Αντίθετα, στην απομακρυσμένη εκτέλεση, με χρήση κάποιων ειδικών προγραμματιστικών οδηγιών σημειώνονται οι περιοχές του κώδικα που προορίζονται για εκτέλεση στον συν-επεξεργαστή. Κατά την εκτέλεση του προγράμματος, ο κώδικας και τα αντίστοιχα δεδομένα μεταφέρονται αυτόματα στον Xeon Phi, εκτελούνται και επιστρέφουν τα αποτελέσματα στον κύριο επεξεργαστή.

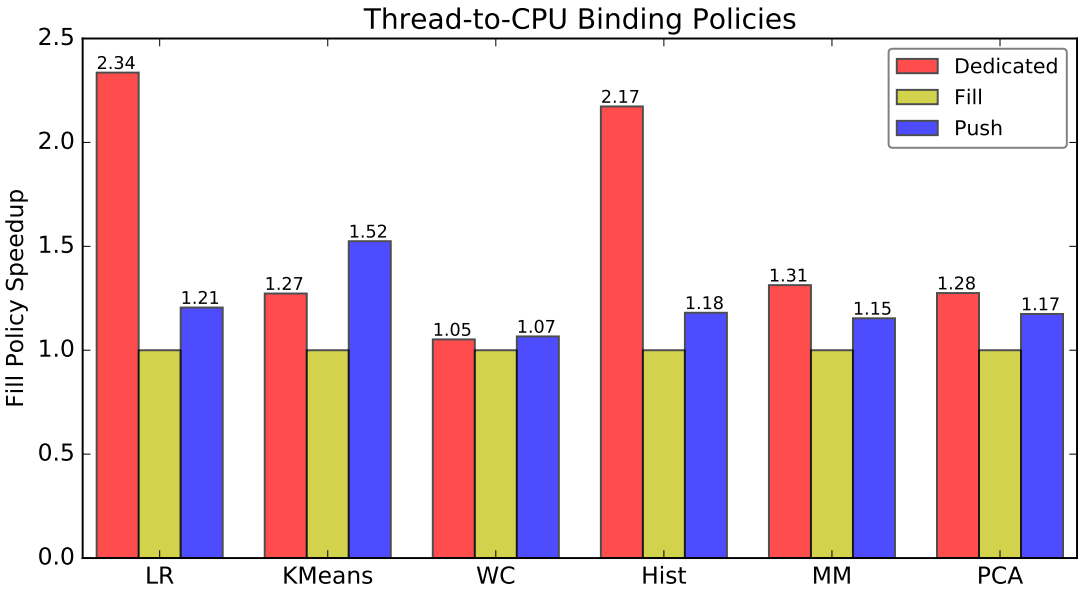
Καθένας από τους περίπου 60 πυρήνες μπορεί να εκτελεί ταυτόχρονα εντολές από 4 νήματα. Μία μονάδα διανυσμάτων (Vector Unit) περιλαμβάνεται σε κάθε πυρήνα, ενώ το μήκος των σχετικών καταχωρητών είναι 512-bits επιτρέποντας την ταυτόχρονη εκτέλεση 16 πράξεων απλής ακρίβειας ή 8 πράξεων διπλής ακρίβειας.

Δύο επίπεδα κρυφής μνήμης είναι διαθέσιμα. Το πρώτο επίπεδο έχει μέγεθος 32KB και το δεύτερο 256KB. Όλες οι κρυφές μνήμες του δευτέρου επιπέδου σχηματίζουν μία κοινή, μοιραζόμενη μνήμη καθώς είναι διασυνδεμένες μέσω ενός διαύλου διπλής κατεύθυνσης.

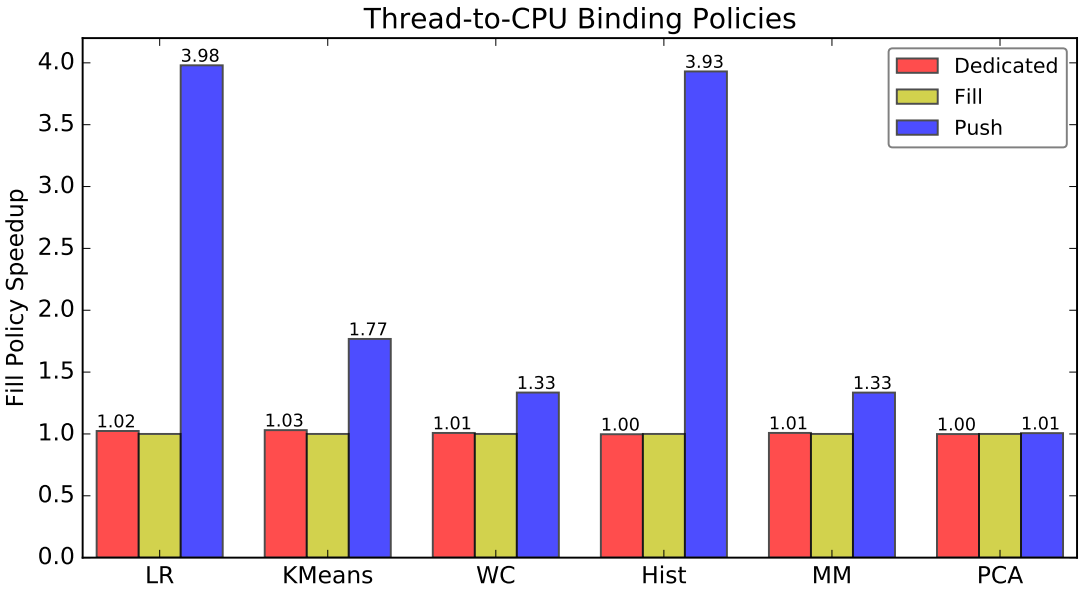
Το μοντέλο που είχαμε στη διάθεση μας διαθέτει 57 πυρήνες, συνολικά 28.5MB L2 κρυφή μνήμη και 6GB RAM. Όλες οι εφαρμογές μεταγλωττίστηκαν στον κύριο επεξεργαστή και εκτελέστηκαν τοπικά στον συν-επεξεργαστή.

1.4.2 Αξιολόγηση των Διαφόρων Πολιτικών

Κοιτάζοντας τα σχήματα 1.7 και 1.8 παρατηρούμε ότι η πολιτική Γεμίσματος υπερτερεί των άλλων δύο, όπως ούτως ή άλλως αναμέναμε. Στον Haswell η διαφορά αυτή είναι αισθητή λόγω του φαινομένου του τρόπου με τον οποίο είναι οργανωμένος ο επεξεργαστής και το γεγονός ότι αποτελεί ένα σύστημα NUMA. Κατά μέσω όρο η βελτίωση ανέρχεται στο 1.43X σε σύγκριση με την πολιτική Αφιέρωσης. Αντιθέτως, ο Xeon Phi είναι ένας απλός κόμβος, στον οποίο μάλιστα όλες οι δευτέρου επιπέδου κρυφές μνήμες είναι διασυνδεμένες, με αποτέλεσμα τα οφέλη της πολιτικής Γεμίσματος να είναι πολύ περιορισμένα, της τάξεως του 1–3%. Το συμπέρασμα είναι ότι μπορούμε με ασφάλεια να θεωρήσουμε ότι η πολιτική Γεμίσματος είναι η πιο αποτελεσματική.



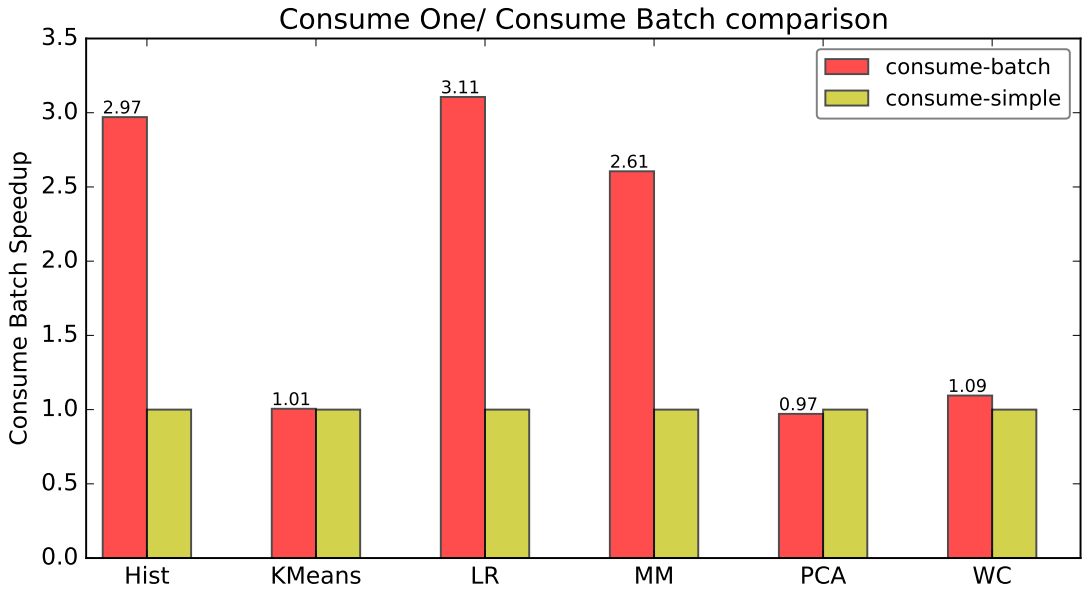
Σχήμα 1.7: Σύγκριση του χρόνος εκτέλεσης μεταξύ των διαφορετικών πολιτικών στο σύστημα Haswell.



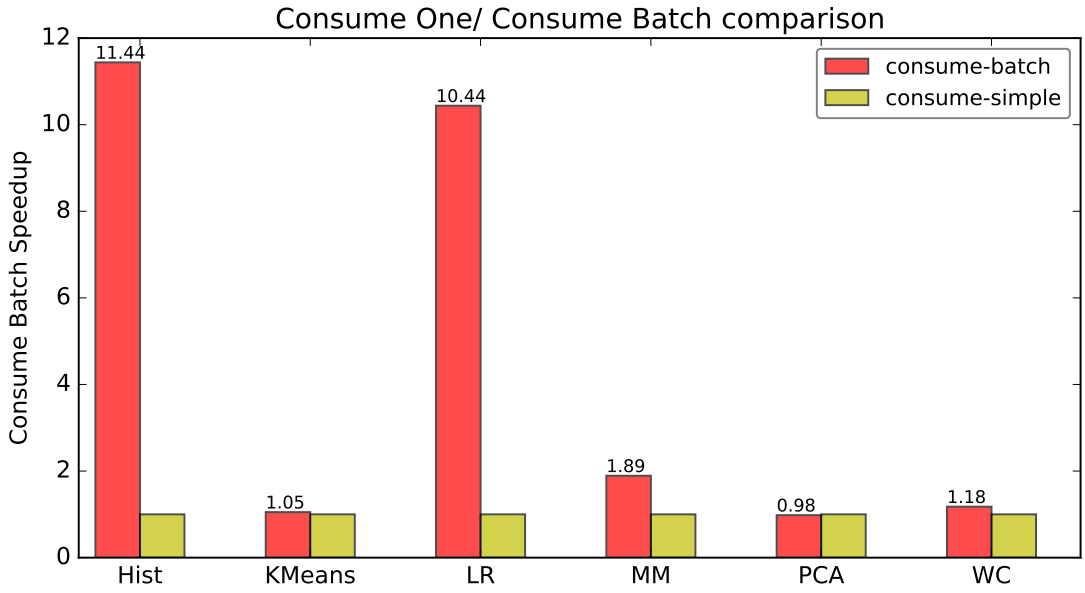
Σχήμα 1.8: Σύγκριση του χρόνος εκτέλεσης μεταξύ των διαφορετικών πολιτικών στο σύστημα Intel Xeon Phi

1.4.3 Κατανάλωση Στοιχείων σε Δέσμες

Όπως εξηγήσαμε στο κεφάλαιο 1.3.2, ένας από τους λόγους που επιλέξαμε την ουρά Boost είναι το πλούσιο API που διαθέτει. Πάνω από αυτό, υλοποιήσαμε μία μέθοδο κατανάλωσης στοιχείων σε δέσμες, χωρίς ανάγκη για μεταφορά δεδομένων. Δέχεται σαν παράμετρο μία συνάρτηση F και έναν ακέραιο `batch_size`. Αν υπάρχουν επαρκή στοιχεία στην ουρά, καταναλώνει μέσω της συνάρτησης F `batch_size` από αυτά. Με τον τρόπο αυτό εκμεταλλευόμαστε την τοπικότητα των δεδομένων.



Σχήμα 1.9: Σύγκριση χρόνου εκτέλεσης με χρήση απλών αναγνώσεων και αναγνώσεων σε δέσμες, στον Haswell.



Σχήμα 1.10: Σύγκριση χρόνου εκτέλεσης με χρήση απλών αναγνώσεων και αναγνώσεων σε δέσμες, στον Intel Xeon Phi

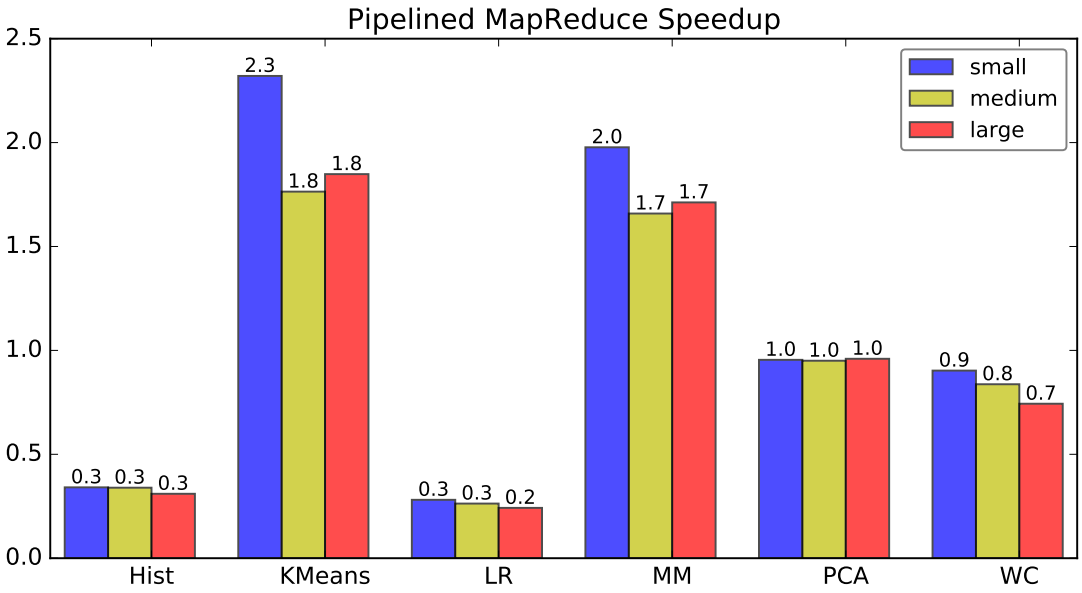
Στα σχήματα 1.9 και 1.10 παρουσιάζονται τα αποτελέσματα χρήσης της τεχνικής αυτής. Στην πράξη αποδείχθηκε ότι χωρίζοντας την κάθε ουρά σε περίπου 20−50 δέσμες επιτυγχάνουμε την μέγιστη απόδοση.

1.4.4 Αξιολόγηση Επίδοσης

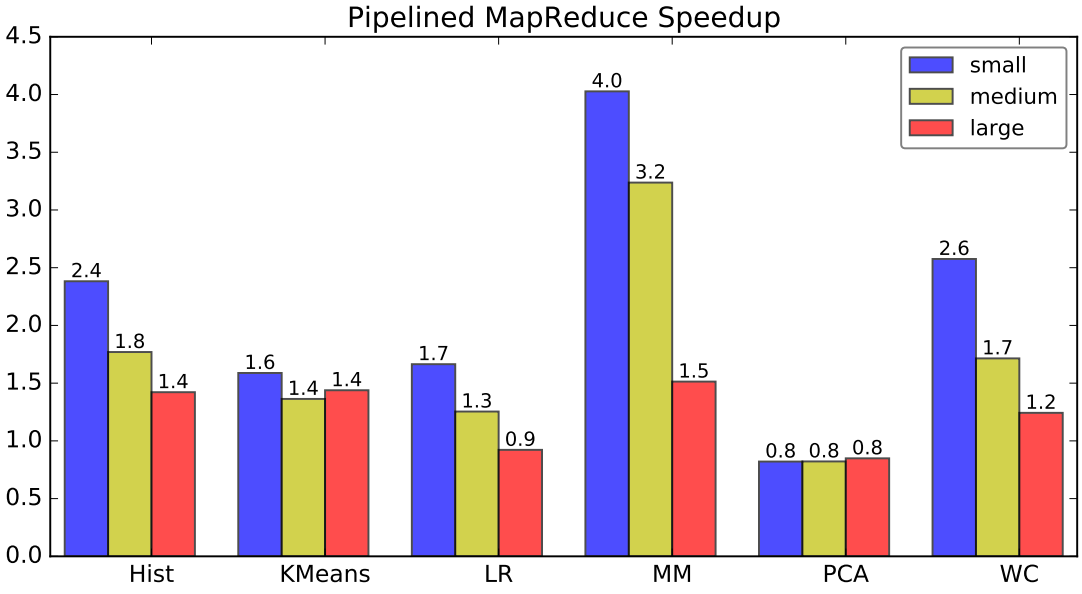
Στην παράγραφο αυτή συγκρίνουμε τον χρόνο εκτέλεσης της αρχιτεκτονικής μας με αυτόν του Phoenix++. Όλα τα πειράματα εκτελέστηκαν και στις δύο διαθέσιμες

πλατφόρμες, ενώ για καλύτερη ακρίβεια πάρθηκες ο μέσος όρος από 20 εκτελέσεις. Η τυπική απόκλιση δεν ξεπέρασε το 1%.

Πλατφόρμα Intel Haswell



Σχήμα 1.11: Επιτάχυνση του Pipelined Phoenix σε σχέση με το Phoenix++ με χρήση προκαθορισμένων Containers, στον Haswell



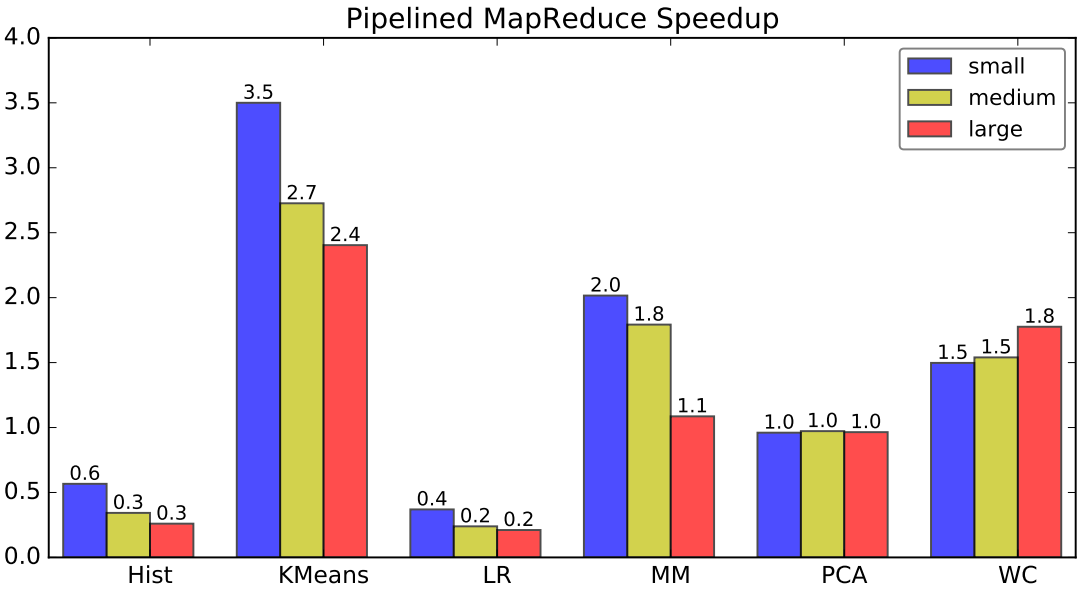
Σχήμα 1.12: Επιτάχυνση του Pipelined Phoenix σε σχέση με το Phoenix++ με χρήση Hash Containers, στον Haswell

Στο σχήμα 1.11 βλέπουμε την επιτάχυνση σε σύγκριση με το Phoenix++, για τρία διαφορετικά μεγέθη εισόδου όταν χρησιμοποιήθηκαν οι προκαθορισμένες

δομές αποθήκευσης ενδιάμεσων δεδομένων. Για όλες τις εφαρμογές η δομή ενδιάμεσων δεδομένων είναι ένας απλός πίνακας, εκτός από το Word Count το οποίο χρησιμοποιεί πίνακα hash. Βλέπουμε ότι το KMeans και το Matrix Multiply επωφελήθηκαν από την αρχιτεκτονική μας κατά 1.95X και 1.77 αντίστοιχα. Το PCA αποδίδει παρόμοια και το το Word Count είναι ελαφρώς χειρότερο κατά 21.6%. Το Histogram και το Linear Regression δεν επωφελούνται από την αρχιτεκτονική μας και είναι κατά 3X και 3.8X φορές αργότερα. Αυτό συμβαίνει λόγω της πολύ χαμηλής τους πολυπλοκότητας, με αποτέλεσμα το επιπρόσθετο κόστος της ουράς να μονοπωλεί τον χρόνο εκτέλεσης.

