ΠΑΝΕΠΙΣΤΗΜΙΟ ΚΡΗΤΗΣ
ΣΧΟΛΗ ΘΕΤΙΚΩΝ ΕΠΙΣΤΗΜΩΝ
ΤΜΗΜΑ ΕΠΙΣΤΗΜΗΣ ΥΠΟΛΟΓΙΣΤΩΝ

ΣΧΕΔΙΑΣΜΟΣ ΚΑΙ ΥΛΟΠΟΙΗΣΗ ΤΟΥ ΜΟΝΤΕΛΟΥ LINDA ΣΕ ΜΗΧΑΝΕΣ ΚΑΤΑΝΕΜΗΜΕΝΗΣ ΜΝΗΜΗΣ

Γιάννης Γ. Καβακλής

Μεταπτυχιακή Εργασία

ΗΡΑΚΛΕΙΟ, ΚΡΗΤΗ
Σεπτέμβριος 1993

ΣΧΕΔΙΑΣΜΟΣ ΚΑΙ ΥΛΟΠΟΙΗΣΗ ΤΟΥ ΜΟΝΤΕΛΟΥ LINDA ΣΕ ΜΗΧΑΝΕΣ ΚΑΤΑΝΕΜΗΜΕΝΗΣ ΜΝΗΜΗΣ

Μεταπτυχιακή εργασία

που υποβλήθηκε στην Επιτροπή Μεταπτυχιακών Σπουδών

του Τμήματος Επιστήμης Υπολογιστών

της Σχολής Θετικών Επιστημών του Πανεπιστημίου Κρήτης

ως μερική απαίτηση για την απόκτηση του

ΔΙΠΛΩΜΑΤΟΣ ΜΕΤΑΠΤΥΧΙΑΚΗΣ ΕΞΕΙΔΙΚΕΥΣΗΣ

Σεπτέμβριος 1993

Συγγραφέας:

Γιάννης Γ. Καβακλής

Τμήμα Επιστήμης Υπολογιστών

Εισηγητική Επιτροπή:

Στέλιος Ορφανουδάκης, Καθηγητής, Επόπτης

Μανόλης Κατεβαίνης, Αναπληρωτής Καθηγητής, Μέλος

Χρήστος Νικολάου, Αναπληρωτής Καθηγητής, Μέλος

Δεκτή:

Πάνος Κωνσταντόπουλος,

Πρόεδρος Επιτροπής Μεταπτυχιακών Σπουδών

Σχεδιασμός και υλοποίηση του μοντέλου Linda σε μηχανές κατανεμημένης μνήμης

Γιάννης Γ. Καβακλής
Μεταπτυχιακή Εργασία
Τμήμα Επιστήμης Υπολογιστών
Πανεπιστήμιο Κρήτης

Περίληψη

Ένας από τους πονοκέφαλους στον κόσμο τον παράλληλων συστημάτων είναι ο προγραμματισμός τους. Αντίθετα με το σειριακό μοντέλο προγραμματισμού, όπου ακολουθίες εντολών μας οδηγούν στο τελικό αποτέλεσμα, στον παράλληλο προγραμματισμό, πολλές διεργασίες αλληλεπιδρούν παράλληλα στα δεδομένα. Ακόμα κι αν υποθέσουμε ότι λύνουμε με επιτυχία το πρόβλημα της εύρεσης του κατάλληλου παράλληλου αλγόριθμου για κάθε πρόβλημα, υπάρχει το πρόβλημα της απόδοσής του στην γλώσσα και το μοντέλο που υποστηρίζει η κάθε μηχανή. Ο προγραμματιστής καλείται να μάθει το προγραμματιστικό περιβάλλον που υποστηρίζει η μηχανή, να λύσει το πρόβλημα της ανάθεσης και της δημιουργίας των διεργασιών στους επεξεργαστές, να καταφέρει να συγχρονίσει τις διάφορες διεργασίες ώστε να λάβει χώρα η ανταλλαγή πληροφορίας και τέλος να βρει πού πρέπει να απευθυνθεί η κάθε διεργασία για να βρει αυτήν την πληροφορία. Επιπλέον σε όλα αυτά πρέπει να επισημάνουμε και το θέμα του portability του προγράμματος, μια και διαφορετικές αρχιτεκτονικές έχουν συνήθως και διαφορετικά πρωτόγονα υπολογιστικά και επικοινωνιακά χαρακτηριστικά (primitives).

Το πρόβλημα που έχει αντιμετωπιστεί μέχρι σήμερα με σχετική επιτυχία είναι το portability, με την δημιουργία βιβλιοθηκών που τις υποστηρίζουν πολλές αρχιτεκτονικές και κρύβουν το στρώμα των calls που υποστηρίζει η κάθε μια.

Ένα σύστημα που προσπαθεί, και σε μεγάλο βαθμό πετυχαίνει, να αντιμετωπίσει συνολικά το πρόβλημα είναι το μοντέλο προγραμματισμού Linda.

Το Linda επιτυγχάνει να αποδεσμεύσει τον προγραμματιστή από το πρόβλημα των χωρικών (πού βρίσκεται το δεδομένο, διεργασία) και χρονικών (πότε πρέπει να γίνει η επικοινωνία και ο συγχρονισμός) αλληλεξαρτήσεων του προγράμματος, ενώ ταυτόχρονα διατίθεται απαράλλαχτο σε πολλές αρχιτεκτονικές.

Το Linda προσφέροντας ένα ασυνήθιστο μοντέλο κοινής μνήμης καταφέρνει να είναι ένα από τα καλύτερα μοντέλα δημιουργίας και συγχρονισμού παράλληλων διεργασιών.

Όλα αυτά βέβαια δεν γίνονται χωρίς κόστος. Το Linda δεν προσπαθεί να πείσει ότι θα παράγει ένα πρόγραμμα πιο γρήγορο από αυτό που θα έκανε ένας έμπειρος προγραμματιστής χρησιμοποιώντας απευθείας τα calls της μηχανής. Εγγυάται όμως ότι ο χρόνος υλοποίησης θα είναι ελάχιστος, η απόδοση πολύ καλή, ενώ ταυτόχρονα το απαιτούμενο υπόβαθρο γνώσεων παράλληλου προγραμματισμού πολύ μικρό.

Το POSYBL είναι μια προσπάθεια υλοποίησης του μοντέλου Linda σε δίκτυο από σταθμούς εργασίας Unix. Εμείς, συνεχίζοντας αυτήν την προσπάθεια αποφασίσαμε να το βελτιώσουμε, πρώτα με την δημιουργία πρωτόκολλων τοπικής (interprocess) και ενδοδικτυακής (internetwork) επικοινωνίας μεταξύ διεργασιών, και στην συνέχεια βελτιώνοντας τους αλγορίθμους αποθήκευσης, ανεύρεσης και κατανομής των δεδομένων. Το αποτέλεσμα αυτής της προσπάθειας λέγεται NPOSYBL.

Χρησιμοποιώντας την πείρα και τα μαθήματα που κερδίσαμε σε αυτήν την διαδικασία, προχωρήσαμε στην δημιουργία του ΙΣΤΟΣ, μιας υλοποίησης Linda σε παράλληλη αρχιτεκτονική, καθώς και τον προσομοιωτή του σε Sun Workstations. Το ΙΣΤΟΣ ξεφεύγει από το στάδιο της πειραματικής υλοποίησης και παρέχει ένα αξιόπιστο περιβάλλον παράλληλου προγραμματισμού που μπορεί να χρησιμοποιηθεί για ερευνητικούς και εκπαιδευτικούς σκοπούς.

Επόπτης: Σ. Ορφανουδάκης,
Καθηγητής του Τμήματος Επιστήμης Υπολογιστών
Πανεπιστημίου Κρήτης.

Design and implementation of the Linda model on distributed memory machines

Yannis Kavaklis
Master of Science Thesis
Department of Computer Science
University of Crete

Abstract

Linda is an easy-to-use language for parallel programming; it is not based on shared variables or message passing, but it uses a novel communication mechanism called the tuple space. It supports (but does not enforce) a programming methodology based on distributed data structures and replicated workers. The goal of this methodology is to release the programmer from thinking in terms of parallel computations and simultaneous events, hence making parallel programming conceptually similar to sequential programming.

POSYBL is a Linda implementation for Unix networks. We decided to extend this work by optimizing the various components of POSYBL.

The first step was to improve the network communication between the various processes that take part in the POSYBL program. Network communication is expensive, and the available protocols that the UNIX operating system offers are not suitable for asynchronous reliable message passing between network nodes. We developed our own protocol which is a subset of the VMTP[12] protocol used in the V[13] distributed kernel.

The second step was to design and implement an efficient protocol for asynchronous message passing between processes on the same node. For this reason we implemented a protocol that uses shared memory to transfer data between processes, and message queues for the synchronization of the shared memory accesses.

The final step was the optimization of the search, distribution and storage of tuples in the tuple space. Regarding storage, a hashing algorithm that uses hashing at two levels was developed. The first level is used to segment the tuple space in orthogonal segments based on the tuple fields. The second level is used for the indexing of tuples in each segment. For the distribution of tuples, statistical information is used to discover the communication pattern of the POSYBL program. This information can be used to avoid unnecessary network traffic. This is achieved by sending the tuples directly to the node that is most likely to search for them.

Using the experience of this design, we developed ISTOS Linda. ISTOS Linda is a Linda implementation on a distributed memory parallel machine based on the transputer architecture and the PARIX operating system. Furthermore, a simulator of ISTOS was developed for Sun Workstations using the Sun LWP package. ISTOS uses distributed hashing to distribute the tuples between the nodes of the parallel machine. We have also developed a precompiler that translates the Linda calls to run time library calls. ISTOS is a full implementation of the Linda model. It is a useful and reliable programming tool that can be used for educational and research purposes.

Supervisor: S. Orphanoudakis,
Professor, Computer Science Department,
University of Crete.

Ευχαριστίες

Θα ήθελα να ευχαριστήσω όλους όσους βοήθησαν στην ολοκλήρωση της εργασίας αυτής αλλά και όσους βοήθησαν, με οποιοδήποτε τρόπο, στην ολοκλήρωση των σπουδών μου στο Πανεπιστήμιο Κρήτης.

Ιδιαίτερα, θα ήθελα να ευχαριστήσω τον ερευνητικό μου σύμβουλο Στέλιο Ορφανουδάκη, Καθηγητή του Πανεπιστημίου Κρήτης, για την βοήθεια και ενθάρρυνση του.

Ακόμα θα ήθελα να ευχαριστήσω την επιτροπή που έκρινε την εργασία αυτή και η οποία αποτελείται από τους Καθηγητές του Πανεπιστημίου Κρήτης, Στέλιο Ορφανουδάκη, Μανόλη Κατεβαίνη και Χρήστο Νικολάου.

Επίσης, θα ήθελα να ευχαριστήσω τον Γιάννη Σχοινά, απόφοιτο του μεταπτυχιακού Τμήματος Επιστήμης Υπολογιστών του Πανεπιστημίου Κρήτης, για την μεγάλη βοήθεια που μου πρόσφερε και τους Ευαγγελία Καβακλή, Μανόλη Λουράκη και Πένη Βατσολάκη.

Τέλος, θα πρέπει να ευχαριστήσω το Ίδρυμα Τεχνολογίας Έρευνας, για την οικονομική ενίσχυση και την υλικοτεχνική υποδομή που μου παρείχε κατά την διάρκεια των μεταπτυχιακών μου σπουδών.

Περιεχόμενα

1 Εισαγωγή **3**

1.1 Προηγούμενη δουλειά 6

1.2 Συνοπτική περιγραφή του Linda model 7

1.3 Αποκλίσεις από το μοντέλο Linda 11

1.3.1 Eval 11

1.3.2 Μια νέα αντιμετώπιση των predicate operations 12

2 Υλοποίηση του μοντέλου Linda σε τοπικό δίκτυο **13**

2.1 Η επικοινωνία των tuple managers 14

2.1.1 Περιγραφή 17

2.1.2 Αλγόριθμος επανεκπομπής πακέτων 20

2.1.3 Η Γενικότητα του πρωτοκόλλου 21

2.1.4 Μετρήσεις απόδοσης 23

2.1.5 Επιλογή της μονάδας MTU 23

2.2 Επικοινωνία process, tuple manager 26

2.2.1 Μετρήσεις απόδοσης 29

2.3 Χρήση gateways για τα broadcasts 29

2.4 Debugging, monitoring εργαλεία 32

3 Μέθοδοι για την βελτιστοποίηση του συστήματος **35**

3.1 Replication για πιο γρήγορα reads 36

3.2 Broadcast based Linda 38

3.2.1 Βελτιώσεις χρησιμοποιώντας στατιστική 39

3.2.2 Κεντρικοποιημένο Tuple Space 41

3.3 Αλγόριθμος ισοκατανομής φόρτου 41
3.4 Αύξηση της ανοχής του συστήματος σε λάθη 43
3.5 Hashing 44
3.5.1 Μετρήσεις απόδοσης 47

4 Συμπεράσματα, επεκτάσεις **49**
4.1 Συμπεράσματα 49
4.2 Επεκτάσεις 50

A ΙΣΤΟΣ Linda: υλοποίηση σε μια παράλληλη μηχανή **53**
A.1 Distributed Hashing 54
A.2 Eval στην παράλληλη μηχανή 55
A.3 ΙΣΤΟΣ-Linda simulator με χρήση threads 56
A.4 Ο μεταφραστής 56
A.5 Συμπεράσματα 57

B Διαχείριση μνήμης **58**

Κατάλογος Σχημάτων

1.1 Κατανεμημένη κοινή μνήμη 4
1.2 Το Linda Model και οι Linda operations 9

2.1 Το μοντέλο λειτουργίας του POSYBL 14
2.2 Η ανταλλαγή μηνύματος μεταξύ δύο κόμβων 17
2.3 Η υπερχείλιση του δέκτη 18
2.4 Η υπερχείλιση του δρομολογητή ή της γέφυρας 19
2.5 Τρόπος λειτουργίας των ασύγχρονων μηνυμάτων 22
2.6 Διάγραμμα σύγκρισης απόδοσης μεταξύ NPOSYBL και RPC. Η συμπεριφορά γύρω από την αρχή των αξόνων φαίνεται σε μεγένθυση στο ένθετο σχήμα. . . . 25
2.7 Η λειτουργία του IPC πρωτόκολλου 28
2.8 Οι gateways και η λειτουργία τους 31
2.9 Το tupleTool 33

3.1 Η πράξη rd χωρίς replication 36
3.2 Η πράξη rd με replication 37
3.3 Η πράξη διαγραφής σε replicated tuple 37
3.4 Η εσωτερική δομή του tuple space σε κάθε κόμβο 46
3.5 Η απόδοση του αλγορίθμου αποθήκευσης και ανεύρεσης των tuples 48

Κεφάλαιο 1

Εισαγωγή

Κάθε μέρα σχεδόν, ανακοινώνονται καινούρια μοντέλα παράλληλων υπολογιστών. Το γεγονός ότι πλησιάζουμε σιγά σιγά τα φυσικά όρια της ταχύτητας των επεξεργαστών και της μνήμης, σε συνδυασμό με την συνεχή διόγκωση των υπολογιστικών αναγκών του επιστημονικού κυρίως τομέα οδηγεί με γοργά βήματα στην καθιέρωση της παράλληλης επεξεργασίας σαν το επόμενο βήμα για την επίτευξη μαζικής υπολογιστικής ισχύος.

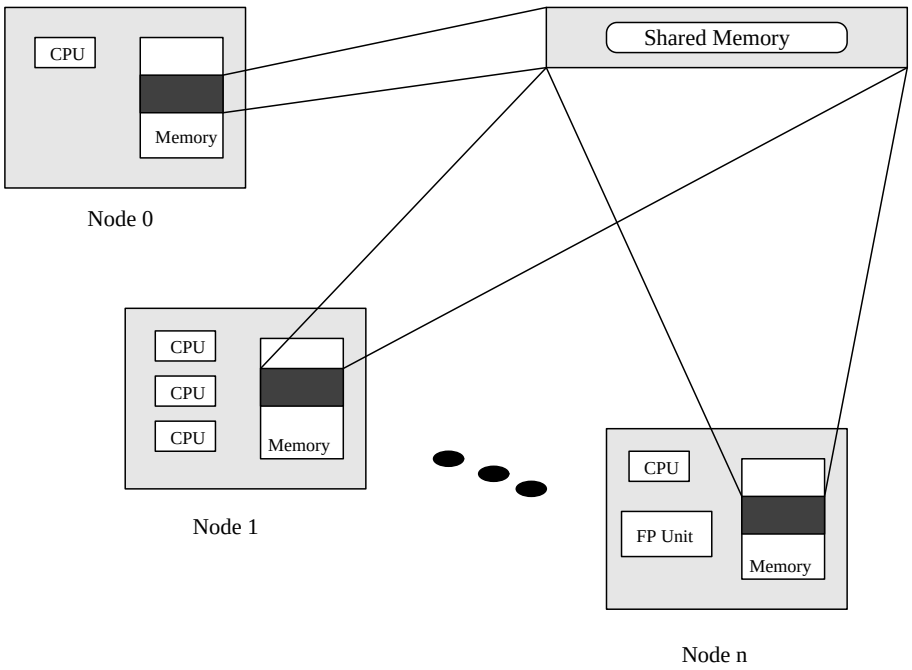
Από τις πολλές παράλληλες αρχιτεκτονικές που έχουν κατά καιρούς προταθεί, δύο έχουν γίνει ιδιαίτερα δημοφιλείς:

- πολυεπεξεργαστές κοινής μνήμης και
- πολυεπεξεργαστές κατανεμημένης μνήμης.

Ένας πολυεπεξεργαστής κοινής μνήμης όμως δεν επεκτείνεται αφού *η κύρια μνήμη χρησιμοποιείται μέσα από ένα κοινό μονοπάτι.* Έτσι περιορίζεται το μέγεθός τους σε μερικές δεκάδες επεξεργαστές.

Οι πολυεπεξεργαστές κατανεμημένης μνήμης δεν έχουν αυτό το πρόβλημα. Αν και είναι δύσκολο να ορίσουμε σαφώς τί είναι υπολογιστής κατανεμημένης μνήμης, μπορούμε με ασφάλεια να πούμε ότι τα συστήματα αυτά αποτελούνται από μια συλλογή ανεξάρτητων επεξεργαστών (computers) συνδεδεμένων μέσω ενός διασυνδετικού δικτύου υψηλής ταχύτητας. Αν ο σχεδιαστής διαλέξει προσεκτικά την τοπολογία του δικτύου, τότε μπορεί να συμπεριλάβει πολύ περισσότερους επεξεργαστές απ' ότι ένας πολυεπεξεργαστής κοινής μνήμης.

Αν και η μέγιστη δυνατή απόδοση των κατανεμημένων συστημάτων είναι θεωρητικά



Σχήμα 1.1: Κατανεμημένη κοινή μνήμη

απεριόριστη, τα μοντέλα κοινής μνήμης έχουν επικρατήσει στο επιστημονικό πεδίο. Αυτό, πέρα από όλους τους άλλους λόγους που μπορεί να σκεφτεί κανείς οφείλεται και στην ευκολία με την οποία προγραμματίζονται. Η πολυεπεξεργασία κοινής μνήμης είναι φυσική επέκταση του μονοεπεξεργαστικού συστήματος, και επομένως πιο εύκολη να προγραμματιστεί.

Αντίθετα, στους πολυεπεξεργαστές κατανεμημένης μνήμης η επικοινωνία μεταξύ των διεργασιών που εκτελούνται ταυτόχρονα γίνεται διαμέσω του δικτύου. Έτσι, το φυσικό μοντέλο προγραμματισμού τους είναι το message passing. Όμως πολλά σύγχρονα συστήματα έχουν αναπτύξει ένα επίπεδο κοινής μνήμης πάνω από τα message passing (Distributed Shared Memory). Αυτό το επίπεδο δίνει την ψευδαίσθηση της κοινής μνήμης (Σχήμα 1.1) στους χρήστες αυτών των συστημάτων, και επιτρέπει την χρήση του ανάλογου μοντέλου προγραμματισμού.

Τα πλεονεκτήματα της DSM μπορούν να επιτευχθούν με σχετικά μικρό επιπλέον κόστος. Συστήματα DSM έχουν υλοποιηθεί χρησιμοποιώντας τις παρακάτω τρεις τεχνικές (ή και συνδυασμούς)[1].

1. Υλοποιήσεις σε hardware που είναι επέκταση των παραδοσιακών τεχνικών caching σε scalable αρχιτεκτονικές.

2. Υλοποιήσεις μέσω operating systems και βιβλιοθηκών που πετυχαίνουν coherence μέσω

μηχανισμών διαχείρισης virtual μνήμης.

3. Υλοποιήσεις μέσω μεταφραστών οι οποίοι μετατρέπουν τα accesses στην κοινή μνήμη σε εντολές (primitives) συγχρονισμού και coherence.

Μια σύντομη περιγραφή των χαρακτηριστικών μερικών DSM υπάρχει στο [1].

Σε αυτήν την εργασία το σύστημα που επιλέχθηκε να υλοποιηθεί είναι το μοντέλο DSM Linda[2, 3], χρησιμοποιώντας MIMD μηχανές και πιο συγκεκριμένα τοπικά δίκτυα από workstations.

Ένα σύστημα που προορίζεται για ευρεία χρήση από χρήστες που το επίπεδο εξοικείωσής τους με τον παράλληλο προγραμματισμό (ή και με τον ίδιο τον προγραμματισμό) κυμαίνεται από στοιχειώδες έως πολύ προχωρημένο πρέπει να είναι σταθερό ώστε να μην ‘‘διαλύεται’’ με το πρώτο λάθος και να είναι γρήγορο και επαρκές ώστε να μην απογοητευθεί ο χρήστης και το εγκαταλείψει στην πρώτη κιόλας προσπάθεια. Πρέπει τέλος να μην κρύβει τελείως την αρχιτεκτονική την οποία χρησιμοποιεί ώστε ο ικανός χρήστης να μπορεί να το χρησιμοποιήσει σαν ένα ακόμα εργαλείο που του προσφέρει επιπλέον δυνατότητες, χωρίς όμως να τον εμποδίζει από το να επιβάλει τις δικές του τεχνικές.

Αν και το αποτέλεσμα αυτής της εργασίας δεν είναι ένα ολοκληρωμένο σύστημα, έχει πολλά από τα στοιχεία εκείνα που θα επιτρέψουν να προχωρήσει στο μέλλον με σταθερά βήματα. Κάθε ένα από τα κομάτια που περιγράφονται είναι λειτουργικό και για επιβεβαίωση χρησιμοποιήθηκαν μικρά αλλά χαρακτηριστικά προβλήματα. Υπάρχουν όμως και πολλά άλλα κομάτια τα οποία είτε είναι στοιχειώδη ώστε να μην αξίζει τον κόπο να αναφερθούν είτε η συμμετοχή μου σε αυτά δεν πίστευα ότι είναι αρκετή ώστε να μπορώ να τα αναφέρω στην μεταπτυχιακή μου εργασία.

