ΑΝΟΧΗ ΣΦΑΛΜΑΤΩΝ ΣΕ ΕΦΑΡΜΟΓΕΣ MPI ΠΟΥ ΑΚΟΛΟΥΘΟΥΝ ΤΟ ΜΟΝΤΕΛΟ MASTER-WORKER

Η
ΜΕΤΑΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ ΕΞΕΙΔΙΚΕΥΣΗΣ

Υποβάλλεται στην

ορισθείσα από την Γενική Συνέλευση Ειδικής Σύνθεσης
του Τμήματος Πληροφορικής
Εξεταστική Επιτροπή

από τον

Φώτη Σιταρά

ως μέρος των Υποχρεώσεων

για τη λήψη

του

ΜΕΤΑΠΤΥΧΙΑΚΟΥ ΔΙΠΛΩΜΑΤΟΣ ΣΤΗΝ ΠΛΗΡΟΦΟΡΙΚΗ

ΜΕ ΕΞΕΙΔΙΚΕΥΣΗ ΣΤΟ ΛΟΓΙΣΜΙΚΟ

Οκτώβριος 2011

ΑΦΙΕΡΩΣΗ

Η συγκεκριμένη εργασία αφιερώνεται σε όλα τα πρόσωπα που μου στάθηκαν με τον καλύτερο δυνατό τρόπο κατά τη διάρκεια των σπουδών μου σε όλους τους τομείς.

ΕΥΧΑΡΙΣΤΙΕΣ

Θα ήθελα να ευχαριστήσω πρώτα από όλα, την οικεγένεια μου που με υποστήριξε σε όλη τη διάρκεια των σπουδών μου και ήταν πάντα δίπλα μου σε οτι χρειαζόμουν.

Επίσης θα ήθελα να ευχαριστήσω τον επιβλέποντα καθηγητή μου κύριο Βασίλειο Δημακόπουλο, για την ανάθεση αυτού του πολύ ενδιαφέροντος θέματος και την πλήρη στήριξη κατά τη διάρκεια εκπόνησής του, καθώς και για την πολύ καλή συνεργασία που είχαμε.

Ιδιαίτερες ευχαριστίες θα ήθελα να δώσω και στον Παναγιώτη Χατζηδούκα, για την πολύτιμη βοήθεια που μου προσέφερε σε όλη τη διάρκεια της εργασίας.

Τέλος, θα ήθελα να πω και ένα μεγάλο ευχαριστώ στα παιδιά από το γραφείο Β.32 για το οτι έκαναν κάποιες δύσκολες ώρες, με τον τρόπο τους ευκολότερες.

ΠΕΡΙΕΧΟΜΕΝΑ

ΕΥΡΕΤΗΡΙΟ ΠΙΝΑΚΩΝ

ΕΥΡΕΤΗΡΙΟ ΣΧΗΜΑΤΩΝ

ΠΕΡΙΛΗΨΗ

Σιταράς Φώτης του Ιωάννη και της Κελαηδινής. MSc, Τμήμα Πληροφορικής Ιωαννίνων, Οκτώβριος, 2011. Ανοχή σφαλμάτων σε εφαρμογές MPI που ακολουθούν το μοντέλο Master-Worker. Επιβλέποντας: Βασίλειος Δημακόπουλος.

Αυτή εργασία επικεντρώνεται στην υποστήριξη ανοχής σφαλμάτων σε κατανεμημένες εφαρμογές MPI που έχουν αναπτυχθεί σύμφωνα με το μοντέλο προγραμματισμού Master-Worker. Όταν μια εφαρμογή εκτελείται για μεγάλα χρονικά διαστήματα σε συστήματα που αποτελούνται από μεγάλο αριθμό υπολογιστικών κόμβων, είναι πολύ πιθανό ένα σφάλμα να συμβεί, είτε λόγω λογισμικού, είτε λόγω υλικού. Η ανοχή σφαλμάτων αναφέρεται στην ικανότητα ενός συστήματος να επιβιώνει και να αντιμετωπίζει τέτοια σφάλματα που μπορεί να εμφανιστούν.