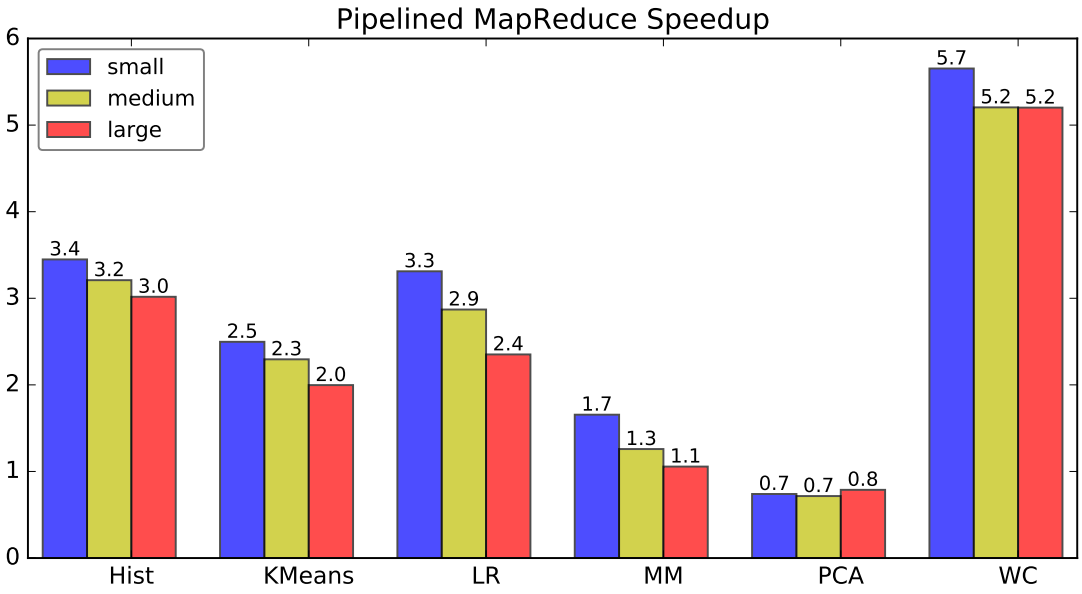
Προκείμενου να αυξήσουμε την πολυπλοκότητα των εφαρμογών, στο σχήμα 1.12 εναλλάσσουμε τις δομές ενδιάμεσων δεδομένων σε πίνακες hash σταθερού ή μεταβλητού μεγέθους. Εδώ βλέπουμε μία άκρως βελτιωμένη εικόνα, με το Pipelined MapReduce να είναι ταχύτερο σε 5 από τις 6 περιπτώσεις και ελαφρώς χειρότερο σε μία. Το συμπέρασμα είναι ότι η αρχιτεκτονική μας είναι καταλληλότερη σε πιο πολύπλοκους τύπους αλγορίθμων.

Συν-επεξεργαστής Intel® Xeon Phi™



Σχήμα 1.13: Επιτάχυνση του Pipelined Phoenix σε σχέση με το Phoenix++ με χρήση προκαθορισμένων Containers, στον Intel Xeon Phi

Τα αντίστοιχα δεδομένα για τον Xeon Phi παρουσιάζονται στις εικόνες 1.13 και 1.14. Στην πρώτη χρησιμοποιήθηκαν οι προκαθορισμένοι καταχωρητές ενδιάμεσων στοιχείων και τα αποτελέσματα είναι ελαφρώς καλύτερα, καθώς και το Word Count επιτυγχάνει καλύτερη συμπεριφορά. Η κατάσταση είναι παρόμοια στις υπόλοιπες εφαρμογές.



Σχήμα 1.14: Επιτάχυνση του Pipelined Phoenix σε σχέση με το Phoenix++ με χρήση Hash Containers, στον Intel Xeon Phi

Περνώντας στη χρήση hash καταχωρητών ενδιάμεσων στοιχείων, για άλλη μία φορά παρατηρούμε σημαντική καλυτέρευση καθώς 5 από τις 6 εφαρμογές είναι βελτιωμένες, με μόνο το PCA να είναι κατά 34% αργότερο. Το συμπέρασμα μας σχετικά με τους τύπων των εφαρμογών που είναι πιο κατάλληλοι για την αρχιτεκτονική μας επιβεβαιώνεται και για την περίπτωση του συν-επεξεργαστή Xeon Phi.

1.5 Συμπεράσματα και Μελλοντικές Επεκτάσεις

1.5.1 Συμπεράσματα

Η εργασία αυτή παρουσιάζει το Pipelined MapReduce, μία εναλλακτική υλοποίηση του συμβατικού MapReduce για συστήματα μοιραζόμενης μνήμης. Το Pipelined MapReduce αποσυνδέει το Map και το Combine σε δύο ξεχωριστές φάσεις. Καθώς δεδομένα πρέπει να μεταφερθούν μεταξύ των δύο φάσεων, εισάγουμε μία μοιραζόμενη ουρά υψηλής επίδοσης, η οποία επιτρέπει στους Mappers και τους Combiners να λειτουργούν ταυτόχρονα. Μετά από προσεκτικές μετρήσεις επιλέξαμε την υλοποίηση του boost [22] ως καταλληλότερη για τις ανάγκες μας. Χτίζοντας πάνω στο ιδιαίτερα πλούσιο API που προσφέρει, προσθέσαμε λειτουργία κατανάλωσης στοιχείων σε ομάδες. Εκτός από την υλοποίηση της ουράς, κεντρικό ρόλο στην απόδοση της αρχιτεκτονικής μας κατέχει και η επικοινωνία των νημάτων. Για να ελαχιστοποιήσουμε το κόστος της, σχεδιάσαμε μία πολιτική δεσίματος νήματος σε ΚΜΕ σύμφωνα με το μοτίβο επικοινωνίας.

Χρησιμοποιήσαμε το Phoenix++ [1], ένα μοντέρνο πλαίσιο MapReduce, ως βάση για την υλοποίηση μας. Διατηρήσαμε ανέπαφο το αρχικό API και όλες οι επεμβάσεις μας αφορούν το κομμάτι της βιβλιοθήκης και είναι εντελώς διαφανής στον χρήστη.

Τέλος, αξιολογήσαμε την επίδοση του Pipelined MapReduce απέναντι στο Phoenix++ σε δύο πλατφόρμες. Η πρώτη αποτελείται από δύο κόμβους Intel Xeon 4ης γενιάς με 14 πυρήνες - 28 νήματα ανά κόμβο ενώ η δεύτερη αποτελείται από έναν συν-επεξεργαστή Intel Xeon Phi με 57 πυρήνες - 228 νήματα. Τα αποτελέσματα παρουσιάζουν μία ιδιαίτερη ποικιλομορφία με το καλύτερη επιτάχυνση να ανέρχεται σε 5.7X και τη χειρότερη επιβράδυνση σε 3.8X. Παρόλα αυτά, τα αποτελέσματα δείχνουν ότι η αρχιτεκτονική μας είναι πιο αποδοτική σε πολύπλοκες εφαρμογές.

Το Pipelined MapReduce κατόρθωσε να εκμεταλλευτεί την αναποτελεσματικότητα που προέρχεται από την σειριοποίηση των Map και Combine έργων. Βασιζόμενοι σε τρεις μετρικές, παρατηρήσαμε ότι οι εφαρμογές με επαρκή πολυπλοκότητα, που αντιμετωπίζουν συχνούς άεργους κύκλους είναι ιδανικοί υποψήφιοι για να επωφεληθούν από την αρχιτεκτονική μας.

1.5.2 Μελλοντικές Επεκτάσεις

Αποδείξαμε ότι υπάρχει ακόμα χώρος για βελτίωση στις υλοποιήσεις MapReduce για συστήματα μοιραζόμενης μνήμης. Σε αυτή την παράγραφο παραθέτουμε κάποιες ιδέες για μελλοντικές επεκτάσεις που μπορούν να βελτιώσουν την υλοποίηση μας ακόμα περισσότερο.

- Για την επίτευξη της καλύτερης επίδοσης, ο χρήστης πρέπει να παραμετροποιήσει σωστά την βιβλιοθήκη. Λόγω της περίπλοκης εξάρτησης μεταξύ των διαφόρων παραμέτρων, αυτό δεν είναι ένα απλό έργο. Η προσθήκη λειτουργίας αυτό-παραμετροποίησης θα είναι ιδιαίτερα χρήσιμη για τους χρήστες.

- Προς το παρόν, η αρχιτεκτονική μας λειτουργεί μόνο με ίσους ή περισσότερους Mappers από Combiners. Σε περιπτώσεις που το έργο του combine είναι βαρύτερο από αυτό του map, θα πρέπει να υπάρχουν περισσότεροι Combiners. Πρέπει να σημειωθεί ότι αυτή είναι μία σπάνια περίπτωση. Στην ακόμα γενικότερη περίπτωση, με την προσθήκη ουρών πολλαπλών παραγωγών - πολλαπλών καταναλωτών, μπορεί να υποστηριχθεί εντελώς αυθαίρετη αντιστοιχία mappers - combiners. Ιδιαίτερη προσοχή πρέπει να δοθεί στο γεγονός ότι το κόστος χρήσης αυτών των ουρών είναι αρκετά αυξημένο σε σχέση με τις ουρές απλού παραγωγού - απλού καταναλωτή.

- Προκειμένου να υποστηρίξουμε την πολιτική δεσίματος νημάτων σε ΚΜΕ, που είναι ενήμερη του μοτίβου επικοινωνίας, έχουμε αποφασίσει να αναθέτουμε στατικά τους combiners με τους mappers. Η επιλογή αυτή υστερεί ως προς την ισοστάθμιση του φόρτου εργασίας. Μία πιο δυναμική στρατηγική μπορεί να ακολουθηθεί. Στην προσέγγιση αυτή, κάθε mapper γράφει τα ενδιάμεσα ζεύγη σε έναν απλό προσωπικό καταχωρητή. Κάθε φορά που αυτός ο καταχωρητής γεμίζει, θα δημιουργείται ένα έργο combine και θα προστίθεται σε μία υψηλής προτεραιότητας ουρά με έργα. Ο πρώτος διαθέσιμος εργάτης θα εκτελέσει το έργο αυτό και έπειτα θα συνεχίσει να εκτελεί τα map έργα. Με τον τρόπο αυτό οι combiners λειτουργούν δυναμικά και δεν χρειάζεται να καθορίσουμε ένα στατικό αριθμό εξαρχής. Το μειονέκτημα της υλοποίησης αυτής είναι η έλλειψη τοπικότητας δεδομένων σε σχέση με την παρούσα υλοποίηση.

Chapter 2

Introduction

2.1 The Big Data Era

Since the dawn of the Internet, high-bandwidth connections, smart phones, social media and other technologies, people are constantly online, sharing their data with the rest of the world, producing new data in form of web-pages, e-mails, pictures, e-books and hundreds of other digital formats. Recent statistics, show that ~2.5 Quintillion bytes or ~2.17 Exabytes are produced daily [27].

All this data has to be stored, and local disks of Personal Computers (PC) are not the choice. The data is stored in remote locations, with large numbers of interconnected servers, called Data Centers, Clusters or Server Farms. These servers provide important services, including data storage, management, recovery, processing, distribution, statistics and many more.

Big Data, is a term that has emerged to characterize extreme volumes of data, with a wide variety of data types, sometimes unstructured, that must be analyzed periodically or in real time. We must clear, that Big Data does not equate to a specific data size, however, the term is often used to describe datasets in the order of terabytes, petabytes or even exabytes.

Unsurprisingly, processing such extreme datasets requires non-conventional programming frameworks. Some of their vital characteristics are:

Distributed Computation Modern, advanced multi-cores are pushing the physical limits set by energy and power consumption. The energy used by a circuit is not shrinking proportionally to its physical dimensions. Lowering the chip's operating voltage lowers the power proportionally but transistors cannot operate below a 200 milli-Volt level. Increasing the clock frequency is also not a solution as, even if signals were transferred at the speed of light, chips clocked above 5GHz would not be able to transmit information fast enough. Extending more circuitry into the third dimension can help, but only a bit. Therefore, multi-processor chips can no longer scale with size of data. The only way to provide the sufficient throughput is by exploiting distributed

computing. Multiple distributed machines across one or more data centers have to co-operate harmoniously to satisfy the computational needs.

Scalability The framework must be able to manage the code execution and the available resources efficiently and ideally maintain a linear performance speed-up with the addition of extra computing resources.

Fault tollerance Supposing that thousands of machines will be combined to process huge datasets in a reasonable amount of time, failures of all types can, and will, occur. The framework must be in position to withstand hardware, software and network errors, and recover expeditiously.

Availability One of the characteristics of Big Data is the constant flow rate. As a result, non-stop processing and high service availability are essential. Software and hardware replication schemes are inevitable.

Simplicity The developer must be able to code using a simple model, independent from data size and framework complexity. Simplicity is always the key.

MapReduce is a functional programming model, accompanied with a corresponding implementation, originally proposed by Google [2] and then open-sourced by Apache Hadoop [3]. The fact that MapReduce combines all the above mentioned characteristics has coupled it with Big Data processing. In this thesis, we discuss the implementation of MapReduce on shared-memory multi-core systems and contribute to its improvement.

2.2 MapReduce with Shared-Memory Properties

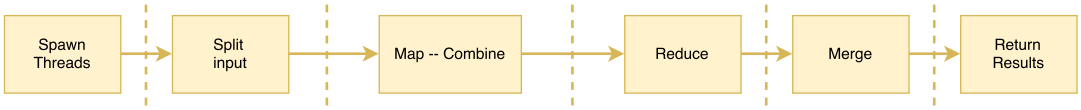
As explained in section 2.1, MapReduce has been designed to facilitate large scale computation on clusters. Modern high performance processors and accelerators, like Intel's recently released Xeon Phi product family [6] and Nvidia's GPGPU cards [7], integrate hundreds to thousands of cores in a single chip, thus they exhibit similarities with cluster systems.

The main advantage of single chip multi-processors compared to distributed servers, is the shared-memory hierarchy between the cores. Therefore, expensive message passing communication over the network is not required. Instead, concurrent data structures and proper synchronization are more crucial to obtain maximum performance out of multi-core and multi-processor platforms.

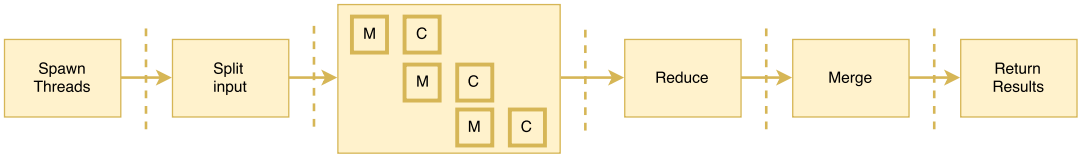
Phoenix [4] and Metis [5], are frameworks that adjust MapReduce to the needs of shared-memory systems. By using them the development process of parallel programs gets notably simplified and, at the same time, competent performance

and scalability are maintained. We discuss the characteristics of Phoenix and Metis in sections 3.3.2 and 3.3.3.

2.3 Proposal Overview



(A) Conventional Work-flow



(B) Decoupled Work-flow

FIGURE 2.1: Traditional MapReduce Work-flow Inefficiency. Our architecture is taking advantage of pipeline parallelism by decoupling map and combine phases.

Having studied and deepened our knowledge of state-of-the-art shared-memory MapReduce libraries, we identify that the traditional Map, Reduce, Merge architecture is sub-optimal. The combine step, coming after every map operation, can slow-down the execution, especially when the map phase is mainly compute-intensive and the combine phase is memory-intensive.

We argue that, *decoupling map and combine into separate phases which operate concurrently in pipelined fashion, leads to a higher degree of parallelism, more effective hardware utilization and, as a consequence, improved run time.* In order to pipeline map and combine phases, we introduce a low-overhead, concurrent, lock-free, fixed-size, circular buffer shared among map and combine workers. The inter-thread communication pattern, between mappers and corresponding combiners, has been taken into consideration, to devise a thread-to-CPU binding policy that minimizes the overhead of data exchange.

The shared-queue has been designed in a flexible way to allow for proper customization depending on specific application workloads. We benchmark the effects of different queue configuration parameters in the overall performance of our framework. Finally, we evaluate, into two diverse multi-core systems, our framework, against Phoenix++ [1] MapReduce library. Our benchmarks indicate that applications with significant amount of work, or complementary workload types during map/combine phases, are favored by our pipelined architecture and demonstrate an up-to 5.34X execution time speed-up. Nevertheless, we underline that our

implementation is not suitable for every type of workload. We identify the decisive application characteristics that affect its suitability to our proposed, pipelined architecture.

2.4 Thesis structure

The rest chapters of thesis are organized as follows:

- Chapter 3 reviews related work and justifies the use of MapReduce in shared memory multi-core systems.
- In Chapter 4 we present the technical details of our proposed decoupled MapReduce architecture.
- We evaluate the performance of our proposed implementation and provide the results discussion in Chapter 5.
- We conclude this thesis and provide a brief discussion on future work in Chapter 6.

Chapter 3

Prior Art

3.1 Introduction

In this chapter, we briefly overview the state-of-the-art on topics essential to this thesis. We begin with a presentation of MapReduce as proposed by Google [2]. We then move on to Phoenix [4], a MapReduce runtime for shared-memory multiprocessors, and its revisions [11], [1]. Afterwards, we outline a tiled implementation of MapReduce [13] and a customization of Phoenix for the Intel® Xeon Phi™ Co-processor [12]. Finally we present Single Producer Single Consumer (SPSC), wait-free, concurrent buffer implementations and conclude with our proposal of Pipelined MapReduce.

3.2 MapReduce Runtime

MapReduce, as originally proposed by Google [2], composes a functional programming model and a corresponding implementation, applicable to large dataset processing on cluster environments. In most of the cases the user needs to provide nothing more than his code, expressed as a Map and a Reduce function. The Map function will be applied to all input data and will generate one or more intermediate key-value pairs. Then, the Reduce function will aggregate all the values associated with the same key. Finally, the aggregated results will be merged, sorted and given as output to the user or as input to another MapReduce task for further processing.

MapReduce facilitates the development of parallel code as it manages subtly, in a transparent to the programmer way, the tedious tasks of data partitioning, dynamic job scheduling, parallelization, error recovery and inter-machine communication. This abstraction frees the programmer from the overwhelming complexity of writing code for parallel distributed systems and substantially accelerates the code development and debugging process. Apart from its usability, MapReduce is known for its ability to endure hardware and software failures with minimal performance overheads and its ability to scale up to thousands of distributed machines.

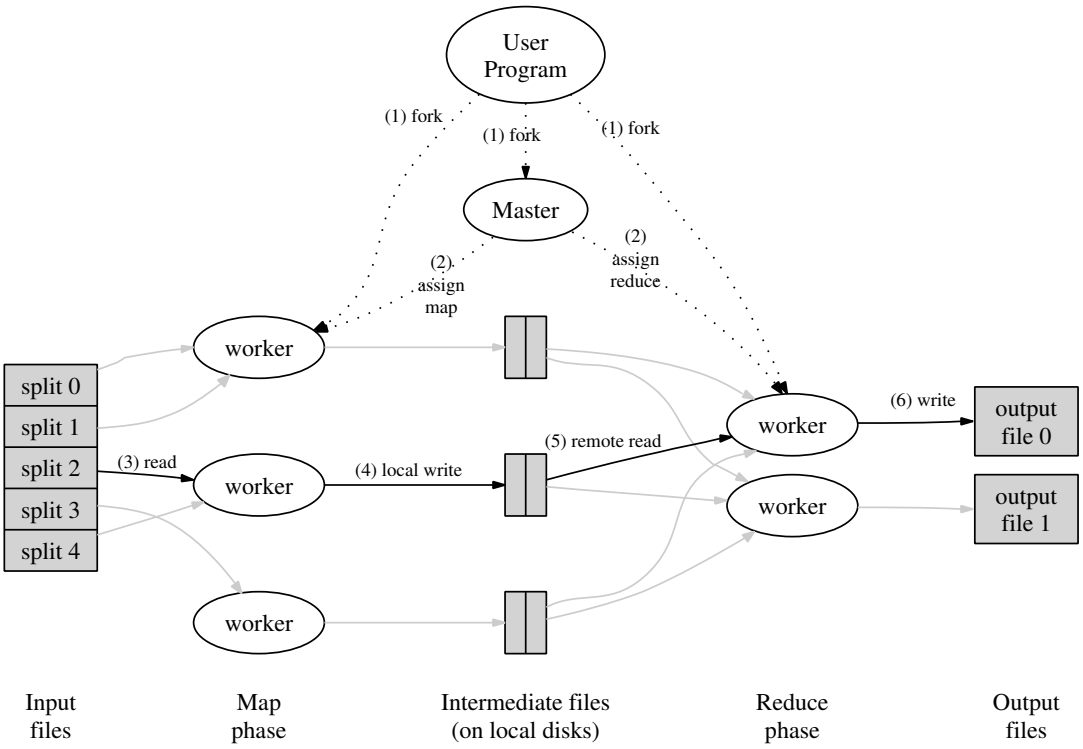


FIGURE 3.1: Google's MapReduce [2] execution overview

3.2.1 MapReduce Architecture

Figure 3.1 shows the execution overview of a MapReduce task. The numbered labels specify the sequence of execution and are briefly described below:

1. At first, the input is split into M equal sized chunks of key-value input pairs by the MapReduce library. Each of these chunks will later correspond to a Map task. Then, the library spawns copies of the program to the available machines.

2. One of these copies plays the role of the Master and is responsible for the MapReduce instance instrumentation. Master assigns tasks to available, idle machines, which are called workers.

3. As map phase begins, each map worker reads an input chunk and passes its key-value pairs to the map function. The intermediate key-value pairs, generated by the map function, are stored in the intermediate buffers.

4. Periodically, a partition function partitions the intermediate buffers on R regions, equal to the number of reduce tasks, and dumps them on local disks. Master is informed about the location of these partitions and will later, communicate them to the reduce workers.

5. After the end of the map phase, the reduce phase starts. Reduce workers, using remote procedure calls, read the intermediate buffers from the local disks of map workers. When a reduce worker has gathered all the necessary key-values pairs, it sorts them by key so that values having the same key are grouped together. Note that the sorting is required, as in general, multiple keys are associated with the same reduce task.