Βασισμένοι στις εμπειρίες που αποκτήσαμε από το σχεδιασμό αυτό, προχωρήσαμε σε ένα δεύτερο βήμα: στην υλοποίησή του σε μια παράλληλη μηχανή και στη δημιουργία ενός προσομοιωτή του σε Sun Workstation. Οι 2 τελευταίες υλοποιήσεις πιστεύουμε ότι έχουν τα χαρακτηριστικά που αναφέραμε σαν επιθυμητά για ένα ευρέως χρησιμοποιούμενο σύστημα.

Ίσως ένα πιο γενικό μοντέλο να πρόσφερε πιο πολύ χώρο για ερευνητική δουλειά. Ο σκοπός μας όμως ήταν να φτιάξουμε ένα εργαλείο για προγραμματιστές που θέλουν να βελτιώσουν την απόδοση των προγραμμάτων τους χρησιμοποιώντας ένα σχετικά εύκολο μοντέλο παράλληλου προγραμματισμού. Πολλές από τις αποφάσεις που πάρθηκαν στηρίζο-

νται σε μετρήσεις σε συγκεκριμένα λειτουργικά συστήματα και μηχανές και ίσως θα πρέπει να αναθεωρηθούν καθώς περνάει ο καιρός και η τεχνολογία βελτιώνεται. Τα προβλήματα όμως που έπρεπε να λυθούν είναι πιο διαχρονικά και εκεί δόθηκε η έμφαση της εργασίας. Φυσικά και οι αποφάσεις οι ίδιες περιγράφονται και ελπίζουμε δικαιολογούνται επαρκώς.

Αυτή η εργασία έχει 2 κύρια μέρη. Πρώτα είναι η περιγραφή της υλοποίησης του NPOSYBL, η επέκταση δηλαδή του POSYBL [4] ώστε να επιτύχει καλύτερη απόδοση, και μετά είναι η συνοπτική περιγραφή του ΙΣΤΟΣ, η υλοποίηση του μοντέλου Linda σε έναν παράλληλο πολυεπεξεργαστή κατανεμημένης μνήμης και ενός προσομοιωτή του για SUN workstations. Ο προσομοιωτής μπορεί να θεωρηθεί και σαν ανεξάρτητο σύστημα, αφού δεν χρησιμοποιεί και δεν επιβάλλει κανένα περιορισμό της παράλληλης μηχανής. Άλλωστε, η Linda είναι ένα από τα πιο portable περιβάλλοντα.

1.1 Προηγούμενη δουλειά

Το μοντέλο της Linda ορίσθηκε το 1982 από τον Gelernter [5]. Από τότε έχουν γίνει πολλές υλοποιήσεις και επεκτάσεις του μοντέλου. Ένα από τα χαρακτηριστικά που μπορεί να επισημανθούν είναι η πληθώρα παράλληλων μηχανών που χρησιμοποιήθηκαν για τις υλοποιήσεις αυτές. Το [6] αναφέρει υλοποιήσεις σε μηχανές κοινής μνήμης όπως Encore Multimax, Sequent Balance και Symmetry και Aliant FX/8, σε μηχανές κατανεμημένης μνήμης, όπως ο Intel IPSC-2 hypercube. σε τοπικό δίκτυο από VAX/VMS και σε πολλές custom αρχιτεκτονικές, όπως το AT&T Bell Labs' S/Net. Επίσης υπάρχουν και εμπορικές εκδόσεις όπως αυτή της Scientific Computing Associates του New Haven, Connecticut που στην αρχή διέθεσε στην αγορά έναν simulator σε Sun και μετά προχώρησε στην υλοποίηση του Linda σε τοπικά δίκτυα υπολογιστών.

Σήμερα πλέον είναι διαθέσιμο το Linda σε σχεδόν όλες τις γνωστές αρχιτεκτονικές workstations, όπως IBM RS6000, Sun κ.λπ.

Οι πιο πολλές υλοποιήσεις είναι χωριστές βιβλιοθήκες που συνδέονται (link) με το πρόγραμμα του χρήστη, και όχι μέρος του λειτουργικού των υπολογιστών. Αξίζει όμως να αναφέρουμε και την υλοποίηση του Kernel Linda[7], όπου το μοντέλο γίνεται πλέον κομάτι ενεργό του λειτουργικού συστήματος της μηχανής.

Ένα άλλο στοιχείο που διαχωρίζει τις διάφορες υλοποιήσεις είναι η χρήση ή όχι compiler

για την βελτιστοποίηση των πράξεων στο tuple space. Πρωτοπορία σε αυτόν τον τομέα έχει η ομάδα του New Haven, που με την χρήση partial evaluation μεθόδων[8] επιτυγχάνει βελτιστοποιήσεις με χρήση πληροφορίας διαθέσιμης κατά την μετάφραση (compiling) του προγράμματος.

Οι υπόλοιπες υλοποιήσεις, και η δική μας επίσης, στηρίζονται στις όποιες βελτιστοποιήσεις μπορούν να επιτευχθούν στον χρόνο τρεξίματος (run time).

Εδώ πρέπει να αναφέρουμε και άλλα δημοφιλή συστήματα κατανεμημένου προγραμματισμού. Το πιο παλιό ίσως είναι το SunRPC (Remote Procedure Calls), που δίνει στον προγραμματιστή την δυνατότητα να καλεί υπορουτίνες προγραμμάτων σε απομακρυσμένους κόμβους. Πάνω από το RPC έχουν υλοποιηθεί πολλά από τα εργαλεία που παρέχουν τα σύγχρονα λειτουργικά. Το πιο γνωστό και διαδεδομένο είναι το NFS (Network File System).

Άλλα τέτοια συστήματα είναι το PVM και το P4. Αυτά είναι portable συστήματα παράλληλου προγραμματισμού με την χρήση message passing. Τα συστήματα αυτά τρέχουν σε μια μεγάλη ποικιλία παράλληλων και κατανεμημένων συστημάτων και δίνουν την δυνατότητα να δημιουργηθούν εύκολα παράλληλα προγράμματα. Το P4 επιπλέον παρέχει την δυνατότητα χρήσης του μοντέλου κοινής μνήμης όταν τρέχει σε υπολογιστές που το υποστηρίζουν.

Τα παραπάνω μοντέλα έχουν γίνει πρόσφατα πολύ δημοφιλή, ειδικά στους κύκλους εκτός της επιστήμης υπολογιστών αν και το μοντέλο που προσφέρουν, εκτός από το portability δεν διαφέρει από τα κλασικά message passing περιβάλλοντα που προσφέρουν όλες οι αρχιτεκτονικές κατανεμημένης μνήμης.

Τέλος μεγάλη κίνηση παρατηρείται τελευταία στον τομέα των data parallel compilers. Παράδειγμα είναι η Data Parallel C και το adaptor από το GMD που είναι μια υλοποίηση της High Performance Fortran.

1.2 Συνοπτική περιγραφή του Linda model

Το Linda model[2, 3] είναι ένα μοντέλο για δημιουργία και συντονισμό πολλαπλών διεργασιών που τρέχουν σε έναν η περισσότερους επεξεργαστές. Το μοντέλο αυτό ενσωματώνεται μέσα σε μια σειριακή γλώσσα προγραμματισμού (π.χ. C, Lisp, κ.λ.π.) και το αποτέλεσμα είναι μια παράλληλη γλώσσα προγραμματισμού.

Το tuple space είναι ο μηχανισμός που παρέχει το Linda model για την δημιουργία και τον συντονισμό πολλαπλών διεργασιών. Το tuple space είναι μια συλλογή από tuples. Κάθε tuple είναι μια σειρά από typed πεδία. Το μοντέλο ορίζει πράξεις για την προσθήκη ενός tuple στο tuple space και την αφαίρεση ή το διάβασμα ενός tuple από το tuple space. Για να βρούμε ένα συγκεκριμένο tuple, χρησιμοποιούμε associative look-up, δηλαδή τα tuples επιλέγονται για αφαίρεση ή διάβασμα με βάση τις τιμές των πεδίων τους. Τα πεδία, τα οποία χρησιμοποιούνται για αυτό τον σκοπό, λέγονται actuals ενώ τα πεδία των οποίων την τιμή διαβάζουμε λέγονται formals. Ένα πεδίο μπορεί να είναι actual σε μια κλήση και formal σε κάποια άλλη. Τα formal πεδία δηλώνονται με ?. Αλλά ας δώσουμε ένα παράδειγμα ενός tuple:

```
("Example", 10, floatvar)
```

και δύο templates που ικανοποιούνται από το παραπάνω tuple:

```
("Example", ? intvar, ? floatvar)
("Example", intvar, ? floatvar)
```

Η δεύτερη template ικανοποιείται από το tuple μόνο εάν η μεταβλητή intvar περιέχει την τιμή 10.

Η δημιουργία μιας διεργασίας γίνεται με τα live tuples. Τα live tuples είναι εκείνα των οποίων τα πεδία δεν υπολογίζονται παρά μόνο αφού το tuple εισαχθεί στο tuple space. Όταν ένα live tuple εισαχθεί στο tuple space υπολογίζεται ανεξάρτητα και παράλληλα με την διεργασία που το εισήγαγε. Όταν ο υπολογισμός τελειώσει, το live tuple γίνεται ένα συνηθισμένο tuple και μπορεί να διαβαστεί ή να αφαιρεθεί όπως ένα οποιοδήποτε άλλο tuple.

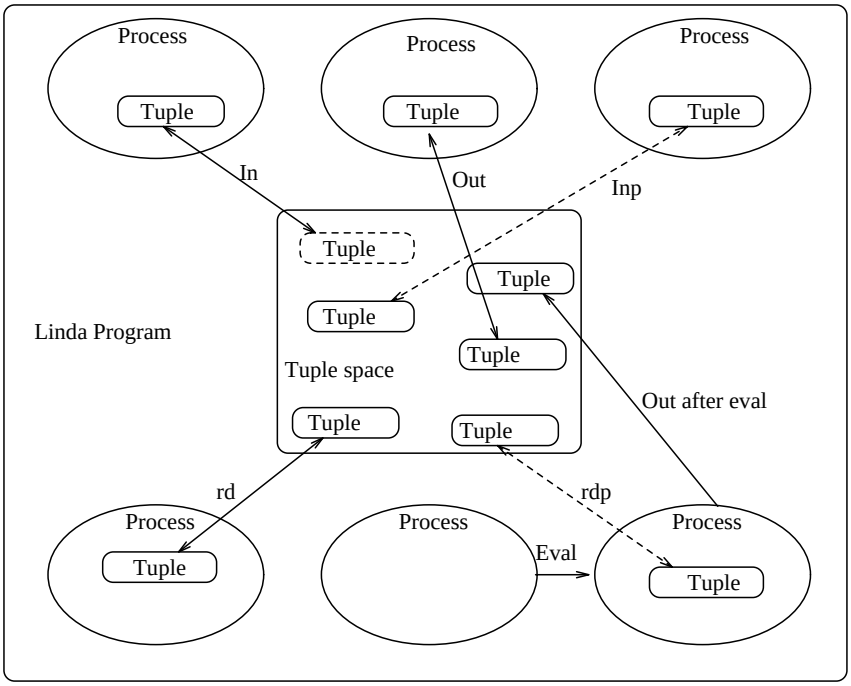
Το Linda model ορίζει τις εξής πράξεις στο tuple space:

- *out(t)*

  Το tuple t γράφεται στο tuple space.

- *in(s)*

  Κάποιο tuple t που ικανοποιεί την template s αποσύρεται από το tuple space. Οι τιμές των actuals του t ανατίθενται στα formals της s και η διεργασία συνεχίζει. Αν κανένα tuple t δεν ικανοποιεί την template s, η διεργασία γίνεται suspended μέχρις ότου υπάρξει κάποιο. Αν υπάρχουν πολλά tuples τα οποία ικανοποιούν την template s, διαλέγεται ένα αυθαίρετα.



Σχήμα 1.2: Το Linda Model και οι Linda operations

- *inp(s)*

  Ανάλογα με την *in()*, με την διαφορά ότι αν κανένα tuple t δεν ικανοποιεί την template s, επιστρέφει False και η διεργασία δεν γίνεται suspended. Αλλιώς, επιστρέφει True.

- *rd(s)*

  Αυτή η πράξη είναι ίδια με την $in(s)$, με την διαφορά ότι το tuple t δεν αποσύρεται από το tuple space.

- *rdp(s)*

  Ανάλογα με την *rd()*, με την διαφορά ότι αν κανένα tuple t δεν ικανοποιεί την template s, επιστρέφει False και η διεργασία δεν γίνεται suspended. Αλλιώς, επιστρέφει True.

- *eval(t)*

  Κάποιο tuple t γράφεται στο tuple space αλλά οι τιμές των πεδίων του t υπολογίζονται αφού εισέλθει στο tuple space και όχι πριν. Μία καινούρια διεργασία δημιουργείται για να εκτελέσει τον υπολογισμό.

Εισαγωγή στην προσέγγιση του Linda model

Το Linda model είναι το πρώτο μοντέλο που προσπάθησε να λύσει το πρόβλημα του προγραμματισμού των παράλληλων υπολογιστών προσφέροντας μια σταθερή δομή για την αποθήκευση πληροφοριών. Το σημαντικότερο πλεονέκτημα αυτής της προσέγγισης είναι ότι ελευθερώνει τον χρήστη από την ανάγκη να προγραμματίζει λαμβάνοντας υπόψη του χρονικές και τοπικές σχέσεις. Αλλά και από πλευράς απόδοσης, η προσέγγιση αυτή έχει το πλεονέκτημα ότι επιτρέπει την αποδέσμευση του χρόνου αποστολής από τον χρόνο παραλαβής της πληροφορίας(decoupling) και επομένως υπάρχουν λιγότερα σημεία συγχρονισμού στο παράλληλο πρόγραμμα.

Χρησιμοποιώντας το Linda model, ο χρήστης μπορεί να αναπτύξει παράλληλα προγράμματα ακολουθώντας τρεις μεθοδολογίες[2]:

- *Message passing:* Μπορεί να χρησιμοποιήσει το tuple space σαν ένα προσωρινό χώρο αποθήκευσης μηνυμάτων. Σε αυτήν την περίπτωση, τα δεδομένα του προγράμματος βρίσκονται μέσα στην ιδιωτική μνήμη κάθε διεργασίας και αυτό που στέλνεται από διεργασία σε διεργασία είναι πράξεις στα δεδομένα.

- *Distributed data structures:* Μπορεί να χρησιμοποιήσει το tuple space για να δημιουργήσει κατανεμημένες δομές δεδομένων. Σε αυτήν την περίπτωση, τα δεδομένα του προγράμματος βρίσκονται στο tuple space προσπελάσιμα από όλες τις διεργασίες.

- *Live data structures:* Μπορεί να χρησιμοποιήσει live tuples για να δημιουργήσει δομές δεδομένων που αποτελούνται από ενεργά στοιχεία τα οποία μπορούν να αντιδρούν στις αλλαγές του περιβάλλοντος τους. Σε αυτή την περίπτωση, δεν υπάρχει διαχωρισμός μεταξύ δεδομένων και διεργασιών.

Παρατηρούμε ότι, κατά κανόνα, οι ανάγκες επικοινωνίας του παράλληλου προγράμματος αυξάνονται (και το grain size μικραίνει) καθώς προχωράμε από το message passing στις live data structures.

1.3 Αποκλίσεις από το μοντέλο Linda

1.3.1 Eval

Η πράξη eval είναι αρκετά ασαφής ώστε να επιτρέπει την διαφορετική αντιμετώπισή της από τον κάθε ένα που υλοποιεί ένα σύστημα Linda. Η πιο βασική πηγή ασάφειας είναι το ποσοστό του περιβάλλοντος (context) που πρέπει να μεταφερθεί από κόμβο σε κόμβο κατά την εκτέλεση του eval. Για παράδειγμα μία διαδικασία (function) που γίνεται eval σε έναν απομακρυσμένο (remote) κόμβο αλλά στηρίζεται στις τιμές καθολικών (global) μεταβλητών δεν θα δουλέψει (θα προσπελαύνει τυχαία δεδομένα) αν δεν μεταφερθούν και οι μεταβλητές αυτές μαζί της. Για να γίνει όμως αυτό χρειάζεται μεγάλη υποστήριξη από τον μεταφραστή, ο οποίος πρέπει να βρει ποιές μεταβλητές είναι απαραίτητες και να παράγει επιπλέον κώδικα ώστε να τις πακετάρει και να τις μεταφέρει στον απομακρυσμένο κόμβο.

Στην δική μας περίπτωση επιλέξαμε μια τελείως διαφορετική αντιμετώπιση, όπου η eval παράγει μια νέα διεργασία σε έναν κόμβο, και όχι την ενεργοποίηση μιας ή παραπάνω ρουτινών μέσα σε διεργασίες που ήδη τρέχουν από την στιγμή της εκκίνησης του συστήματος. Δηλαδή η eval πλέον παίρνει σαν όρισμα ένα εκτελέσιμο πρόγραμμα και όχι μια ή παραπάνω ρουτίνες. Επίσης, η eval πλέον δεν παράγει στο τέλος κανένα tuple.

Αυτό επιβάλει το κάθε πρόγραμμα που θα γίνει eval να είναι αυτόνομο με δικό του main() και ανεξάρτητες λειτουργίες, αλλά από την άλλη μας επιτρέπει την συγκόληση ξεχωριστών προγραμμάτων που τρέχουν ανεξάρτητα αλλά επικοινωνούν μεταξύ τους μέσω των πράξεων της Linda. Με τον συνδυασμό του hashing πού επιτρέπει χωριστή μετάφραση (separate compilation) των προγραμμάτων μπορεί κανείς να συνδυάσει προγράμματα μεταξύ τους πού έχουν αναπτυχθεί για τελείως διαφορετικό σκοπό, αλλά έχουν κοινό format στις πράξεις Linda.

Το αποτέλεσμα είναι ένα σύστημα όπου το eval είναι σχετικά ακριβό αφού παράγει μια δαπανηρή (heavyweight) διεργασία σε έναν απομακρυσμένο κόμβο. Ο χρήστης δεν μπορεί να παράγει αλόγιστα διεργασίες και άρα θέλει ιδιαίτερη προσοχή κατά τον προγραμματισμό.

Από την άλλη μεριά όμως δίνει παραπάνω δυνατότητες στον προγραμματιστή αφού του επιτρέπει να συνδυάσει ανεξάρτητα προγράμματα και να καλέσει εξωτερικές διαδικασίες.

1.3.2 Μια νέα αντιμετώπιση των predicate operations

Τα predicate operations **inp** και **rdp** δεν τα υλοποιήσαμε. Ο λόγος είναι ότι οι πράξεις αυτές δίνουν έναν ισχυρό χαρακτηρισμό της κατάστασης της μνήμης (tuple space) σε μια συγκεκριμένη χρονική στιγμή. Για να υλοποιηθεί αυτό όμως πρέπει να εκτελεστεί μια γενική φάση συγχρονισμού μεταξύ όλων των κόμβων ώστε να δοθεί μια θετική ή αρνητική απάντηση.

Εμείς προτιμήσαμε να υλοποιήσουμε δύο νέες πράξεις που δεν δίνουν απόλυτη αλλά πιθανή απάντηση για την ύπαρξη ή όχι ενός tuple στο tuple space. Οι πράξεις αυτές είναι οι **tin** και **trd**, και χρησιμοποιούν χρονομετρητές (timers) για να καθορίσουν την απάντησή τους. Δηλαδή στην περίπτωση που επιστρέφεται κάποιο tuple που να ικανοποιεί την template του tin, trd μέσα σε ορισμένο χρόνο t, τότε όλα είναι όπως και στην περίπτωση του inp, rdp, και επιστρέφουμε TRUE. Όταν περάσει αυτός ο χρόνος και δεν έχουμε λάβει κανένα tuple, τότε δεν μπορούμε να πούμε ότι δεν υπάρχει τέτοιο tuple, αλλά ότι μέσα σε χρόνο t κανείς κόμβος δεν μας έδωσε μια θετική απάντηση και επιστρέφουμε FALSE.

Με δεδομένη την ταχύτητα των κόμβων και του δικτύου, μπορεί κανείς να εκλάβει και σαν τελική αυτήν την απάντηση. Πάντως υπάρχει περίπτωση να μην είναι αληθής, μια και ένας κόμβος μπορεί να έχασε το broadcast πακέτο, ή να είναι πολύ φορτωμένος για να απαντήσει μέσα στον προκαθορισμένο χρόνο.

Ο χρόνος t ονομάζεται timeout, και είναι εσωτερικά ορισμένος από το σύστημα. Θα μπορούσε όμως εύκολα να επεκταθεί ώστε να είναι ορισμένος από τον χρήστη. Με την χρήση timeouts μπορούμε να επιτύχουμε το ίδιο σχεδόν αποτέλεσμα. Μας δίνονται μεγαλύτερες δυνατότητες μια και τα inp, rdp είναι εύκολο να υλοποιηθούν έμμεσα από τον χρήστη ,με αρκετό επιπλέον κόστος, ενώ με το tin, trd μας δίνει νέες δυνατότητες αφού μας επιτρέπει να προσθέσουμε και χρονικές παραμέτρους στη εκτέλεση του προγράμματος.

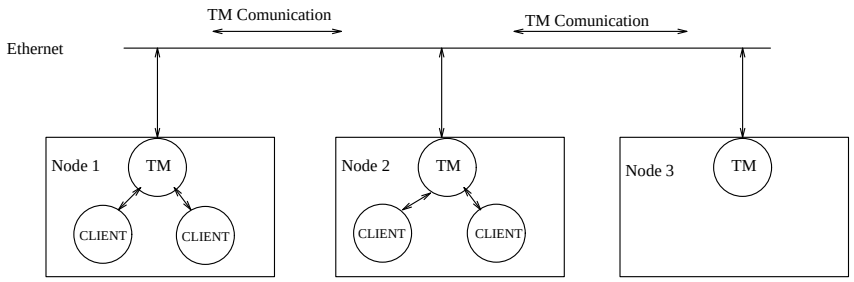
Κεφάλαιο 2

Υλοποίηση του μοντέλου Linda σε τοπικό δίκτυο

Η πρώτη προσπάθεια υλοποίησης του Linda σε τοπικό δίκτυο στο πανεπιστήμιό μας είναι το POSYBL[9]. Η αρχιτεκτονική του είναι βασισμένη στην δουλειά του Leighter [10] που έγινε πάνω από το λειτουργικό VMS. Τα βασικά του μέρη είναι οι tuple managers, που είναι διεργασίες UNIX υπεύθυνες για την διατήρηση του tuple-space, και οι διεργασίες POSYBL, ανεξάρτητα προγράμματα που επικοινωνούν με το tuple-space μέσω των tuple-managers. Οι διεργασίες POSYBL πού συνεργάζονται μεταξύ τους μέσω του tuple space ονομάζονται πρόγραμμα POSYBL. Όλες οι πράξεις Linda που εκτελούν οι διεργασίες απευθύνονται αρχικά στους tuple-managers, και κατόπιν αυτοί διευθετούνε μεταξύ τους την εξυπηρέτηση των πράξεων Linda, όπως φαίνεται στο σχήμα 2.1.