Στην εργασία αναλύεται το πρότυπο MPI, για ανάπτυξη κατανεμημένων εφαρμογών. Παρουσιάζονται οι διάφορες εκδόσεις του αλλά και οι πιο διαδεδομένες υλοποιήσεις του. Παρουσιάζεται και περιγράφεται επίσης και το μοντέλο προγραμματισμού Master-Worker.

Αναλύεται το πως το πρότυπο επιλέγει να χειριστεί τα σφάλματα και το κατά πόσο οι υλοποιήσεις μπορούν να το υποστηρίξουν σε αυτόν τον τομέα. Τι υλοποιήσεις έχουν γίνει, για να προσφέρουν ανοχή σφαλμάτων στο πρότυπο και τι επιτυγχάνουν. Επίσης περιγράφτηκαν και διάφορες τεχνικές αντιμετώπισης των σφαλμάτων, μεμονομένα.

Σημαντικό μέρος της εργασίας αφορά το σχεδιασμό και την υλοποίηση ενός μηχανισμού (FTMW) ανοχής σφαλμάτων σε εφαρμογές MPI που χρησιμοποιούν το μοντέλο προγραμματισμού Master-Worker. Η υλοποίηση είναι ένα ενδιάμεσο επίπεδο που τοποθετείται ανάμεσα στην υλοποίηση MPI και στην εφαρμογή MPI. Η υλοποίηση προσφέρει τη δυνατότητα αξιόπιστης ανίχνευσης σφαλμάτων, επανεκκίνησης των διεργασίων που εμφανίζουν σφάλμα, καταγραφής των μηνυμάτων και εκτέλεσης με πολλαπλές διεργασίες Master, για την περίπτωση που το σφάλμα εμφανιστεί στην πλευρά της διεργασίας Master.

Η υλοποίηση, για το επίπεδο της ορθότητας, ελέγθηκε με απλές εφαρμογές Master-Worker. Στις εφαρμογές αυτές επιλέγαμε να ρίχνουμε έναν ή περισσότερες διεργασίες Worker, σε τυχαία χρονικά διαστήματα ώστε να διαπιστώσουμε αν τόσο η βασική διεργασία Master, όσο και οι υπόλοιπες που μπορεί να εκτελούνται παράλληλα, ανιχνεύουν το σφάλμα στο σωστό σημείο. Επίσης χρησιμοποιήσαμε και μια εφαρμογή αριθμητικής βελτιστοποίησης που χρησιμοποιεί τη μέθοδο Multistart, για παράλληλη γραμμική βελτιστοποίηση στο χώρο αναζήτησης από πολλά σημεία.

Τα πειράματα αυτά αποσκοπούσαν στο να διαπιστωθεί το πόσο επιπλέον φόρτο εκτέλεσης, εισάγει η υλοποίηση του FTMW σε σχέση με κάποια συμβατική υλοποίηση που δεν προσφέρει ανοχή σφαλμάτων και το πόσο επιπλέον φόρτο εισάγουν οι λειτουργίες αντιμετώπισης σφαλμάτων της υλοποίησης. Τα αποτελέσματα δείχνουν ότι οι επιβαρύνσεις της υλοποίησης μας στις τελικές επιδόσεις είναι ελάχιστες.

EXTENDED ABSTRACT IN ENGLISH

Sitaras Fotis, MSc, Computer Science Department, University of Ioannina, Greece. October, 2011.Fault tolerance in MPI applications that use the Master-Worker model. Thesis Supervisor: Vasileios Dimakopoulos.

This thesis focuses on supporting fault tolerance over distributed MPI applications that have been developed according to the Master-Worker programs executed for extended periods on top of systems that consists of a large number of nodes, face an increasing probability of failure, either in terms of software, or hardware. Fault tolerance is the ability of a system to survive and to handle these failures.

We presented the MPI, a programming prototype for developing distributed applications, throught messages exchange. We presented the versions of the MPI and the various implementations as well and the Master-Worker scheme for developing applications.