6. Afterwards, the reduce worker passes the keys with their corresponding set of values to the user defined reduce function which aggregates them. The result is appended to the final output and written to a file.

7. At the end of the reduce phase, the control returns back to the user code.

Upon a successful run of a MapReduce instance, the output of each reduce worker will be available in a separate file. The user can merge them together or process them further via a new MapReduce instance.

3.2.2 The "Hello World" of MapReduce

Word count is a simple problem, in which we are asked to count the occurrences of each word in a set of documents. Due to the simplicity of the MapReduce style solution to this problem, word count is considered as a perfect example to demonstrate the MapReduce programming model. In listing 3.1 we can see the pseudo-code of map and reduce functions.

As we can see the map function tokenizes the document text into words, and for each word emits to the intermediate buffer a key-value pair using the word as a key and the occurrence counter, which is always one, as the value. The reduce function sums up all counters associated with the same word and emits to the total number of occurrences.

3.2.3 Success and Extensive Applicability

Since Google's MapReduce was implemented it has enjoyed much attention and appreciation among the computing industry. Hadoop [3] is the most popular open-source implementation of the MapReduce runtime in Java. MapReduce has been tested in a wide range of domains, including machine learning, scientific simulations, bioinformatics, image processing, word processing and a lot more.

```
1  void map(string key, string value){
2      // key: document name
3      // value: the text of the document
4      for(each word in value){
5          emit_intermediate(word, 1);
6      }
7  }
8
9  void reduce(string key, int values[]){
10     // key: a word
11     // values[]: an array of numbers
12     int result = 0;
13     for(each value in values){
14         result += value;
15     }
16     emit_out(result);
17 }
```

Listing 3.1: Map and Reduce functions for the Word Count application

3.3 Phoenix: MapReduce runtime for multi-cores and multi-processor systems

Over the last years, shared memory multi-cores and multi-processor systems have become increasingly popular. Quad-core or eight-core chips are typical on modern processors, while machines with tens or hundreds of cores on a single chip have been commercialized. Given this trend, it is exceedingly important to implement efficient, easy to use frameworks in order to harness the computational power of such systems.

3.3.1 Modern Parallel Programming Models

As a result of the multi-processors trend, a significant amount of research has been conducted and novel programming models have emerged. A narrow subset of them is listed below:

MPI Message Passing Interface [28] is the implementation of a communication protocol used in parallel computing without the use of shared memory primitives. Practical collective operations such as scatter, gather and reduce simplify the implementation of parallel code.

OpenMP Open Multi-Processing [29] is a directive based API, applicable in C, C++ and Fortran that supports shared memory multi-processing. OpenMP provides a higher interface to the fork-join idiom and due to its simplicity it is one of the most widely used and mature parallel programming models.

CilkPlus CilkPlus [30], [31], [32] is a programming language, based on C++, with extended capabilities to construct parallel loops and write task parallel programs. CilkPlus demonstrates an interface similar to that of OpenMP combined with an advanced work-stealing scheduler.

TBB Thread building blocks [33] is a C++ library, developed by Intel, for parallel programming. The TBB library decomposes the program into multiple tasks, which are then parallelized with respect to their inter-dependencies. In a similar fashion to STL [34], TBB implements a variety of algorithms, data structures and other constructs that promote parallelism in a native way.

OpenCL Open Computing Language [35] is a programming language based on C, that provides an interface to task and data based parallelism, but also manages the execution of the code across heterogeneous devices such as CPUs, GPUs and FPGAs. Programs written in OpenCL are mean to be compiled at run time so that they are portable across different systems.

Despite the generous efforts of the above mentioned programming models, coding efficient parallel programs still remains a cumbersome task for most developers. Explicit concurrency management, thread scheduling, load balancing and error recovery are non-trivial.

3.3.2 Original Phoenix Library

Phoenix [4] is a library that implements the MapReduce programming model for shared memory systems. In a similar fashion to that of Google's MapReduce [2], Phoenix attempts to simplify parallel code development by exposing the practical MapReduce model to the user and implementing an efficient runtime library that supports dynamic task scheduling, fault tolerance and deliberate resource management.

Phoenix API

The Phoenix library implements an API for C or C++, though it can be easily extended for languages like C# and Java. The functions provided by the Phoenix library are separated into two sets. The first set includes the functions that are defined by the library and must be used by the programmer in order to perform necessary

Function Description	R/O
Functions Provided by Runtime	
`int phoenix_scheduler (scheduler_args_t * args)` Initializes the runtime system. The `scheduler_args_t` struct provides the needed function & data pointers	R
`void emit_intermediate(void *key, void *val, int key_size)` Used in Map to emit an intermediate output <key,value> pair. Required if the Reduce is defined	O
`void emit(void *key, void *val)` Used in Reduce to emit a final output pair	O
Functions Defined by User	
`int (*splitter_t)(void *, int, map_args_t *)` Splits the input data across Map tasks. The arguments are the input data pointer, the unit size for each task, and the input buffer pointer for each Map task	R
`void (*map_t)(map_args_t*)` The Map function. Each Map task executes this function on its input	R
`int (*partition_t)(int, void *, int)` Partitions intermediate pair for Reduce tasks based on their keys. The arguments are the number of Reduce tasks, a pointer to the keys, and a the size of the key. Phoenix provides a default partitioning function based on key hashing	O
`void (*reduce_t)(void *, void **, int)` The Reduce function. Each reduce task executes this on its input. The arguments are a pointer to a key, a pointer to the associated values, and value count. If not specified, Phoenix uses a default *identity* function	O
`int (*key_cmp_t)(const void *, const void*)` Function that compares two keys	R

FIGURE 3.2: The Phoenix API, grouped into two sets. R and O stand for required and optional functions respectively

tasks such as the initialization of the MapReduce instance and the emitting of key-value pairs from map and reduce stages. The second set contains functions that are defined by the user. Function arguments throughout the API are declared as void pointers so that they are type agnostic.

In order to initialize a MapReduce instance, user code has to call the `phoenix_scheduler()` function. The `scheduler_args_t` 3.3 data structure is used to pass all the required information to the library. The basic fields must be properly set by the programmer. Optional fields are used to enable custom optimizations.

The Phoenix API 3.2 guarantees that each reduce worker will process its partition of intermediate key-value pairs in key order, however it does not guarantee any specific order in the way the original input data will be processed during the map phase. The programmer has to ensure that his algorithm is compatible with the MapReduce programming model and the restrictions applied by the Phoenix API.

Phoenix Runtime

Figure 3.4 gives an overview of the execution and basic data flow of Phoenix. The scheduler orchestrates the MapReduce job. At first, using the splitter function, the input is divided into equally sized chunks that will be processed by the map tasks. Each input chunk corresponds to a map task. The size of a chunk is adjusted so that it fits the L1 cache. In general, the optimal chunk size is a trade-off between better load balancing (smaller chunks) and lower library overhead (fewer bigger chunks).

Field	Description
Basic Fields	
`Input_data`	Input data pointer; passed to the Splitter by the runtime
`Data_size`	Input dataset size
`Output_data`	Output data pointer; buffer space allocated by user
`Splitter`	Pointer to Splitter function
`Map`	Pointer to Map function
`Reduce`	Pointer to Reduce function
`Partition`	Pointer to Partition function
`Key_cmp`	Pointer to key compare function
Optional Fields for Performance Tuning	
`Unit_size`	Pairs processed per Map/Reduce task
`L1_cache_size`	L1 data cache size in bytes
`Num_Map_workers`	Maximum number of threads (workers) for Map tasks
`Num_Reduce_workers`	Maximum number of threads (workers) for Reduce tasks
`Num_Merge_workers`	Maximum number of threads (workers) for Merge tasks
`Num_procs`	Maximum number of processors cores used

FIGURE 3.3: Phoenix scheduler_args_t data structure type

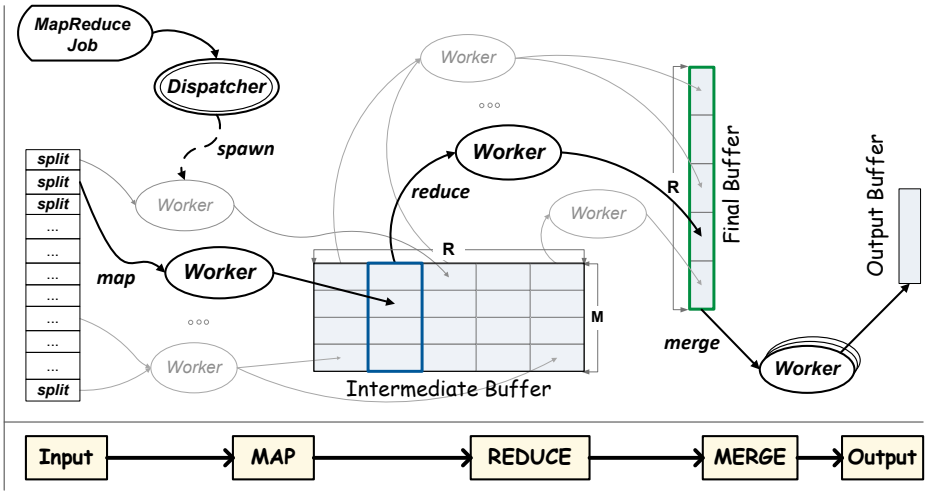


FIGURE 3.4: The execution overview and data flow of the Phoenix Runtime

When splitting the input into chunks is over the scheduler spawns the worker threads. Their number depends on the number of available cores in the system. Typically, one worker per logical core maximizes the throughput, but this is subject to tuning, by the developer.

Map tasks are assigned dynamically to workers. After applying the user-defined map function to the input chunks, they emit intermediate key-value pairs. The partition function splits these pairs into input units for the reduce tasks. Further, it ensures that all the values associated with the same key will end up in the same unit.

Once the map phase is over the scheduler initiates reduce phase. Again, reduce tasks are assigned dynamically to workers and apply the user-defined reduce function to the set of values associated with each key. As the size of each unit depends

on the distribution of intermediate key-value pairs reduce phase may suffer from load imbalance. The output of each reduce task is placed in an output buffer, ordered by key. Finally, when all reduce tasks are over their outputs are merged together in a single buffer, which is the final output to the user.

Phoenix Test-cases

The source code of Phoenix comes with a set of test-cases to help users get started. They are widely used algorithms from the domains of enterprise computing (Word count, Reverse index, String match), scientific computing (Matrix Multiply), artificial intelligence (KMeans, PCA, Linear Regression) and image processing (Histogram). Below is a brief description of these test-cases.

Histogram Generates the frequency histogram of an 8-bit RGB pixel bitmap.

KMeans Iteratively groups n-dimensional points into a user specified number of clusters, until convergence is reached.

Linear Regression Computes the line that best fits a set of coordinates.

Matrix Multiply Calculates the product of two equal-size, square matrices.

PCA Calculates the mean and covariance vector of a set of points, as the first steps of performing the Principal Component Analysis algorithm.

Reverse index Creates an inverted index for HTML files, which is a mapping from the links found in the HTML files to the files it was found.

Word count Counts the frequency of each unique word in a text file.

String match Searches through a file for the occurrences of a given set of words.

Performance Summary

Overall, Phoenix managed to demonstrate the advantages and potential of the MapReduce programming model on shared memory systems. The time and effort required by the programmer to develop parallel code is significantly reduced. Furthermore, Phoenix provided reasonably scalable performance on multi-core chips. Out of the eight test-cases mentioned in section 3.3.2 Phoenix achieved better run time in three test-cases, similar though a little worse in two test-cases, and was clearly outperformed in three test-cases, compared to custom P-thread parallel implementations.

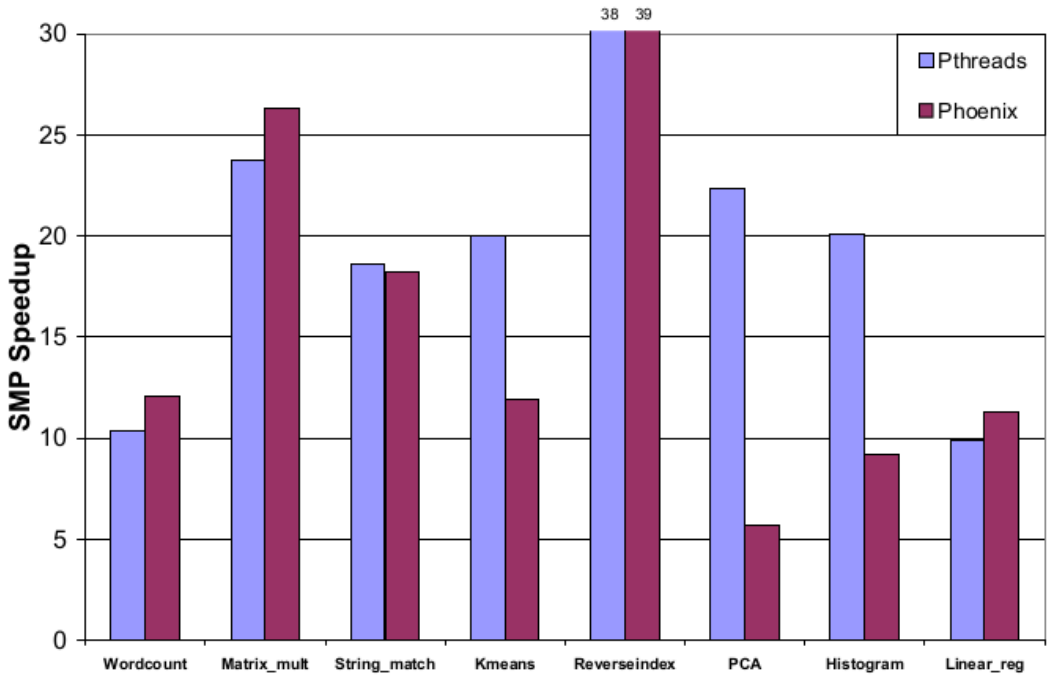


FIGURE 3.5: Phoenix vs Pthreads implementation in a 24-core multi-processor. The speedup is with respect to the same sequential code. We can see that despite of the library overhead, the overall performance and scalability are competent. [4]

3.3.3 Phoenix Rebirth

As a proof of concept, original Phoenix [4], established that a careful implementation of the MapReduce programming model for shared memory multi-cores can be invaluable. It frees the programmer from the burdening tasks of thread concurrency and proper data locality. Despite the runtime overheads Phoenix managed to perform competently on small-scale systems with uniform memory access (UMA).

Phoenix Rebirth [11], the second revision of Phoenix, is trying to tackle inefficiencies occurring on larger scale systems with non-uniform memory access (NUMA) characteristics. The optimizations are organized into three categories: (i) Algorithms, (ii) Implementation, and (iii) OS interaction.

Algorithmic Optimizations

The thread scheduler of the Original Phoenix was lacking NUMA awareness as all workers were treated identically. This can become a major bottleneck on a large scale NUMA environment. To overcome this issue, Phoenix Rebirth uses a separate task queue per locality group [36]. A locality group is defined as a set of CPUs in which each CPU accesses any memory in the system within a certain latency interval. By distributing tasks according to the location of the related data, map

workers operate mainly on local data. For better load balancing map workers will attempt to steal tasks from a remote task queue when the local task queue is empty.

Implementation Optimizations

Unlike Distributed MapReduce, where the network bandwidth dictates the data storage and retrieval rate, in Phoenix the data structures used to store intermediate data are critical for the system's performance. Figure 3.6 depicts the data structure used for the intermediate buffer as well as the access pattern during the map phase.

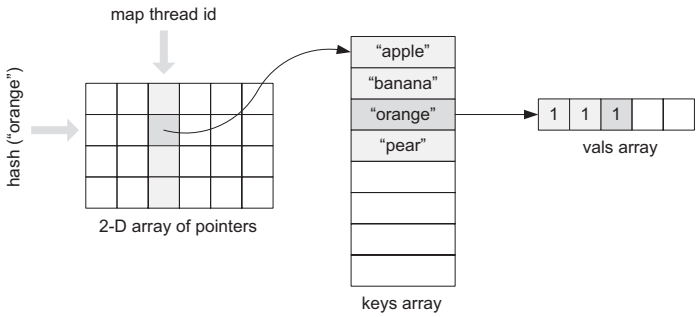


FIGURE 3.6: Original Phoenix data structure for intermediate key-value pairs

When a map worker emits a value, it uses its thread map id to index the array column-wise. The hash value of the key is then used to calculate the row of the array to insert the emitted key-value pair. Each row forms a separate keys array. Keys arrays are implemented as contiguous buffers, sorted by key, to enable fast binary searching. Each entry on the keys array maintains a pointer to another contiguous buffer for the values. As arrays in C are row-major arranged in memory, when accessing the intermediate buffer, map workers exhibit little to no locality. On the contrary, reduce workers operate row-wise on the intermediate buffer, hence this data structure is optimized for the reduce phase.

This implementation has several downsides. As the keys array is a fixed size, sorted buffer, every time it runs out of space, a reallocation is needed. To make things worse, every time a new key is added, all the entries below need to be shifted accordingly to maintain proper order. Vals array suffers from the reallocation issue too.

Phoenix Rebirth attempts to mitigate these issues by a course of implementation improvements. Firstly, the number of hash buckets, i.e. rows of the intermediate storage data structure, is increased significantly so that on average only one key resides in every keys array. This avoids regular buffer reallocation and key reordering. Unfortunately, workloads with a fixed number of possible keys, did not profit

from this optimization. In the end, it was decided to make the hash table size user tunable and propose default values for the existing workloads. Furthermore, the vals array was implemented as a buffer of disjoint memory chunks, to tackle the re-allocation issue. An iterator interface was implemented and exposed to the reduce workers in order to maintain sequential accessibility.

OS Interaction Optimizations

In Phoenix the workers are forked in the beginning of a MapReduce instance and joined back at the end. The `mmap()` system call is used to allocate memory for the threads' stacks and, subsequently, the `munmap()` system call is used to deallocate the memory during thread joining. These two calls were detected to bottleneck the scalability of Phoenix. To overcome this obstacle, a thread pool was implemented that reuses the threads, thus keeping the number of calls to `mmap()` and `munmap()` to a minimum.

Performance Summary

Compared to original Phoenix, its second, heavily optimized version demonstrates improved run time in all workloads and thread counts. In particular, for single chip executions, with less than 64 threads, the average speedup is 1.5x, while the maximum reaches 2.8x. However, the speedup gets significantly bigger in NUMA environments with up to 256 threads. In this case, the average speedup achieved was 2.53x, while the maximum was 19x.

3.3.4 Phoenix++: The Latest Implementation

Phoenix++ [1] is the third, and to our knowledge, final revision of Phoenix. Although Phoenix Rebirth successfully demonstrated the applicability of MapReduce on shared memory systems, the uniform implementation of intermediate buffers and the inefficient combiners are limiting the framework from supporting a wider range of applications.

Phoenix++ offers a modular and flexible pipeline, which allows the programmers to effortlessly adapt the runtime to their workload characteristics. In particular, intermediate key-value grouping and storage is exposed to the users through the *container* and *combiner* interfaces. Moreover, the memory allocator, used by the runtime, has been made easily configurable and the final key-value sorting optional. Nevertheless, increased modularity can have opposite results due to additional function calls and less opportunities for compiler optimizations. To deal with this issue, Phoenix++ is re-written in C++ [34] and uses templates [10] extensively to enable static code inlining.

Containers

In Phoenix++, containers substitute the intermediate key-value data structure, and together with combiners they provide a high performance key-value storage, customizable for a variety of workloads. Each map worker has a thread local container, where it emits the intermediate key-value pairs. Depending on the distribution of keys per map task three different types of containers are provided:

hash container A variable width hash table is used if every map task can emit any key.

array container When every map task can emit any key in a predefined range a fixed-size, thread-local array is used.

common array container In the case where each task emits a single, unique key a non-blocking, global array shared among all the workers is used.

Combiners

A combiner object is used to store all the emitted values associated with the same key. Notice that combiner used to be a function that was applied at the end of the map phase in Phoenix Rebirth [11], but now it is a stateful object. Again, following the modular fashion of Phoenix++, two disparate types of combiners are provided:

buffer_combiner The `buffer_combiner` is used to provide the traditional functionality of MapReduce. It is buffering up all the values having the same key, in an array composed of disjointed memory regions, as implemented in Phoenix Rebirth.

associative_combiner Contrary to `buffer_combiner`, `associative_combiner` is not buffering up the values, but combines them together and maintains a single aggregated value. This eliminates all the issues and overheads due to dynamic memory allocation. The combine function, that will be used to combine a newly emitted value with the aggregated value stored in the combiner object, is provided by the user.

Performance summary

The combination of customizable containers, efficient combiners and function call overhead elimination, offers Phoenix++ a 4.7x speedup on average, compared to the second Phoenix version, Phoenix rebirth.

3.3.5 Alternative Shared-Memory MapReduce Libraries

In the following section we describe some alternatives to the Phoenix MapReduce runtime. Their attempt is to optimize or customize the traditional MapReduce architecture to fulfill particular use cases.

Tiled map reduce

Tiled MapReduce [13] is an interesting variation, which argues that it is more efficient to iteratively process smaller chunks of data through multiple MapReduce invocations, than processing a large chunk of data at once. Processing smaller jobs significantly reduces memory footprint and leads to a more advantageous use of the memory hierarchy. Therefore, Tiled MapReduce utilizes the tiling strategy to partition a MapReduce job in smaller sub-jobs, process them iteratively and in the end, merge the output of all the sub-jobs together. To reduce the overhead of reallocation of data structures on every new sub-job run Tiled- MapReduce reuses input and intermediate buffers among sub-jobs.

Tiled-MapReduce reports a speedup over Phoenix [4], by a factor of up to 3.3x, while saving up to 85% of memory.