Οι tuple managers είναι διεργασίες τύπου daemon που τρέχουν σε κάθε μηχάνημα που υποστηρίζει το POSYBL. Τα tuples που παράγει κάθε διεργασία POSYBL αποθηκεύονται στον τοπικό tuple manager. Επίσης, κάθε template απευθύνεται στον τοπικό tuple manager. Όταν μια template δεν μπορεί να εξυπηρετηθεί από τον τοπικό tuple manager, γίνεται broadcast σε όλους, ώστε να γίνει γνωστή και να εξυπηρετηθεί από κάποιον άλλο tuple manager. Αν υπάρξει matching tuple τότε αυτό στέλνεται στον τοπικό tuple manager της διεργασίας που το ζήτησε και μέσω αυτού επιστρέφεται στην διεργασία που το είχε ζητήσει. Όλοι οι tuple managers που τυχόν έχουν tuple που ικανοποιεί την template στέλνουν απάντηση στον tuple manager της διεργασίας που παρήγαγε το template.

Η επικοινωνία δηλαδή γίνεται σε δύο επίπεδα: διεργασίες χρήστη με τους τοπικούς tuple



Σχήμα 2.1: Το μοντέλο λειτουργίας του POSYBL

managers, και οι tuple managers μεταξύ τους.

Αυτή η λογική διατηρήθηκε και στο NPOSYBL. Αυτό που επιχειρήθηκε είναι η βελτίωση της απόδοσης χρησιμοποιώντας καλύτερους αλγόριθμους για την υλοποίηση των επιμέρους κομματιών καθώς και νέων τεχνικών όπου οι προηγούμενες ήταν ανεπαρκείς.

2.1 Η επικοινωνία των tuple managers

Οι tuple managers είναι UNIX daemons που τρέχουν σε κάθε κόμβο που συμμετέχει ή μπορεί να συμμετέχει σε ένα POSYBL πρόγραμμα. Η επικοινωνία του γίνεται μέσω του δικτύου που ενώνει τους σταθμούς εργασίας, και είναι είτε point to point όταν 2 tuple managers επικοινωνούν μεταξύ τους, είτε broadcast messages όταν κάποιος tuple manager ανακοινώνει κάποια template. Οι μοντέρνοι σταθμοί εργασίας που χρησιμοποιούν το λειτουργικό UNIX προσφέρουν σαν standard αξιόπιστο πρωτόκολλο μεταφοράς το TCP-IP. Το TCP-IP είναι ένα γνωστό και πολύ αποτελεσματικό πρωτόκολλο επικοινωνίας μεταξύ υπολογιστών διασυνδεδεμένων μέσω δικτύου. Παρόλα αυτά, και προσπαθώντας να είναι όσο το δυνατόν πιο γενικό, αποτυγχάνει να αξιοποιήσει πλήρως τις δυνατότητες ενός γρήγορου τοπικού δικτύου, χωρίς να υπονοούμε σχεδιαστικά λάθη ή γενικότερη αναποτελεσματικότητα. Πιστεύουμε όμως ότι δεν είναι προσαρμοσμένο σε transaction processing (μπορεί κανείς να δει τα in και out σαν πράξεις σε μια βάση δεδομένων). Τα μειονεκτήματα που το έκαναν ανεπαρκές για την εφαρμογή μας είναι τα παρακάτω:

- Είναι connection oriented. Κάθε tuple manager πρέπει να έχει, ή να μπορεί να δημιουργήσει σύνδεση με όλους τους υπόλοιπους. Αυτό σημαίνει ότι πρέπει να έχουμε αρκετές συνδέσεις ενεργές, και άρα πολλούς file descriptors. Το UNIX όμως είναι αρκετά φειδωλό σε αυτόν τον τομέα.

- Το TCP απαιτεί πολλά επιπλέον πακέτα για RPC, ειδικά για μεμονωμένες κλήσεις. Συγκεκριμένα η εγκατάσταση μονοπατιού επικοινωνίας (connection setup) και η καταστροφή του μονοπατιού απαιτούν επιπλέον πακέτα.

- Το πρωτόκολλο δουλεύει με το ονομαζόμενο "cold start", δηλαδή στην αρχή η επικοινωνία γίνεται σε μικρά κομμάτια δεδομένων, που σταδιακά αυξάνουν σε μέγεθος. Στην ειδική περίπτωση που εξετάζουμε, αυτό επιβραδύνει την επικοινωνία χωρίς να υπάρχει σημαντικός λόγος.

Η άλλη λύση που προσφέρει μέχρι και πρόσφατα το UNIX είναι το connectionless UDP πρωτόκολλο, το οποίο όμως έχει το σοβαρό μειονέκτημα της έλλειψης αξιοπιστίας. Δηλαδή η ασφαλής αποστολή και άφιξη δεδομένων δεν είναι εξασφαλισμένη. Τέλος τα τελευταία 2 χρόνια ένα νέο μοντέλο έχει εμφανιστεί, και ονομάζεται TLI (Transport Library Interface)[11] το οποίο παρέχει πρόσβαση σε διάφορους Transport Providers και με διάφορα χαρακτηριστικά. Παρόλο που φαίνεται να είναι το μέλλον στον προγραμματισμό μέσω δικτύου, δεν είναι αρκετά ώριμο ώστε να το χρησιμοποιήσουμε, αφού ακόμα δεν υποστηρίζεται από όλες τις εκδόσεις λειτουργικών συστημάτων .

Στην περίπτωσή μας, το περιβάλλον της Linda είναι ελεγχόμενο. Ξέρουμε ότι χρησιμοποιούμε γρήγορα τοπικά δίκτυα, όπως και το γεγονός ότι οι κόμβοι του δικτύου είναι σχετικά ομοιογενείς ως προς την υπολογιστική τους ισχύ. Αυτό μας οδήγησε σε μια μερική υλοποίηση του VMTP[12], ενός πρωτόκολλου από το V[13] distributed kernel.

Το VMTP μπορεί να θεωρηθεί μια μορφή αξιόπιστου message passing, που δίνει την δυνατότητα σε 2 υπολογιστές να επικοινωνούν με μηνύματα, χωρίς προκαθορισμένα (preestablished) μονοπάτια και χρόνο setup. Είναι βελτιστοποιημένο για εφαρμογές που χρησιμοποιούν request-response τύπο επικοινωνίας όπως για παράδειγμα η υλοποίηση του μοντέλου Linda σε τοπικό δίκτυο.

Η υλοποίησή μας είναι τμήμα του συνόλου των προδιαγραφών του VMTP και έχει κύριο στόχο την παροχή secure message passing μεταξύ 2 κόμβων. Το μέρος που υλοποιήθηκε ήταν αυτό που αφορούσε τα transactions. Ιδιαίτερη έμφαση δόθηκε στην ελαχιστοποίηση των μηνυμάτων (πακέτων) που εκπέμπονται στο δίκτυο, αφού αυτός είναι και ο μεγαλύτερος παράγοντας καθυστέρησης σε ένα κατανεμημένο σύστημα, όπως επίσης και παράγοντας γενικής καθυστέρησης αφού το μέσο είναι συνήθως μοναδικό και καταλαμβάνεται από έναν

κάθε φορά.

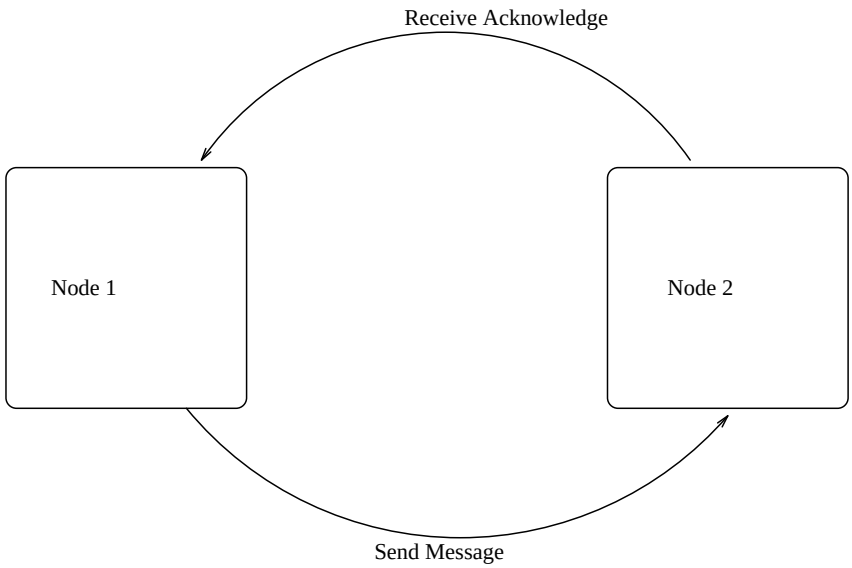
Οι πράξεις δικτύου που υλοποιήθηκαν είναι οι παρακάτω.

- send_message(), για την αποστολή μηνύματος σε κάποιον κόμβο. Ο κόμβος λαμβάνει το μήνυμα και επιβεβαιώνει.
- send_message_bcack(), για την αποστολή μηνύματος σε όλους τους γνωστούς κόμβους. Κάθε κόμβος που λαμβάνει το μήνυμα επιβεβαιώνει. Όλοι οι κόμβοι πρέπει να λάβουν το μήνυμα. Σε αυτήν την πράξη δώσαμε το όνομα secure broadcast, αν και δεν είναι πραγματικό broadcast.
- send_message_noack(), για την αποστολή ενός μηνύματος σε κάποιον κόμβο. Ο κόμβος ακόμα και αν λάβει το μήνυμα δεν επιβεβαιώνει.
- send_message_bcast(), για την αποστολή UDP broadcast μηνύματος στο δίκτυο.

Μετά την απόφαση να υλοποιήσουμε αυτό το πρωτόκολλο, μας δόθηκε η δυνατότητα να το προσαρμόσουμε ακριβώς στο μοντέλο του POSYBL ώστε να επιτύχουμε την βέλτιστη απόδοση. Έτσι, οι παραπάνω πράξεις, χρησιμοποιήθηκαν για την υλοποίηση ενός RPC (Remote Procedure Call) πακέτου όπου πάνω του χτίστηκαν οι tuple managers. Στην δική μας περίπτωση, το πρωτόκολλο αυτό είναι χτισμένο πάνω από μια UDP socket που χρησιμοποιείται σαν το interface του δικτύου. Το μέγιστο μήνυμα που μπορεί να αποστείλει το πρωτόκολλό μας είναι 120 Kbytes.

Οι πράξεις receive είναι έμμεσα υλοποιημένες, αφού το πρωτόκολλο είναι ενσωματωμένο στο σύστημά μας. Δηλαδή όταν ο μηχανισμός παραλαβής μηνυμάτων διαπιστώσει ότι έλαβε ένα ολόκληρο μήνυμα, τότε καλεί απευθείας την κατάλληλη RPC ρουτίνα. Σε περίπτωση που θα έπρεπε να αποχωρισθεί το πρωτόκολλο και να γίνει αυτόνομο, πρέπει να υλοποιηθεί κάποιου τύπου mailbox όπου θα αποθηκεύονται τα εισερχόμενα μηνύματα ώσπου κάποια διεργασία να τα διαβάσει.

Ένας άλλος πολύ σημαντικός λόγος που μας οδήγησε στην ανάπτυξη ενός νέου πρωτοκόλλου είναι η απαίτηση για ασύγχρονη επικοινωνία μεταξύ των κόμβων. Αυτό είναι αναγκαίο μια και δεν είναι λογικό η αποστολή ενός μεγάλου μηνύματος από τον tuple manager να τον κάνει να αγνοεί όλα τα μικρότερα μηνύματα που περιμένουν, είτε για παραλαβή είτε για εκπομπή.



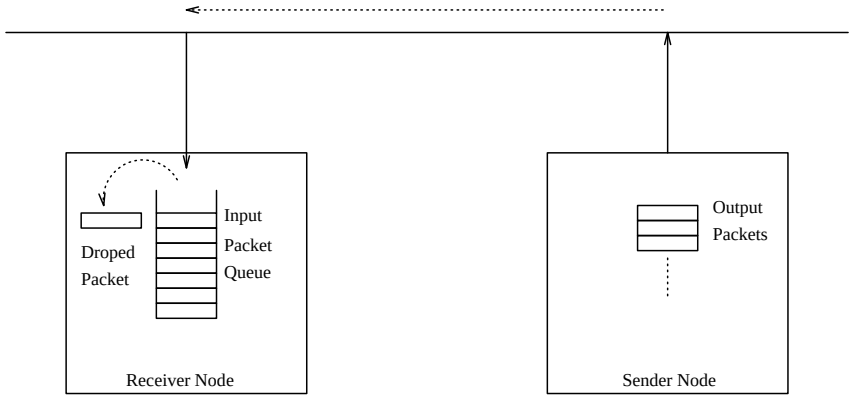
Σχήμα 2.2: Η ανταλλαγή μηνύματος μεταξύ δύο κόμβων

Τέλος και για τον λόγο ακριβώς ότι χρειάζεται ασύγχρονη επικοινωνία και για μεγαλύτερα μηνύματα από αυτά που υποστηρίζει το πρωτόκολλο που υλοποιήσαμε, χρησιμοποιούμε μια απλή public domain έκδοση του TCP που ονομάζεται TinyTCP.

Γενικά πρέπει να παρατηρηθεί ότι η υλοποίηση ενός πρωτόκολλου είναι αρκετά δύσκολο να επαληθευτεί. Εμείς δεν κάναμε καμιά προσπάθεια θεωρητικής (formal) επαλήθευσης και πιστεύουμε ότι είναι ένα πολύ ενδιαφέρον θέμα για μελλοντική δουλειά. Το πρωτόκολλο δοκιμάστηκε όμως εκτενώς στην πράξη χωρίς να παρουσιάζει προβλήματα. Είναι αυτήν την στιγμή ένα λειτουργικό κομάτι του NPOSYBL και παίζει σημαντικό ρόλο στην βελτίωση της απόδοσής του.

2.1.1 Περιγραφή

Το πρωτόκολλο αποτελείται από δύο μέρη. Τον πομπό και τον δέκτη. Ο πομπός στέλνει ένα μήνυμα ενώ ο δέκτης επιβεβαιώνει την παραλαβή του. Τα πράγματα όμως δεν είναι τόσο απλά. Το φυσικό δίκτυο δεν έχει την δυνατότητα να στέλνει μηνύματα τυχαίου μεγέθους. Υπάρχει ένα συγκεκριμένο μέγιστο μέγεθος αποστολής (MTU, Maximum Transmition Unit) το οποίο και δεν μπορούμε να υπερβούμε. Έτσι για να σταλεί ένα μήνυμα με μέγεθος μεγαλύτερο του MTU κόβεται σε μικρότερα κομάτια που είναι μικρότερο ή ίσο του MTU και τα ονομάζουμε πακέτα. Ένα μήνυμα με μέγεθος μικρότερο ή ίσο του MTU εγκλείεται μέσα σε ένα πακέτο. Ο δέκτης αναλαμβάνει να ‘‘ξανακολλήσει’’ αυτά τα πακέτα πριν παραδώσει



Σχήμα 2.3: Η υπερχείλιση του δέκτη

το μήνυμα στην εφαρμογή.

Αν και γενικά ο αριθμός των πακέτων που χάνεται είναι μικρός, και πάλι το κύριο πρόβλημα που αντιμετωπίζουμε είναι η απώλεια πακέτων στην διαδρομή. Δηλαδή, ενώ ο πομπός στέλνει N πακέτα, ο δέκτης τελικά αντιλαμβάνεται λιγότερα. Αυτό συμβαίνει για δύο κυρίως λόγους [1]:

- Ο δέκτης δεν έχει την δυνατότητα να διαβάσει τα πακέτα με τον ρυθμό που τα στέλνει ο πομπός (Σχήμα 2.3).
- Οι παρεμβαλλόμενοι δρομολογητές (routers) ή γέφυρες (bridges) δεν προλαβαίνουν να αναμεταδόσουν τα πακέτα από δίκτυο σε δίκτυο (Σχήμα 2.4).

Παρατηρούμε από τα παραπάνω ότι κύρια πηγή απώλειας πακέτων είναι η υπερχείλιση (overflow) των διάφορων συσκευών (interfaces) του δικτύου με αποτέλεσμα την απώλεια πληροφορίας.

Στην υλοποίησή μας ο κάθε πομπός διατηρεί μια ουρά από outgoing messages για κάθε κόμβο με τον οποίο μπορεί να επικοινωνήσει. Ανάλογα και κάθε δέκτης διατηρεί ουρές από εισερχόμενα μηνύματα από κάθε πομπό. Όταν θέλουμε να στείλουμε ένα μήνυμα σε κάποιον κόμβο το βάζουμε στην κατάλληλη ουρά και από κει και πέρα αναλαμβάνει το πρωτόκολλο να το στείλει με ασφάλεια στον προορισμό του.

Οι διαφορετικές ουρές για κάθε κόμβο υλοποιήθηκαν ώστε να μπορούμε να κρατάμε

[1] Παλιότερα υπήρχε και σημαντικός αριθμός πακέτων που έφταναν στον προορισμό τους αλλοιωμένα από το ίδιο το μέσο μεταφοράς (θόρυβος στις γραμμές κλπ.) που σε ένα λειτουργικό τοπικό δίκτυο δεν θεωρείται πλέον σημαντικό πρόβλημα.

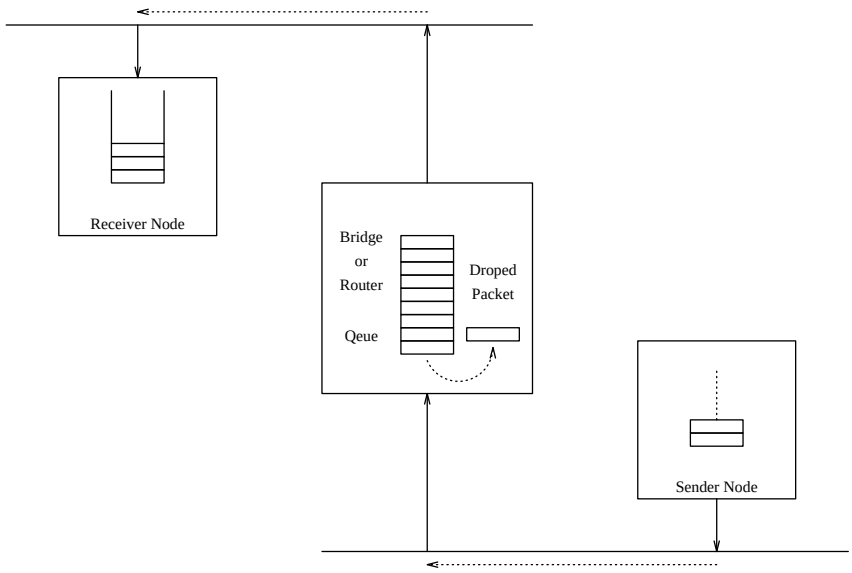
τα στοιχεία της επικοινωνίας με κάθε κόμβο χωριστά, και να πράττουμε ανάλογα με την συμπεριφορά του στις προηγούμενες εκπομπές. Το κύριο στοιχείο που καταχωρείται για κάθε κόμβο προς τον οποίο στέλνουμε messages είναι το *παράθυρο δεδομένων* που μπορεί να καταναλώσει χωρίς να συμβεί υπερχείλιση στο interface του καθώς και ο χρόνος που πρέπει να μεσολαβεί μεταξύ δύο εκπομπών.

Τα χαρακτηριστικά του πρωτόκολλου είναι: το ότι είναι ασύγχρονο, δεν έχει κόστος για τον εγκαινιασμό επικοινωνίας, ο πομπός είναι αισιόδοξος (την θεωρεί μεγάλη) για την ικανότητα του δέκτη να λάβει πακέτα, υλοποιεί την επιλεκτική επανεκπομπή (selective retransmit) σε περίπτωση απώλειας πακέτων και προσπαθεί να παράγει όσο το δυνατόν λιγότερες εκπομπές στο δίκτυο.

Η ασύγχρονη λειτουργία επιτυγχάνεται με τον κλασικό τρόπο των events (γεγονότων) όπου ο πομπός και ο δέκτης ανταποκρίνονται σε διάφορα γεγονότα όπως η λήψη ενός πακέτου, η εξάντληση ενός χρονομετρητή (timer) και η εντολή αποστολής ενός μηνύματος.

Οι επικοινωνίες μεταξύ δύο κόμβων εγκαινιάζονται έμμεσα (implicit connection setup). Με την παραλαβή του πρώτου μηνύματος προς κάποιον κόμβο στον πομπό, και αντίθετα με την άφιξη του πρώτου πακέτου στον δέκτη γίνεται και έμμεσα ο εγκαινιασμός της σύνδεσης η οποία θεωρείται λειτουργική από κει και πέρα.

Το πρωτόκολλο στέλνει χωρίς καθόλου έλεγχο ροής το πρώτο μήνυμα όσο μεγάλο και αν είναι. Στην περίπτωση (που είναι και αρκετά συχνή όταν ο δέκτης είναι ίσης ή μεγαλύτερης



Σχήμα 2.4: Η υπερχείλιση του δρομολογητή ή της γέφυρας

ισχύος από τον πομπό) που ο δέκτης επιβεβαιώσει την παραλαβή χωρίς λάθος, τότε και οι επόμενες αποστολές συνεχίζονται με αυτόν τον τρόπο. Σε αυτήν την περίπτωση έχουμε βέλτιστη χρήση και μέγιστη απόδοση, αφού κανένα επιπλέον πακέτο δεν εμφανίζεται στο δίκτυο.

2.1.2 Αλγόριθμος επανεκπομπής πακέτων

Τα πακέτα ενός μηνύματος είναι αριθμημένα. Ο δέκτης στέλνει την επιβεβαίωση παραλαβής του μηνύματος αφού διαβάσει και το τελευταίο πακέτο και έχει επιτυχώς ξανασυναρμολογήσει το μήνυμα.

Αν ο δέκτης λάβει κάποιο πακέτο εκτός της κανονικής σειράς, τότε υποθέτει ότι τα ενδιάμεσα πακέτα έχουν χαθεί, και στέλνει ένα μήνυμα στον πομπό ώστε να τα ξαναστείλει.

Σε περίπτωση που δεν ξαναλάβει καμιά απάντηση από τον πομπό ο δέκτης συνεχίζει να στέλνει τέτοια μηνύματα στον πομπό ώσπου αυτά να ξεπεράσουν κάποιο όριο οπότε υποθέτει ότι έχει χαθεί η επικοινωνία πλέον με αυτόν τον κόμβο.