We analyze the way that the MPI, handles faults and in what extent, the implementations of the prototype can support it. We survey related work on supporting fault on supporting fault tolerance in MPI and what they achieve. We also present a variety of techniques that have been proposed to handle failures.

A significant part of this work was the design and development of FTMW (Fault Tolerance Master Worker), a fault tolerant system for MPI applications that use the Master-Worker scheme. The implementation is a middleware that is located between the MPI implementation and the MPI application. We offer the possibilities for detecting and reporting the failure, respawning the failure Worker process, communication logging and employing redundant Master processes, for the case a fault occurrs in the main process.

Our implementation has been tested with a variety of synthetic Master-Worker applications, in which we choose to terminate Worker processes in different parts of the code, in order to demonstrate the accuracy of detecting and handling the failures. We also experimented with a full-fledged application, wich performs numerical optimization and utilizing the Multistart method. These experiments aim to estimate the additional execution time that our system introduces, as compared to a conventional MPI implementation that does not offer fault tolerance. Our results shows that FTMW introduces negligible overheads.

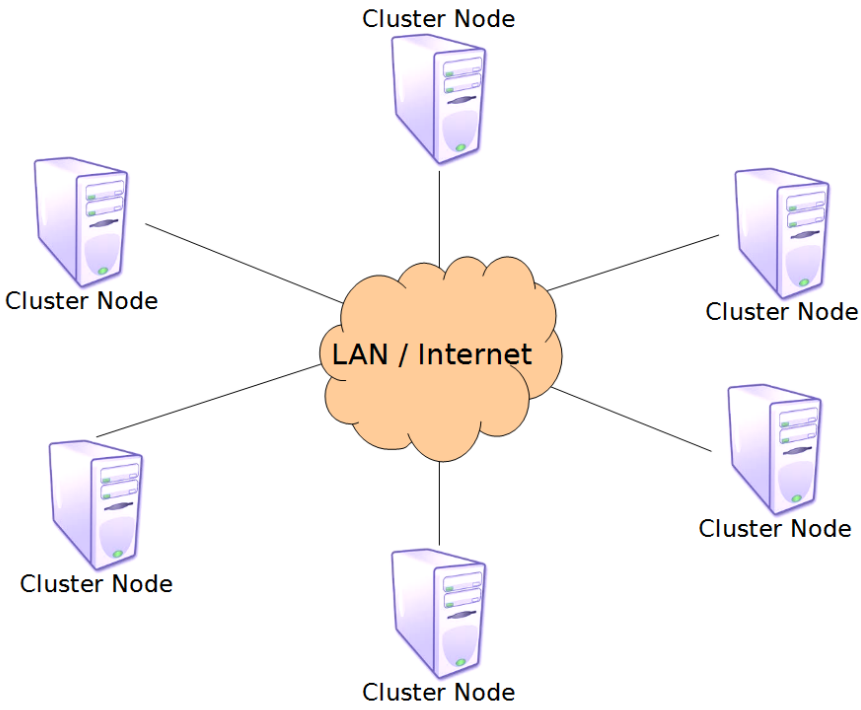
ΚΕΦΑΛΑΙΟ 1. ΕΙΣΑΓΩΓΗ

Προγραμματισμός Συστάδων Υπολογιστών
Στόχοι
Δομή της Διατριβής

Η ανάγκη για όλο και περισσότερη επεξεργαστική ισχύ και η περιορισμένη επίδοση που προσφέρει ένας απλός υπολογιστής, έχει οδηγήσει στη σύνθεση και αξιοποίηση συστάδων υπολογιστών. Μεγάλα κατανεμημένα συστήματα, αποτελούμενα από εκατοντάδες ή ακόμα και χιλιάδες ανεξάρτητους υπολογιστές είναι διαθέσιμα πλέον. Συνδυάζοντας απλούς συμβατικούς υπολογιστές που συνδέονται μέσω ενός δικτύου, διαμοιράζονται τους συνολικούς πόρους του συστήματος και συνεργάζονται για την επίτευξη ενός αποτελέσματος, συνθέτεται ουσιαστικά ένας υπερυπολογιστής με πολλές επεξεργαστικές μονάδες. Εκτός όμως από επεξεργαστική δύναμη, αυξάνεται και η διαθεσιμότητα αποθηκευτικού χώρου, καθώς κάθε υπολογιστής ή αλλιώς κόμβος της συστάδας, μπορεί