Metis

Metis [5] highlights the decisive role of data structures used to store the intermediate key-value pairs, in the performance of MapReduce on shared-memory multi-core processors. Stepping in this direction Metis implements a data structure designed to perform adequately on most workloads. The main targets are applications with a relatively large number of intermediate key-value pairs, and a low amount of computation so that the effects of data storage and retrieval are not overshadowed by the computational part.

The proposed data structure for intermediate buffers is a fixed-size hash table with a b+tree in each entry, called hash+tree. In the case where the hash table size is sufficient for the keys, the cost of insertion and retrieval is $O(1)$, because of the hash table. If there a lot more keys than hash table entries, the insert and retrieve cost fall back to $O(\log(n))$ due to the b+tree. Finally, if the keys are less than the hash table entries, the overhead of allocation and deallocation of the hash table will be sub-optimal, but nevertheless insignificant, compared to the total execution time. In overall, the suggested, hybrid data structure offers a reasonable compromise for most workloads.

Key to the effective behavior of the hast+tree structure, described above, is an accurate estimation of the number of intermediate keys. To predict the number of keys, Metis transacts a test run on a 7% input sample and counts the number of

emitted keys. It then sets the hash table size so that each entry (b+tree), will likely contain 10 keys.

Compared to Phoenix [4] Metis performs similarly or better on most applications, without requiring extra tunning by the programmer.

MRPhi

MRPhi [12] adapts the MapReduce framework in the needs of Intel® Xeon Phi™ Co-processor. Xeon Phi, a recently released product family by Intel, features a set of wide (512-bit) vector units to employ Single Instruction Multiple Data (SIMD) computation. Furthermore, it contains many more cores (~60) than conventional CPUs, and through hyper-threading each core can utilize up to 4 logical threads. Noticeably Xeon Phi is a promising platform that has already demonstrated its potential [37], [38], [39], [40].

In order to fully utilize the advanced characteristics of the Xeon Phi Co-processor MRPhi suggests a MapReduce runtime with the following characteristics:

- Map phase is implemented in a SIMD friendly way.
- SIMD hash functions are used to utilize the wide vector units.
- Pipelined map and reduce phases take advantage of hyper-threading and lead to better resource utilization.
- Instead of having thread local containers a global container is used to address the limited memory issue. Atomic operations are used to modify the global array.

With these optimizations implemented MRPhi demonstrated a surprising speedup of up to 38x over Phoenix++ on Xeon Phi Co-processor.

3.4 High-Performance Shared Queue Implementations

The shared data-structure that is used for the mappers-combiners communication is placed in the core of our architecture and its performance has an immediate impact on our framework's overall performance. In this section, we present some of the most advanced shared queue implementations that we found in the literature. Basic knowledge as well as further sources about the original single-producer, single-consumer, shared buffer implementation can be found in Appendix B.

3.4.1 MCRingBuffer

MCRingBuffer [14] is an implementation of a lock-free, concurrent, circular buffer, optimized for cache efficiency. Two aspects have been taken into account to achieve cache-efficiency, (i) cache line protection, and (ii) batched updates of control variables.

Cache-line protection

When multiple threads are accessing different data placed on the same cache line, they invalidate the cache line state. Thus, the cache system will require other threads to reload their data from memory even though their data were not actually changed. This access pattern results in performance degradation and is known as false sharing. To avoid false sharing [41] such data must be placed on different cache lines. MCRingBuffer places the control variables (i.e. variables used by the buffer implementation to ensure proper thread synchronization, like number of elements written or read) on different cache lines by using the padding technique.

Batched updates of control variables

On a basic implementation of a circular buffer control variables are updated every time a consumer is reading from the buffer, or a producer is writing to it. However, MCRingBuffer divides the ring buffer into blocks of `batch_size` elements and increments the read or write control variable to the next block only after `batch_size` read or write operations. This way the frequency of writing the shared variables is reduced. Note that this batched update scheme is based on the assumption that, new data are always available, which is not always the case. Proper modification by the programmer is needed to support applications with finite amount of elements pushed in the queue.

3.4.2 Fast Forward

FastForward [15] is a system for pipelining parallelism on multi-core architectures built on top of an efficient, concurrent, lock-free queue to provide low latency core-to-core communication. The motivation is to implement a queue suitable for network frame processing with a target throughput rate of ~1.5 million frames per second or ~ 672ns per frame. To accomplish that FastForward couples control and data together, so that the producer and consumer can operate independently when the queue has at least one element on it. Listing 3.2 presents the pseudo-code for enqueue and dequeue functions, taking advantage of this method.

```
1  enqueue_nonblock(data) {
2      if (NULL != buffer[head]){
3          return EWOULDBLOCK;
4      }
5      buffer[head] = data;
6      head = NEXT(head);
7      return 0;
8  }
9
10 dequeue_nonblock(data) {
11     if (NULL == buffer[tail]){
12         return EWOULDBLOCK;
13     }
14     data = buffer[tail];
15     buffer[tail] = NULL;
16     tail = NEXT(tail);
17     return 0;
18 }
```

Listing 3.2: FastForward [15] enqueue and dequeue functions

3.5 MapReduce Architecture Related Work

Summarizing the MapReduce Work-flows, proposed in the literature, we see that Phoenix [4] initially adopted the traditional Map, Reduce, Merge scheme, as proposed by Google MapReduce [2]. Metis [5], based on Phoenix, implemented a hybrid intermediate buffer data structure capable of performing adequately on most workloads, and maintained the conventional MapReduce scheme. Phoenix Rebirth [11] added the concept of local combining, but with little success. Phoenix++ [1] optimized the combine process by introducing stateful, modular, combiner objects and local containers. MRPhi [12], a customized to the needs of Intel® Xeon Phi™, shared-memory MapReduce library, interleaved map and reduce phases to take advantage of MIMD hyper-threading and recorded an overall performance improvement of 8.5%. Tiled MapReduce [13], following the compilers' loop tiling technique, divides a MapReduce task into smaller parts and process them iteratively to reduce the memory footprint and enable locality-aware optimizations.

Non-conventional MapReduce architectures are not exclusive to shared-memory systems. MapReduce Online [42], based on Hadoop [3], extends its range of capabilities by pipelining data between mappers and reducers. The implementation, called Hadoop Online Prototype (HOP), allows users to have early returns from yet incomplete jobs, and also supports stream processing and event monitoring applications.

This is done by modifying the classic MapReduce work-flow, and, instead of reducers pulling data from mappers in the end of map phase, mappers pro-actively push their data, in batches of configurable size, to reducers.

Regarding high-performance, concurrent queue implementations, MCRingBuffer [14] avoids false sharing of control variables, using the padding technique. To reduce further the queue overhead, it divides the queue into smaller blocks and updates the control variables only when a whole block has been processed. Fast Forward [15] minimizes the producer - consumer interference, by coupling control variables together with data, using a simple technique, described in section 3.4.2.

3.6 Pipelined MapReduce

After having studied thoroughly and acquired a concrete understanding of the innovation, implementation specific details, assets and deficiencies of current, state-of-the-art shared-memory MapReduce implementations, we identify that the MapReduce runtime is deteriorated by the blocking point between map and reduce phase.

The removal of this barrier results in a further parallel runtime, as the execution of mappers is overlapped with that of reducers. In addition, as map phase is traditionally compute intensive and reduce phase is memory intensive, overlapping their execution leads to more efficient hardware utilization.

Given a flawless overlapping of two tasks, the expected time speedup is equal to the run time of the lighter task. In the best case, where the two tasks have equal execution times, the overall speedup will be ~50%. After a close lookup at the run time breakdown of different applications, we saw that map phase dominates the run time in almost every case. This was expected due to the use of combiners, proposed by Phoenix++ and their action was described analytically in section 3.3.4. Therefore, pipelining map and reduce phase would only induce a limited speedup.

We propose a novel MapReduce architecture, where *map and combine are decoupled in separate phases*. We introduce a concurrent, lock-free, circular queue which is shared between mappers and combiners. After executing a map task, mappers push intermediate key-value pairs in the queue. At the same time, in a pipelined fashion, combiners are consuming the pushed elements by running the combine function on them. We consider the overlap of mappers and combiners more preferable than that of mappers and reducers, as the combine function execution time is comparable with that of map function in many cases. Thus, the potential speedup is more auspicious. Chapter 4 covers the implementation details of our architecture and chapter 5 presents the performance evaluation and discussion.

Chapter 4

Implementation Details

4.1 Introduction

The purpose of this Chapter is to give the reader a thorough understanding of the implementation and architecture specific details of our proposed Pipelined MapReduce runtime. The chapter is organized as follows: in Section 4.2 we present the work-flow of our suggested runtime, in Section 4.3 we describe and reason about our choice of the mappers-combiners, shared data structure. In Section 4.4, we demonstrate the different thread-to-CPU binding policies that we created. Section 4.5, summarizes the role of all programmer tunable variables, essential to MapReduce runtime, and finally, Section 4.6 contains our secondary contributions.

4.2 Pipelined Map Reduce Architecture

In this section, we describe assiduously the high-level work-flow of our proposed pipelined MapReduce architecture. We will use figure 4.1 as the basis of our description.

Looking at the leftmost part of the diagram, we see the invocation of MapReduce runtime with a call to `MapReduce.run()` class method. At first, a set of necessary initializations and configurations take place. Variables such as, the total number of threads that will be used, the number of mappers and combiners, the thread-to-CPU binding policy, the number of map/ reduce tasks and a set of parameters that configure the SPSC queues, are initialized. Programmer can dynamically tune the framework according to application and platform needs, by exporting environmental variables. After having set up the library specific variables, two separate thread pools are initiated. In figure 4.1, the green-colored pool contains general-purpose workers and will be used for all the phases, except from combine. More specifically, the general-purpose thread pool is used to execute the tasks of Map, Reduce and Merge. On the contrary, the orange-colored thread pool is used only during combine phase and has typically less or at most equal number of workers compared to the general-purpose thread pool.

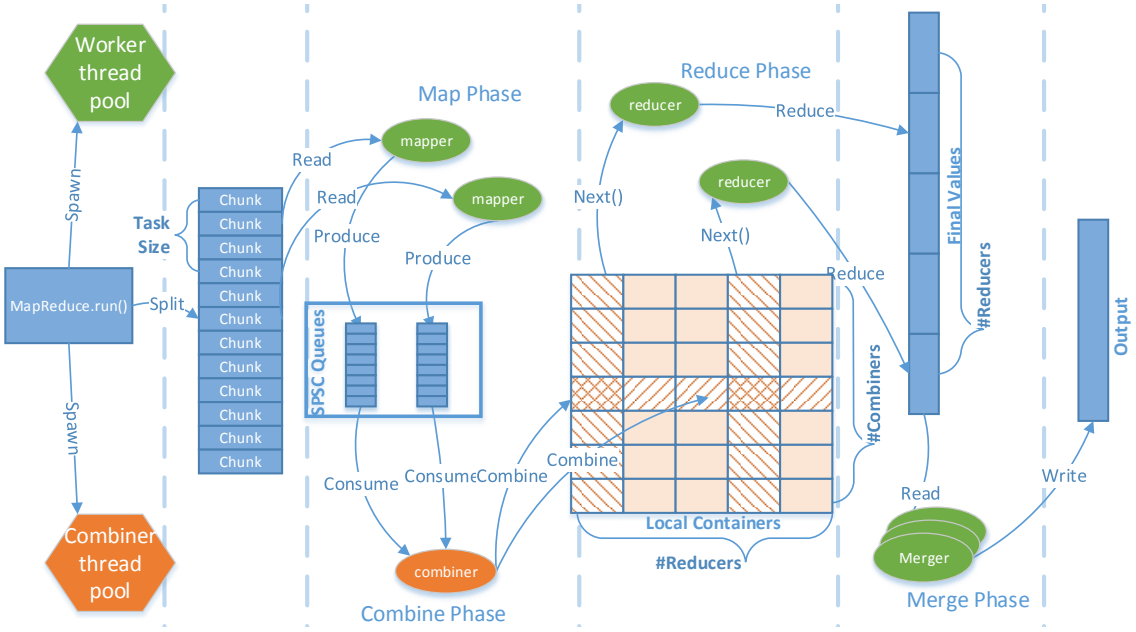


FIGURE 4.1: Pipelined MapReduce Architecture

Splitting the raw input into chunks comes next. If the input has to be formatted on a particular way, the user has to provide the split function, otherwise the input is splitted using the default function provided by the library. Task size defines the number of input splits that correspond to a task and is subject to dynamic tunning. Picking the optimal task size is not trivial, as big task sizes will result in substandard load balancing, while very small task sizes will result in non-negligible library overhead with regards to computation.

Map-Combine phases start, right after input splitting is done. All the map tasks are added in the available task queues – one for each locality group. Map workers, iteratively, dequeue tasks from their local queue and apply the user-defined map function on the task data. Map function produces intermediate key-value pairs that would be, according to the traditional MapReduce execution, directly combined by a combiner object and then stored in the worker's local container. Instead, in our proposed variation, for the purpose of overlapping the execution of map and combine functions, the intermediate data are pipelined through a set of Single-Producer, Single-Consumer concurrent queues. As soon as the queues get partially filled, combine workers start their operation. They pop batches of key-value pairs out of the queue, combine them using the user-defined combine function, and finally store the result in their local container. To allow multiple combiners to operate simultaneously, a separate container is allocated for each combiner. We argue that, *overlapping map and combine executions, results in more efficient hardware utilization and improved run times.*

Map phase ends when all input tasks have been processed. An atomic, boolean

flag is used to notify the combiners that map phase is over. Before exiting, combine workers, will loop once more over their assigned shared queues, and consume any remaining data. This final check is essential, as combiners process data in batches, and it is possible to have queues occupied with less than batch size elements. The rest MapReduce execution remains unchanged. Reduce phase takes over, with the reduce tasks being added in the task queues. Reduce workers, dequeue and execute the tasks dynamically. For each task, they iteratively, pass a pair, composed of a key and the set of all the values associated with this key, to the user-defined reduce function. The reduce function aggregates all the values and stores the result in the final values buffer. Each reduce worker uses a different buffer. When all reduce tasks have finished, merge phase begins. Merging the final buffers is done in parallel, in a binary tree like fashion. Optionally, if the output result needs to be sorted, every merge worker first sorts, in place, a piece of the final values buffer and then the merging is done, respecting the elements order, similarly to mergesort's merge step.

The result buffer is finally returned to the user. It is composed of a set of key-value pairs, so it can be directly fed into a new MapReduce instance for further processing.

4.3 Concurrent Shared Buffer

Vital role for the effective pipelining of map and combine phase plays the queue used by the mappers to forward data to the combiners.

4.3.1 Queue Characteristics

As we can see in figure 4.1, every mapper has its own queue. Thus, a single producer queue suffices. On the contrary, since combine tasks can be lighter than map tasks, we allow a combiner to correspond to one or more map queues. However, after assigning map queues to combiners, each combiner has exclusive access to its set of queues. This implies that there is only one combine worker for every queue, so a single consumer queue is also sufficient. In the case where a queue is partially filled, but not empty, a mapper and a combiner can operate on it concurrently. As a result, the semantics of our design are satisfied by a Single-Producer, Single-Consumer (SPSC), concurrent queue.

L.Lamport has proven that, under sequential consistency memory model, an SPSC concurrent queue can be implemented without using explicit synchronization mechanisms between the producer and the consumer [17], [16], [18]. With minimal modifications to Lamport's circular buffer implementation, it is possible to support weaker consistency models, such as the Total-Store-Order (TSO) consistency model [20].

Assuming hardware support for atomic instructions, such as compare-and-swap (CAS), it has been proven by Maurice Herlihy [43] that, is feasible to implement a multiple producer, multiple consumer (MPMC), lock-free, wait-free queue. As SPSC queues are specialization of MPMC queues, an MPMC queue could also be an acceptable candidate for our design. Unfortunately, the extra overhead of using atomic operations is not-negligible on modern processors, and as a result, even the most optimized MPMC queues are outperformed by Lamport's circular buffer with one producer and one consumer.

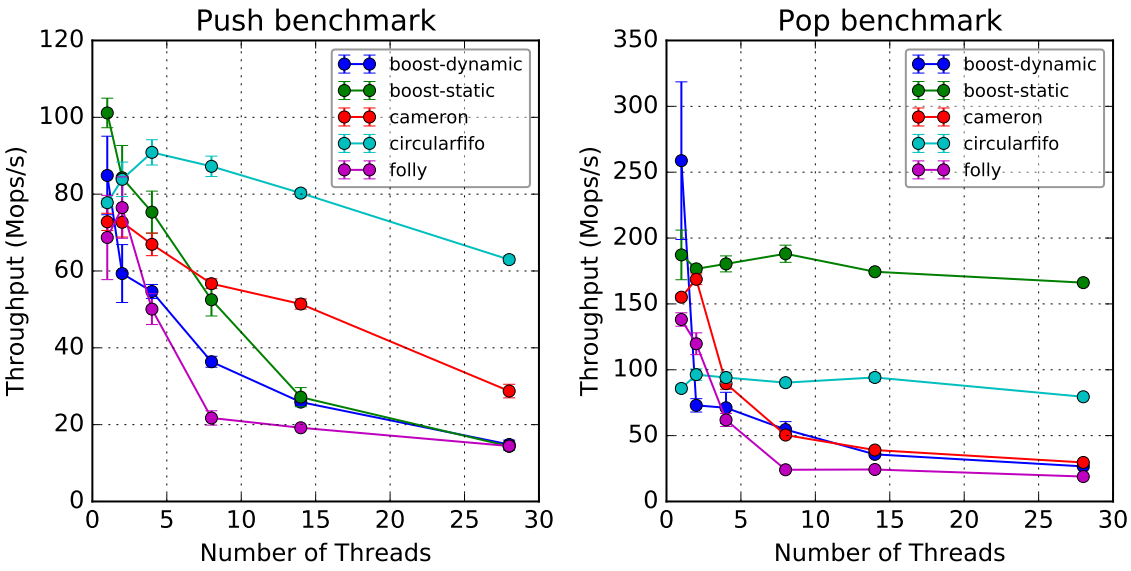
Considering all the above, we conclude that, a concurrent, Single-Producer, Single-Consumer, lock-free, wait-free, circular, fixed-size buffer is the most appropriate data structure for our needs.

4.3.2 Queue Implementations Benchmark

Luckily for us, our problem has been well studied and heavily optimized, so, instead of re-inventing the wheel, we found some lock-free, wait-free, SPSC queue implementations and benchmarked them in order to select the most efficient.

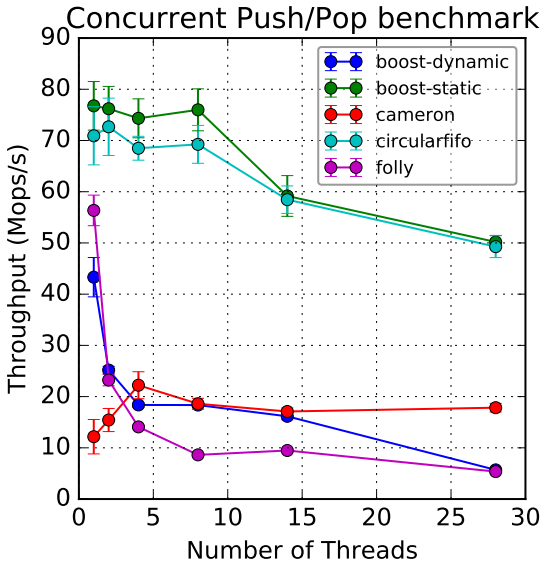
Three different benchmarks have been conducted over a set of five different concurrent SPSC queue implementations. In push benchmark, the throughput of producers, pushing in an empty queue of sufficient size is tested. Correspondingly, in pop benchmark, consumers are popping elements out of a full queue. In concurrent push/pop benchmark, one producer and one consumer are operating on the same queue, concurrently. The elements pushed/popped are key-value pairs of two 64-bit integers, because this format often appears in our MapReduce target applications. The y-axis denotes the average throughput in million operations per second. X-axis denotes the number of producer or consumer threads used for push or pop benchmarks respectively, while for the concurrent benchmark, x-axis denotes the producer-consumer pairs used, thus the total number of threads is twice the x-axis value. Note that, in all the cases, no two producers are sharing the same queue. The same applies to consumers.

The benchmarks were evaluated on two heterogeneous platforms, which are described adequately in section 5.2. Figure 4.2 summarizes the benchmark results for the Intel Haswell multi-core platform. Boost [22] with static memory allocation and Circularfifo [21] perform similarly, with Circulafifo achieving better push throughput and boost-static better pop throughput. Circularfifo is also using static memory allocation, meaning that the size of the buffer is known at the compile time. On the contrary, folly [44], cameron [23] and boost-dynamic define the buffer's size at run time. The results from Intel® Xeon Phi™ Co-processor card, are summarized in



(A) haswell-push

(B) haswell-pop

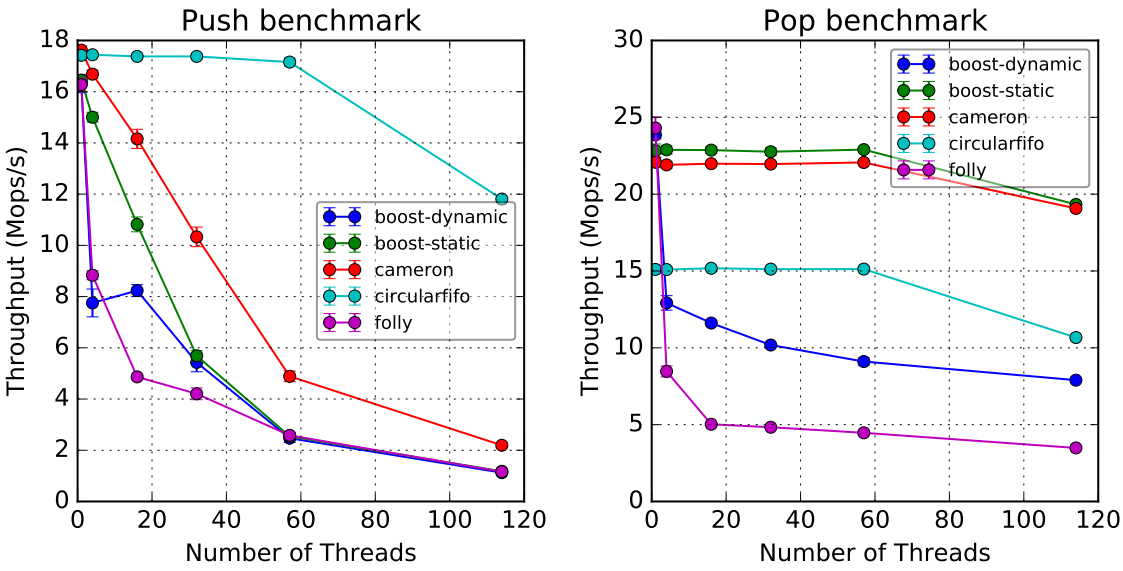


(C) haswell-concurrent

FIGURE 4.2: Comparison of five different SPSC queue implementations on Intel Haswell Platform.