Ο πομπός στέλνει τα πακέτα στον δέκτη και μετά περιμένει να έρθει η επιβεβαίωση. Αν στην διάρκεια της εκπομπής λάβει μήνυμα ότι ο δέκτης έχασε κάποια πακέτα, τότε σταματά την αποστολή, ξαναμαρκάρει τα χαμένα πακέτα για αποστολή και στέλνει την επόμενη φορά τόσα πακέτα, όσα κατάφερε να απορροφήσει ο δέκτης (παράθυρο αποστολής). Αυτό γίνεται με σκοπό να στέλνονται στον δέκτη τόσα πακέτα, όσα μπορεί με επιτυχία να απορροφήσει. Η παραπάνω διαδικασία επαναλαμβάνεται για τον νέο αριθμό πακέτων που υπολογίστηκε, δηλαδή το μήνυμα πλέον γίνεται το παράθυρο αποστολής, και επομένως κάθε μήνυμα κόβεται σε κομάτια μικρότερα ή ίσα με το παράθυρο αποστολής.

Στην περίπτωση που ο δέκτης ‘‘χάσει’’ όλα τα πακέτα από ένα σημείο του μηνύματος μέχρι το τέλος (για παράδειγμα το τελευταίο), τότε δεν μπορεί ούτε καν να διαπιστώσει την απώλειά τους. Για την αντιμετώπιση αυτού του προβλήματος και οι δυο πλευρές ενεργοποιούνται. Ο δέκτης μετά από τον χρόνο του round trip time στέλνει μηνύματα που ρωτούν τον πομπό γι αυτό το μήνυμα. Τα μηνύματα αυτά συνεχίζονται έως ένα προκαθορισμένο όριο, και μετά ο πομπός μαρκάρεται σαν μη προσπελάσιμος. Ο χρόνος κάθε φορά διπλασιάζεται. Ο πομπός αντίστοιχα, στέλνει σε τακτά χρονικά διαστήματα το τελευταίο πακέτο του μηνύματος στον δέκτη ώστε αυτός να αντιδράσει με τον τρόπο που περιγράφηκε παραπάνω. Όταν γίνει ανεπιτυχώς αυτή η αποστολή μερικές φορές, μέχρι ένα προκαθορισμένο όριο, τότε ο

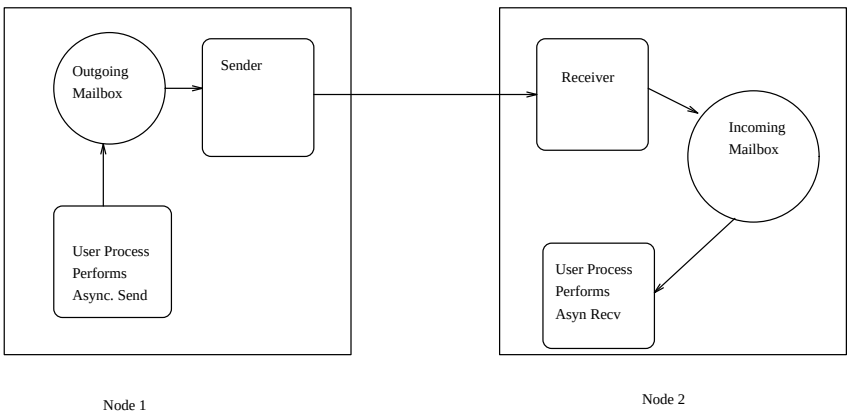
δέκτης μαρκάρεται σαν μη προσπελάσιμος. Την ίδια ενέργεια κάνει ο πομπός και στην περίπτωση που ο δέκτης δεν έλαβε κανένα πακέτο του μηνύματος και επομένως δεν μπορεί να αντιδράσει.

Το VMTP ορίζει και την ύπαρξη χρονικών διαστημάτων μεταξύ της αποστολής δύο συνεχόμενων πακέτων. Αυτό αν και θα ήταν πάρα πολύ βοηθητικό για την αποφυγή των υπερχειλίσεων των interfaces, ήταν αδύνατον να υλοποιηθεί με την αρχιτεκτονική των SparcStations που διαθέτει το τμήμα μας, αφού δεν διαθέτουν ρολόι μεγάλης ανάλυσης. Έτσι το μικρότερο χρονικό διάστημα που μπορεί να μετρηθεί είναι το $\frac{1}{60}$ του δευτερολέπτου που είναι πολύ μεγάλος χρόνος για να παρεμβληθεί μεταξύ δύο πακέτων. Στο μέλλον και με την αρχιτεκτονική SuperSparc, μαθαίνουμε ότι θα μπορεί να χρησιμοποιηθεί ρολόι υψηλής ανάλυσης. Για την επίλυση αυτού του προβλήματος χρησιμοποιείται η τεχνική του ‘‘self timing’’, δηλαδή ο χρόνος που σπαταλείται για τα πακέτα επιβεβαίωσης που στέλνει ο δέκτης για να επιβεβαιώσει την λήψη ενός παράθυρου πακέτων χρησιμοποιείται και σαν το χρονικό διάστημα που παρεμβάλλεται μεταξύ της αποστολής δύο συνεχόμενων παραθύρων.

Ένα άλλο χαρακτηριστικό της υλοποίησης είναι ότι ο δέκτης είναι ενεργός (active), δηλαδή δεν περιμένει από τον πομπό να καθορίζει τον χρόνο αποστολής του μηνύματος αλλά και τα timeouts, αλλά από μόνος του δηλώνει πότε είναι έτοιμος να δεχτεί πακέτα, είτε με την επιβεβαίωση παραλαβής ενός παραθύρου, είτε με τις συχνές (round trip time είναι ο χρόνος μεταξύ ερωτήσεων) ερωτήσεις που κάνει στον πομπό για την κατάσταση του μηνύματος, σε περίπτωση που χαθεί κάποιο πακέτο, ή δεν λαμβάνει πλέον πακέτα. Έτσι η αποστολή πακέτων γίνεται με τον γρηγορότερο δυνατό ρυθμό, αφού ο δέκτης καθορίζει τις δυνατότητές του.

2.1.3 Η Γενικότητα του πρωτοκόλλου

Το ερώτημα που εύλογα τίθεται είναι κατά πόσον ένα τέτοιου είδους πρωτόκολλο ασύγχρονης επικοινωνίας μπορεί να χρησιμοποιηθεί από εφαρμογές που τρέχουν σε σταθμούς εργασίας γενικής χρήσης. Ο τρόπος που δουλεύει συνήθως η ασύγχρονη επικοινωνία μεταξύ δύο κόμβων ενός κατανεμημένου συστήματος είναι με την χρήση mailbox. Δηλαδή σε κάθε κόμβο υπάρχει μια διαδικασία (π.χ. thread, interrupt handler κ.λπ.) πού αναλαμβάνει να φυλάει τα εισερχόμενα ασύγχρονα μηνύματα σε συγκεκριμένη δομή δεδομένων στην μνήμη. Ανάλογα υπάρχει μια άλλη διαδικασία που αναλαμβάνει την αποστολή των εξερχόμενων



Σχήμα 2.5: Τρόπος λειτουργίας των ασύγχρονων μηνυμάτων

μηνυμάτων. Η γενική ιδέα της ασύγχρονης επικοινωνίας φαίνεται στο σχήμα 2.5 Όταν τώρα κάποια διαδικασία χρειαστεί κάποιο μήνυμα το διαβάζει μέσα από την δομή των εισερχόμενων μηνυμάτων αν υπάρχει. Αν θέλει αντίστοιχα να στείλει μήνυμα το τοποθετεί στην δομή των εξερχόμενων μηνυμάτων.

Το πρόβλημα είναι ο καθορισμός του ανώτατου ορίου μνήμης και γενικότερα των resources που μπορούν να καταναλωθούν για την φύλαξη τέτοιων μηνυμάτων. Για παράδειγμα, αν υποθέσουμε ότι δεν υπάρχει κανένα όριο, τότε κάποια ακολουθία από λάθος μηνύματα θα μπορούσε να καταναλώσει όλη την μνήμη ενός κόμβου. Αυτό πιθανόν να είναι αποδεκτό σε συστήματα πολυεπεξεργαστών ενός χρήστη (single user), είναι όμως αδιανόητο για συστήματα με πολλούς χρήστες (multi user) αφού κάποιοι χρήστες μπορεί να αδικηθούν με αυτόν τον τρόπο. Η λύση που συνήθως δίνεται είναι ο ορισμός ανώτατων ορίων για την αποθήκευση, όπου μόλις αυτά τα όρια ξεπεραστούν τότε είτε τα νέα μηνύματα χάνονται, είτε σβήνεται κάποιο από τα είδη υπάρχοντα. Αυτό όμως κάνει και πάλι unreliable και ειδική την χρήση του πρωτοκόλλου και απαιτείται επιπλέον συγχρονισμός για να επιβεβαιώσουμε ότι δεν ξεπερνιούνται τα όρια αυτά με την αποστολή ενός μηνύματος.

Τέλος πρέπει να επισημάνουμε αυτό που έμμεσα ειπώθηκε στην αρχή. Ότι δηλαδή το πρωτόκολλο είναι προσαρμοσμένο σε τοπικά δίκτυα υψηλής ταχύτητας. Η χρήση του πάνω από αργές γραμμές ή γραμμές που επιφέρουν αλλοίωση των δεδομένων είναι μη αποδοτική.

2.1.4 Μετρήσεις απόδοσης

Το πρωτόκολλο όπως υλοποιήθηκε ήταν πλήρως ενσωματομένο με τον tuple manager. Για να γίνουν οι μετρήσεις απόδοσης φτιάξαμε ένα αυτόνομο πρόγραμμα που χρησιμοποιεί το πρωτόκολλο, και το οποίο στέλνει μηνύματα σε έναν απομακρυσμένο κόμβο. Τα μηνύματα αυτά περιέχουν τυχαία δεδομένα, και δεν απαιτείται χρόνος για να προετοιμασθούν. Τα μηνύματα διαφέρουν σε αριθμό και μέγεθος, ανάλογα με τα ορίσματα που δίνουμε στο πρόγραμμα. Οι μετρήσεις αφορούν τον χρόνο που στέλνεται ένα μήνυμα. Τα αποτελέσματα συγκρίνονται με αυτά του SunRPC. Για την μέτρηση της απόδοσης του SunRPC χρησιμοποιήσαμε ένα πρόγραμμα που στέλνει άδεια RPC calls σε ένα server. Όταν το μήνυμα είναι μικρό (μικρότερο από 8Kbytes) τότε το RPC χρησιμοποιεί UDP Datagrams, αλλιώς χρησιμοποιεί TCP/IP.

Ο φόρτος του δικτύου κατά την στιγμή της μέτρησης δεν λαμβάνεται υπόψην, αφού τα αποτελέσματα συγκρίνονται με αυτά του RPC που πάρθηκαν στα ίδια χρονικά διαστήματα και από τις ίδιες μηχανές.

Για τις μετρήσεις χρησιμοποιήσαμε την παρακάτω μέθοδο. Πραγματοποιήσαμε 100 κλήσεις ρουτίνας σε απομακρυσμένο κόμβο. Η ρουτίνα δεν περιείχε καθόλου επεξεργασία, και άρα ο χρόνος που μετρήθηκε είναι ο χρόνος αποστολής του μηνύματος.

Στο σχήμα 2.6 φένεται το διάγραμμα σύγκρισης των δύο μεθόδων αποστολής μηνυμάτων. Από το σχήμα φαίνεται καθαρά πόσο υπερτερεί σε ταχύτητα το πρωτόκολλο του NPOSYBL.

Στους πίνακες 2.1 φαίνονται αναλυτικά τα αποτελέσματα από τις μετρήσεις.

2.1.5 Επιλογή της μονάδας MTU

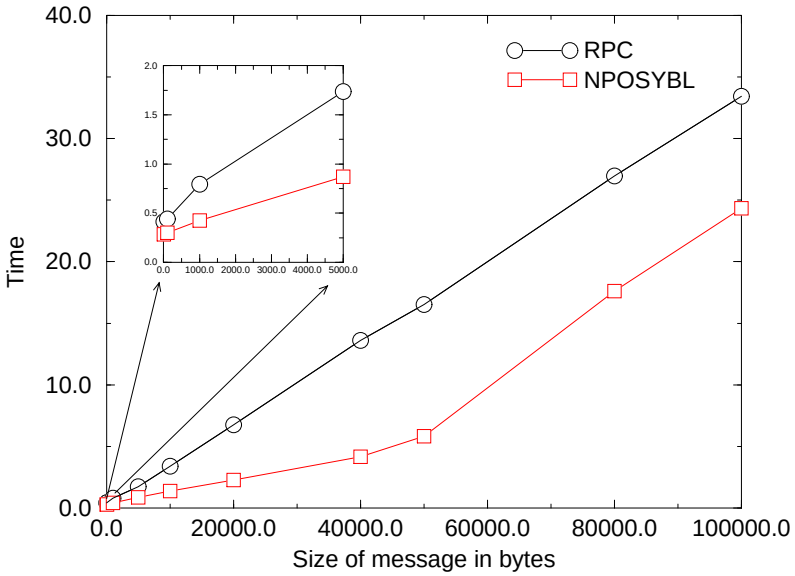
Στην περίπτωσή μας που χρησιμοποιούμε μια UDP socket σαν συσκευή (interface) δικτύου, το MTU μας είναι ορισμένο σε 4Kbytes. Οι τιμές που θα μπορούσε να πάρει το MTU είναι μέχρι και 8Kbytes, που είναι και το μέγιστο UDP Datagram. Η απόφαση να γίνει 4Kbytes πάρθηκε διότι αυτή η τιμή μας δίνει την καλύτερη μέση απόδοση. Μεγαλύτερο MTU δεν λειτουργεί καλά στην περίπτωση που παρεμβάλλονται routers και bridges, αφού μεγαλώνει πολύ η πιθανότητα απώλειας του πακέτου. Για να επαληθεύσουμε τους παραπάνω ισχυρισμούς κάναμε μετρήσεις. Οι μετρήσεις έγιναν με την εντολή του UNIX spray, όπου ένας πομπός στέλνει μια σειρά από πακέτα ορισμένου μήκους σε έναν δέκτη, και παρακολουθεί

NPOSYBL			RPC		
Μέγεθος Πακέτου σε Bytes	Χρόνος ενός μηνύματος	Bytes/sec	Μέγεθος Πακέτου σε Bytes	Χρόνος ενός μηνύματος	Bytes/sec
1	0.00285761	349.943	1	0.00417578	239.476
100	0.00301735	33141.6	100	0.00445613	22441
1000	0.00427989	233651	1000	0.00800552	124914
5000	0.00878788	568966	5000	0.017545	284981
10000	0.0139394	717389	10000	0.0344206	290524
20000	0.0227899	877583	20000	0.0682516	293034
40000	0.042	952380	40000	0.137635	290624
50000	0.0588415	849741	50000	0.166792	299774
80000	0.177994	449453	80000	0.272253	293845
100000	0.245935	406612	100000	0.337837	296001

Πίνακας 2.1: Χρόνος αποστολής κενού μηνύματος για τα πρωτόκολλα NPOSYBL και RPC.

From host	To host	size	number	Elapsed Time	Dropped Packets	%dropped
crete	melia	8000	5000	62.0	2562	51.24
crete	melia	4000	10000	105.6	1835	18.35
cerberus	leros	8000	5000	60.5	3575	71.5
cerberus	leros	4000	10000	100.4	1189	11.89
crete	rodos	8000	5000	51.7	5	0.10
crete	rodos	4000	10000	102.5	4	0.04
crete	cyprus	8000	5000	51.7	33	0.66
crete	cyprus	4000	10000	102.9	5	0.05

Πίνακας 2.2: Αποτελέσματα πειραμάτων με την χρήση της εντολής spray



Σχήμα 2.6: Διάγραμμα σύγκρισης απόδοσης μεταξύ NPOSYBL και RPC. Η συμπεριφορά γύρω από την αρχή των αξόνων φαίνεται σε μεγένθυση στο ένθετο σχήμα.

πόσα από αυτά έφτασαν και με ποιον ρυθμό. Αυτό που μετρήσαμε είναι πόσα από τα πακέτα χάνονται κατά την διαδρομή μέσα από το δίκτυο, σε σχέση με το μέγεθος των πακέτων. Τα μεγέθη που χρησιμοποιήθηκαν είναι 8000 και 4000 bytes. Ο συνολικός αριθμός δεδομένων που στέλναμε ήταν σταθερός, δηλαδή 5000 πακέτα των 8000 Bytes και 10000 πακέτα των 5000 bytes. Επίσης μεταξύ των πακέτων βάζουμε και κάποια σταθερή καθυστέρηση (παράμετρος -d της εντολής spray) ώστε οι απώλειες λόγω ανικανότητας του δέκτη να απορροφήσει πακέτα να ελαχιστοποιηθούν. Τα ορίσματα και η χρήση της εντολής υποδεικνύεται στο [14]. Στον παραπάνω πίνακα βλέπουμε καθαρά ότι τα μικρότερα πακέτα περνάνε με μεγαλύτερη πιθανότητα. Οι παραπάνω μετρήσεις έγιναν συνεχόμενα (μικρές αλλαγές στον φόρτο του δικτύου).

Μετά από συνεχείς μετρήσεις κατά την διάρκεια της ημέρας βλέπουμε ότι τα ποσοστά ίσως αλλάζουν, αλλά γενικά είναι πολύ λιγότερα τα πακέτα που χάνονται όταν το μέγεθος πακέτου είναι 4000 Bytes.

2.2 Επικοινωνία process, tuple manager

Η επικοινωνία των διεργασιών linda με το tuple space γίνεται μέσω των tuple managers που μέσα στο address space τους αποθηκεύουν τα tuples. Ιδεατά, οι διεργασίες που τρέχουν στον ίδιο κόμβο θα έπρεπε να μοιράζονται το τοπικό tuple space και να έχουν απευθείας πρόσβαση σε αυτό. Οι περισσότερες υλοποιήσεις του UNIX, παρέχουν την δυνατότητα κοινής μνήμης (shared memory) που επιτρέπει σε δύο διεργασίες να έχουν απευθείας πρόσβαση σε κοινά κομμάτια μνήμης. Αυτή η λύση όμως αντιμετωπίζει το πρόβλημα ότι η κοινή μνήμη δεν μπορεί να υπερβεί το μέγεθος της φυσικής μνήμης της μηχανής, δεν υπάρχει τρόπος προστασίας της από λάθη του προγραμματιστή και τέλος δεν μπορούμε να χρησιμοποιήσουμε τα εργαλεία του συστήματος για memory management. Το τελευταίο πρόβλημα δεν είναι πολύ σοβαρό αφού μπορεί να αντιμετωπισθεί εύκολα, είτε υλοποιώντας ξανά τις ρουτίνες διαχείρισης μνήμης, είτε χρησιμοποιώντας κάποια άλλη έτοιμη υλοποίηση που διατίθεται μαζί με τον κώδικα.

Έτσι το βάρος δόθηκε στην βελτιστοποίηση της επικοινωνίας μεταξύ των τοπικών διεργασιών και του tuple manager.

Τα κριτήρια που καθορίζουν την αποδοχή του IPC, είναι το μικρό κόστος σε απόλυτο χρόνο, το μικρό κόστος σε χρόνο εξυπηρέτησης από το σύστημα (system time), η δυνατότητα για χρήση select και τέλος η συμπεριφορά του σε επικοινωνία πολλών διεργασιών ταυτόχρονα. (πολλές διεργασίες που επικοινωνούν ταυτόχρονα με τον tuple manager).

Οι επιλογές είναι πολλές στην περίπτωση του UNIX.

- Πρώτη επιλογή είναι η χρήση UNIX domain socket, που μας δίνει καλό bandwidth, αλλά καταναλώνει πολλούς file descriptors (έναν για κάθε τοπική διεργασία) και δεν συμπεριφέρεται καλά στην περίπτωση της multiuser επικοινωνίας αφού έχει υψηλό system time. Επίσης δεν είναι ασύγχρονη η επικοινωνία που επιτυγχάνεται με την χρήση τους. Παρόλα αυτά, δίνει την δυνατότητα select και έχει πολύ καλή απόδοση για επικοινωνία μεταξύ δύο διεργασιών (single user). Μια παραλλαγή θα ήταν η χρήση fifo πού έχει όμως τα ίδια προβλήματα.

- Η χρήση του πρωτοκόλλου που υλοποιήσαμε για την inter-node επικοινωνία αλλά με χρήση UNIX domain UDP socket. Αυτή η λύση δεν εξετάστηκε καθόλου λόγω έλλειψης χρόνου.

- Η χρήση κοινής μνήμης για επικοινωνία, και κάποια μορφή συγχρονισμού. Εδώ εξετάστηκε η χρήση semaphores, που όμως δεν επιτρέπουν το select και είναι πολύ ακριβές, η χρήση message queue σαν binary semaphore, που έχει και πάλι το μειονέκτημα ότι δεν υποστηρίζει το select, και τέλος η χρήση μιας fifo και μιας message queue που επιτρέπει το select.

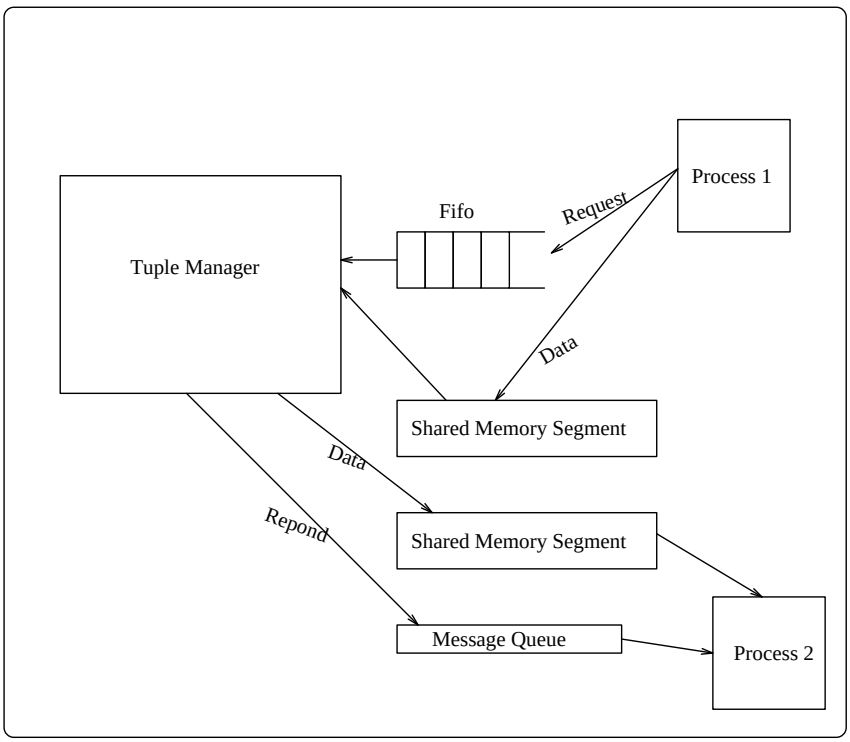
Η τελική υλοποίηση χρησιμοποιεί fifo και message queue για συγχρονισμό, και κοινή μνήμη για επικοινωνία. Η fifo χρησιμοποιείται για να στέλνει control μηνύματα μια διεργασία στον tuple manager. Η message queue χρησιμοποιείται για την αποστολή control μηνυμάτων από τον tuple manager προς τις διεργασίες. Τέλος για κάθε διεργασία που επικοινωνεί με τον tuple manager δημιουργείται ένα shared memory segment που χρησιμοποιείται για την ανταλλαγή των δεδομένων. Η fifo και η message queue χρησιμοποιούνται και για τον σκοπό του συγχρονισμού αφού οι κλήσεις σε αυτές είναι blocking. Η fifo όπως αναφέρθηκε και προηγουμένως μας δίνει την δυνατότητα να κάνει select ο tuple manager για να περιμένει μήνυμα από τις διεργασίες. Ένα επιπλέον πλεονέκτημα που μας δίνει η χρήση αυτών των συγκεκριμένων εργαλείων για επικοινωνία είναι η επιπλέον δυνατότητα να περνάμε μαζί με τα μηνύματα ελέγχου επιπλέον πληροφορίες. Αντίθετα η χρήση σηματοφόρων για παράδειγμα δεν επιτρέπει το πέρασμα πληροφορίας μέσα από αυτές εκτός ίσως από την πληροφορία που μπορεί να περιληφθεί στην τιμή της σηματοφόρου.