να έχει διαθέσιμα και προσβάσιμα και τα δεδομένα αλλά και τον αποθηκευτικό χώρο οποιουδήποτε άλλου υπολογιστή συμμετέχει στη συστάδα. Επίσης το δίκτυο που συνδέει τους υπολογιστές που αποτελούν τη συστάδα μεταξύ τους, μπορεί να είναι είτε τοπικό, είτε να πρόκειται για το internet. Τυπικό παράδειγμα μιας συστάδας υπολογιστών φαίνεται στην Εικόνα 1.1. Η δημιουργία συστάδων υπολογιστών, ουσιαστικά προσέφερε τη δυνατότητα σε αρκετές επιστημονικές ομάδες και εργαστήρια, να δημιουργήσουν με οικονομικό τρόπο και να αξιοποιήσουν, μεγάλη επεξεργαστική δύναμη.

Οι συστάδες υπολογιστών, αποτελούν ιδανική πλατφόρμα για την εκτέλεση απαιτητικών μεγάλου μεγέθους εφαρμογών. Τέτοιου είδους εφαρμογές, που χρήζουν μεγάλου χρόνου εκτέλεσης για την ολοκλήρωσή τους και άρα παραγωγή

αποτελεσμάτων τους, απαιτούν και μεγαλύτερη αναγκαιότητα για αξιοπιστία απέναντι σε πιθανά σφάλματα που μπορούν να εμφανιστούν κατά τη διάρκεια εκτέλεσής τους. Πολύ περισσότερο οι χρήστες θα επιθυμούσαν, η εφαρμογή τους να συνεχίσει την ορθή εκτέλεση της και να αποφύγει πιθανές συνέπειες που θα μπορούσαν να επηρεάσουν τη συνολική εκτέλεσή της. Είναι βέβαια αρκετά σπάνια η εμφάνιση σφάλματος, αλλά όσο προσθέτουμε μηχανήματα στο γενικότερο σύστημα, μιας και αυτό είναι και το ζητούμενο προκειμένου να αποκτήσουμε μεγαλύτερη επεξεργαστική ισχύ, αυξάνεται ανάλογα και η πιθανότητα να συμβεί σφάλμα στο σύστημα κατά τη διάρκεια της εκτέλεσης της εφαρμογής. Πόσο μάλλον αν βάλουμε και το γεγονός οτι οι εφαρμογές τρέχουν για μεγάλα χρονικά διαστήματα.



Εικόνα 1.1: Συστάδα Υπολογιστών

Πέρα λοιπόν από την ανάγκη για μεγαλύτερη επεξεργαστική ισχύ που προσφέρουν οι συστάδες υπολογιστών εγείρεται και το κρίσιμο ζήτημα της αξιοπιστίας που αυτά μπορούν να προσφέρουν, δηλαδή το ζήτημα της συμπεριφοράς τους στην εμφάνιση σφαλμάτων. Το επιθυμητό από ένα τέτοιο σύστημα είναι η διατήριση της ορθότητας

εκτέλεσης του σε περίπτωση σφάλματος που μπορεί να υπάρξει κατά τη διάρκεια εκτέλεσης κάποιας εφαρμογής. Το πρόβλημα αυτό, δεν αφορά μόνο το λειτουργικό σύστημα ή την αρχιτεκτονική που υπάρχει στο γενικότερο σύστημα. Είναι μια ιδιότητα που θα πρέπει να απασχολεί και τον ίδιο τον προγραμματιστή κατά τη διαδικασία ανάπτυξης της εφαρμογής του. Η παρούσα εργασία, ασχολείται με αυτό το αντικείμενο.