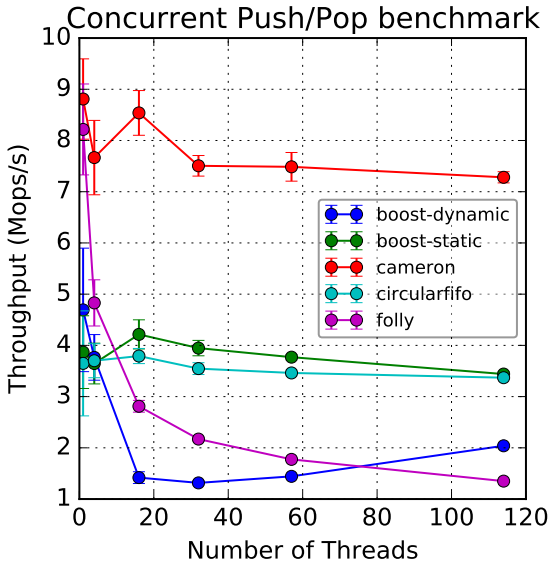
figure 4.3. The overall picture is different than Haswell, with Circularfifo demonstrating the best push performance, cameron and boost-static sharing the best pop throughput and cameron having the best concurrent push/pop performance.

In order to select the best queue implementation, except from the benchmarks shown in figures 4.2, 4.3, we considered their API. Excluding boost dynamic/ static queues, all the res offer a limited API, containing the basic functions of `push()`, `pop()`, `top()` plus a `size()` approximation. On the other hand, boost offers a very rich API, with enhanced capabilities such as pushing/ popping elements in batches, and also a function that pops and consumes one or all available elements



(A) phi-push

(B) phi-pop



(C) phi-concurrent

FIGURE 4.3: Comparison of five different SPSC queue implementations on the Intel Xeon Phi platform.

with a single call. We discovered that these capabilities, if used properly, perform better than the basic operations. This settled us on using boost, with static memory allocation, as the core of our SPSC queue.

4.3.3 Queue Optimizations

In the previous section, we explained the reasons that led us to select boost with static memory allocation over the rest benchmarked implementations.

Building on top of it, we added some extra optimizations. They are presented below and evaluated in Chapter 5.

Producer Sleep time The way that our architecture is designed, requires that, pushing an element in a queue succeeds always. Of course, discarding elements or overwriting existing ones, is not allowed. This means that, when a mapper tries to push data on a full queue, he will have to wait until some space is freed. A classic busy wait loop can be used for this purpose. Instead, we noticed that, having the mapper to fall asleep for a sort amount of time, after a failed trial, can result in performance improvement. The default value for this sleep interval is 100 nanoseconds, but can be modified, as described in section 4.5.2.

Bathced Reads Normally, combiners are looping over their associated queues, read available data and aggregate them through the user-defined combine function. Instead of letting them consume single elements, we divide the buffer into blocks of `batch_size` elements, that will be processed contiguously. This reduces the mappers-combiners contention on shared queue control variables and increases spatial data locality. We benchmark the effect of this optimization in section 5.7.

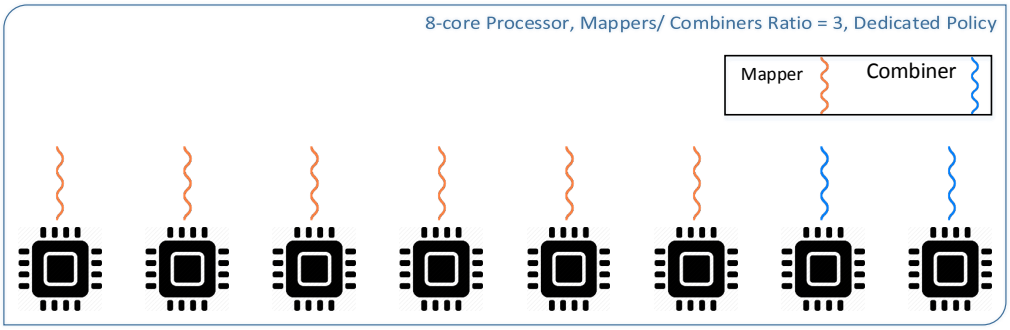
Read and Consume As we previously mentioned, boost extends the traditional queue API and supplies functions with extended capabilities. We take advantage of the `consume_all(Function f)` method, that reads and consumes all available elements on a single call. Based on it, we implemented a `consume_batch(Function f, size_t batch)` function, that consumes batch elements at once. Notice that, we pass the function that will consume the data as an argument. This reduces the number of function calls to an absolute minimum and, at the same time, eliminates completely the need of data copying between the queue and the combine workers.

4.4 Memory Aware Thread-to-CPU Binding Policies

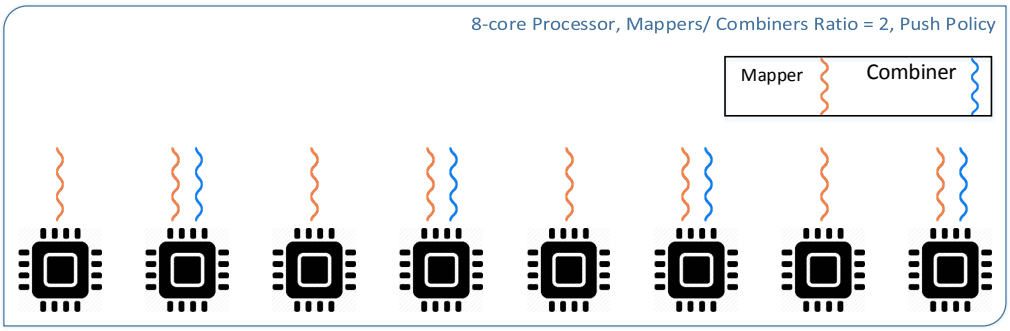
In order to pipeline Map and Combine phases, we introduced a set of SPSC shared queues. Mappers use them to write the products of map functions, and, concurrently, combiners consume these data through the combine function. Due to this extensive communication, additional pressure is posed on the memory subsystem. In the absence of a careful design, the communication overhead between Mappers and Combiners can counter any probable speedup.

Since the very beginning of our implementation, it was made clear, that the communication overhead has to be minimized, and it is essential to pack as close as possible threads that access the same resources. We tackle this issue in three steps. Firstly, depending on the mappers/ combiners ratio we correspond statically every

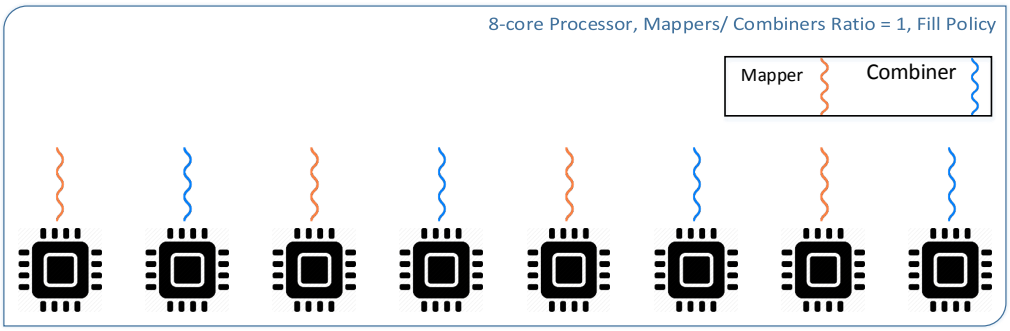
combiner to a set of mappers. This correspondence will remain unchanged till the end of map-combine phase. Secondly, we select the most appropriate thread binding policy, out of the three different policies that we have implemented, pictured in figure 4.4. Finally, using the system call `sched_setaffinity()` mapper and combiner threads are binded to specific logical cores.



(A) Dedicated Policy



(B) Push Policy



(C) Fill Policy

FIGURE 4.4: Graphical demonstration of the three different thread-to-CPU binding policies

Figure 4.4 represents graphically the three different binding policies that are available. A short description follows, while their performance is benchmarked in section 5.4.

Dedicated Policy This is the first and most basic policy we implemented. As we see, it is not taking into consideration the communication pattern between mappers and combiners.

Push Policy An alternative policy that is employing more threads than logical cores. Its logic is to bind a mapper thread to every logical core and in addition, according to the specified mappers/ combiners ratio, bind the combiners side by side with mappers. Though promising, it is outperformed by Fill policy.

Fill Policy This policy targets the incompetency of the Dedicated policy. It interleaves mapper and combiner threads in order to minimize their distance in logical core units. Our results show that this policy is the most efficient for all the test-cases.

4.5 Proper Configuration is Crucial

Fine tuning of the MapReduce runtime and queue characteristics is optional, but nevertheless, vital to competent performance. Different types of workload may require completely diverge set of parameters to operate efficiently. This section exposes to the reader, the role and importance of all the runtime configurable variables.

4.5.1 MapReduce Runtime Configuration

Variable name	Description	Default Value
MR_NUMTHREADS	Number of Threads	Number of logical cores
MR_NUMMAPTASKS	Number of Map Tasks	$map\_threads \times 16$
MR_NUMREDUCETASKS	Number of Reduce Tasks	Number of Map threads
MR_CHUNKSIZE	Multiplier of Task size	1

Table 4.1: MapReduce Runtime Configurable Variables

Table 4.1 presents the MapReduce runtime related sizes that are subject to programmer tunning. The MR_NUMTHREADS variable defines the total number of threads that will be used by the library and defaults to the number of logical cores. This is an acceptable choice for most applications and specifically for compute intensive workloads. MR_NUMMAPTASKS sets the amount of map reduce tasks that will be created, and, as every input split has to be in one and only task, the task size derives from the formula:

$$task\_size = \frac{input\_splits\_count}{num\_map\_tasks} \tag{4.1}$$

In practice, we found little usage for MR_NUMMAPTASKS variable as, picking an optimal number of map tasks, is both application and input size specific. However, MR_CHUNKSIZE is a factor that scales the derived task size. When defined, the actual task size is calculated as:

$$task\_size = \text{MR\_CHUNKSIZE} \times \frac{input\_splits\_count}{num\_map\_tasks} \tag{4.2}$$

Our benchmarks show that, some applications may profit from smaller task sizes, meaning MR_CHUNKSIZE values around $1/4, 1/8$. Finally, the number of reduce tasks that will be generated is controlled by MR_NUMREDUCETASKS. By default, there will be one reduce task per map worker. As in all our target applications, reduce phase occupies only a minor fraction of the total execution time, we didn't experiment with the number of reduce tasks.

4.5.2 Map-Combine Pipeline Configuration

Variable name	Description	Default Value
MR_MAP_COMBINE_RATIO	Mappers / Combiners ratio	1
MR_THR_TO_CPU_POLICY	Thread to CPU binding policy	1 (Fill policy)
MR_BUF_SIZE	SPSC Queue fixed-size	1000
MR_BATCH_SIZE	Minimum number of elements to consume	100
MR_MAP_SLEEP_TIME	Sleep interval in case of full buffer	100 nano sec

Table 4.2: Map-Combine Pipeline Configurable Variables

A variety of environmental variables can be used to properly tune pipeline and queue specific sizes. Table 4.2 summarizes them and a more accurate description follows.

The MR_MAP_COMBINE_RATIO is perhaps the most important variable and affects heavily the execution time. As its name suggests, it defines the mappers to combiners ratio, or more specifically:

$$\text{MR\_MAP\_COMBINE\_RATIO} = \frac{num\_of\_mappers}{num\_of\_combiners} \tag{4.3}$$

Its value is a positive integer above 1, as mappers have to be at least equal to combiners. The optimal ratio is defined by the heaviness of the workload of map and combine phases. The heavier the map workload is, compared to combine, the greater the ratio should be. MR_THR_TO_CPU_POLICY is used to select one of the three different policies, described in section 4.4. Dedicated policy corresponds to a value of 0, Fill policy to 1 and Push policy to 2. MR_BUF_SIZE sets the size of concurrent SPSC queues. A small value will cause the buffers to be frequently full, thus mappers will be unable to push data, while a very big value will be a waste of memory and lead to limited spatial locality. In practice, values ranging from 1 to 10 thousands performed the best. In section 4.3.3, we explained how combiners consume batches of data instead of single elements. The environmental variable MR_BATCH_SIZE is used to set the minimum number of elements that should be in the queue before the combiner starts consuming them. Finally, MR_MAP_SLEEP_TIME is the interval in nanoseconds, that mappers will sleep, in case the buffer is full, before checking again. By experimentation we discovered that a value of 100 nanoseconds achieves adequate performance, while zero sleep time between each re-trial is sub-optimal.

4.6 Secondary Contributions

This section contains our secondary contributions, which were, nevertheless, essential to help us better conceive the behavior of our pipelined implementation.

4.6.1 Synthetic Test Suite

While developing our architecture, we realized that our set of test-cases was limited in terms of map/ combine workload type and weight. More specifically, both Linear Regression and Histogram, have extremely light map and combine workloads. PCA has practically no combine phase, while the rest test-cases, i.e. Word Count, Matrix Multiply and KMeans, have significantly heavier map than combine phase. Though this is the most common case, it is not the only one. We wanted to make sure that our implementation is tested under as many as possible different workload types.

Implementing some new MapReduce applications would be a possibility. This way, we would be able to test different types of map-combine workloads. However, again, we would have little to no control over their weight. We decided to implement a synthetic test suite, that would allows us to easily configure the type, as well as the amount of work of map and combine phases.

To imitate a CPU-intensive workload, we combined computationally heavy trigonometric, exponential and logarithmic functions, which operate on a small contiguous dataset. On the contrary, to reproduce a Memory-intensive workload, we

apply computationally light operations on a wide dataset with pseudo-random memory access pattern. A set of runtime configurable variables are used to adjust the heaviness of map and combine tasks independently.

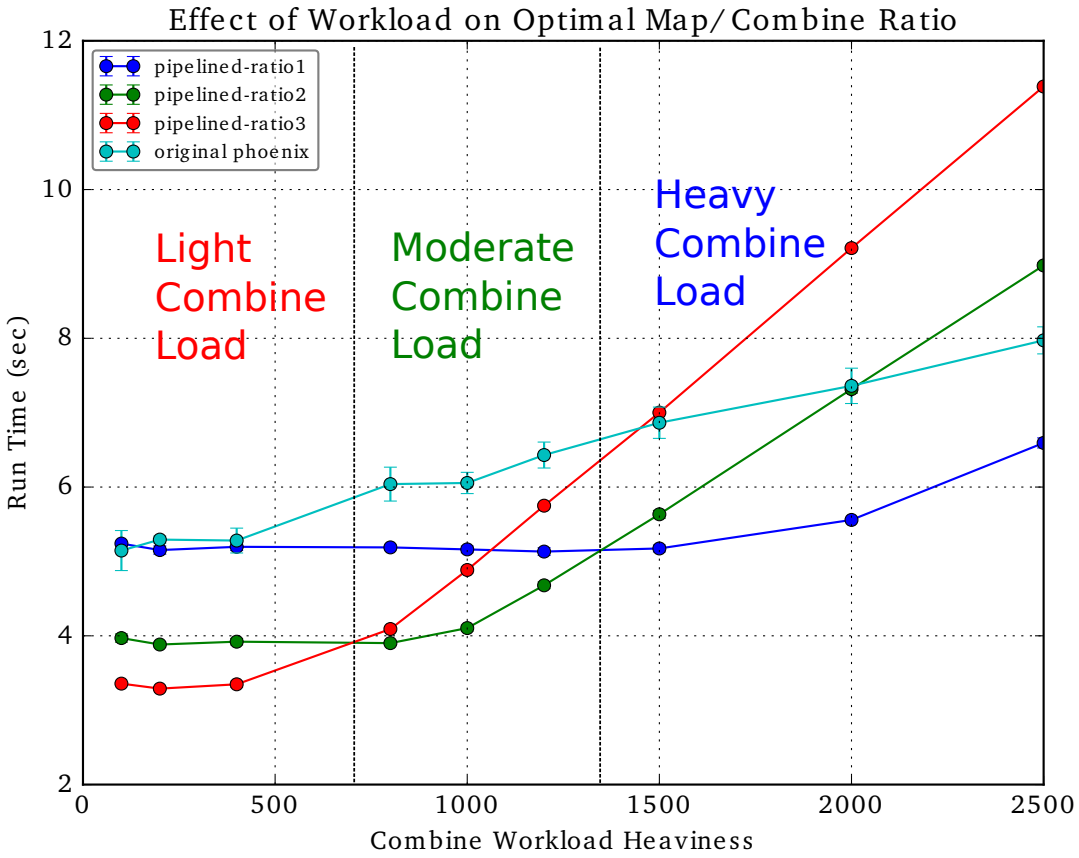


FIGURE 4.5: Varying Combine Workloads with Synthetic Test-case

Figure 4.5, demonstrates the usability of the test suite. Map task was fixed as compute-intensive, while combine task as memory intensive. The x-axis denotes the different combine phase workloads that were tried. The cyan line shows the run-time of Phoenix++ [1] without any modifications, and the rest three lines correspond to our pipelined framework with different mapper/ combiner ratios. We can clearly see, how the optimal ratio depends on the workload weight distribution over map and combine phases.

4.6.2 Application Characterization

In this section, we attempt to analyze the behavior of different applications and characterize their compatibility with Pipelined MapReduce. We base the quantification of applications' fitness on three simple metrics:

$$IPB = \frac{Instructions}{Input\_Bytes} \tag{4.4}$$

$$MSPI = \frac{Memory\_Stalls}{Instructions} \tag{4.5}$$

$$RSPI = \frac{Resource\_Stalls}{Instructions} \tag{4.6}$$

IPB measures the complexity of an application. *Applications with very low complexity, like Histogram and Linear Regression, are not well-fitted for our framework, simply because there is not enough work, to decouple and overlap efficiently.* On top of that, the introduced complexity by our framework, due to queue manipulations, can dominate the execution time of light applications, with low IPB values.

On the other side, applications with high IPB, are not necessarily well-fitted to our architecture. If map and combine workloads, are complex enough, but are not competing for any memory or CPU resources, they will not profit from our framework's, more efficient hardware utilization. This means that IPB by itself is not sufficient for our characterization, thus we introduce two more stall-related metrics.

Memory Stalls per Instruction (MSPI) and Resource Stalls per Instruction (RSPI) represent how often an application experiences memory subsystem and resource lated stalled cycles, respectively. Resource stalls can be due to a full Reorder buffer, not eligible Reservation Station entry available, no store or load buffers available and register renaming. Memory related stalls contain stalled cycles due to L1D and L2 cache misses. *Applications with frequent stalls, indicate sub-optimal hardware utilization, and as consequence, they are good candidates for our decoupled implementation.* Based on IPB, MSPI and RSPI, we conclude that, *workloads with sufficient complexity, that suffer from frequent memory or resource stalls, are perfect candidates for our runtime.*

Figure 4.6 presents the IPB, MSPI and RSPI metrics for our six target applications. The metrics were calculated based on relevant hardware counters, that were extracted using the Intel® VTune™ Amplifier [45] performance profiler and refer to Map and Combine functions only. The figure has two y-axis scales, the left one is used for IPB and it is logarithmic, while the right y-axis is used for memory and resource stalls. For the intermediate key-value pairs, the default container was used, which is a hash table for Word Count and a fixed-size array for the rest. We can see

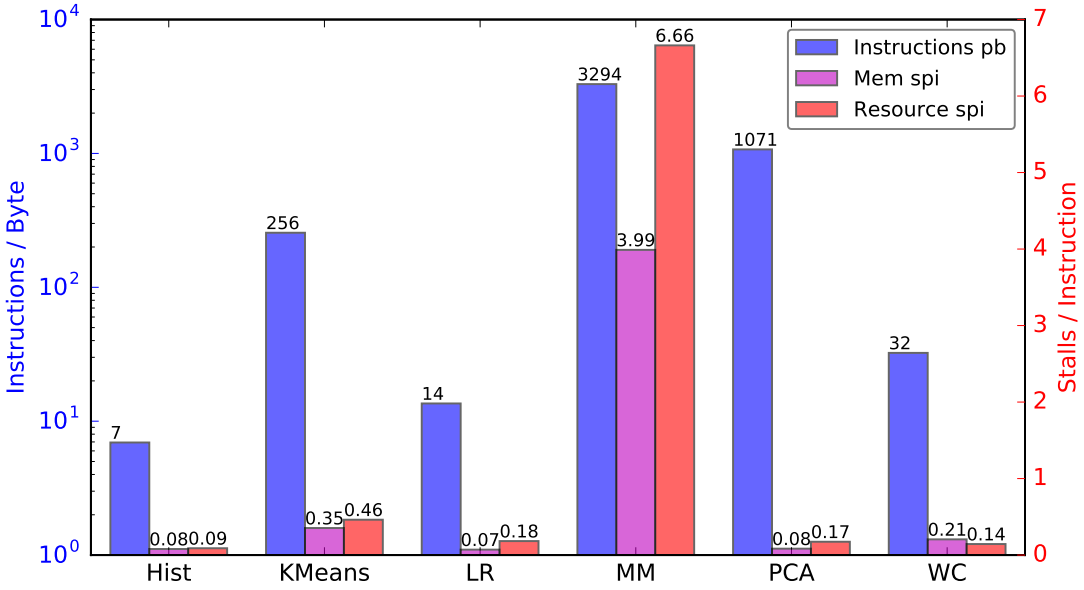


FIGURE 4.6: Instructions per byte, Resource Stalls per Instruction and Memory Stalls per Instructions of Phoenix++ with the Default Containers

that Histogram and Linear Regression are not well-fitted for our runtime, as they have very light workload and relatively few memory and resource related stalls. KMeans and Matrix Multiply, on the other hand, seem suitable, as their workload is both complex enough and suffers from regular stalled cycles. Finally, PCA and Word Count are neither fully well-fitted nor incompatible with our library, as they have sufficient complexity, but not many stalled cycles per instruction.