Μόλις ξεκινάει ο tuple manager σε κάποιον κόμβο, δημιουργεί μια UNIX fifo και μια message queue, που είναι μοναδικές και δεν αλλάζουν όσο ‘‘ζει’’ ο tuple manager.

Κάθε διεργασία χρήστη που δημιουργείται ζητάει από το λειτουργικό την δημιουργία ενός κομματιού κοινής μνήμης με σταθερό για όλους μέγεθος. Το μέγεθος ορίζεται εσωτερικά από το σύστημα. Με την χρήση της fifo ειδοποιεί τον tuple manager για την ύπαρξή της και για το ποιό κομμάτι κοινής μνήμης θα χρησιμοποιηθεί για την μεταξύ τους επικοινωνία.

Από εκείνο το σημείο και μετά χρησιμοποιείται ένα πρωτόκολλο επικοινωνίας που αναλαμβάνει την επικοινωνία μεταξύ διεργασίας χρήστη και tuple manager.

Στο πρωτόκολλο αυτό η διεργασία χρήστη βάζει τα δεδομένα που έχει για αποστολή στον tuple manager στην κοινή μνήμη και στέλνει μέσω της fifo μήνυμα που ειδοποιεί τον tuple manager να παραλάβει τα δεδομένα. Μόλις ο tuple manager διαβάσει τα δεδομένα από την κοινή μνήμη, ειδοποιεί την διεργασία χρήστη μέσω της message queue. Αν έχει και ο tuple manager δεδομένα να στείλει προς την διεργασία χρήστη, πριν ειδοποιήσει για την παραλαβή



Σχήμα 2.7: Η λειτουργία του IPC πρωτόκολλου

των δεδομένων τα τοποθετεί στην κοινή μνήμη.

Σε περίπτωση που η κοινή μνήμη δεν είναι αρκετή για να χωρέσει όλα τα δεδομένα που πρέπει να ανταλλαχθούν, τότε αυτά κόβονται σε παραπάνω από ένα μέρη και επαναλαμβάνεται η παραπάνω διαδικασία.

Ένα σημείο που πρέπει να προσεχθεί στην χρήση fifos είναι ότι οι πράξεις write σε αυτές είναι ατομικές μόνο όταν υπάρχει αρκετή μνήμη προσωρινής αποθήκευσης (buffer space). Σε περίπτωση που δεν υπάρχει, τότε οι διεργασίες που θέλουν να γράψουν γίνονται blocked. Όταν αδειάσει ξανά χώρος στην μνήμη προσωρινής αποθήκευσης της fifo, τότε δεν μπορούμε να ξέρουμε ποια από τις διεργασίες θα ξεμπλοκάρει πρώτη και θα συνεχίσει να γράφει στην fifo. Το πρόβλημα αυτό αντιμετωπίστηκε με την χρήση μηνυμάτων που ήταν ακέραιο υποπολλαπλάσιο της μνήμης προσωρινής αποθήκευσης της fifo, και πάντα σταθερού μεγέθους. Το ότι τα μηνύματα έγιναν σταθερού μεγέθους σημαίνει ότι κάποιο τμήμα τους μπορεί να έμενε αχρησιμοποίητο. Αυτό το τμήμα όμως το χρησιμοποιήσαμε για την μεταφορά κάποιων δεδομένων. Έτσι στην πολύ συχνή περίπτωση που η επικοινωνία γίνεται από τις διαδικασίες χρήστη προς τον tuple manager με μηνύματα μερικών bytes, αυτά μεταφέρονται μέσα από την fifo και δεν χρειάζεται η χρησιμοποίηση της κοινής μνήμης και

έξτρα συγχρονισμού.

Και πάλι η μέθοδος αυτή επικοινωνίας είναι ειδικά βελτιστοποιημένη για την χρήση του NPOSYBL. Δηλαδή υποστηρίζει το μοντέλο όπου πολλές διεργασίες (clients) επικοινωνούν με μια άλλη (master). Οι διαδικασίες που υποστηρίζει είναι οι:

- client_send(), μια διεργασία στέλνει δεδομένα. Η διεργασία γίνεται block έως ότου διαβάσει τα δεδομένα η διεργασία master.
- client_receive(), μια διεργασία περιμένει να λάβει δεδομένα από τον master. Η διεργασία γίνεται block μέχρι να λάβει τα δεδομένα.
- master_send(), Ο master στέλνει ασύγχρονα δεδομένα σε έναν client. Πολλά τέτοια calls μπορεί να είναι ενεργοποιημένα ανά πάσα στιγμή.

Η διαδικασία master_receive() είναι έμμεσα υλοποιημένη αφού ο master βρίσκεται πάντα σε receive mode. Όταν ο master διαβάσει ένα ολόκληρο μήνυμα τότε καλεί την κατάλληλη RPC ρουτίνα.

2.2.1 Μετρήσεις απόδοσης

Για την μέτρηση της απόδοσης του μηχανισμού τοπικής επικοινωνίας φτιάξαμε 2 νέα προγράμματα που χρησιμοποιούν αυτό το πρωτόκολλο για να ανταλλάσσουν άδεια μηνύματα.

Τα αποτελέσματα τα συγκρίνουμε με την χρήση μιας *UNIX domain STREAM socket*.

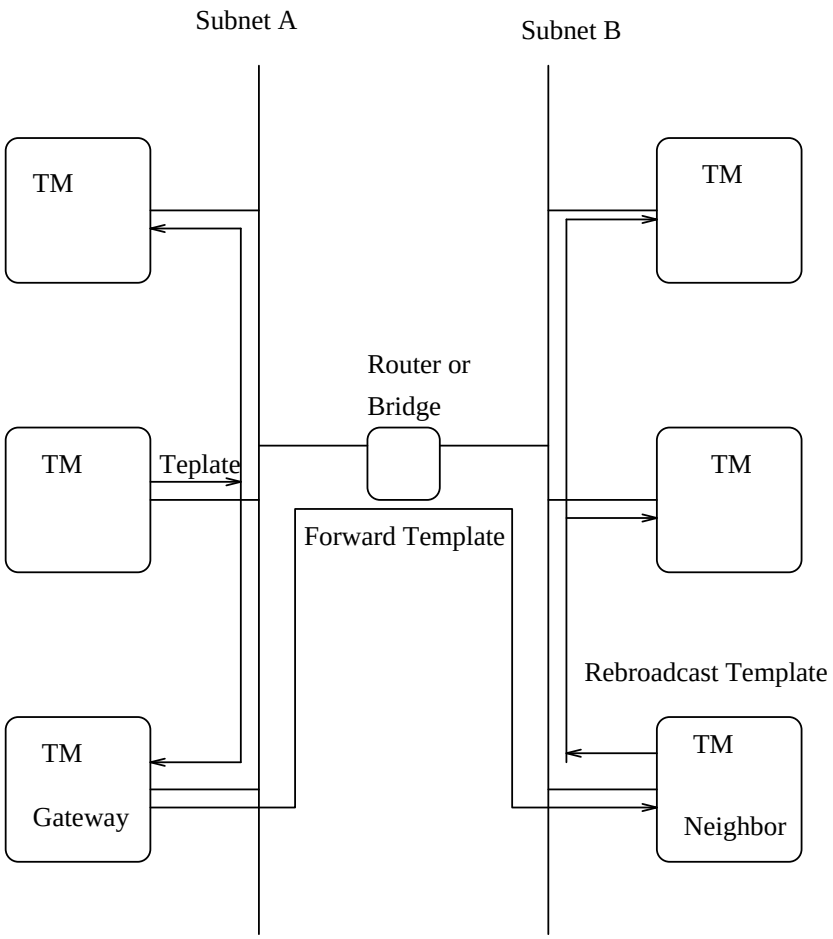
Όπως φαίνεται και στον πίνακα 2.3 ό δικός μας τρόπος επικοινωνίας είναι καλός όσο το μέγεθος της κοινής μνήμης είναι μεγαλύτερο από τα μεταφερόμενα μηνύματα. Μετά αρχίζει να πέφτει η απόδοσή του πάλι με ρυθμό που εξαρτάται από το μέγεθος της κοινής μνήμης.

2.3 Χρήση gateways για τα broadcasts

Στις περισσότερες εγκαταστάσεις δικτύων τα broadcasts είναι δυνατά μόνον μέσα σε ένα segment και απαγορεύεται η αναμετάδοσή τους από τα bridges ή τα gateways. Το POSYBL βασίζεται στα broadcasts για την μετάδοση των templates, και άρα δεν θα μπορούσε

	Χρόνος σε δευτερόλεπτα			
Μέγεθος σε Bytes	UNIX	Μέγεθος κοινής μνήμης 10K	30K	100K
1	0.242158	0.364004	0.367347	0.372511
100	0.385989	0.375474	0.374913	0.425222
1000	0.567381	1.069450	1.076425	1.076070
5000	1.449594	1.509998	1.517930	1.510448
10000	2.505338	1.948985	1.946979	1.947390
20000	4.654010	6.049740	2.878756	2.874110
40000	8.937574	11.920722	10.262180	4.882620
50000	11.072116	14.969802	12.329462	6.021644
80000	17.921936	24.311254	19.785696	9.177687
100000	23.157321	30.324900	25.175392	11.127055

Πίνακας 2.3: Αποτελέσματα μετρήσεων αποστολής 1000 μηνυμάτων με χρήση Unix Domain socket και του NPOSYBL με διάφορα μεγέθη κοινής μνήμης



Σχήμα 2.8: Οι gateways και η λειτουργία τους

να χρησιμοποιηθεί διαμέσου υποδικτύων χωρίς κάποιον τρόπο μεταφοράς και επανεκπομπής των templates σε κάθε ένα υποδίκτυο.

Για τον σκοπό αυτό, κάποιοι tuple managers αναλαμβάνουν αυτό το καθήκον και ονομάζονται και gateway tuple managers.

Ο αλγόριθμος είναι ο παρακάτω: Κάθε tuple manager ξέρει από το configuration file που διαβάζει στο ξεκίνημά του αν είναι και gateway. Επίσης γνωρίζει και σε ποιόν tuple manager θα πρέπει να κάνει forward τις templates.

Κάθε φορά που ο gateway tuple manager αντιληφθεί μια template, τότε αφού εξετάσει αν ο ίδιος μπορεί να την εξυπηρετήσει ώστε να αποφύγει την περαιτέρω αποστολή την αποστέλει χρησιμοποιώντας το send_message_noack() στον tuple manager πού του έχουμε ορίσει σαν γείτονα. Ο γείτονας εξετάζει και αυτός αν μπορεί να εξυπηρετήσει την template και αν και αυτός δεν μπορεί τότε την κάνει broadcast στο τοπικό του υποδίκτυο.

Στο σχήμα 2.3 φαίνεται ένα παράδειγμα όπου το σύστημα τρέχει χρησιμοποιώντας 2 υποδίκτυα. Ένας tuple manager στο υποδίκτυο A κάνει broadcast μια template και οι υπόλοιποι την διαβάζουν από το δίκτυο. Ο gateway tuple manager, αφού εξετάσει αν αυτός μπορεί να εξυπηρετήσει την template, την στέλνει στον γείτονά του (neighbor TM) στο υποδίκτυο B. Αφού και ο γείτονας εξετάσει αν μπορεί να εξυπηρετήσει την template, την κάνει broadcast με την σειρά του στο υποδίκτυο B.

Παρατηρούμε ότι για την μετάδοση της template δημιουργήθηκαν τρία μηνύματα. Ένα το αρχικό broadcast, ένα για την επικοινωνία των gateway tuple managers, και ένα για το broadcast στο υποδίκτυο B. Στην περίπτωση που η γέφυρα ή ο δρομολογητής έκανε αυτήν την επανεκπομπή θα είχαμε μόνο το αρχικό και το τελικό broadcast.

2.4 Debugging, monitoring εργαλεία

Ένα πλήρες σύστημα ανάπτυξης απαιτεί και την ύπαρξη εργαλείων για debugging και monitoring της εκτέλεσης του προγράμματος. Γι' αυτόν τον σκοπό υλοποιήθηκαν 2 εργαλεία.

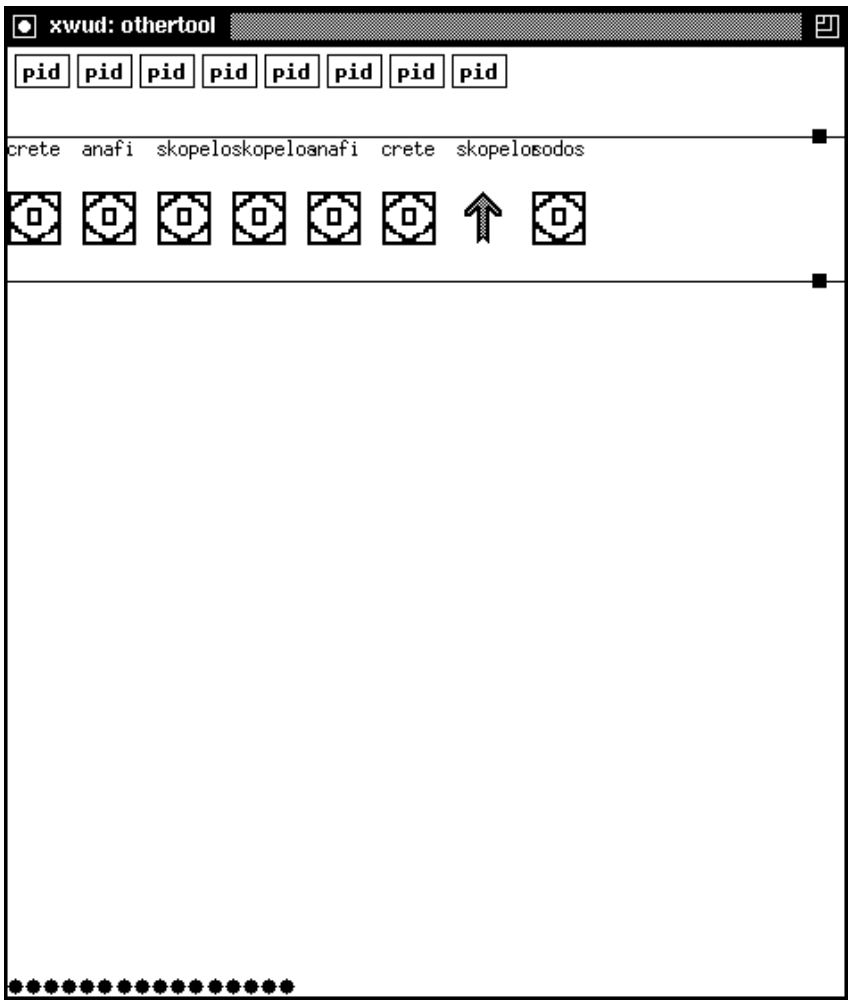
1. Ένα γραφικό display της συχνότητας των πράξεων που συμβαίνουν σε έναν κόμβο, (in, out, rd etc).

2. Ένα εργαλείο πού δείχνει τις πράξεις που εκτελεί κάθε process, όπως και τις process που υπάρχουν.

Η παρατήρηση των πράξεων στο tuple space είναι πολύτιμη, και μπορεί να μας δώσει στοιχεία για να αντιληφθούμε το communication και computation pattern του προγράμματός μας.

Για την υλοποίηση του πρώτου εργαλείου χρησιμοποιήθηκαν τα στατιστικά στοιχεία που κρατά κάθε tuple manager. Η όψη του εργαλείου είναι σαν αυτή του xload. Το εργαλείο δείχνει σε μια γραφική παράσταση τον αριθμό πράξεων Linda που συμβαίνουν μεταξύ 2 χρονικών διαστημάτων μέτρησης. Το αντίστοιχο X widget προσαρμόστηκε, ώστε εκτός από το γράφημα να δείχνει τον αριθμό των πράξεων σε αριθμητική μορφή, καθώς και το τη μετράμε κάθε φορά. Με την χρήση ενός menu επιλέγουμε την μέτρηση που θέλουμε να βλεπουμε κάθε φορά. Το εργαλείο μέτρησης ζητάει από τον κόμβο που έχει προκαθοριστεί

t



Σχήμα 2.9: Το tupleTool

στο ξεκίνημά του τα στατιστικά στοιχεία που διατηρεί. Αυτό γίνεται σε προκαθορισμένα χρονικά διαστήματα.

Το δεύτερο εργαλείο, διατηρεί μια πιο κατατοπιστική εικόνα του POSYBL προγράμματος, δείχνοντας τις ενεργές διεργασίες, τις πράξεις που εκτελεί η κάθε μια και αναλυτικά τα tuples που υπάρχουν στο tuple space (Σχήμα reffig-ttool). Προς το παρόν το εργαλείο αυτό κάνει μόνο monitoring, αλλά μπορεί εύκολα να επεκταθεί ώστε διαλογικά ο χρήστης να μπορεί να ελέγχει τις πράξεις και να καθορίζει το αν και πότε θα εκτελεστούν.

Επίσης στο μέλλον θα δείχνονται πληροφορίες όπως σε ποια γραμμή του κώδικα βρίσκεται η κάθε πράξη και σε ποιό αρχείο, ποια είναι τα πεδία του tuple ή template και ποιες είναι οι τιμές των actual πεδίων και τέλος η δυνατότητα εξέτασης των τιμών που έχουν τα tuples μέσα στο tuple space.

Για την υλοποίηση αυτού του εργαλείου, το ίδιο το σύστημα του POSYBL χρησιμοποιείται. Δηλαδή ο ”debugger” είναι μια POSYBL process που συμμετέχει στην εκτέλεση του προγράμματος. Πριν μια πράξη Linda εκτελεστεί, το σύστημα παράγει ένα info tuple, το οποίο στην συνέχεια διαβάζει ο debugger. Για να εκτελεστεί η πράξη περιμένει να διαβάσει το permition tuple που παράγει ο debugger. Στην παρούσα υλοποίηση ο debugger παράγει αμέσως αυτό το permition tuple. Στο μέλλον θα ζητείται και η διαλογική έγκριση του χρήστη, στην περίπτωση που έχει επιλέξει να συμμετέχει στην εκτέλεση του προγράμματος.

Τα εργαλεία αυτά δίνουν στοιχεία σε communication level. Δεν είναι δηλαδή πλήρεις debuggers με grain εντολής, αλλά το στοιχείο που εξετάζουν είναι οι πράξεις Linda (κύρια οι πράξεις επικοινωνίας). Υποθέτουμε ότι ο χρήστης θα έχει κάνει το debugging των αλγορίθμων του ξεχωριστά και κατόπιν θα τους συνδυάσει σε ένα NPOSYBL πρόγραμμα.

Κεφάλαιο 3

Μέθοδοι για την βελτιστοποίηση του συστήματος

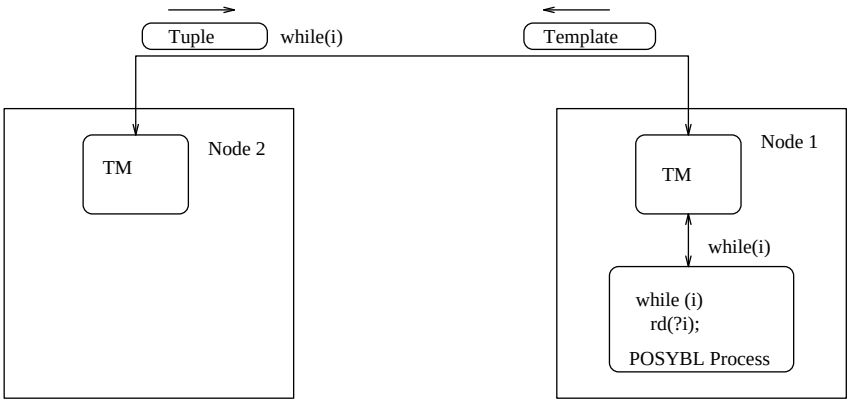
Όσο γρήγορος κι αν είναι ο τρόπος επικοινωνίας μεταξύ των διεργασιών δεν παύει να αποτελεί το μεγαλύτερο παράγοντα καθυστέρησης. Η αποστολή του πιο μικρού πακέτου στο ethernet διαρκεί τουλάχιστον 1ms, άσχετα με το πόσο καλός ή κακός είναι ο αλγόριθμος αποστολής μηνυμάτων. Ένα σύστημα πρέπει να περιορίσει όσο το δυνατόν την εκπομπή άσκοπων μηνυμάτων για να βελτιστοποιήσει τον χρόνο εκτέλεσής των προγραμμάτων.

Ένας άλλος παράγοντας που επηρεάζει την αποτελεσματικότητα της εκτέλεσης του προγράμματος είναι η ισοκατανομή του φόρτου στις διάφορες μηχανές. Και ο πιο τέλειος αλγόριθμος θα αποτύχει αν δεν καταφέρει να μοιράσει την εργασία κατάλληλα ώστε κάθε κόμβος να πάρει υπολογιστικό φορτίο ανάλογα με τις δυνατότητές του.

Σε ένα κατανεμημένο περιβάλλον υπάρχει και η περίπτωση να μην ολοκληρωθεί η εκτέλεση κάποιου προγράμματος λόγω διακοπής λειτουργίας ενός από τους κόμβους που συμμετέχουν.