1.1. Προγραμματισμός Συστάδων Υπολογιστών

Η διαχείριση των προσφερόμενων πόρων σε συστήματα συστάδων υπολογιστών απαιτεί περισσότερες γνώσεις και προσοχή από τον προγραμματιστή κατά τη διάρκεια ανάπτυξης της εφαρμογής. Η τοπολογία και η αρχιτεκτονική των συστάδων υπολογιστών είναι τα δύο χαρακτηριστικά που καθορίζουν και τον προγραμματισμό τους. Η μνήμη σε ένα τέτοιο σύστημα είναι κατανεμημένη και ο πιο συνηθισμένος τρόπος επικοινωνίας είναι η ανταλλαγή μηνυμάτων μέσω ενός εγκατεστημένου δικτύου μεταξύ των υπολογιστών που αποτελούν τη συστάδα. Θεμελιώδης απαίτηση σε τέτοια συστήματα είναι και η ικανότητα διατήρησης της εκτέλεσης της εφαρμογής ανεπηρέαστα σε περίπτωση εμφάνισης σφάλματος σε κάποιον κόμβο ή γενικότερα σε κάποιο επίπεδο του συστήματος. Για την επίτευξη του παραπάνω, υπάρχουν μια σειρά από μεθοδολογίες αλλά και κάποιες βασικές απαιτήσεις όπως η έγκαιρη ανίχνευση του σφάλματος, η διαθεσιμότητα της απαιτούμενης πληροφορίας ώστε ο υπολογισμός να συνεχιστεί και αν είναι δυνατόν να επανεκκινήσει. Παρόλα αυτά, η βασική αρχή της ανοχής σφαλμάτων είναι η ανεπηρέαστη εκτέλεση των υπολογισμών στους επεξεργαστές σε περίπτωση εμφάνισης σφάλματος σε κάποιον άλλον.

Ο πιο διαδεδομένος τρόπος προγραμματισμού τέτοιων συστημάτων είναι μέσω του Message Passing Interface (MPI) [22], [5]. Το πρότυπο αυτό προσφέρει μια πλατφόρμα για ανάπτυξη αφαιρετικών κατανεμημένων αλγορίθμων με καλό επίπεδο κλιμακωσιμότητας για συστάδες υπολογιστών, αποκρύπτοντας λεπτομέρειες της αρχιτεκτονικής από τον προγραμματιστή της εφαρμογής. Η διεπαφή του MPI, προσφέρει κλήσεις αποστολής και λήψης μηνυμάτων μεταξύ δύο υπολογιστών, κλήσεις για την μαζική και αποδοτική επικοινωνία μεταξύ ομάδων υπολογιστών αλλά

και γενικότερες διαχειριστικές κλήσεις. Παρόλα αυτά το MPI ως πρότυπο δεν ορίζει κάποια ενιαία στρατηγική αντιμετώπισης σφαλμάτων, πολύ περισσότερο δεν αναφέρει ούτε καν τον όρο. Η παρούσα εργασία εστιάζει στην αντιμετώπιση αυτού του κενού.

Ένα άλλο κομμάτι που αφορά τον προγραμματισμό τέτοιων συστημάτων είναι και το προγραμματιστικό μοντέλο/σχήμα πάνω στο οποίο θα βασιστεί η ανάπτυξη της εφαρμογής. Ένα τυπικό και ευρέως χρησιμοποιούμενο παράδειγμα είναι το μοντέλο Master-Worker το οποίο θα μας απασχολήσει και στην εργασία. Η διεργασία που έχει το ρόλο του Master είναι υπεύθυνη για την κατανομή της εργασίας και των δεδομένων στις υπόλοιπες διεργασίες που έχουν το ρόλο του Worker. Οι τελευταίες υπολογίζουν ένα (επιμέρους) αποτέλεσμα και το στέλνουν πίσω στον Master ο οποίος θα συνθέσει το συνολικό αποτέλεσμα. Το μοντέλο αυτό είναι ιδιαίτερα διαδεδομένο για την ανάπτυξη εφαρμογών που εκτελούνται σε συστάδες υπολογιστών, και κατά επέκταση για εφαρμογές MPI, λόγω της εκ φύσεως δομής του, που ευνοεί την κατανομή εργασίας ανάμεσα στις υπολογιστικές οντότητες που το αποτελούν, δηλαδή στις διεργασίες Worker και την εξασφάλιση ανεξάρτητης εκτέλεσης μεταξύ των υπολογισμών της κάθε μιας.