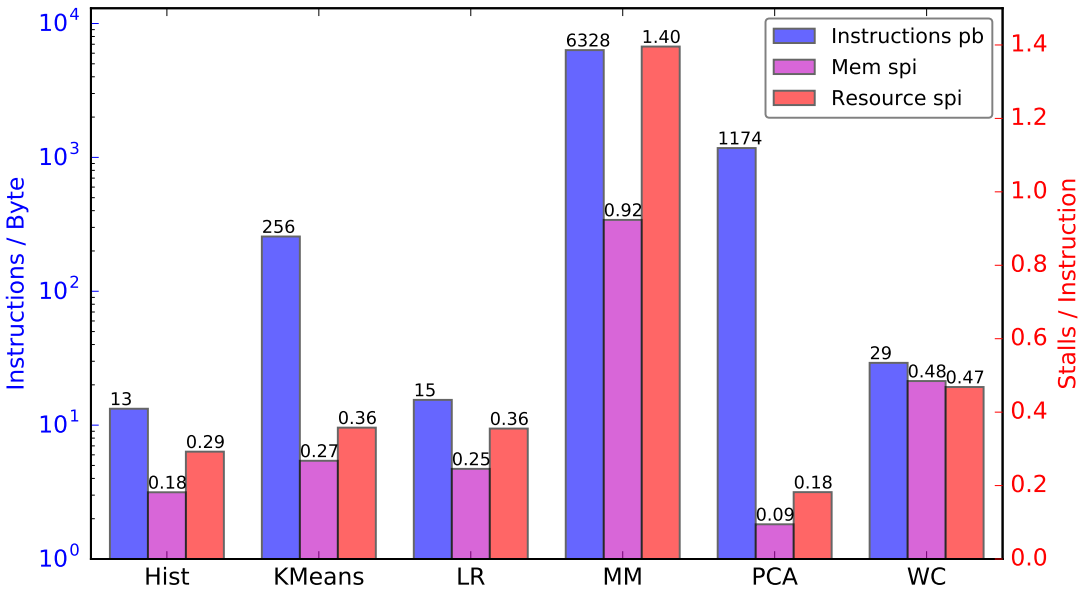


FIGURE 4.7: Instructions per byte, Resource Stalls per Instruction and Memory Stalls per Instructions of Phoenix++, using Hash Containers

Figure 4.7 presents the same metrics, but this time we used either regular or

fixed-size hash table intermediate containers. More particularly, a fixed-size hash was used in Histogram, KMeans, Linear regression and Word Count, and a regular hash for Matrix Multiply and PCA. In general, the use of hash containers, adds hash calculation overheads so we expect see an increase in the IPB metric. KMeans is the only exception to this rule. KMeans uses the intermediate container to update the cluster centroids and count the amount of points that belong to the clusters. The number of clusters is small compared to the number of points. For this specific test-case, we used 500k, 4-D points and 100 clusters. The fixed-hash has 100 entries, meaning that in each resides at most one cluster and the space needed for the container will be around $100 \times 4 \times 4 = 1.6MB$. The default size for the buffer container is also 100, so it has the same space needs. Furthermore, as the points and the clusters are not ordered in any way, there will be little to no data locality during map phase, regardless the container type. Practically, there is no difference between the two containers, and this is why we see no increase in IPB of KMeans. Word Count has a slightly lower IPB, which is reasonable if we consider that in figure 4.6, a hash container was used, so the hash calculation overhead was included, too.

Stalls are increased in all cases, except Matrix Multiply and KMeans. This happens due to the increased computational complexity, that can result in CPU resources being exhausted and/ or inadequate cache behavior, if the fixed-hash size is sub-optimal. In Matrix multiply, we see a significant improvement in stalled cycles when using a regular hash container. To explain this, we have to think of the way Matrix multiply is designed. When a regular array is used, each worker thread allocates a fixed size array, that will be used to combine and store the intermediate key-value pairs. The size of this array in our case is 1M integer elements. However, as every mapper emits keys from a limited range, the hash container is significantly smaller than the fixed-buffer. This improves cache locality, and reduces memory and resource related stalls. For the same reason, KMeans has also slightly improved MSPI and RSPI metrics. In general, a map worker may not encounter points from every cluster, so there is no need to allocate an array container with size equal to the number of clusters.

Looking at figure 4.7, we conclude that KMeans, Matrix Multiply and Word Count are suitable to our implementation, as they have both enough complexity and frequent stalls. Histogram and Linear Regression are not completely incompatible, as they experience memory and resource related stalls often, but still their workload complexity is relatively low. PCA will practically demonstrate the same behavior as with the default array container, because although the workload is sufficient, the amount of resource and memory stalls is very low.

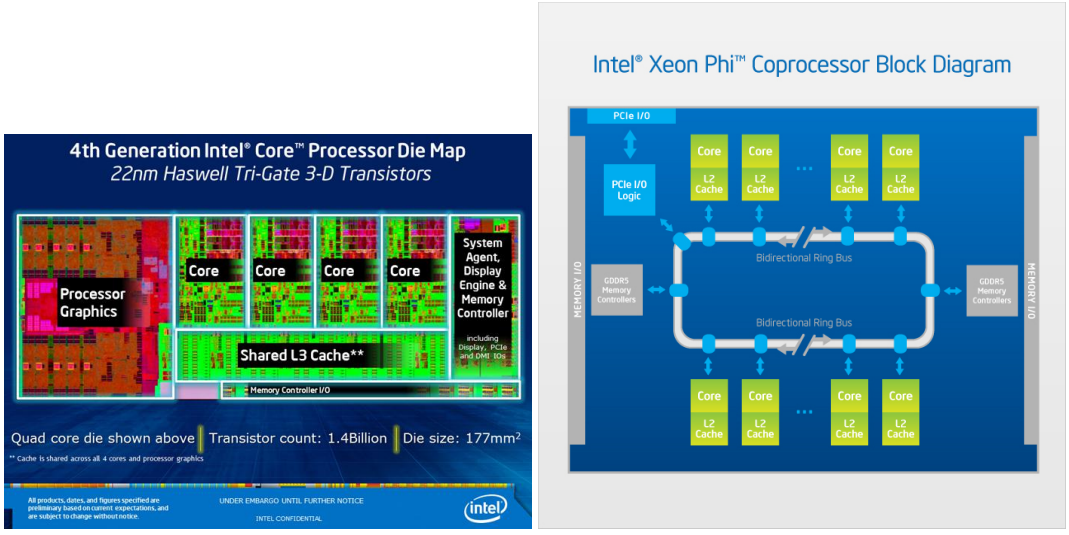
Chapter 5

Results

5.1 Introduction

In the following chapter we evaluate the different aspects of our proposed pipelined implementation. We start by describing the hardware setup that hosted our experiments. Then, we demonstrate the effect of the various programmer tunable variables on run-time. Finally, we benchmark our custom architecture against Phoenix++ [1] and also attempt to characterize the suitability of various workloads to our implementation.

5.2 Experimental Setup



(A) Intel® Haswell Architecture [25] (B) Intel® Xeon Phi™ Architecture [6]

FIGURE 5.1: High Level Architecture of the systems on which we tested our implementation

For our experiments, we used two inherently diverse system. The first is a regular multi-core server platform and the latter is a high performance Co-processor.

Both are sufficient for multi-threaded applications as the are capable of executing concurrently up to 56 and 227 hardware threads respectively.

5.2.1 Intel® Haswell

Haswell [25] is the brand-name for the 4th generation of Intel processor micro-architecture, and it is the successor of the Intel Ivy Bridge [26] micro-architecture. Compared to its predecessor, Haswell achieves an overall 3% performance improvement.

It features 22nm, 3D tri-gate transistors, 14 to 19 stage pipeline, x86 64-bit instruction set and three levels of cache. L1 cache is divided into 32KB instruction and 32KB data cache, L2 cache is 256 KB per core and L3 cache is 35MB per chip.

The platform we used for our measurements is a dual slot Intel® Xeon® E5-2683 v3, clocked at 2GHz up to 3GHz with Intel® Turbo Boost, 14 cores per chip, 2 hardware threads per core, and 64GB of RAM in total. Each of the two slots forms a separate NUMA node of 28 logical cores. During tests, Intel® Turbo Boost was disabled for a more stable and reproducible behavior.

5.2.2 Intel® Xeon Phi™

The Intel® Xeon Phi™ Co-processor is a relatively young product family announced by Intel in late 2012. It is established in the area of High Performance Hardware accelerators, competing, amongst others, with NVIDIA's Tesla-branded [7] product lines. It features up to 61 Intel® Many Integrated Core (MIC) Architecture in-order processor cores.

The Intel® Xeon Phi™ Coprocessor is connected to an Intel Xeon processor, usually referred as the host, via a PCIe bus. It supports two execution modes – native and offload. In native execution, the code is cross-compiled in the host using a special flag and then the binary is copied and executed in the co-processor. In offload mode, OpenMP-like pragmas are used to mark code regions that will be offloaded and executed in the co-processor. The compiler generates two binary versions for the marked regions, one for the host and one for the co-processor. During runtime, if no co-processor is detected, code will be executed on the host. Otherwise, the code will be automatically transfered and executed in the co-processor. One of Intel's Xeon Phi advantages over other accelerators, is it's flexible and simple programming model. Programs written in C/C++ or Fortran and use libraries such as OpenMP, MPI, TBB, Cilk Plus and Intel MKL need no changes to be compiled and run natively on Xeon Phi.

Each of the ~60 cores of the Co-processor is clocked at 1GHz and can fetch and decode instructions from four hardware threads. One Vector floating-point unit

(VPU) is provided for each core, which contains 32 512-bit wide registers, and is capable of performing, 16 32-bit integer, or 16 single-precision, or 8 double-precision packed operations per cycle.

Two levels of cache memory are used. L1 hosts a 32KB instruction and a 32KB data cache with a latency of 1 cycle. L1, as well as L2, caches are 8-way associative with 64 byte cache lines. The latency of the L2 cache is 11 cyles and its size is 512KB per core. A bidirectional, ring bus is used to connect the L2 memory controllers of each core together, thus the total effective L2 cache size is ~32MB per chip.

Our experimental platform consists of an Ivy Bridge E5-2609 v2 Intel® Xeon® Processor with 4 cores at 2.5GHz combined with an Intel® Xeon Phi™ Coprocessor 3120A with 57 cores at 1.1GHz, 22nm lithography, 28.5MB L2 size and 6GB of RAM. All the applications where compiled on Xeon Processor and executed natively on the co-processor.

5.3 Static-Dynamic queue comparison

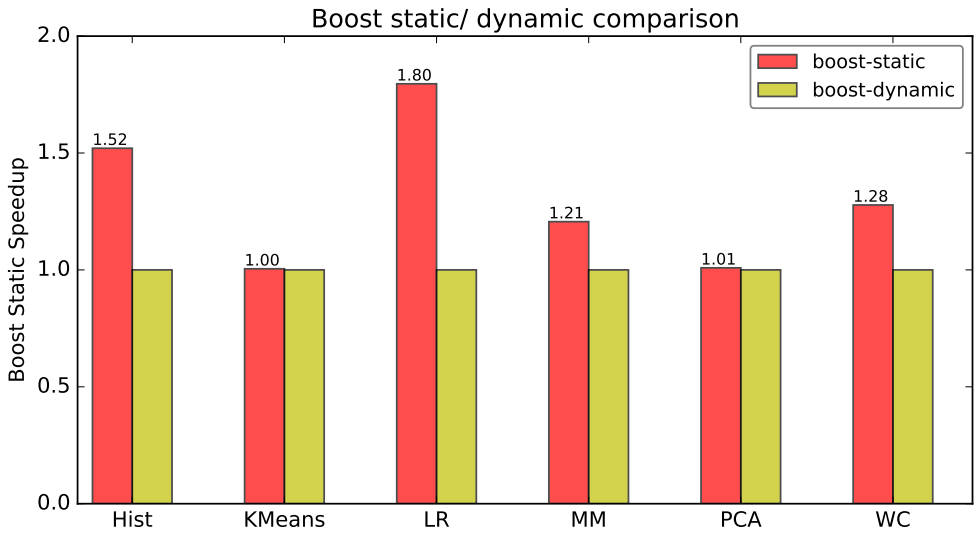


FIGURE 5.2: Run Time Comparison Between Static and Dynamic Buffer Allocation on Haswell

In section 4.3.2, we benchmarked different SPSC queue implementations to find the most efficient. In this section, we compare the static and dynamic boost [22] SPSC queue variants in actual test-cases. Boost-static uses static memory allocation, and as a consequence the size of the buffer needs to be defined at compile time. On the other side, boost-dynamic uses dynamic memory allocation so the definition of the buffer size can be delayed until run time. Except from this discrepancy, their API is exactly the same.

Figures 5.2, 5.3 summarize the test results from Haswell and Intel Xeon Phi respectively. Y-axis displays the normalized time in boost-dynamic. On the X-axis we

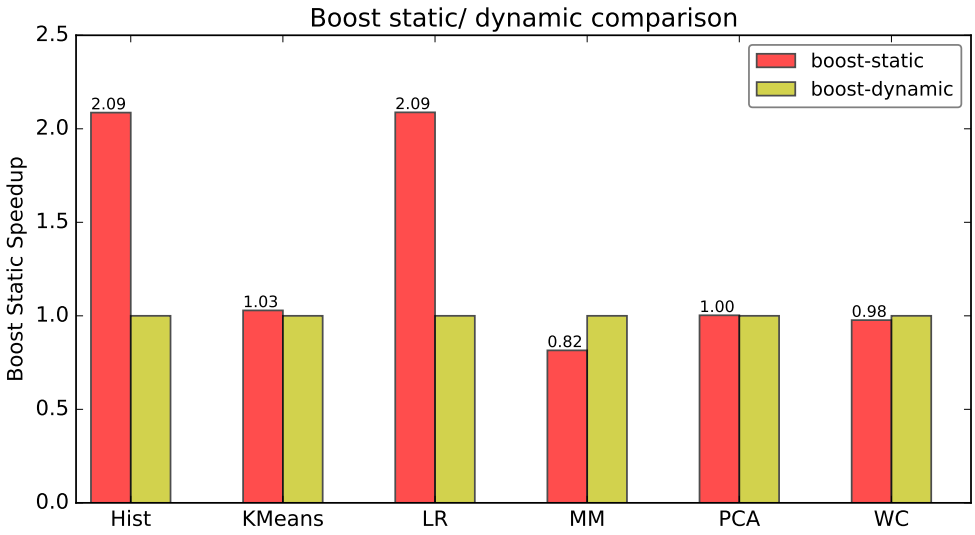


FIGURE 5.3: Run Time Comparison Between Static and Dynamic Buffer Allocation on Intel Xeon Phi

can see all the different applications that were tested. LR stands for Linear Regression, WC for Word Count, Hist for Histogram, MM for Matrix Multiply and PCA for Principal Component Analysis.

In Haswell, static memory allocation improves the run-time up to 44%. In the worst cases, which are KMeans and PCA, the run-time remains unaffected. The results are a little different on Xeon Phi. The MM test-case is 23% slower with static compared to dynamic memory allocation. Hist and LR are 52% improved and the rest applications remain unaffected.

5.4 Different policies comparison

This section benchmarks the efficiency of the different Thread-to-CPU binding policies that we implemented and presented in section 4.4. Figures 5.4, 5.5 summarize the results taken from Haswell and Xeon Phi.

In general, Push policy is outperformed by Fill policy, but also Dedicated in most cases. While on Haswell, Fill performs clearly better than Dedicated policy, on Phi the two policies perform equally. In order to explain this fact, we have to take into account the memory-hierarchy of the two systems. Our Haswell setup is composed of two NUMA nodes, and thus inter-thread communication and placement has to be considered to get the maximum performance. On average, we see an 1.43x speedup, with a peak value at 2.33x and 2.17x on Linear regression and Histogram respectively. On the contrary, Intel Xeon Phi is a single UMA node and due to the bidirectional ring, that inter-connects the memory controllers of all cores, the L2 caches are coherent and almost instantaneously accessible. As a result, the

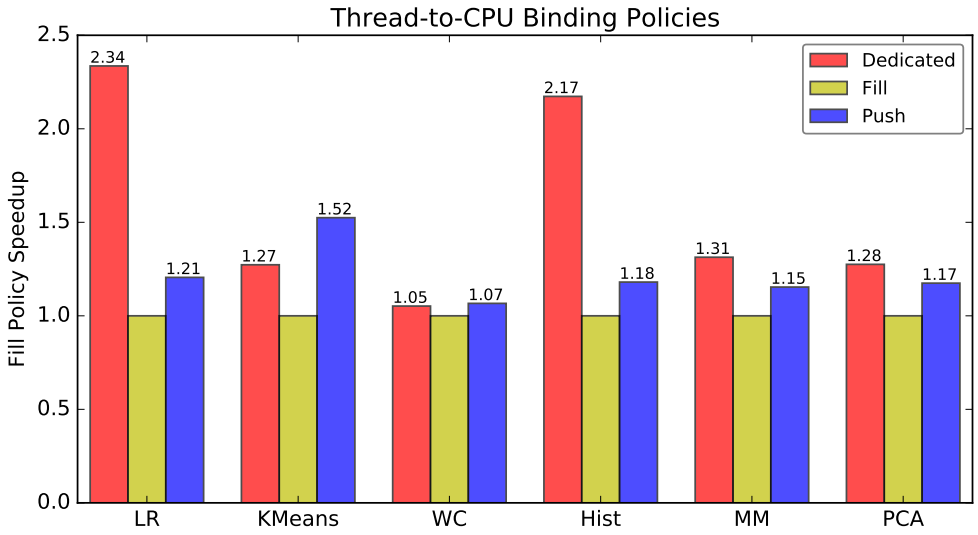


FIGURE 5.4: Run Time Comparison of Different Thread-to-CPU Binding Policies on Haswell

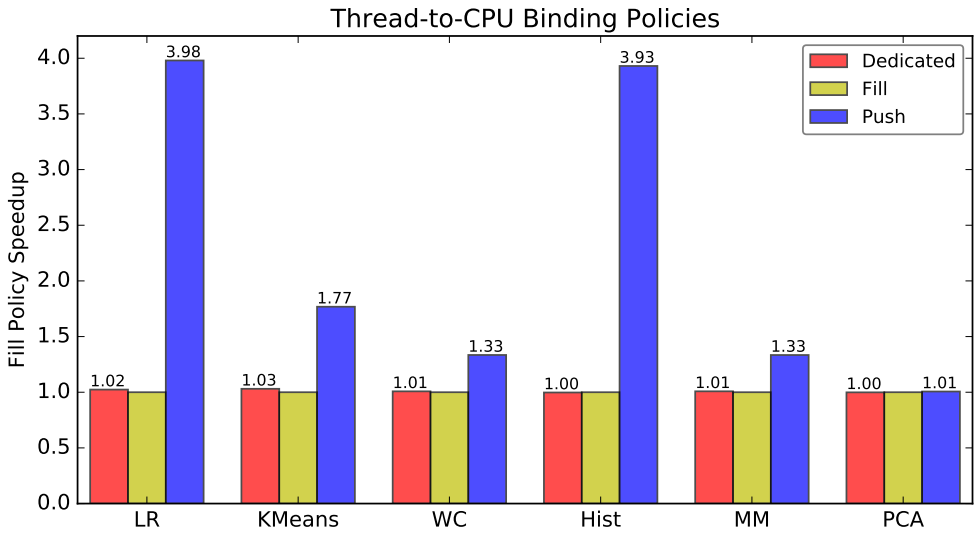


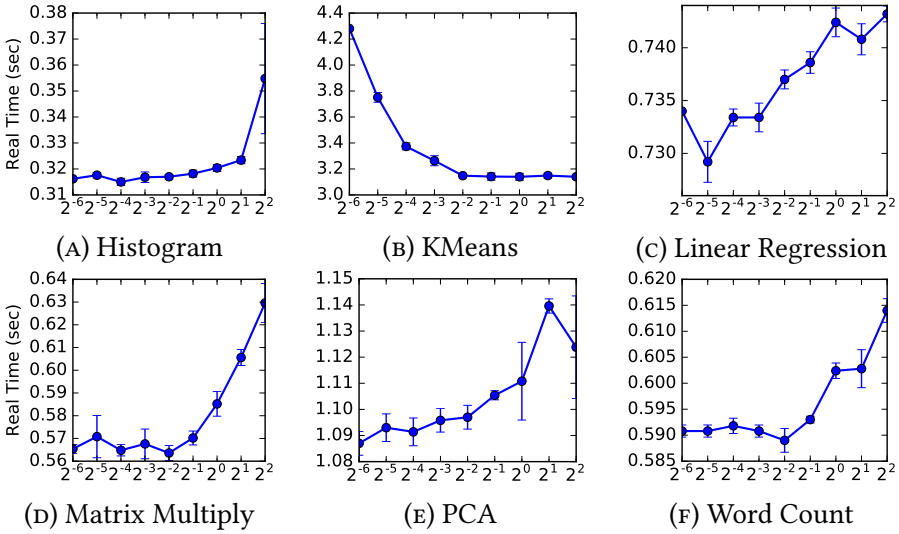
FIGURE 5.5: Run Time Comparison of Different Thread-to-CPU Binding Policies on Intel Xeon Phi

performance gain due to communication-aware thread placement is very limited, i.e. 1–3%.