Τέλος, ιδιαίτερη σημασία έχει ο αλγόριθμος με τον οποίο αποθηκεύονται τα tuples μέσα στο tuple space. Το tuple space είναι ένας χώρος όπου τα δεδομένα φυλάσσονται για μικρά χρονικά διαστήματα. Ταυτόχρονα πρέπει να είναι όσο το δυνατόν πιο γρήγορη και αποτελεσματική η ανεύρεσή τους μέσα σε αυτό. Άρα ο αλγόριθμος πρέπει να επιτρέπει πολύ γρήγορο ψάξιμο, και ταυτόχρονα να μην σπαταλάει ιδιαίτερο χρόνο στην αποθήκευση αφού το πιθανότερο είναι ότι κάθε tuple σύντομα θα αφαιρεθεί.

Στις παρακάτω παραγράφους θα δούμε μερικές μεθόδους που υλοποιήσαμε για να



Σχήμα 3.1: Η πράξη rd χωρίς replication

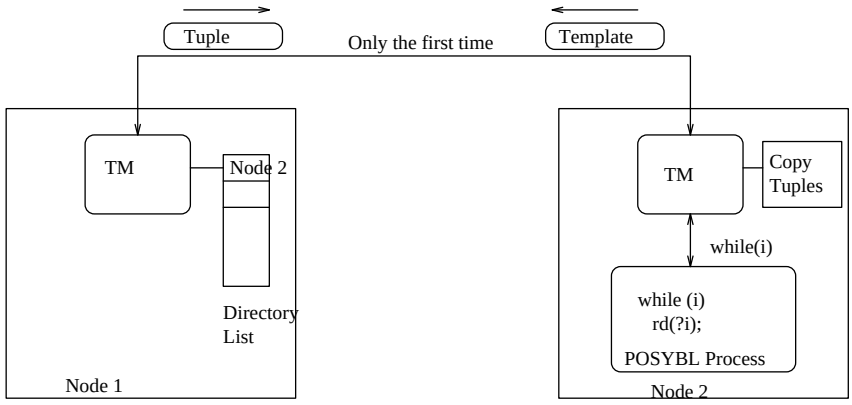
επιλύσουν ή να απαλύνουν τα παραπάνω προβλήματα.

3.1 Replication για πιο γρήγορα reads

Όπως αναφέρθηκε και παραπάνω, το **rd** είναι η πράξη της Linda που επιτρέπει την ανάγνωση πληροφορίας, χωρίς να διαγράφεται από το tuple-space. Αυτή η πράξη μπορεί να χρησιμοποιηθεί για την δημιουργία spin-locks, την μεταφορά write-once read-many πληροφορίας κτλ. Η αρχική υλοποίηση του POSYBL μεταχειριζόταν το rd με κεντρικοποιημένο τρόπο. Κάθε rd εξυπηρετούνταν από τον ιδιοκτήτη του tuple, άσχετα με το πόσες φορές διαβάζονταν, εισάγοντας σοβαρό επιπλέον κόστος (Σχήμα 3.1) όταν το ζητούμενο tuple βρίσκονταν σε άλλο κόμβο από αυτό που τρέχει η process.

Η λύση που υιοθετήθηκε είναι το replication. Δηλαδή το rd ενός tuple από remote tuple manager γίνεται μόνο την πρώτη φορά. Τις υπόλοιπες υπάρχει ένα τοπικό αντίγραφο που χρησιμοποιείται για την περαιτέρω εξυπηρέτηση των rd πράξεων που ικανοποιεί αυτό το tuple (Σχήμα 3.1).

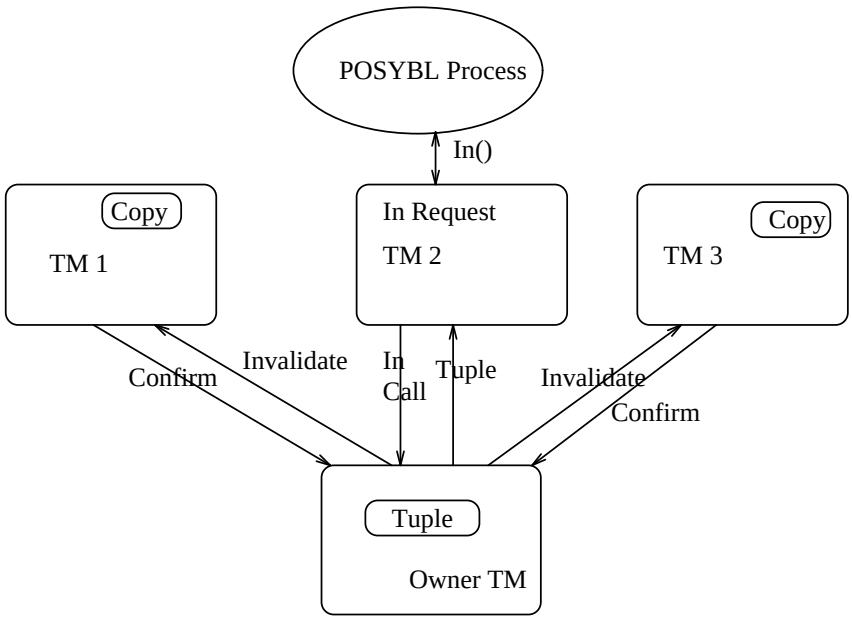
Το replication όμως εισάγει το νέο πρόβλημα της συμβατότητας (consistency) των αντιγράφων. Αυτό συμβαίνει όταν κάποιος αποσύρει (κάνει in) το tuple από το tuple space. Τότε πρέπει, και πριν ολοκληρωθεί η πράξη in, να ακυρώσουμε τα αντίγραφα του tuple που πιθανός υπάρχουν σε κάποιον απομακρυσμένο (remote) tuple manager. Το σχήμα συμβατότητας (consistency) που υλοποιήθηκε είναι directory[15], όπου υπάρχει ένας ιδιοκτήτης και αυτοί που έχουνε αντίτυπα. Ο ιδιοκτήτης κρατάει στο directory τους κόμβους που έχουν αντίτυπο του tuple. Η ύπαρξη αντίτυπων αυξάνει φυσικά τις απαιτήσεις σε



Σχήμα 3.2: Η πράξη rd με replication

μνήμη, μια και κάθε αντίτυπο καταναλώνει ίση μνήμη με το πρωτότυπο tuple. Επίσης δεν υλοποιήθηκε κανένας μηχανισμός που να ακυρώνει αυτόματα ένα αντίτυπο αν δεν ζητηθεί για πολύ χρόνο. Η βελτίωση αυτή όμως είναι εύκολο να υλοποιηθεί.

Η αφαίρεση του tuple από το tuple-spase, γίνεται με ένα broadcast που ειδοποιεί όλους τους κατόχους αντίτυπου να το ακυρώσουν. Μόλις τελειώσει αυτή η διαδικασία τότε εκτελείται και η πράξη διαγραφής (in) (Σχήμα 3.1). Αν και εισέρχεται επιπλέον overhead στην διαγραφή tuple, αυτό είναι πολύ μικρό μπρος στο όφελος που έχουμε λόγω του replication. Όμως ο προγραμματιστής πρέπει να το λαμβάνει υπόψη του όταν πρόκειται να χρησιμοποιήσει **rd**. Για read once δεδομένα το κόστος του **rd** είναι το ίδιο με αυτό του in. Στην περίπτωση



Σχήμα 3.3: Η πράξη διαγραφής σε replicated tuple

μάλιστα που ακολουθεί in το κόστος είναι μεγαλύτερο αφού πρέπει να ακυρωθούν και τα αντίτυπα. Μια άλλη αντιμετώπιση που θα μπορούσε να εξετασθεί γι αυτήν την περίπτωση είναι η εισαγωγή μιας επιπλέον πράξης παρόμοιας με το rd που όμως δεν θα χρησιμοποιεί replication.

Ένα σημείο που απλοποιεί σημαντικά την υλοποίηση, είναι η χαλαρή έννοια του consistency που υπάρχει στην Linda. Όταν αρχίσει να εκτελείται το in δεν υπάρχει πρόβλημα να συνεχίσει κάποιος να εκτελεί rd. Για παράδειγμα η παρακάτω ακολουθία εντολών, ενώ είναι σε συμφωνία με το μοντέλο της Linda, δεν έχει μοναδικό αποτέλεσμα στην μεταβλητή i:

```
out(”int”, 7);
out(”barrier”);
rd(”barrier”);          rd(”barrier”);
in(”int”, ?i);
out(”int”, i+1);          rd(”int”,?i);
```

Το i μετά το rd μπορεί να έχει την τιμή 7, ή την τιμή 8.

Για να δοκιμάσουμε αυτήν την λύση χρησιμοποιήσαμε το πρόγραμμα πολλαπλασιασμού πινάκων, το οποίο εκτελεί πολλά rd, κάνοντας rd όλες τις στήλες, για κάθε γραμμή που πολλαπλασιάζει. Τα αποτελέσματα είναι θεαματικά μια και ο χρόνος εκτέλεσης του προγράμματος υποδιπλασιάζεται.

3.2 Broadcast based Linda

Το POSYBL σε όλες του τις εκδόσεις χρησιμοποιεί broadcasts για την γνωστοποίηση των template’s στους υπόλοιπους tuple managers.

Αυτή η μέθοδος μας επιτρέπει να αποθηκεύουμε τα tuples που παράγει η κάθε διεργασία στον τοπικό της κόμβο και μας δίνεται η δυνατότητα να εκμεταλλευτούμε την τοπικότητα των αναφορών (locality of reference) στο tuple space. Για παράδειγμα, σε μια εφαρμογή επεξεργασίας εικόνων με πολλές φάσεις υπολογισμών, τα δεδομένα που παράγει κάθε διεργασία σε μια αρχική φάση υπολογισμού, πιθανότατα θα τα χρησιμοποιήσει σε μια από τις επόμενες φάσεις.

Αντίθετα, στην περίπτωση που χρησιμοποιούμε το tuple space σαν ένα μέσο ανταλλαγής μηνυμάτων (message passing model), τότε το μοντέλο δεν λειτουργεί αποδοτικά για δυο

λόγους. Πρώτος λόγος είναι το επιπλέον κόστος που προστίθεται για την αναζήτηση ενός tuple μέσω δικτύου (αυτό το κόστος υπάρχει σε όλες τις υλοποιήσεις Linda) και δεύτερος λόγος είναι το μεγάλο κόστος του broadcast. Όπως έχουμε πει, το broadcast είναι απαραίτητο αφού δεν ξέρουμε εκ των προτέρων σε ποιόν tuple manager είναι πιθανόν να βρίσκεται το tuple που ζητάμε. Κάνοντας όμως broadcast αναγκάζουμε όλους τους tuple managers να επεξεργαστούν την template ακόμα και αν δεν συμμετέχουν καν στο συγκεκριμένο πρόγραμμα. Αυτό σημαίνει ότι κάτω από βαριά χρήση και συχνές αναζητήσεις tuples, μεγάλο τμήμα του χρόνου καταναλώνεται στην εξυπηρέτηση των templates. Αυτός ο χρόνος όμως είναι χρόνος που χάνεται από την επεξεργασία, και άρα το πρόγραμμα δεν κλιμακώνει την απόδοσή του ικανοποιητικά.

Εδώ συναντώνται τα εξής δύο μοντέλα:

1. distributed data structures, όπου το πρόγραμμα σπάνια χρησιμοποιεί την κοινή μνήμη για να συμβουλευθεί πληροφορίες. Οι πληροφορίες έχουν μεγάλο όγκο και είναι αδύνατον να τις κρατήσει ένας κόμβος.

2. message passing programming, όπου το πρόγραμμα χρησιμοποιεί την κοινή μνήμη πολύ συχνά για την ανταλλαγή πληροφορίας.

Παρακάτω θα εξετάσουμε μερικές απλές μεθόδους που υλοποιήσαμε για να ξεπεράσουμε το πρόβλημα του broadcast στην περίπτωση που το μοντέλου προγραμματισμού είναι message passing ή το σχήμα επικοινωνίας μπορεί να ανιχνευθεί εύκολα από το σύστημα.

3.2.1 Βελτιώσεις χρησιμοποιώντας στατιστική

Η ανάγκη του broadcast προέρχεται από το γεγονός ότι στο POSYBL δεν μπορούμε να ξέρουμε σε ποιόν κόμβο υπάρχει το tuple που ζητάμε. Παρόλα αυτά είναι πολλές φορές εύκολο να αναγνωρίσουμε το πρότυπο (pattern) της επικοινωνίας που διέπει το πρόγραμμα. Για παράδειγμα στην περίπτωση που έχουμε πολλούς workers που κάποιος master τους αναθέτει εργασία και μετά μαζεύει τα αποτελέσματα, ξέρουμε ότι τα αποτελέσματα πρέπει να σταλούν στον κόμβο που τρέχει η master process.

Επίσης στην περίπτωση του domain decomposition όπου το πρόβλημα κόβεται σε κομμάτια και οι επεξεργαστές ανταλλάσσουν μεταξύ τους τα boundaries, μπορούμε και πάλι να

ξέρουμε πού πρέπει να σταλούν τα δεδομένα μια και μόνο γειτονικοί κόμβοι πρέπει να επικοινωνήσουν.

Ταυτόχρονα συμβαίνει και το αντίθετο, δηλαδή μπορεί ο κάθε επεξεργαστής να ξέρει από ποιό κόμβο θα ζητήσει τα δεδομένα. Έτσι στην πρώτη περίπτωση του μοντέλου επεξεργασίας αφέντη σκλάβων (master slave) όλοι οι workers πρέπει να απευθύνονται στον κόμβο που τρέχει ο master για να ζητάνε tuples που περιγράφουν εργασία.

Στα συμβατικά μοντέλα προγραμματισμού, ο προγραμματιστής αναλαμβάνει να επιλύσει αυτά τα προβλήματα με τις διάφορες φάσεις computation και communication που υπάρχουν στο πρόγραμμα. Αντίθετα στην περίπτωση της Linda δεν υπάρχει τρόπος για να δηλώσει κάποιος αυτού του είδους τις αλληλοεξαρτήσεις και μάλιστα μια από τις σημαντικότερες καινοτομίες που εισάγει είναι η απαλλαγή του προγραμματιστή από αυτές.

Έτσι πρέπει να αναζητηθούν τρόποι, είτε μέσω του compiler είτε μέσα από το runtime σύστημα να βελτιστοποιηθεί η απόδοση του συστήματος αναγνωρίζοντας αυτές τις σχέσεις μεταξύ των μερών (processes) του παράλληλου προγράμματος.

Η λύση που υιοθετήθηκε είναι η χρήση της εμπειρίας του συστήματος από το άμεσο παρελθόν ώστε να προβλέψει και την συμπεριφορά του προγράμματος στο μέλλον. Έτσι καταλήξαμε στα παρακάτω:

1. Για την περίπτωση που παράγουμε ένα tuple, το στέλνουμε στον κόμβο από τον οποίο μας είχε ζητηθεί τις προηγούμενες n φορές (όπου το n είναι σταθερά που ορίζεται εσωτερικά από εμάς)

2. Για την περίπτωση που στέλνουμε template που δεν μπορεί να εξυπηρετηθεί από τον τοπικό tuple manager τότε την απευθύνουμε στον κόμβο που μας εξυπηρέτησε τις προηγούμενες n φορές (και πάλι το n είναι εσωτερική σταθερά του συστήματος).

Στην περίπτωση πού δεν υπάρχει συμπληρωμένος αριθμός του n, τότε ακολουθούμε την αρχική μέθοδο, δηλαδή out στον τοπικό tuple manager, και broadcast των template που δεν μπορουν να εξυπηρετηθούν τοπικά.

Στην περίπτωση που ο αλγόριθμος μας οδηγήσει σε λανθασμένες αποφάσεις, η αρχιτεκτονική δεν παρουσιάζει πρόβλημα, μια και σε αυτήν την περίπτωση ακολουθείται η μέθοδος του broadcast.

3.2.2 Κεντρικοποιημένο Tuple Space

Μια άλλη εκδοχή που υλοποιήσαμε είναι το κεντρικοποιημένο tuple space. Δηλαδή να μην γίνεται πλέον η αποθήκευση και αναζήτηση των tuples από τους tuple managers όλων των κόμβων που συμμετέχουν στο πρόγραμμα, αλλά μόνο από έναν. Ο tuple manager αυτός ορίζεται να είναι αυτός στον οποίο ξεκίνησε το πρόγραμμα. Κατά το ξεκίνημα του προγράμματος ο προγραμματιστής δηλώνει αν θέλει να χρησιμοποιήσει την κατανεμημένη ή την κεντρικοποιημένη έκδοση.

Αν και η λέξη ‘‘κεντρικοποιημένη’’ ίσως ακούγεται παράξενη για ένα κατανεμημένο σύστημα, είναι απόλυτα λογική η χρήση της σε πολλές περιπτώσεις. Το πιο χαρακτηριστικό παράδειγμα είναι αυτό των replicated workers. Όταν δηλαδή υπάρχουν πολλοί workers που διαβάζουν τις περιγραφές εργασίας που παράγει κάποιος master, κάνουν τους υπολογισμούς και επιστρέφουν τα αποτελέσματα πίσω στον master.

Στην περίπτωση αυτή είναι προφανές ότι η κεντρικοποιημένη μέθοδος θα δώσει καλύτερη χρονοβελτίωση από την κατανεμημένη, και είναι καλό να προτιμηθεί

3.3 Αλγόριθμος ισοκατανομής φόρτου

Ένα από τα προβλήματα που αντιμετωπίζει ένα τέτοιο σύστημα, ακόμα και σε ομοιογενή δίκτυα, είναι η εξισορρόπηση του φόρτου μεταξύ των κόμβων. Αυτό συμβαίνει για τους παρακάτω λόγους:

- Το περιβάλλον δεν είναι απομονωμένο. Εξωτερικοί παράγοντες όπως χρήστες, άλλα προγράμματα που τρέχουν στον κόμβο, αλλάζουν δυναμικά τις επιδόσεις του κόμβου σημαντικά.
- Το δίκτυο δεν είναι συνήθως ομοιογενές. Κάθε κόμβος έχει διαφορετικές δυνατότητες ως προς την υπολογιστική ισχύ, την δυνατότητα εισόδου εξόδου από τον δίσκο, μέγεθος κύριας μνήμης, ταχύτητα επικοινωνίας με τους άλλους κόμβους.
- Ανά πάσα στιγμή υπάρχει πιθανότητα να μήν είναι λειτουργικοί όλοι οι κόμβοι.

Τα παραπάνω προβλήματα πρέπει να αντιμετωπιστούν ώστε το τελικό αποτέλεσμα να είναι όσο το δυνατόν πιο αποδοτικό, όχι μόνον σε κατανάλωση resources αλλά και σε χρόνο

τρεξίματος. Η αντιμετώπιση αυτών των προβλημάτων ήταν η εφαρμογή ενός δυναμικού αλγόριθμου κατανομής φορτίου στο δίκτυο. Κάθε φορά που πρέπει να γίνει ένα νέο eval συμβουλεύεται το σύστημα τα αποτελέσματα του αλγόριθμου και η νέα διεργασία ξεκινά στον κόμβο που υποδεικνύεται. Ο αλγόριθμος βασίζεται στην ανταλλαγή χαιρετιστήριων μηνυμάτων, και σε βάρη (ένδειξη ικανοτήτων) που γίνεται assigned κάθε κόμβος. Όπως έχουμε περιγράψει σε κάθε κόμβο τρέχει σαν daemon ο tuple manager. Οι tuple managers ανταλλάσσουν πληροφορία για το φορτίο του κάθε κόμβου χρησιμοποιώντας το reliable broadcast του πρωτόκολλου επικοινωνίας (χαιρετιστήρια μηνύματα). Κάθε tuple manager κρατάει μια ταξινομημένη λίστα από το πηλίκο του φορτίου του κόμβου δια το βάρος του κόμβου. Όσο πιο ισχυρός είναι ένας κόμβος, τόσο μεγαλύτερη είναι η πιθανότητα να ξεκινήσει σε αυτόν μια διεργασία με την χρήση eval. Στην περίπτωση που κάποιος κόμβος δεν απαντήσει στο secure broadcast τότε αυτός διαγράφεται από τους πιθανούς κόμβους για κάποιο eval. Μόλις ο κόμβος ”ξαναεμφανιστεί”, τότε ξαναμπαίνει αυτομάτως στην λίστα. Η διαδικασία πληροφόρησης των κόμβων γίνεται ενεργητικά. Δηλαδή κάθε κόμβος δηλώνει μόνος του το φορτίο του στους υπόλοιπους.

Στην παραπάνω μέθοδο υποθέσαμε ότι οι αλλαγές στο φορτίο των κόμβων δεν συμβαίνουν ακαριαία, αλλά με ρυθμό μικρότερο από αυτόν της πληροφόρησης. Αυτό εν μέρει δεν είναι αλήθεια μια και ένα πρόγραμμα χρειάζεται κλάσματα του δευτερολέπτου για να καταναλώσει όλη την μνήμη μιας μηχανής.

Πολλές επεκτάσεις μπορούν να γίνουν εύκολα σε αυτό το σχήμα. Οι πληροφορίες που αφορούν τις δυνατότητες και το configuration ενός workstation (μνήμη, δίσκος κλπ) είναι σχετικά εύκολο να αποκτηθούν. Επίσης είναι εύκολο το να προσθέτει ο χρήστης μια ποσοτική εκτίμηση του αν το πρόγραμμα που θα κάνει eval είναι CPU, IO ή Communication bounded.

Αυτές οι πληροφορίες σε συνδυασμό μπορούν να μας δώσουν μια καλή εκτίμηση του πού θα ήταν στατιστικά καλύτερο το να ξεκινήσει η νέα διεργασία.

Ένα πιθανό πρόβλημα που μπορεί να εμφανιστεί σε αυτήν την αντιμετώπιση είναι η υπερφόρτωση του δικτύου λόγω των χαιρετιστήριων μηνυμάτων. Πραγματικά αν υπάρχουν πολλοί κόμβοι που συμμετέχουν σε αυτήν την ανταλλαγή τότε το κόστος μπορεί να γίνει πολύ μεγάλο. Η λύση είναι είτε να αραιώσουν τα χρονικά διαστήματα που γίνεται αυτή η πληροφόρηση, είτε να γίνεται η πληροφόρηση μόνο σε περίπτωση σημαντικής αλλαγής

της κατάστασης ενός κόμβου, είτε τέλος να γίνεται μόνο στην περίπτωση που πρέπει να εκτελεστεί μια eval. Από τις τρεις παραπάνω λύσεις η προτιμότερη είναι η τελευταία. Δηλαδή να γίνεται αυτή η πληροφόρηση μόνο όταν πρόκειται να εκτελεστεί μια eval. Αυτό όμως θα είχε σαν συνέπεια την σημαντική αύξηση του κόστους της eval, αφού θα έπρεπε να ανταλλαγεί σημαντικός αριθμός μηνυμάτων πριν να παρθεί η τελική απόφαση.