1.2. Στόχοι

Στόχος της παρούσας εργασίας είναι να προσφέρει ένα αξιόπιστο επίπεδο χειρισμού σφαλμάτων που μπορεί να εμφανιστούν σε κάποια διεργασία MPI. Ουσιαστικά να γεφυρώσει το κενό αξιοπιστίας του προτύπου σε περιπτώσεις σφαλμάτων, προσφέροντας τη δυνατότητα έγκαιρης ανίχνευσης και αναφοράς του σφάλματος στο επίπεδο της εφαρμογής αλλά και εύκολης αντιμετώπισής του από τη σκοπιά του χρήστη.

Για το λόγο αυτό, μελετήθηκε η συμπεριφορά των πιο διαδεδομένων υλοποιήσεων του MPI, σε περιπτώσεις εμφάνισης σφαλμάτων κατά την εκτέλεση της εφαρμογής MPI για να διαπιστωθεί το κατά πόσο αυτές, υπακούν στο πρότυπο για τις περιπτώσεις σφαλμάτων. Από τα πειράματα διαπιστώσαμε οτι δεν παρουσιάζουν μια

αξιόπιστη συμπεριφορά, κυρίως στο θέμα της ανίχνευσης και αναφοράς του σφάλματος στο επίπεδο της εφαρμογής και έτσι σχεδιάσαμε ένα ενδιάμεσο επίπεδο ανάμεσα στην εφαρμογή MPI και στην υλοποίηση MPI, με σκοπό να προσφέρει αξιόπιστη ανίχνευση και αναφορά σφαλμάτων στο επίπεδο του χρήστη.

Σε δεύτερο επίπεδο σχεδιάσαμε και υλοποιήσαμε ένα μηχανισμό που προσφέρει λειτουργίες για αντιμετώπιση σφαλμάτων κατά την εκτέλεση της εφαρμογής σε σχήμα Master-Worker. Με αυτόν τον μηχανισμό ο χρήστης, μπορεί να εξασφαλίσει αξιοπιστία τόσο στην πλευρά του Worker, δίνοντας τη δυνατότητα να τον επανεκκινήσει σε περίπτωση που τερματιστεί η λειτουργία του λόγω σφάλματος, όσο και στην πλευρά του Master, προσφέροντας τη δυνατότητα ορισμού περισσότερων διεργασιών Master οι οποίες εκτελούνται παράλληλα με τη βασική και μπορούν να λάβουν τη θέση της σε περίπτωση σφάλματος.

Η ώθηση για το θέμα που πραγματεύεται η εργασία, δόθηκε κυρίως από την αυξημένη ανάγκη αξιόπιστης εκτέλεσης εφαρμογών που χρήζουν μεγάλης επεξεργαστικής ισχύος από εκατοντάδες ή ακόμα και χιλιάδες μηχανήματα για μεγάλα χρονικά διαστήματα. Σε μεγάλα χρονικά διαστήματα εκτέλεσης η πιθανότητα εμφάνισης του σφάλματος κατά τη διάρκεια των υπολογισμών είναι πολύ μεγαλύτερη. Η διατήρηση της εκτέλεσης τους ανεξάρτητα από την εμφάνιση σφαλμάτων, αποτελεί βασικό ζητούμενο για τέτοιες εφαρμογές.