Nevertheless, it is safe to use Fill policy over Dedicated and Push, as it performs equally or better in all test-cases.

5.5 Different chunk size

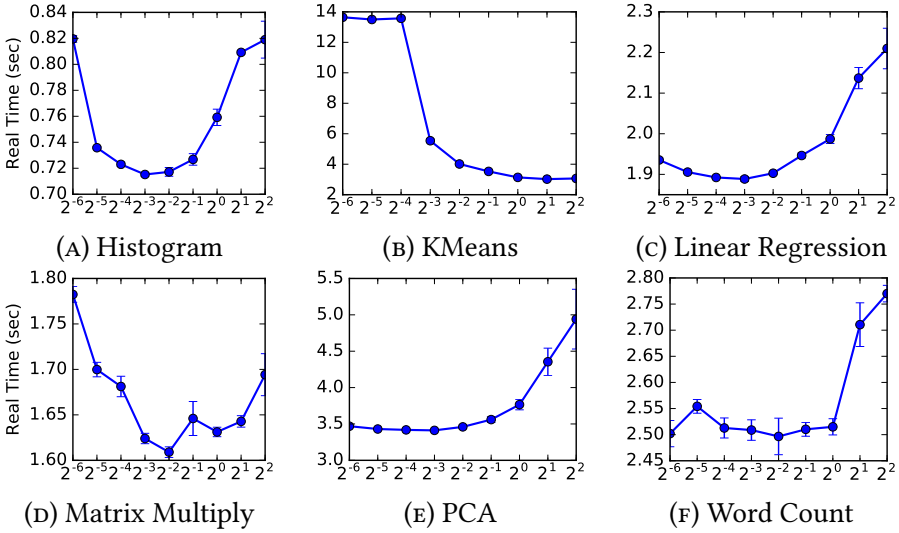
Chunk size is one of the programmer configurable variables that we described in section 4.5.1. It factors the actual task size and it also affects the number of Map tasks as calculated by equation 4.2. Figure 5.6 shows the effect of chunk size in the run time on Haswell, while figure 5.7 shows the effects of chunk size on Xeon Phi.



(A) Histogram (B) KMeans (C) Linear Regression

(D) Matrix Multiply (E) PCA (F) Word Count

FIGURE 5.6: Chunk Size Effect on Haswell



(A) Histogram (B) KMeans (C) Linear Regression

(D) Matrix Multiply (E) PCA (F) Word Count

FIGURE 5.7: Chunk Size Effect on Xeon Phi

As a common rule, we can conclude that applications profit from around 4 to 8 times smaller chunk sizes. The advantages of smaller chunk sizes is that they fit more comfortably in cache memory and lead to better load balancing due to higher number of map tasks. On average, on Xeon Phi there is a 4.3% improvement with optimal chunk size, compared to the default, and on Haswell the average improvement is 1.9%.

5.6 Different buffer size

A set of experiments where conducted in order to find the optimal buffer size for every application. The results are presented in Figures 5.8 and 5.9. We observe that the run-time is heavily affected by the buffer size. More specifically, very big sizes

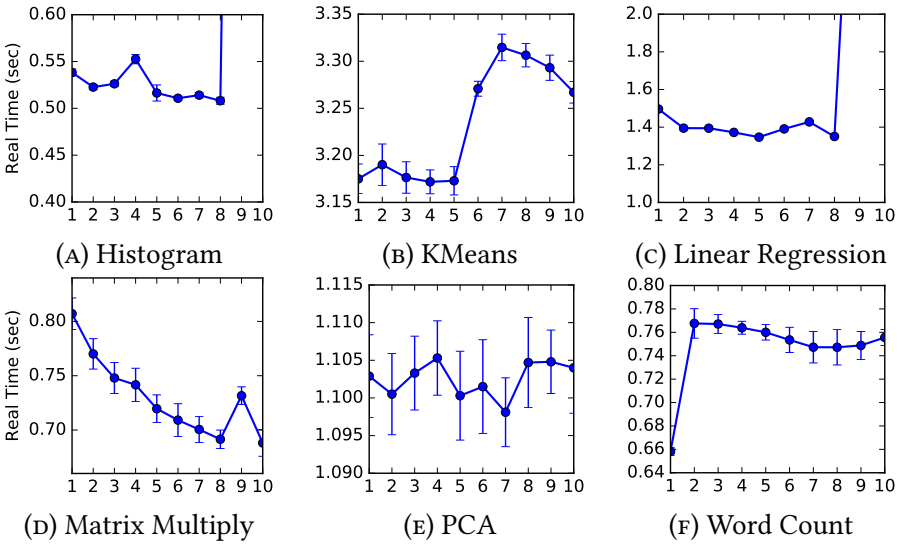


FIGURE 5.8: Buffer Size Effect on Haswell

can result in bad memory locality while very small buffer sizes can be often full, thus the producers will have to sleep, as explained in section 4.3.3. An adequate compromise is achieved, for sizes around 5–6 thousand elements.

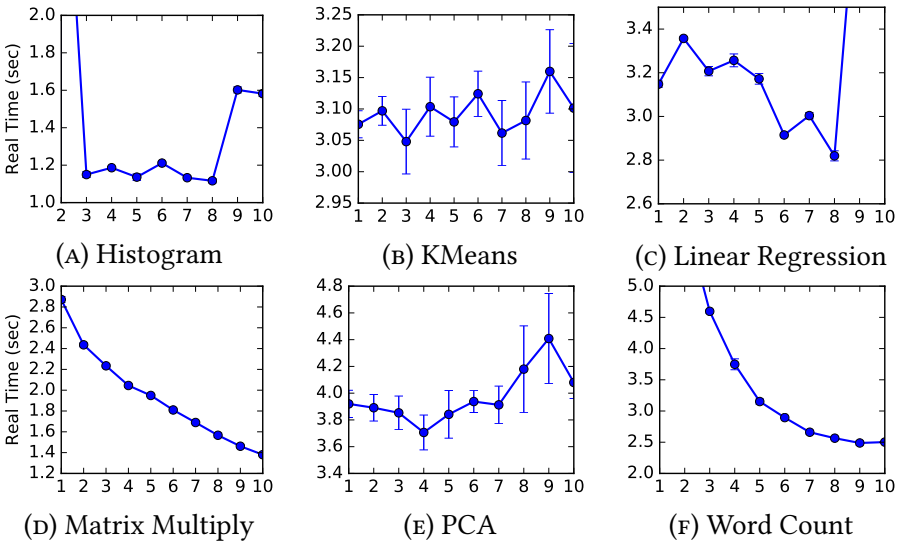


FIGURE 5.9: Buffer Size Effect on Xeon Phi

5.7 Different batch size

As we described in section 4.3, one of the reasons we chose boost was the rich API that it offers. Building on top of it, we implemented a batched, data-copy-free consume method. It takes as an argument a functor `f` and an integer `batch_size`. If there are enough elements in the queue, the functor `f` is applied to `batch_size`

elements. Performing operations in batches takes advantage of cache locality, reduces contention on queue control variables, and results in improved performance. The effect of batched reads is depicted in figure 5.10 for the Haswell system and in figure 5.11 for Intel® Xeon Phi™. We see an impressive speedup of more than 10X for Histogram and Linear Regression on Xeon Phi, and a fair speedup up to 3X for Histogram, Linear Regression and Matrix Multiply on Haswell.

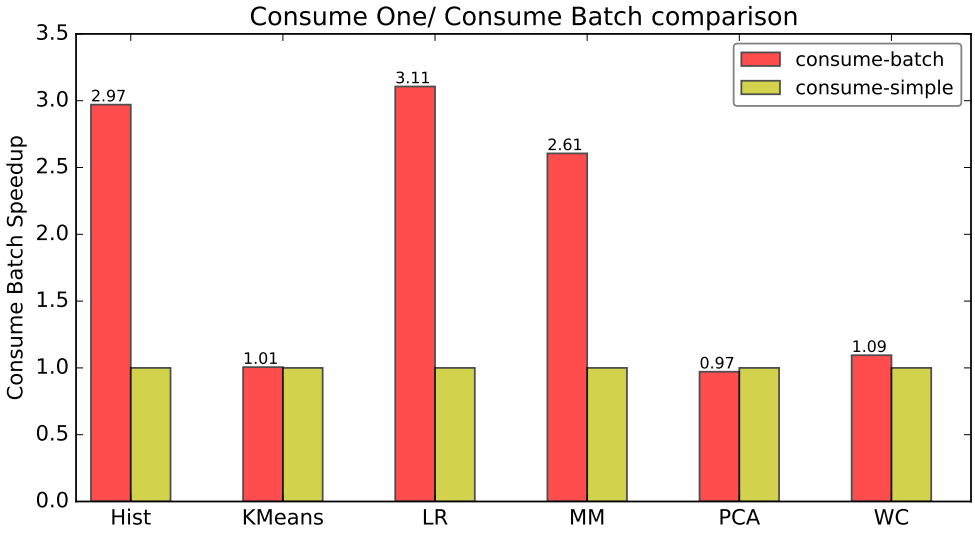


FIGURE 5.10: Run Time Comparison of Simple and Batched Reads on Haswell

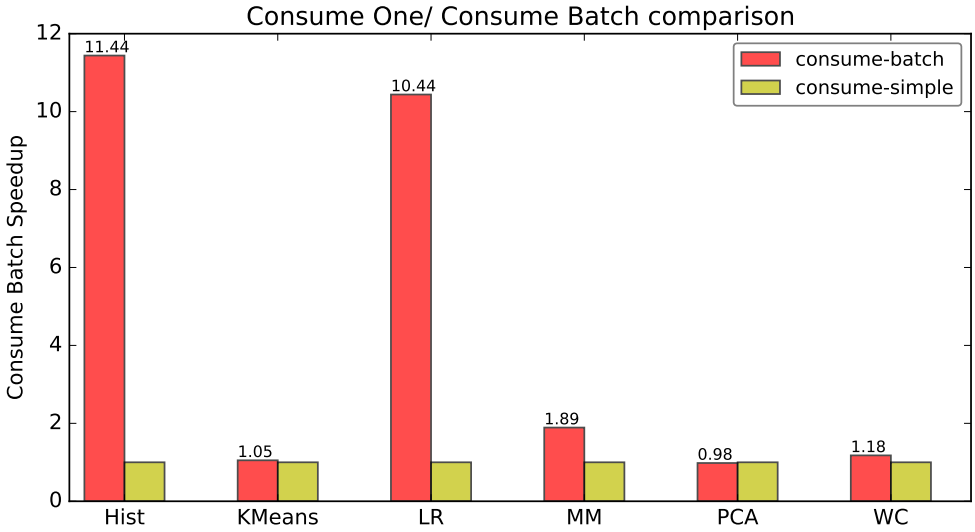
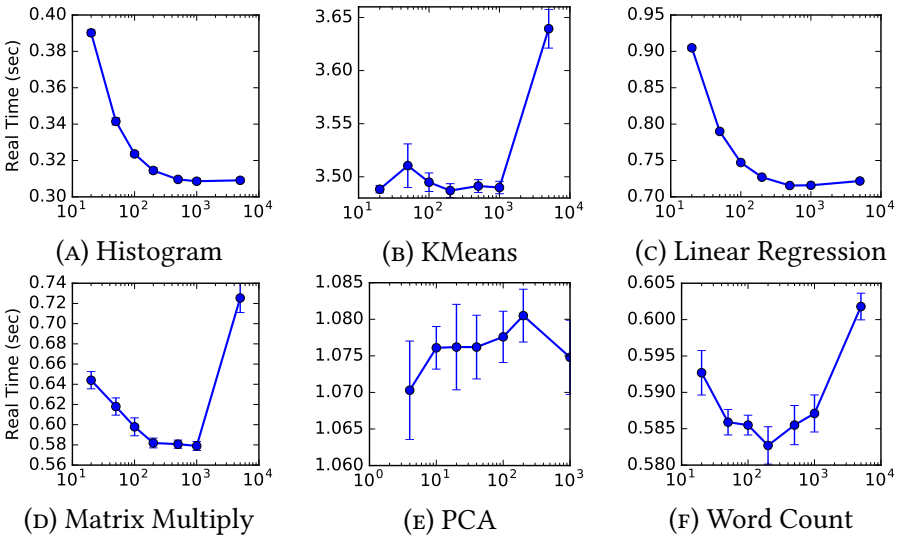


FIGURE 5.11: Run Time Comparison of Simple and Batched Reads on Intel Xeon Phi

Batch size, defines the size in elements of every consume operation. Bigger batch sizes may increase combiners' idle time, but improve spatial locality when significant amount of data are processed contiguously. Figure 5.12 summarizes the effect of different batch sizes, on each of the six target applications, on Haswell.

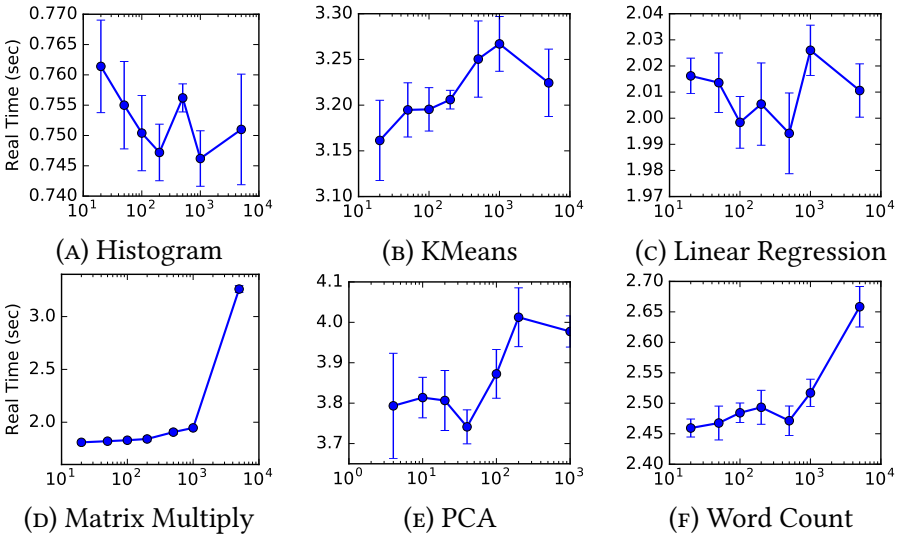
The buffer size was 10k elements for every application, except PCA, which had 2k buffer size. We can see that almost all applications profit from 200 or 500 elements per batch. Similar results can be noticed in figure 5.13, which present the effects of batch size on Xeon Phi, for each application.



(A) Histogram (B) KMeans (C) Linear Regression

(D) Matrix Multiply (E) PCA (F) Word Count

FIGURE 5.12: Batch Size Effect on Haswell



(A) Histogram (B) KMeans (C) Linear Regression

(D) Matrix Multiply (E) PCA (F) Word Count

FIGURE 5.13: Batch Size Effect on Xeon Phi

We can safely assume that 20–50 batches per buffer offer a good compromise between combiners idleness, caused by substantial batch sizes, and poor data locality, induced by limited, consecutive data, processing.

5.8 Performance Evaluation

In this section we evaluate the run time performance of our Pipelined MapReduce implementation against Phoenix++. All the benchmarks were conducted on both our available platforms. For better accuracy, the average of 20 runs was used, and the maximum reported standard deviation was ~1%.

5.8.1 Intel Haswell Platform

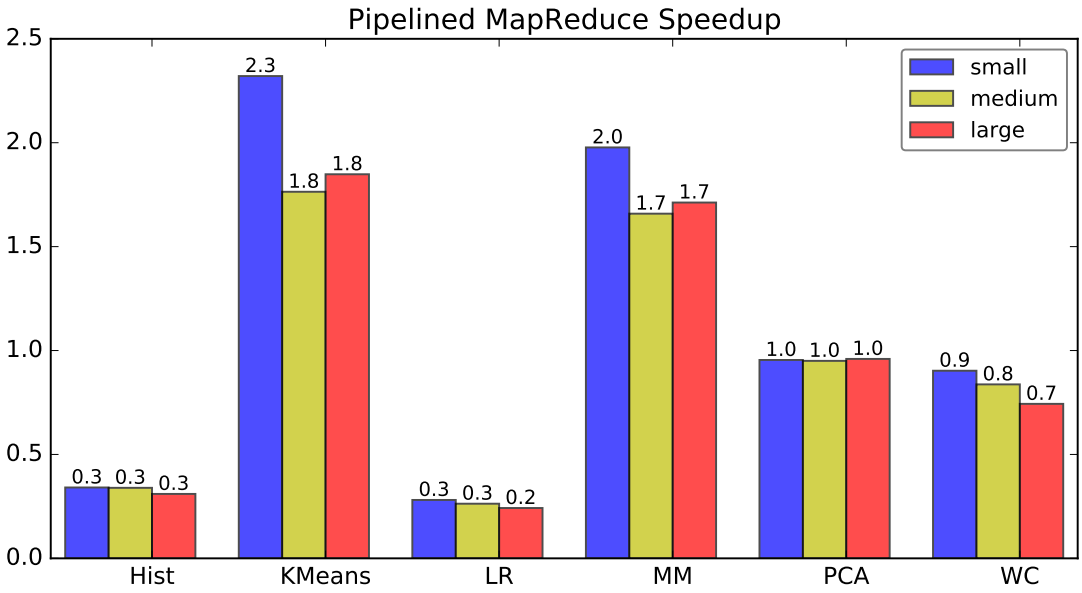


FIGURE 5.14: Pipelined Phoenix Speedup over Phoenix++ with Default Containers in Haswell

Figure 5.14, shows the normalized to Phoenix++ execution time for variable input sizes, when using the default intermediate container for every application. The intermediate container, as described in section 3.3.4, is the data structure that holds the intermediate key-value pairs. The default container for all applications, except Word Count is an array container, because the number of possible keys is either known a priory, or is bounded. Word Count uses a hash container by default, as it is suitable for storing string keys. KMeans and Matrix Multiply profit from our architecture and demonstrate a speedup of 1.95x and 1.77x respectively. PCA performs equally with Phoenix++ and Word Count is slightly slowed down by 21.6%. However, Histogram and Linear Regression are clearly outperformed by a factor of 3x and 3.8x respectively. This due to the very light map and combine workload, causing the overhead induced by the shared-queue to dominate the run-time.

In figure 5.15, we used different intermediate data containers. For Histogram, KMeans, Linear Regression and Word Count we use a fixed-size Hash as intermediate container and for Matrix Multiply and PCA we use a regular Hash container.

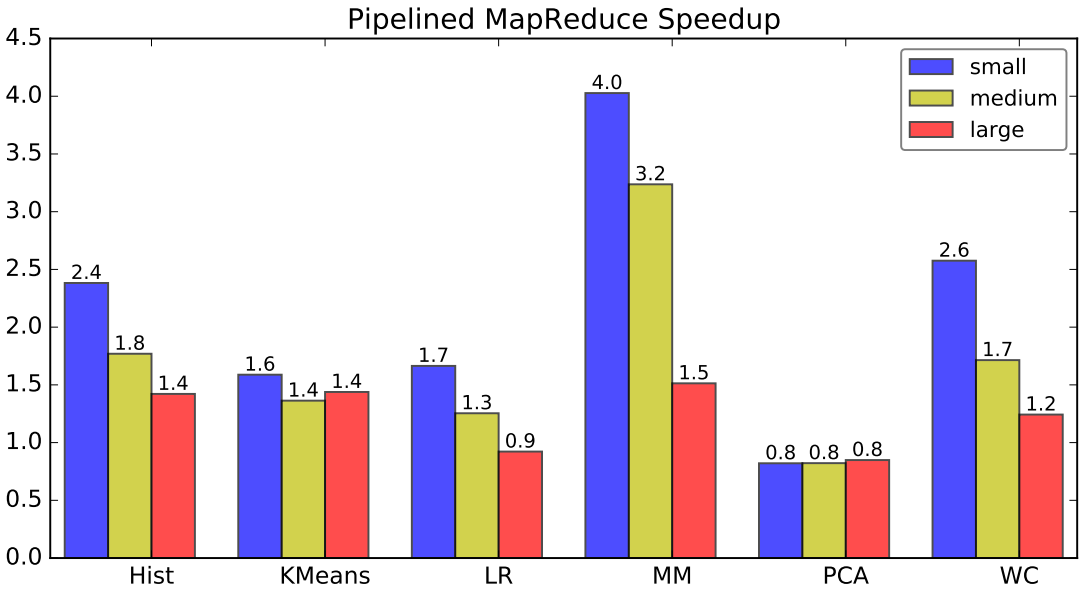


FIGURE 5.15: Pipelined Phoenix Speedup over Phoenix++ with Hash Containers in Haswell

This modification affects the combine workload, as a hash value must computed and memory has to be allocated every time a new element is stored in the container. The outcome is that Pipelined MapReduce outperforms Phoenix++ in 5 out of 6 applications. Matrix Multiply exhibits a 2.46x average speedup, Histogram and Word Count a speedup of 1.77x and 1.69x respectively, KMeans is 1.46x times faster on average and Linear Regression 1.21x. PCA still performs 20.4% worse than Phoenix++.