3.4 Αύξηση της ανοχής του συστήματος σε λάθη

Ένα δίκτυο σταθμών εργασίας έχει την ιδιότητα να θεωρείται λειτουργικό ακόμα και αν ένας κόμβος του δεν δουλεύει. Αυτό δημιουργεί απαιτήσεις στο κατανεμημένο σύστημα για την απεξάρτηση της λειτουργίας του από συγκεκριμένους κόμβους πού δεν συμμετέχουν σε ένα πρόγραμμα, άλλα συμμετείχαν στο παρελθόν, και πιθανώς να συμμετάσχουν στο μέλλον αν λειτουργήσουν.

Η αντιμετώπιση του προβλήματος από το NPOSYBL βασίζεται στην άμεση απεξάρτηση του προγράμματος από κόμβους που έπαψαν να συμμετέχουν στο πρόγραμμα αφού δεν υπάρχει διεργασία POSYBL που τρέχει σε αυτούς.

Όπως έχει περιγραφεί παραπάνω, τα tuples που παράγει κάθε διεργασία αποθηκεύονται στον τοπικό της tuple manager. Αυτό έχει σαν αποτέλεσμα το POSYBL πρόγραμμα να εξαρτάται από αυτόν τον κόμβο, ακόμα και στην περίπτωση που δεν υπάρχει πλέον διεργασία POSYBL σε αυτόν αλλά περιέχει tuples χρήσιμα για την συνέχιση της εκτέλεσης του προγράμματος.

Με την νέα αντιμετώπιση, μόλις ο κόμβος αυτός αδειάσει από διεργασίες ενός POSYBL προγράμματος, μεταφέρει όλα τα tuples στον tuple-manager που ξεκίνησε η πρώτη διεργασία toy προγράμματος. Έτσι ακόμα και αν αυτός ο κόμβος πάψει να λειτουργεί, το πρόγραμμα θα μείνει ανεπηρέαστο.

Αυτή η λύση είναι εφικτή με την συγκεκριμένη αρχιτεκτονική μια και το ψάξιμο για tuples είναι βασισμένο σε broadcasts κι έτσι είναι σχετικά αδιάφορο το πού βρίσκονται.

Το επιπλέον κόστος που παράγεται με αυτήν την υλοποίηση δεν είναι σοβαρό, αφού το μόνο που συμβαίνει είναι η μεταφορά των tuples σε έναν άλλο tuple manager.

Η λύση που χρησιμοποιήσαμε δεν λύνει το πρόβλημα των failures. Απλά προσπαθεί να μειώσει την πιθανότητα της διακοπής του προγράμματος λόγω δυσλειτουργίας μεμονωμένων

κόμβων του δικτύου που δεν συμμετέχουν στο πρόγραμμα. Αντίθετα, αν ένας κόμβος που περιέχει διεργασία του προγράμματος σταματήσει να δουλεύει, τότε το συνολικό πρόγραμμα διακόπτεται.

Τα persistent tuple spaces είναι μια άλλη ελκυστική λύση πού όχι μόνον επιτρέπει την διατήρηση της κατάστασης του tuple-space και μετά την αστοχία ενός κόμβου του δικτύου, αλλά ταυτόχρονα επιτρέπει και πολλές επεκτάσεις. Προτιμήθηκε όμως η προηγούμενη λύση μια και είναι απλούστερη στην υλοποίηση.

3.5 Hashing

Όπως περιγράψαμε και παραπάνω η Linda χρησιμοποιεί associative search για να εντοπίσει κάποιο tuple με δοσμένη μια template. Η ιδιότητα αυτή του μοντέλου κάνει πολύ σημαντική την προσεκτική επιλογή και υλοποίηση του αλγόριθμου αποθήκευσης και αναζήτησης των tuples. Η μέθοδος που χρησιμοποιείται σε όλες τις υλοποιήσεις που έχουμε συναντήσει είναι hashing.

Όταν μια template φτάσει στο σύστημα, ο tuple manager πρέπει να συγκρίνει τα actual πεδία της με τα αντίστοιχα πεδία των διαθέσιμων tuples ώστε να διαλέξει πιο από όλα θα δώσει σαν απάντηση στην διεργασία που εκτέλεσε το in ή το rd. Στην περίπτωση που η template δεν περιέχει actuals, δεν χρειάζεται καθόλου matching, και το πρώτο διαθέσιμο tuple που περιέχει την ίδια ακολουθία πεδίων με την template (ταυτότητα) επιστρέφεται. Αν η template περιέχει actual πεδία τότε αυτά πρέπει να συγκριθούν με τα αντίστοιχα πεδία των tuples του ίδιου τύπου μέχρι να βρεθεί το κατάλληλο tuple ή να εξαντληθούν.

Η εξέταση της ταυτότητας του κάθε tuple είναι απλή και γρήγορη. Παρόλα αυτά μπορεί να αποβεί χρονοβόρα για δυο λόγους. Πρώτα, γιατί στην χειρότερη περίπτωση πρέπει να εξετάσουμε την ταυτότητα κάθε tuple μέσα στο tuple space, και δεύτερο γιατί η διάσχιση περίπλοκων δομών με χρήση virtual memory παράγει πολλά page faults πού καθυστερούν σημαντικά την εκτέλεση του προγράμματος.

Για να επιταχυνθεί αυτή η διαδικασία μπορούμε να χρησιμοποιήσουμε την τμηματοποίηση του tuple space με βάση την ταυτότητα των tuples.

Αφού χωριστούν τα tuples σε τμήματα, το ψάξιμο μέσα σε ένα από αυτά μπορεί και πάλι να είναι εξουθενωτικό και άρα πολύ αργό μια και τα πεδία που πρέπει να συγκριθούν ίσως

είναι μεγάλα arrays ή πολύπλοκες structures. Η λύση σε αυτό το πρόβλημα είναι η χρήση δεικτών. Οι δείκτες μπορεί να μας οδηγήσουν απευθείας στο ζητούμενο tuple ή σε ένα μικρό σύνολο πιθανών tuples.

Για ένα τμήμα του tuple space που περιέχει tuples με ταυτότητα (int, int, int) οι πιθανοί συνδυασμοί formal-actual πεδίων είναι οκτώ. Αν παραστήσουμε τα actual πεδία μιας template με 1 και τα formal με 0, η παρακάτω ακολουθία από bit-vectors μπορεί να παραχθεί: 000, 001, 010, 011, 100, 101, 110 και 111. Αποκαλούμε αυτά τα διανύσματα τύπο της template. Για παράδειγμα, η template

(?i, 10, ?j)

έχει τύπο 010, ενώ η template

(?i, 10, 30)

έχει τύπο 011. Γενικά ένα tuple με n πεδία έχει 2^n διαφορετικούς τύπους template.

Για την template

(?i, 10, 30)

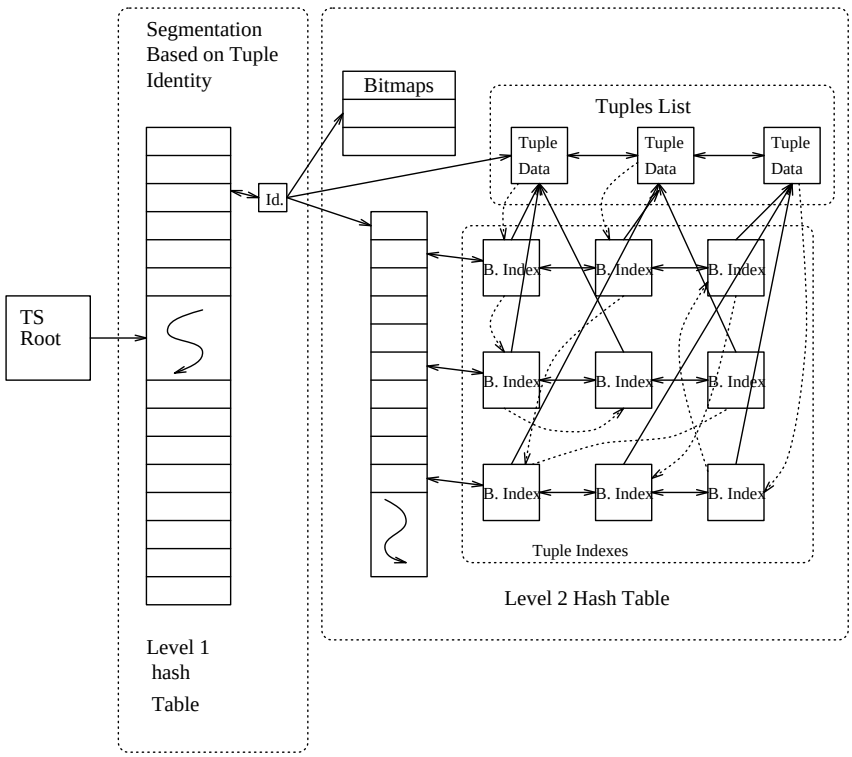
το δεύτερο και το τρίτο πεδίο των tuples πρέπει να συγκριθεί με το 10 και το 30 αντίστοιχα μέχρι να βρεθεί το κατάλληλο tuple. Γενικά, τα πεδία των tuples πρέπει να συγκριθούν με τα αντίστοιχα actual πεδία μιας template. Αν η template δεν περιέχει actuals δεν υπάρχει και ανάγκη σύγκρισης.

Για το αποδοτικό ψάξιμο θα μπορούσε να χρησιμοποιηθεί ένας δείκτης. Για παράδειγμα για την template:

(?i, k, ?j)

ένας δείκτης με βάση το δεύτερο πεδίο των tuples μπορεί να βελτιώσει δραστικά το ψάξιμο. Παρόμοια το δεύτερο και τρίτο πεδίο των tuples μπορεί να χρησιμοποιηθεί.

(?i, k, j)



Σχήμα 3.4: Η εσωτερική δομή του tuple space σε κάθε κόμβο

Το πρόβλημα που αντιμετωπίζουν οι δείκτες είναι ότι πρέπει να είναι γνωστοί εκ των προτέρων οι διαφορετικοί τύποι templates που πρόκειται να χρησιμοποιηθούν κατά την διάρκεια του προγράμματος. Φυσικά υπάρχει η δυνατότητα της παραγωγής όλων των δυνατών δεικτών αλλά αυτό είναι απαράδεκτο ως προς την απόδοση και τις απαιτήσεις σε μνήμη, για tuples με παραπάνω από δύο ή τρία πεδία.

Ο compiler μπορεί να κάνει αυτήν την προεργασία αναγνωρίζοντας τα tuples και τις αντίστοιχες templates μέσα στο πρόγραμμα. Ο compiler, μπορεί να παράγει πληροφορία που θα χρησιμοποιηθεί μετέπειτα από το runtime σύστημα για να παραχθούν οι κατάλληλοι δείκτες. Το ίδιο όμως μπορεί να γίνει και κατά την διάρκεια της εκτέλεσης του προγράμματος, δημιουργώντας τον δείκτη δυναμικά όπως έρχονται οι νέες template. Αυτό υπονοεί ότι μόλις έρθει μια template νέου τύπου πρέπει να παραχθούν δείκτες για τα tuples που είναι είδη αποθηκευμένα. Επίσης όταν φτάνει ένα νέο tuple θα πρέπει να παραχθούν δείκτες γι αυτό με βάση όλες τις template που έχουν εμφανιστεί μέχρις εκείνο το σημείο της εκτέλεσης του προγράμματος.

Το POSYBL χρησιμοποιεί διεπίπεδο hashing και runtime δημιουργία των δεικτών. Η ιδέα είναι βασισμένη στο [16] Η δομή του tuple space φαίνεται στην εικόνα 3.5. Το tuple space

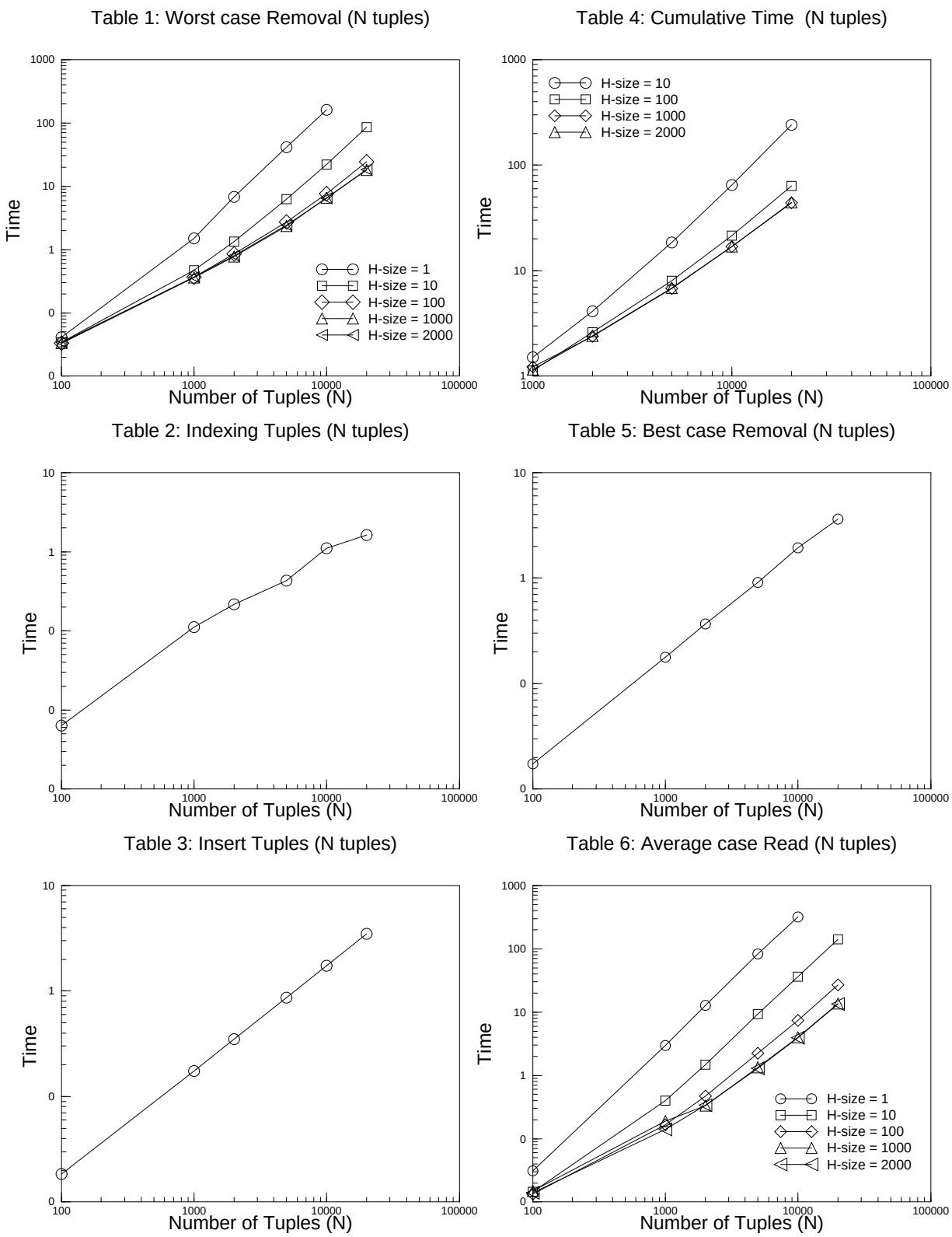
αποτελείται από το δύο επίπεδα hash tables (level1 hash table και level2 hash table). Το hashing στο πρώτο επίπεδο γίνεται με βάση την ταυτότητα του tuple. Με αυτό επιτυγχάνεται ο διαχωρισμός μεταξύ τελείως ανόμοιων tuples ώστε με μια πολύ γρήγορη πράξη στην αρχή να εντοπίζουμε το σύνολο των tuples που έχουν ίδια ταυτότητα. Το δεύτερο επίπεδο ορίζει και ένα τμήμα του tuple space. Σε αυτήν την δομή γίνεται και το πραγματικό indexing των tuples. Τα δεδομένα των tuples φυλάσσονται σε μια διπλά διασυνδεδεμένη λίστα, την tuples list. Αν δεν εμφανιστεί template με actual πεδία, καμιά άλλη πράξη εκτός από insert delete στην tuples list δεν γίνεται. Έτσι αποφεύγεται οποιοδήποτε overhead στις περιπτώσεις που δεν χρειάζεται. Όταν έρθει κάποια template με actual πεδία, όλα τα tuples που βρίσκονται στην tuples list γίνονται index με βάση αυτήν την template. Η template καταχωρείται στο bitmaps table, ώστε την επόμενη φορά που θα εμφανιστεί template αυτού του τύπου να μην ξαναγίνει το indexing. Το bitmaps table επίσης, χρησιμοποιείται για το indexing των νέων tuples που έρχονται με βάση τα είδη υπάρχοντα templates. Όλα τα διαφορετικά indexes με βάση τους τύπους των templates αποθηκεύονται στο ίδιο hash table. Οι μετρήσεις έδειξαν ότι αυτό δεν επηρεάζει την απόδοση για μεγέθη του hash table που ξεπερνούν τα 100 στοιχεία.

Όπως φαίνεται και στο σχήμα, οι δείκτες του κάθε tuple είναι ενωμένοι μεταξύ τους σε μια linked list, που χρησιμοποιείται για την αφαίρεση και των δεικτών στην περίπτωση αφαίρεσης κάποιου tuple.

3.5.1 Μετρήσεις απόδοσης

Για την μέτρηση της απόδοσης του αλγόριθμου hashing αφαιρέσαμε το κόστος επικοινωνίας μεταξύ διεργασίας και tuple manager. Έτσι στα παρακάτω διαγράμματα φαίνεται το καθαρό κόστος του hashing για διάφορες πράξεις και για διάφορα μεγέθη του hash table. Βλέπουμε ότι μετά από hash table μεγέθους 100 μικρή διαφορά παρατηρείται στην απόδοση. Επίσης ο χρόνος εισαγωγής, indexing και διαγραφής tuple είναι σταθερός, ανεξάρτητος από το μέγεθος του hash table και του αριθμού των tuples.

Συγκριτικές μετρήσεις έγιναν σε σχέση με το εμπορικό πακέτο της SCA C-Linda, που έδειξαν ότι ο δικός μας αλγόριθμος είναι πιο γρήγορος. Παρόλα αυτά δεν παραθέτουμε εδώ λεπτομέρειες αφού δεν γνωρίζουμε τις λεπτομέρειες υλοποίησης της C-Linda και υπάρχει κίνδυνος να μην λαμβάνουμε υπόψην μας κάποιο επιπλέον κόστος που δεν είναι προφανές.



Σχήμα 3.5: Η απόδοση του αλγορίθμου αποθήκευσης και ανεύρεσης των tuples

Κεφάλαιο 4

Συμπεράσματα, επεκτάσεις

4.1 Συμπεράσματα

Τα περισσότερα τμήματα που παρουσιάστηκαν παραπάνω είναι λειτουργικά αυτόνομα κομμάτια που τα ενώσαμε μεταξύ τους για να δημιουργήσουμε το NPOSYBL. Πολλά από αυτά θα μπορούσαν να χρησιμοποιηθούν και σε άλλες εφαρμογές. Για παράδειγμα το πρωτόκολλο δικτύου και το πρωτόκολλο IPC μπορούν να χρησιμοποιηθούν για την υλοποίηση transaction processing συστημάτων. Η ιδέα και η υλοποίηση του hashing είναι κατάλληλη για προσωρινή αποθήκευση δεδομένων με χρήση πολλαπλών κλειδιών για την αναζήτησή τους. Άλλα πάλι κομμάτια είναι ειδικά σχεδιασμένα και υλοποιημένα για την εφαρμογή του NPOSYBL, και αυτούσια δύσκολα θα έβρισκαν χρήση σε άλλες εφαρμογές.

Κερδίσαμε πολλά μαθήματα στην διάρκεια της προσπάθειας αυτής. Το πιο βασικό κατά την γνώμη μας είναι η συνειδητοποίηση του κόστους που έχουν οι διάφορες υπηρεσίες που προσφέρει το λειτουργικό σύστημα ενός υπολογιστή στις διεργασίες χρήστη.

Η βασική αρχή πάντως που πιστεύουμε ότι ισχύει είναι να δίνουμε στο λειτουργικό σύστημα τη δυνατότητα να κάνει την δουλειά του, και να μήν προσπαθούμε να το υποκαταστήσουμε όταν δεν υπάρχει σοβαρός λόγος.

Ένα άλλο σημείο που πρέπει να επισημάνουμε είναι η διαδικασία επιλογής των τελικών αποφάσεων της υλοποίησης. Πολλές εναλλακτικές λύσεις όχι μόνο συζητήθηκαν αλλά και υλοποιήθηκαν. Για παράδειγμα το πρωτόκολλο δικτύου υλοποιήθηκε και με την χρήση του raw ethernet device, και με την χρήση raw sockets. Τα πειράματα έδειξαν ότι η έλλειψη του buffering και η αργή ανταπόκριση που έχουν οι διεργασίες χρήστη στα γεγονότα δεν έκανε

αποδοτικές αυτές τις υλοποιήσεις.

Το τελικό σύστημα είναι ένα αρκετά σταθερό και αποδοτικό περιβάλλον Linda.

Ο χρήστης πρέπει να κατανοήσει μερικά βασικά πράγματα για το σύστημά του και η κυριότερη διαφορά του NPOSYBL από το κλασικό μοντέλο Linda είναι ότι χρησιμοποιεί μερικά επιπλέον calls για την σύνδεση και αποσύνδεσή του με τον tuple manager και την επιλογή κεντρικοποιημένου ή κατανεμημένου αλγόριθμου αποθήκευσης των tuples. Οι tuple managers είναι διεργασίες daemon και πρέπει να ξεκινάει στο boot time των μηχανών, και επίσης να υπάρχει και κάποιος administrator που να κάνει την εγκατάσταση του software και την επιλογή των μηχανών που μπορούν να συμμετέχουν σε ένα NPOSYBL πρόγραμμα.

4.2 Επεκτάσεις

Το μοντέλο Linda δεν είναι κλειστός δρόμος. Το γεγονός ότι υλοποιήσαμε αυτό το μοντέλο με τον τρόπο που περιγράψαμε στα παραπάνω κεφάλαια δεν σημαίνει ότι έχουμε φτάσει και στο τέρμα αυτής της προσπάθειας. Από τώρα και μπρος μας δίνεται η ευκαιρία να κάνουμε το απλό και αποδοτικό αυτό μοντέλο πιο ισχυρό.

Το μοντέλο Linda, έχει κάνει νομίζουμε μια αρχή ώστε να πάψουμε να βλέπουμε την μνήμη μας σαν ένα απλό πίνακα διευθύνσεων που δεν ξέρουμε που βρίσκεται και πώς τελικά προσπελαύνεται. Οι δυνατότητες που δίνονται στον προγραμματιστή είναι πολύ πιο εύχρηστες από το message passing μοντέλο και ταυτόχρονα δείχνουν καθαρά το πότε υπάρχει επικοινωνία ή συγχρονισμός. Σαν εκφραστικό εργαλείο πιστεύουμε ότι έχει πετύχει αφού όχι μόνον κάνει πολύ εύκολη την δουλειά του προγραμματισμού αλλά ταυτόχρονα δεν κρύβει και τις λεπτομέρειες.