Στην εργασία γίνεται επίσης και χρήση του υλοποιημένου συστήματος, πάνω σε μια εφαρμογή τύπου Master-Worker σε περιβάλλον εκτέλεσης συστάδας υπολογιστών τόσο σε εκτέλεση χωρίς σφάλματα, όσο και σε εκτέλεση με εμφάνιση σφαλμάτων σε διάφορα σημεία του κώδικα της εφαρμογής. Πρόκειται για μια εφαρμογή ολικής βελτιστοποίησης, που χρησιμοποιεί μια τυπική μέθοδο τοπικής γραμμικής βελτιστοποίησης στο χώρο αναζήτησης σε συνδυασμό με τη μέθοδο Multistart. Σύμφωνα με την τελευταία, πολλές μέθοδοι τοπικής βελτιστοποίησης ξεκινούν παράλληλα από διαφορετικά σημεία του χώρου και από τα επιμέρους αποτελέσματα εξάγεται το βέλτιστο. Τα πειράματα αφορούν τόσο το θέμα της απόδειξης της ορθότητας εκτέλεσης της υλοποίησης, κυρίως με την ύπαρξη πολλαπλών διεργασιών Master, όσο και της σύγκρισης της εκτέλεσης της υλοποίησης με συμβατικές

υλοποιήσεις MPI που δεν προσφέρουν κάποια επιπλέον χαρακτηριστικά ανοχής σφαλμάτων, για να διαπιστωθεί το επιπλέον κόστος εκτέλεσης που εισάγει η υλοποίηση μας. Άλλα πειράματα, αφορούσαν το πόσο επιπλέον κόστος εισάγουν οι λειτουργίες της υλοποίησης για επανεκκίνηση μιας εσφαλμένης διεργασίας καθώς και για ύπαρξη πολλαπλών διεργασιών Master.

1.3. Δομή της Διατριβής

Ακολουθεί μια περιληπτική περιγραφή της δομής των κεφαλαίων της παρούσας εργασίας.

ΚΕΦΑΛΑΙΟ 2: Γίνεται μια παρουσίαση και περιγραφή του προτύπου MPI, ενός προτύπου για τον προγραμματισμό κατανεμημένων συστημάτων μέσω ανταλλαγής μηνυμάτων. Παρουσιάζονται βασικές έννοιες του προτύπου και παράδειγμα χρήσης του. Επίσης παρουσιάζονται οι διάφορες εκδόσεις του προτύπου και υλοποιήσεις του. Τέλος περιγράφεται και το μοντέλο ανάπτυξης εφαρμογών Master-Worker.

ΚΕΦΑΛΑΙΟ 3: Γίνεται μια κουβέντα γύρω από τα σφάλματα και την ιδιότητα της ανοχής σφαλμάτων και για το πως αυτή μπορεί να συνδυαστεί με το πρότυπο MPI. Παρουσιάζονται τεχνικές που έχουν χρησιμοποιηθεί για υποστήριξη ανοχής σφαλμάτων στο MPI. Παρουσιάζονται επίσης τα αποτελέσματα πειραματισμού που κάναμε με διάφορες υλοποιήσεις του MPI για να διαπιστώσουμε τη συμπεριφορά τους σε περιπτώσεις σφαλμάτων.

ΚΕΦΑΛΑΙΟ 4: Γίνεται η παρουσίαση και η περιγραφή του μηχανισμού που αναπτύξαμε. Περιγράφεται ο τρόπος που ανιχνεύει και αναφέρει τα σφάλματα στο επίπεδο της εφαρμογής, οι λειτουργίες ανοχής σφαλμάτων που προσφέρει στον προγραμματιστή και η συμπεριφορά των διεργασιών που συμμετέχουν στην εκτέλεση.

ΚΕΦΑΛΑΙΟ 5: Παρουσίαση και συζήτηση γύρω από τα πειραματικά αποτελέσματα της εκτέλεσης της υλοποίησης. Γίνεται περιγραφή του περιβάλλοντος εκτέλεσης των πειραμάτων αλλά και των εφαρμογών σύμφωνα με τις οποίες έγινε ο έλεγχος ορθότητας και της απόδοσης του μηχανισμού.

ΚΕΦΑΛΑΙΟ 6: Γίνεται ο επίλογος της εργασίας και δίνονται κάποιες συμβουλές για περαιτέρω μελλοντική εργασία.