Application	Small	Medium	Large
Hist	400MB	800MB	1.6GB
KMeans	P:100K, C:100, D:4	P:200K,-,-	P:500K,-,-
LR	400MB	800MB	1.6GB
MM	500x500	800x800	1000x1000
PCA	R:2K, C:2K	R:3K, C:3K	R:4K, C:4K
WC	200MB	400MB	1GB

TABLE 5.1: Input sizes used in the Haswell system

5.8.2 Intel® Xeon Phi™ Co-Processor

Figure 5.16 summarizes the normalized to Phoenix++ time with various input sizes, using the default intermediate key-value containers. The result are comparable to that of Haswell. Word Count performs better by 1.59x on average, KMeans by 2.8x and Matrix Multiply by 1.52x. PCA, again, performs equally, while Histogram and Linear Regression are outperformed by Phoenix++ by 2.84x and 2.87x on average, respectively. The reason is their extremely light workload, that lets the queue overhead to overshadow the Map - Combine decoupling advantageous effects.

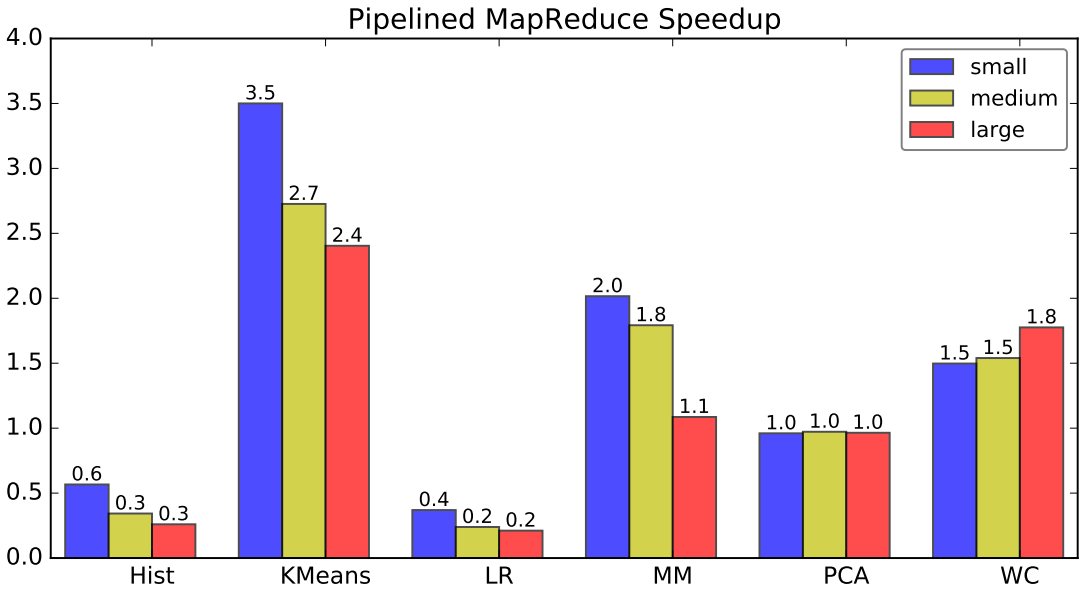


FIGURE 5.16: Pipelined Phoenix Speedup over Phoenix++ with Default Containers in Intel Xeon Phi

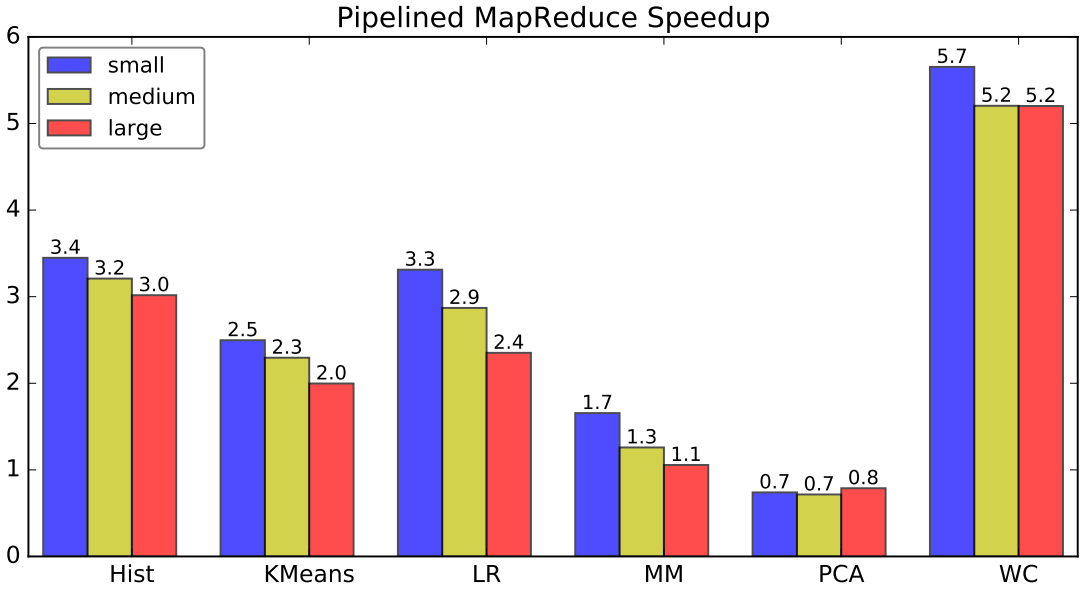


FIGURE 5.17: Pipelined Phoenix Speedup over Phoenix++ with Hash Containers in Intel Xeon Phi

Application	Small	Medium	Large
Hist	200MB	400MB	800MB
KMeans	P:50K, C:100, D:4	P:100K,-,-	P:200K,-,-
LR	200MB	400MB	800MB
MM	300x300	500x500	800x800
PCA	R:2K, C:2K	R:3K, C:3K	R:4K, C:4K
WC	200MB	400MB	600MB

TABLE 5.2: Input sizes used in the Xeon Phi system

In Figure 5.17, we used fixed-size hash containers for Histogram, KMeans, Linear Regression and Word Count, and regular Hash container for Matrix Multiply and PCA. Similarly to Haswell respective benchmarks, the behavior of Pipelined MapReduce is favored in comparison to Phoenix++. Our implementation performs better in 5 out of 6 test-cases and worse only in PCA, by 34% on average. Histogram is faster by 3.2x, Word Count by 5.34x, Linear Regression by 2.79x, Kmeans by 2.24x and Matrix Multiply by 1.28x on average.

Chapter 6

Conclusions and Future Work

6.1 Conclusions

This thesis presents Pipelined MapReduce, an alternative to the conventional MapReduce Runtime for shared-memory systems. Pipelined MapReduce, decouples Map and Combine tasks into separate, independent phases. As data need to be communicated from Map to Combine, we introduced a high performance concurrent data-structure that allows Mappers and Combiners to operate concurrently in a Pipelined fashion. Our choice for this data-structure is a Single-Producer, Single-Consumer, wait-free, concurrent, fixed size, circular buffer. After thorough benchmarking, we found that boost's [22] `spsc_queue`, with static memory allocation is the most efficient and well suited to our needs implementation. Building on top of its rich API, we enhanced it with batched, copy-free data consuming and we added a subtle sleep interval between every, write on full buffer, retrial. Apart from the shared queue implementation, vital role for the performance of our architecture, plays the overhead of inter-thread communication. To minimize it, we implemented a thread-to-CPU binding policy with awareness of the producers-consumers communication pattern.

We used Phoenix++ [1], a state-of-the-art MapReduce library, as the basis of our architecture. While developing our framework, we made sure that the user's API has remained untouched, all our modifications affected the MapReduce Library back-end and are completely transparent to the applications' code.

Finally, we evaluated the performance of Pipelined MapReduce against Phoenix++ into two platforms. The first is composed of two Intel® Xeon generation Haswell nodes, with 14-cores/ 28-threads at 2GHz, and the later is an Intel® Xeon Phi™, with 57-cores/ 228-threads at 1.1GHz, Co-processor. The target applications were six widely used algorithms from the domains of enterprise computing (Word Count), scientific computing (Matrix Multiply), artificial intelligence (KMeans, PCA, Linear Regression) and image processing (Histogram). Using the default, intermediate data containers, on Haswell, our architecture recorded a speedup in two applications (Kmeans, Matrix Multiply), a slowdown in two others (Hist, Liner Regression)

and performed equally in PCA and WC. Similarly, on Intel® Xeon Phi™, our implementation performed better in three applications (KMeans, Matrix Multiply and WC), equally in PCA and worse in Histogram and Linear Regression. However, when changing the intermediate container to fixed hash, the results were impressive, with our architecture achieving a speedup, up to 5.7x, in all applications except PCA, which was 20% slower, both on Haswell and Xeon Phi.

Our Pipelined MapReduce implementation managed to exploit the inefficiency of the traditional architecture that requires serialization of Map and Combine tasks. Nevertheless, we observed that test-cases like Histogram and Linear Regression are not well-fitted for our alternative MapReduce runtime. We used three metrics, derived from PMU counters, to categorize our target applications in suitable and non-suitable with our implementation. We conclude that, *applications with sufficient amount of work per input element, that experience frequent resource-related or memory-related stalls are perfect candidates for our fully-decoupled architecture.*

6.2 Future Work

Pipelined MapReduce proved that there is still space for improvement in the shared-memory implementation of MapReduce. In this section, we underline the incompetencies of our framework and propose ideas for future work and further optimization.

- Four, dynamically defined parameters, read batch size, write on full buffer sleep interval, mappers to combiners ratio and fixed-buffer size, need proper adjustment for an optimized execution time. Due to their interference, defining the optimal values can be tedious. Adding auto-tune capability to our framework would be of great benefit for the users as it would free them from this burdensome task.

- For the time being, our framework supports only one-to-one or many-to-one mappers-combiners correspondence. We took this design decision based on the fact that combine task is usually lighter than map task, so one combiner thread is sufficient to consume the data produced by one or more mapper threads. However, if the combine task is significantly heavier than map, one mapper could feed multiple combiners. In the most generic case, many-to-many correspondence could be implemented, but this requires the presence of multiple-producers, multiple-consumers queue which has a considerable overhead compared to single-producer, single-consumer queues.

- In order to support memory and communication pattern aware thread-to-CPU binding, we have chosen to statically assign mappers to combiners. However, this can result in sub-optimal load balancing, and also, makes the mappers to combiners ratio very essential for good performance. A more dynamic approach could tackle these issues. The execution of Map phase could start as normal, with workers being dynamically assigned map tasks, and writing the intermediate data in a private buffer. Every time the buffer gets full, a combine task will be created and added to a high-priority queue. The first worker thread that completes its map task, will be assigned and execute the combine task. Though, this approach has the advantages of dynamic load balancing and automatic mappers-to-combiners ratio adjustment, it lacks memory and communication pattern awareness.

Appendix A

Source Code

A.1 Source code repository

The source code of Pipelined MapReduce can be found at:
https://github.com/kiliakis/pipelined-mapreduce.

A.2 Original License

As the code was developed based on Phoenix [4], it is distributed with Phoenix original BSD License. *The copyright is held by Stanford University. Phoenix is provided "as is" without any guarantees of any kind.*

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

- Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
- Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
- Neither the name of Stanford University nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL STANFORD UNIVERSITY BE LIABLE FOR

ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Appendix B

Producer Consumer Problem

B.1 Solutions to Producer Consumer Problem

```
1  int buffer[N];
2  int read_idx = 0, write_idx = 0, count = 0;
3
4  void enqueue(int item) {
5      while (count == N) ; // spin
6      buffer[write_idx] = item;
7      write_idx++;
8      if (write_idx == N)
9          write_idx = 0;
10     count++;
11 }
12
13 int dequeue() {
14     while (count == 0) ; // spin
15     int item = buffer[read_idx];
16     read_idx++;
17     if(read_idx == N)
18         read_idx = 0;
19     count--;
20     return item;
21 }
```

Listing B.1: Naive enqueue and dequeue functions

The producer-consumer problem is one of the classic examples of multi-process synchronization. In this example, two processes, one producer and one consumer, are sharing a common fixed-size buffer. The producer process, generates data and stores them in the buffer. Concurrently, the consumer process is removing data from the buffer. If no proper synchronization is applied, consumer will try to remove data from an empty buffer and/or producer will try add data to a full buffer. Assuming that a cyclic, fixed-size buffer, a naive implementation of the enqueue

function, called by the producer, and the dequeue function called by the consumer is given in listing addref.

```
1  int buffer[N];
2  int read_idx = 0, write_idx = 0, count = 0;
3  mutex_t lock;
4  mutex_init(&lock);
5
6  void enqueue(int item) {
7      while (count == N) ; // spin
8      buffer[write_idx] = item;
9      write_idx++;
10     if (write_idx == N)
11         write_idx = 0;
12     mutex_lock(&lock);
13     count++;
14     mutex_unlock(&lock);
15 }
16
17 int dequeue() {
18     while (count == 0) ; // spin
19     int item = buffer[read_idx];
20     read_idx++;
21     if(read_idx == N)
22         read_idx = 0;
23     mutex_lock(&lock);
24     count--;
25     mutex_unlock(&lock);
26     return item;
27 }
```

Listing B.2: Enqueue and dequeue functions

Obviously, the code in listing B.1 is pathogenic because of the concurrent access to the shared variable `count`. On the consumer's side, a lost update on the `count` variable could result in reading the same element twice or skipping one. Respectively, a lost update on the producer's side could result in overwriting an element. The code in listing B.2 deals with this problem using a lock to ensure explicit access on the shared variable `count`.

Other synchronization mechanisms such as semaphores, atomic operations and monitors could have been used. However, L. Lamport in his paper *Specifying concurrent program modules* [19], proved that, under a sequential consistency memory

model [18] a concurrent, wait-free implementation of the single-producer/ single-consumer circular buffer is possible, without using explicit synchronization primitives. A wait-free algorithm is guaranteed to complete after a finite number of steps, regardless of the timing behavior of other operations. With minimal modifications to Lamport's original implementation, it is possible to extend this lemma for other weaker consistency models. The proof of correctness of these lemmas are out of the scope of this thesis and will not be discussed further. The reader can study them in the relative articles [16], [17], [18], [19], [43].

Bibliography

[1] Justin Talbot, Richard M Yoo, and Christos Kozyrakis. "Phoenix++: modular MapReduce for shared-memory systems". In: *Proceedings of the second international workshop on MapReduce and its applications*. ACM. 2011, pp. 9–16.

[2] Jeffrey Dean and Sanjay Ghemawat. "MapReduce: simplified data processing on large clusters". In: *Communications of the ACM* 51.1 (2008), pp. 107–113.

[3] Apache Hadoop. *Hadoop*. 2009.

[4] Colby Ranger et al. "Evaluating mapreduce for multi-core and multiprocessor systems". In: *High Performance Computer Architecture, 2007. HPCA 2007. IEEE 13th International Symposium on*. Ieee. 2007, pp. 13–24.

[5] Yandong Mao, Robert Morris, and M Frans Kaashoek. "Optimizing MapReduce for multicore architectures". In: *Computer Science and Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Tech. Rep.* Citeseer. 2010.

[6] George Chrysos. "Intel® Xeon Phi™ coprocessor-the architecture". In: *Intel Whitepaper* 176 (2014).

[7] Erik Lindholm et al. "NVIDIA Tesla: A unified graphics and computing architecture". In: *IEEE micro* 28.2 (2008).

[8] *Events for Intel® Microarchitecture Code Name Haswell*. URL: https://software.intel.com/en-us/node/589935 (visited on 01/19/2017).

[9] *Events for Intel® Xeon Phi™ Coprocessor (Code Name: Knights Corner)*. URL: https://software.intel.com/en-us/node/589941 (visited on 01/19/2017).

[10] James O Coplien. "Curiously recurring template patterns". In: *C++ Report* 7.2 (1995), pp. 24–27.

[11] Richard M Yoo, Anthony Romano, and Christos Kozyrakis. "Phoenix rebirth: Scalable MapReduce on a large-scale shared-memory system". In: *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*. IEEE. 2009, pp. 198–207.

[12] Mian Lu et al. “Optimizing the mapreduce framework on intel xeon phi coprocessor”. In: *Big Data, 2013 IEEE International Conference on*. IEEE. 2013, pp. 125–130.

[13] Rong Chen, Haibo Chen, and Binyu Zang. “Tiled-MapReduce: optimizing resource usages of data-parallel applications on multicore with tiling”. In: *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*. ACM. 2010, pp. 523–534.

[14] Patrick PC Lee, Tian Bu, and Girish Chandranmenon. “A lock-free, cache-efficient shared ring buffer for multi-core architectures”. In: *Proceedings of the 5th ACM/IEEE Symposium on Architectures for Networking and Communications Systems*. ACM. 2009, pp. 78–79.

[15] John Giacomoni, Tipp Moseley, and Manish Vachharajani. “FastForward for efficient pipeline parallelism: a cache-optimized concurrent lock-free queue”. In: *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*. ACM. 2008, pp. 43–52.

[16] Leslie Lamport. “Concurrent reading and writing”. In: *Communications of the ACM* 20.11 (1977), pp. 806–811.

[17] Leslie Lamport. “Proving the correctness of multiprocess programs”. In: *IEEE transactions on software engineering* 2 (1977), pp. 125–143.

[18] Leslie Lamport. “Correctly Executes Multiprocess Program”. In: *IEEE transactions on computers* 28.9 (1979).

[19] Leslie Lamport. “Specifying concurrent program modules”. In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 5.2 (1983), pp. 190–222.

[20] Massimo Torquati. “Single-producer/single-consumer queues on shared cache multi-core systems”. In: *arXiv preprint arXiv:1012.1824* (2010).

[21] Kjell Kod. *Lock-Free Single-Producer - Single Consumer Circular Queue*. 2014. URL: https://www.codeproject.com/articles/43510/lock-free-single-producer-single-consumer-circular (visited on 01/19/2017).

[22] Tim Blechmann. *boost::lockfree::spsc_queue*. 2013. URL: http://www.boost.org/doc/libs/1_61_0/doc/html/boost/lockfree/spsc_queue.html (visited on 01/19/2017).

[23] Cameron. *A Fast Lock-Free Queue for C++*. 2014. URL: http://moodycamel.com/blog/2013/a-fast-lock-free-queue-for-c++#benchmarks (visited on 01/19/2017).

[24] Rama Kishan Malladi. "Using Intel® VTune™ Performance Analyzer Events/Ratios & Optimizing Applications". In: *http:/software.intel.com* (2009).

[25] Per Hammarlund et al. "Haswell: The fourth-generation intel core processor". In: *IEEE Micro* 34.2 (2014), pp. 6–20.

[26] Satish Damaraju et al. "A 22nm IA multi-CPU and GPU system-on-chip". In: *Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2012 IEEE International*. IEEE. 2012, pp. 56–57.

[27] *IBM Big Data*. URL: http://www.ibm.com/big-data/us/en/ (visited on 02/12/2017).

[28] William Gropp, Ewing Lusk, and Anthony Skjellum. *Using MPI: portable parallel programming with the message-passing interface*. Vol. 1. MIT press, 1999.

[29] Leonardo Dagum and Ramesh Menon. "OpenMP: an industry standard API for shared-memory programming". In: *IEEE computational science and engineering* 5.1 (1998), pp. 46–55.

[30] Matteo Frigo, Charles E Leiserson, and Keith H Randall. "The implementation of the Cilk-5 multithreaded language". In: *ACM Sigplan Notices*. Vol. 33. 5. ACM. 1998, pp. 212–223.

[31] Robert D Blumofe et al. *Cilk: An efficient multithreaded runtime system*. Vol. 30. 8. ACM, 1995.

[32] *Intel® Cilk™ Plus*. URL: https://www.cilkplus.org/ (visited on 01/19/2017).

[33] Chuck Pheatt. "Intel® threading building blocks". In: *Journal of Computing Sciences in Colleges* 23.4 (2008), pp. 298–298.

[34] David R Musser, Gillmer J Derge, and Atul Saini. *STL tutorial and reference guide: C++ programming with the standard template library*. Addison-Wesley Professional, 2009.

[35] John E Stone, David Gohara, and Guochun Shi. "OpenCL: A parallel programming standard for heterogeneous computing systems". In: *Computing in science & engineering* 12.3 (2010), pp. 66–73.

[36] *Oracle Locality Groups*. URL: https://docs.oracle.com/cd/E18752_01/html/817-4415/lgroups-2.html (visited on 02/12/2017).

[37] Alexander Heinecke et al. "Design and implementation of the linpack benchmark for single and multi-node systems based on intel® xeon phi coprocessor". In: *Parallel & Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on*. IEEE. 2013, pp. 126–137.

[38] Simon J Pennycook et al. “Exploring simd for molecular dynamics, using intel® xeon® processors and intel® xeon phi coprocessors”. In: *Parallel & Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on*. IEEE. 2013, pp. 1085–1097.

[39] *Stampede-KNL TOP500*. URL: https://www.top500.org/system/178914 (visited on 02/12/2017).

[40] *Tianhe-2 (MilkyWay-2) TOP500*. URL: https://www.top500.org/system/177999 (visited on 02/12/2017).

[41] Josep Torrellas, HS Lam, and John L. Hennessy. “False sharing and spatial locality in multiprocessor caches”. In: *IEEE Transactions on Computers* 43.6 (1994), pp. 651–663.

[42] Tyson Condie et al. “MapReduce online.” In: *Nsdi*. Vol. 10. 4. 2010, p. 20.

[43] Maurice Herlihy and J Eliot B Moss. *Transactional memory: Architectural support for lock-free data structures*. Vol. 21. 2. ACM, 1993.

[44] Bo Hu and Jordan DeLong. *folly::ProducerConsumerQueue*. 2017. URL: https://github.com/facebook/folly/blob/master/folly/docs/ProducerConsumerQueue.md (visited on 01/19/2017).

[45] *Intel® VTune™ Amplifier 2017*. 2017. URL: https://software.intel.com/en-us/intel-vtune-amplifier-xe (visited on 01/19/2017).