Στην προσπάθεια της υλοποίησης του μοντέλου νομίζουμε ότι γίναμε και απαιτητικοί του χρήστες. Πολλές φορές χρειαστήκαμε πράγματα που δεν τα παρείχε απευθείας ή ακόμα όταν τα υλοποιούσαμε ήταν μη αποδοτικά.

Στις παρακάτω παραγράφους θα περιγράψουμε με συντομία διάφορες ιδέες που θα μπορούσαν να επεκτείνουν το μοντέλο και να το κάνουν πιο ισχυρό.

Πολλαπλά tuple spaces

Το tuple space είναι μια ‘‘επίπεδη’’ δομή αποθήκευσης δεδομένων. Όλα τα tuples συγκεντρώνονται μαζί και είναι ευθύνη του αλγόριθμου αποθήκευσης να τα διαχωρίσει. Μια επέκταση που προτείνεται είναι η δυνατότητα χρήσης και δημιουργίας πολλών tuple spaces. Αυτό δίνει την δυνατότητα για πιο αποδοτικά προγράμματα, αφού είναι πιο γρήγορο το να ψάχνουμε σε μικρό tuple space.

Μια άλλη δυνατότητα που μπορεί να μας δώσει αυτή η επέκταση είναι ο λογικός διαχωρισμός των διεργασιών που εκτελούν όμοιες εργασίες. Έτσι θα μπορούμε να αποφύγουμε την άσκοπη εκπομπή templates σε κόμβους που δεν έχουν σχέση με κάποιο tuple space.

Break - points και persistent tuple spaces

Σε ένα κατανεμημένο σύστημα είναι πολύ ελκυστική η ιδέα της αποθήκευσης της κατάστασης του συστήματος σε κάποιες φάσεις της εκτέλεσης του προγράμματος.

Μια τέτοια υλοποίηση θα μας έδινε την δυνατότητα να σταματάμε το πρόγραμμα, και να το ξεκινάμε αργότερα, να δημιουργούμε fault tolerant εφαρμογές, οι οποίες θα μπορούσαν να ξαναξεκινήσουν μετά από μια βλάβη (failure) του συστήματος.

Άλλες εφαρμογές αυτής της ιδέας είναι η δημιουργία μεταναστευτικών προγραμμάτων. Για παράδειγμα αν κάποια διεργασία μπορεί να σώσει την κατάσταση της μνήμης της σε κάποια στιγμή μπορεί να ξαναξεκινήσει σε έναν άλλο κόμβο λιγότερο φορτωμένο ή πιο ισχυρό μόλις αυτός γίνει διαθέσιμος. Άλλο παράδειγμα είναι η δημιουργία και αποθήκευση δεδομένων που αλλάζουν αργά με τον χρόνο και είναι χρήσιμα για την εκτέλεση πολλών προγραμμάτων

Αν σκεφτούμε το Linda σαν ένα σύστημα που επιτρέπει την δημιουργία system utilities, όπως printer queues, ηλεκτρονικό ταχυδρομείο κλπ. πολλά τμήματα του configuration μπορεί να βρίσκονται συνεχώς μέσα στο persistent tuple space.

Compiler optimizations

Ο πιο πρόσφορος τρόπος για την επέκταση και βελτιστοποίηση του μοντέλου είναι μέσω του compiler και των βελτιστοποιήσεων που μπορεί να γίνουν κατά την διάρκεια της μετάφρασης.

Ο μεταφραστής μπορεί να ανακαλύψει τα matching tuples και templates, ώστε να βελτιστοποιηθεί η αποθήκευσή τους. Μπορεί επίσης να βρει τα communication patterns ώστε με τα κατάλληλα directives να μπορεί να βελτιστοποιήσει την ανάθεση των εργασιών στους κόμβους.

Footprinting in Linda

Μια άλλη πολύ σημαντική βελτιστοποίηση που μπορεί να επιτευχθεί μέσω του compiler είναι η διάσπαση των in και rd πράξεων σε δυο κομμάτια: 1) αυτό που κάνει request μια template και 2) αυτό που λαμβάνει την απάντηση. Μεταξύ αυτών των δύο μερών μπορεί να παρεμβληθεί ανεξάρτητος κώδικας και έτσι να αποφύγουμε την περιττή σπατάλη χρόνου που συνεπάγεται η αναμονή για την εξυπηρέτηση μιας template.

Παράρτημα Α

ΙΣΤΟΣ Linda: υλοποίηση σε μια παράλληλη μηχανή

Η απόκτηση μιας παράλληλης μηχανής Parsytec με 4 κόμβους T805 της INMOS, ήταν μια ευκαιρία και μια πρόκληση για την υλοποίηση του Linda σε αυτήν. Τα βασικά πλεονεκτήματα μιας παράλληλης μηχανής είναι η ομοιογένεια, η μεγάλη ταχύτητα επικοινωνίας και το reliability της επικοινωνίας. Έτσι πολλά από τα προβλήματα που αντιμετωπίσαμε στην υλοποίηση τοπικού δικτύου τώρα πλέον δεν υφίστανται. Αντίθετα, πιεστικά προβλήματα μνήμης υπάρχουν που πρέπει να ληφθούν υπόψην κατά την υλοποίηση αλλά και κατά την χρήση του συστήματος.

Αυτήν τη φορά η έμφαση δόθηκε στην σταθερότητα του συστήματος. Ο σκοπός μας τώρα είναι να φτιάξουμε ένα πλήρες περιβάλλον Linda όπου ο τυχαίος χρήστης με κάποιες γνώσεις προγραμματισμού και κάποιο manual να μπορεί να παράγει αποδοτικά παράλληλα προγράμματα. Έτσι υλοποιήθηκαν οι πράξεις inp, rdp και eval ώστε το περιβάλλον να είναι τελείως συμβατό με τον ορισμό του μοντέλου.

Ένα από τα στοιχεία που χαρακτηρίζει το σύστημα είναι οι ελάχιστες απαιτήσεις από τον χρήστη. Δηλαδή, ο χρήστης δεν χρειάζεται να γνωρίζει τίποτε για την αρχιτεκτονική ή την συγκεκριμένη υλοποίηση, εκτός από το μέγεθος της μνήμης που διαθέτει. Ταυτόχρονα ο χρήστης, αν θέλει μπορεί να χρησιμοποιήσει όλα τα εργαλεία που παρέχει το συγκεκριμένο λειτουργικό της παράλληλης μηχανής, θυσιάζοντας όμως το portability του προγράμματός

[0]Η υλοποίηση της ΙΣΤΟΣ Linda χρηματοδοτήθηκε εν μέρει από την KNOSSOS Technologies, S.A. η οποία παρείχε και σχετική τεχνική υποστήριξη.

του.

A.1 Distributed Hashing

Η κατανομή των tuples στους διάφορους κόμβους, είναι ένα από τα πιο περίπλοκα προβλήματα που έχει να αντιμετωπίσει ένα σύστημα Linda. Για ένα σύστημα με πιο ισσοροπημένη σχέση computation/communication το distributed hashing είναι και η πιο πρόσφορη λύση. Δηλαδή τα tuples κατανέμονται στους κόμβους με βάση την ταυτότητά τους. Μετά, μέσα στον κόμβο και σε δεύτερο επίπεδο μπορεί να γίνει το κλασικό hashing που εφαρμόζουμε και στο POSYBL. Στην περίπτωση μάλιστα του ΙΣΤΟΣ, όπου έχουμε σύνολο επεξεργαστών με μικρή σχετικά τοπική μνήμη, η μέθοδος αυτή είναι αναγκαία μια και επιτρέπει την κατανομή των δεδομένων (tuples) μεταξύ των επεξεργαστών. Από την άλλη μεριά, επειδή η κατανομή γίνεται με βάση το identity των tuples, όλα τα tuples με το ίδιο identity θα πάνε στον ίδιο επεξεργαστή. Έτσι μπορεί να εμφανιστούν bottlenecks και επίσης να καταναλωθεί όλη η μνήμη ενός επεξεργαστή.

Η λύση που δώσαμε σε αυτό το πρόβλημα είναι η μεταφορά στον χρήστη της ευθύνης για την σωστή κατανομή των tuples. Αυτό γίνεται με την χρήση ενός νέου τελεστή, του @, ο οποίος όταν προηγείται μιας μεταβλητής τότε αυτή χρησιμοποιείται για τον καθορισμό του κόμβου στον οποίο θα αποθηκευτεί το tuple. Αντίθετα, όταν αυτός ο τελεστής χρησιμοποιηθεί σε μια template, τότε καθορίζει από ποιό κόμβο θα ζητηθεί το tuple. Η χρήση αυτού του τελεστή είναι προαιρετική. Άλλες υλοποιήσεις χρησιμοποιούν τον compiler ώστε με τον εντοπισμό των actual πεδίων στα ζευγάρια tuples - templates, να χρησιμοποιεί κάποιο από αυτά για την χρήση που προαναφέρθηκε. Άλλες πάλι υλοποιήσεις υποχρεώνουν την χρήση ενός actual σαν πρώτο πεδίο σε όλα τα tuples και τα templates και χρησιμοποιούν αυτό για την κατανομή και το indexing των tuples.

Η συνεργασία όμως από τον χρήστη κρύβει πάντα τον κίνδυνο λανθασμένης απόφασής του. Έτσι, η χρήση αυτών των τελεστών δεν ενθαρρύνεται παρά μόνον στην περίπτωση εξοικειωμένου χρήστη. Αυτός ο τελεστής πάντως μπορεί να μετατρέψει το Linda πρόγραμμα σε ένα κοινό message passing πρόγραμμα, με το πρόσθετο κόστος της πρόσβασης στα δεδομένα μέσω του tuple space.

A.2 Eval στην παράλληλη μηχανή

Στην παράλληλη μηχανή όπου δεν υπάρχουν προβλήματα ανομοιογένειας μπορέσαμε να πειραματιστούμε και να υλοποιήσουμε την eval. Η γενική παρατήρηση είναι ότι η δυνατότητα να κάνουμε remote invocation μιας διαδικασίας (function) με τα ορίσματά της μας επιτρέπει να επιτύχουμε την συμπεριφορά της eval. Έτσι στην δική μας υλοποίηση, αυτό που κάναμε είναι η μετατροπή της eval σε μια μικρή function που κάνει out τα fields της eval. Έτσι η κλήση:

```
eval(f(x),y);
```

μετατρέπεται στην ρουτίνα:

```
__threadX(x,y)
{
out(f(x),y);
}
```

Η ρουτίνα αυτή όταν κληθεί η eval ενεργοποιείται μέσω μιας thread σε κάποιον processor.

Ο περιορισμός που τίθεται είναι ότι τα ορίσματα της eval γίνονται evaluated σειριακά. Έτσι για κάθε eval υπάρχει ένα μόνο thread of execution. Ο αλγόριθμος κατανομής των threads στους επεξεργαστές γίνεται με έναν στατικό αλγόριθμο που επιβάλει το εξής:
Κάθε επεξεργαστής ξεκινάει τις νέες threads ξεκινώντας από τον επόμενό του και συνεχίζοντας κυκλικά.

Θα μπορούσαμε να κάνουμε ένα κεντρικοποιημένο σύστημα ανάθεσης εργασίας, όπου θα κρατούσαμε κεντρικά το πόσες δουλειές (threads) έχει ο κάθε επεξεργαστής και θα αναθέταμε την δουλειά στον λιγότερο φορτωμένο. Αυτό έχει όμως το πρόβλημα ότι θα αύξανε σημαντικά το κόστος του eval και άρα δεν θα μπορούσε κάποιος να το χρησιμοποιήσει για στοιχειώδεις πράξεις. Επίσης στην περίπτωση που έχουμε δουλειές με άνισες απαιτήσεις σε υπολογιστική ικανότητα θα καταλήγαμε και πάλι σε παρόμοιο αποτέλεσμα παρόλη την προσπάθεια.

Ένα άλλο σημείο στο οποίο πρέπει να δοθεί προσοχή είναι ότι 2 threads μπορεί να υπάρχουν στον ίδιο επεξεργαστή. Αυτό σημαίνει ότι έχουν και οι 2 access στις ίδιες global

variables, και άρα χρειάζεται προσοχή από τον προγραμματιστή ώστε είτε να χρησιμοποιεί locking, είτε να μήν προσπελαύνει κοινές μεταβλητές.

A.3 ΙΣΤΟΣ-Linda simulator με χρήση threads

Ένας από τους πιο απλούς τρόπους για την υλοποίηση concurrent threads of execution στο SunOS είναι η χρήση των light weight processes. Με αυτόν τον τρόπο αποφασίσαμε να υλοποιήσουμε έναν προσομοιωτή του ΙΣΤΟΣ σε Sun workstation ώστε να προσφέρουμε κάποιον φτηνό τρόπο για την εξοικείωση του χρήστη με το μοντέλο του Linda και επίσης να μπορεί να χωρίσει κάποιος την φάση ανάπτυξης από την πραγματική παράλληλη μηχανή που στην γενική περίπτωση είναι ασύμφορη η χρήση της για ανάπτυξη software.

Τελικά αποδείχτηκε ένα πολύ αξιόπιστο εργαλείο προσομοίωσης, debugging και εκπαίδευσης, που μπορεί να αξιοποιήσει όλες τις δυνατότητες που προσφέρει το συγκεκριμένο λειτουργικό σύστημα.

A.4 Ο μεταφραστής

Για την πλήρη υποστήριξη του run time συστήματος του NPOSYBL και του ΙΣΤΟΣ, χρειάζεται και ο μεταφραστής. Το σύστημα είναι σχεδιασμένο ώστε μην είναι απαιτητικό από τον μεταφραστή. Δεν χρειάζεται δηλαδή κανενός είδους compile time πληροφορίας για τα τις πράξεις Linda, εκτός από τους τύπους των πεδίων των tuples και των templates.

Ο μεταφραστής λοιπόν που φτιάχτηκε στα πλαίσια αυτού του project, παίρνει το πρόγραμμα, ανιχνεύει τις πράξεις Linda, τα μετατρέπει σε κλήσεις βιβλιοθήκης και προσθέτει πληροφορία για τους τύπους των πεδίων.

Το άλλο έργο που καλείται να επιτελέσει ο μεταφραστής είναι στην μεν περίπτωση του NPOSYBL να προστεθεί επιπλέον κώδικας για την μετατροπή των ορισμάτων σε architecture independent format, στο δε ΙΣΤΟΣ, στην κατάλληλη μεταμόρφωση της eval όπως περιγράφηκε παραπάνω.

Ο μεταφραστής είναι βασισμένος στον GNU gcc 1.37 και το πραγματικά δύσκολο έργο της μετατροπής του ώστε να υποστηρίζει το μοντέλο Linda όπως εμείς το ορίσαμε επιτελέσθηκε από τους Βαγγελιό Καβακλή και Μανόλη Λουράκη τους οποίους και θερμά ευχαριστώ.

Α.5 Συμπεράσματα

Ο γράφων για πρώτη φορά έθεσε σαν στόχο της υλοποίησης όχι μόνον την εφαρμογή καλών ιδεών (κατά την γνώμη του καλών, φυσικά) αλλά και το τελικό αποτέλεσμα να είναι εύχρηστο από οποιονδήποτε. Τελικά νομίζουμε ότι αυτός ο στόχος είναι και ο πιο χρονοβόρος να υλοποιηθεί. Μπορούμε να αναφέρουμε ότι ο χρόνος να φτιαχτεί η πρώτη έκδοση του ΙΣΤΟΣ ήταν γύρω στην μία εβδομάδα, ενώ χρειάστηκαν τρεις με τέσσερις μήνες για να γίνει εύχρηστος.

Ένα άλλο σημείο που υποχρεωθήκαμε να εξετάσουμε εκτενέστερα είναι και η αξιοπιστία. Έτσι λεπτομερές testing έγινε όχι μόνον για την ταχύτητα, αλλά και για το portability των ΙΣΤΟΣ-Linda προγραμμάτων, και την αντοχή του συστήματος. Αφήσαμε για πολλές ώρες προγράμματα να τρέχουν συνεχώς ώστε να δούμε αν έχουμε απώλειες στην απόδοση αλλά και χάσιμο μνήμης.

Νομίζουμε, ότι το σύστημα αυτή τη στιγμή είναι πολύ σταθερό. Το έχουμε χρησιμοποιήσει για να τρέξουμε πολλές ασκήσεις φοιτητών, public domain Linda προγράμματα, αλλά και πολλά δικά μας tests με επιτυχία. Το σύστημα έχει τρέξει χωρίς κανένα πρόβλημα σε μηχανή 512 επεξεργαστών για την δημιουργία raytraced σκηνών με την χρήση του αλγόριθμου rayshade.

Πιστεύουμε ότι είναι ένα ολοκληρωμένο εργαλείο που η χρησιμοποίησή του από έμπειρους προγραμματιστές δεν είναι χάσιμο χρόνου.

Θα ήμασταν εξαιρετικά χαρούμενοι να το δούμε να χρησιμοποιείται και σε μαθήματα στο Πανεπιστήμιό μας.

Παράρτημα Β

Διαχείριση μνήμης

Η δημιουργία και διατήρηση του tuple space γίνεται μέσα στο virtual address space των tuple managers. Η διαχείριση της μνήμης γίνεται μέσω των ρουτινών βιβλιοθήκης malloc και free. Το UNIX παρέχει την δυνατότητα εύκολης διαχείρισης μνήμης κρύβοντας όμως το κόστος που αυτό συνεπάγεται. Το κόστος αυτό γίνεται μάλιστα απαγορευτικό στη περίπτωση υλοποίησης daemon processes, δηλαδή διεργασιών που παραμένουν ‘‘ζωντανές’’ όσο λειτουργεί και ο σταθμός εργασίας. Μια καλή τακτική είναι να μην χρησιμοποιείται καθόλου το malloc σε daemon processes, ή αν δεν γίνεται αλλιώς να γίνονται allocate ίσου μεγέθους κομμάτια μνήμης που μπορούν να ξαναχρησιμοποιηθούν.

Αυτό ισχύει για 2 λόγους. Πρώτον διότι (στις περιπτώσεις που εμείς ξέρουμε) όταν πάρουμε ένα κομάτι μνήμης με την χρήση malloc, αυτό δεν επιστρέφεται στο σύστημα με την χρήση του free, αλλά κρατιέται στην free-list της διεργασίας (εσωτερική υλοποίηση βιβλιοθήκης) για να χρησιμοποιηθεί αργότερα. Δεύτερο, και είναι συνέπεια και του πρώτου, το allocation άνισων κομματιών μνήμης προκαλεί fragmentation και κατά συνέπεια σπατάλη.

Το πρόβλημα είναι γνωστό στην κοινότητα των προγραμματιστών (ευτυχώς) και έτσι ψάχνοντας στο δίκτυο, βρήκαμε μια υλοποίηση της βιβλιοθήκης malloc που κάνει πραγματικό free, δηλαδή επιστρέφει στο σύστημα κομάτια μνήμης που δεν χρησιμοποιούνται και είναι συνεχόμενα στο τέλος της heap.

Αυτό βοηθάει αφού όταν το σύστημα έρθει σε κατάσταση ηρεμίας, δηλαδή καμιά POSYBL process δεν τρέχει, και δεν υπάρχουν καθόλου tuples στο tuple space, η μνήμη που είχε γίνει allocate αποδίδεται πίσω στο σύστημα.

Ένα δεύτερο σημείο που βελτιώνει αρκετά την κατάσταση είναι η χρήση preallocated

χώρου για ποσότητες γνωστού μεγέθους που ζούνε για λίγο χρόνο. Για παράδειγμα για τις δομές δεδομένων που κρατούν την περιγραφή των γεγονότων (events) στα οποία πρέπει να ανταποκριθεί το σύστημα ο χώρος δεσμεύεται από την αρχή. Αν σε κάποια στιγμή της εκτέλεσης ο χώρος δεν είναι επαρκής τότε νέος δεσμεύεται, ο οποίος όμως ελευθερώνεται μόλις δεν χρειάζεται.

Γενικά πάντως η διαχείριση της μνήμης είναι ένα πρόβλημα που πρέπει να λαμβάνεται σοβαρά υπόψιν σε μεγάλες εφαρμογές. Όσο και αν αυξάνει σημαντικά η δυνατότητα σε μνήμη των σταθμών εργασίας παραμένει ακόμα πολύ εύκολη η ολική της κατανάλωση!!

Βιβλιογραφία

[1] Bill Nitzberg and Virginia Lo. Distributed shared memory: A survey of issues and algorithms. *IEEE Computer*, pages 52--60, August 1991.

[2] N. Carriero and D. Gelernter. How to Write Parallel Programs:A Guide to the Perplexed. *ACM Computing Surveys*, 21:323--357, September 1989.

[3] N. Carriero and D. Gelernter. Linda in context. *Communications of the ACM*, 32, Number 4:444--558, April 1989.

[4] G. Schoinas. Issues on the implementation of a PrOgramming SYstem for distriButed appLications, January 1991. CSD, University of Crete.

[5] David H. Gelernter. An Integrated Microcomputer Network for Experiments in Distributed Computing. Technical report, PHD thesis, State University of New York at Stony Brook, 1982.

[6] David Gelernter. Getting the Job Done. *Byte Magazine*, pages 301--308, November 1988.

[7] W. Leler. Linda meets Unix. *IEEE Computer*, 23(2), February 1990.

[8] N. Carriero and D. Gelernter. Tuple analysis and partial evaluation strategies in the Linda precompiler. In *Second Workshop on Languages and Compilers for Paralellism*, August 1989.

[9] G. Schoinas. POSYBL: a PrOgramming SYstem for distriButed appLications. In G. Wilson, editor, *Workshop on Linda-like systems and their implementations*. Edinburg Parallel Computer Centre, June 1991.

[10] J. Leichter. Shared Tuple Memories, Shared Memories, Buses and LAN's - Linda Implementations Across the Spectrum of Connectivity. Technical Report YALEU/DCS/RR-714, DCS, Yale University, July 1989.

[11] SUN Microsystems Inc. *Unix Network Programming Manual.*

[12] David Cheriton. Vmtp: Versatile message transaction protocol (protocol specification). Technical Report rfc1045, Computer Science Department, Stanford University, February 1988.

[13] David R. Cheriton. THE V DISTRIBUTED SYSTEM. *Communications of the ACM*, 31(3), March 1988.

[14] Sun Microsystems Inc. Networks and File Servers, A Performance Tuning Guide. Technical report, Sun Microsystems, Inc, December 1990.

[15] Daniel Lenoski James Laudon Kourosh Gharachorloo Anoop Gupta and John Hennesy. The Directory-Based Cache Coherence Protocol for the DASH Multiprocessor. In *17th Annual International Symposium on Computer Architecture.* ACM SIGARCH 18.2, June 1990.

[16] Edward J. Segall. Multiple search keys for linda. Technical report, CAIP Center and Electrical and Computer Engineering Departme nt, Rutgers University Piscataway, NJ 08855-1390.