ΚΕΦΑΛΑΙΟ 2. ΠΡΟΓΡΑΜΜΑΤΙΣΜΟΣ ΜΕ MPI

Όπως αναφέρθηκε στην εισαγωγή, αυτή η εργασία ασχολείται με θέματα που έχουν να κάνουν με τον προγραμματισμό συστάδων υπολογιστών και τα ζητήματα που αφορούν τη συμπεριφορά του συστήματος σε περιπτώσεις σφαλμάτων που μπορούν να συμβούν σε αυτό. Ένας από τους πιο διαδεδομένους τρόπους προγραμματισμού τέτοιων συστημάτων που αποτελούνται από πολλούς υπολογιστές, με κατανεμημένη μνήμη είναι η ανταλλαγή μηνυμάτων μεταξύ των κόμβων και το πιο διαδεδομένο πρότυπο για τη διευκόλυνση αυτής τη διαδικασίας είναι το MPI. Σε αυτό το κεφάλαιο θα γίνει μια περιγραφή του προτύπου αυτού, των υλοποιήσεών του καθώς και μια παρουσίαση σχετικής εργασίας για την αντιμετώπιση σφαλμάτων.

2.1. Το Πρότυπο MPI

Το Message Passing Interface (MPI) [22], [5], είναι μια βιβλιοθήκη συναρτήσεων για την ανταλλαγή μηνυμάτων μεταξύ διεργασιών. Είναι ένα επίπεδο διεπαφής για τον προγραμματιστή της εφαρμογής. Δεν καθορίζει το πρωτόκολλο επικοινωνίας και ανταλλαγής των μηνυμάτων ή της διαχείρισης των πακέτων που θα διαδοθούν στο επίπεδο του δικτύου που συνδέει τους κόμβους. Τα δύο χαρακτηριστικά που προσφέρει το MPI είναι η κλιμακωσιμότητα που μπορεί να προσφέρει στην ανάπτυξη

της εφαρμογής και βέβαια από τη στιγμή που είναι ένα πρότυπο οδηγιών, η μεταφερσιμότητα του. Παρόλα αυτά, αν και προσφέρει μια διεπαφή που απαλλάσσει τον προγραμματιστή από αρκετές λεπτομέρειες της αρχιτεκτονικής και του δικτύου, πρόκειται για χαμηλού επιπέδου βιβλιοθήκη. Ο προγραμματιστής θα πρέπει να αναπτύξει ρητά και όχι αόριστα παραλληλοποιήσιμο και κατανεμημένο κώδικα.

Οι υλοποιήσεις του MPI αποτελούνται από ένα σύνολο κλήσεων συναρτήσεων για γλώσσες προγραμματισμού C, C++ και Fortran. Βέβαια και για οποιαδήποτε γλώσσα που διαθέτει διεπαφή για τη χρήση αυτών των γλωσσών. Όπως προαναφέρθηκε, η μεταφερσιμότητα είναι ένα από τα βασικά πλεονεκτήματα μιας εφαρμογής MPI, όχι μόνο γιατί πρόκειται για μια προτυποποίηση οδηγιών αλλά και γιατί υλοποιήσεις του υποστηρίζουν μια σειρά από διάφορες παράλληλες αρχιτεκτονικές κατανεμημένης μνήμης. Εκτός αυτού όμως, υπάρχουν και υλοποιήσεις για συστήματα κοινόχρηστης μνήμης αλλά και αρχιτεκτονικές NUMA [12].

Η βασική λειτουργία που προσφέρει το MPI, είναι η ανταλλαγή μηνυμάτων μεταξύ των διεργασιών που συμμετέχουν στο πρόγραμμα. Σε κάθε διεργασία στην αρχή της εκτέλεσης του προγράμματος, ανατίθεται ένας μοναδικός αριθμός ως αναγνωριστικό από το 0 μέχρι το πλήθος των διεργασιών μείον ένα (N-1), ο οποίος χρησιμεύει ως ταυτότητα για κάποια συγκεκριμένη διεργασία. Οι επικοινωνίες επιτυγχάνονται είτε μεταξύ δύο διεργασιών που έχουν ανάγκη να ανταλλάξουν δεδομένα, είτε συγκεντρωτικά ανάμεσα σε ένα σύνολο διεργασιών που ανήκει σε μια ομάδα. Τέτοιου είδους επικοινωνία μπορούμε για παράδειγμα να έχουμε όταν μια διεργασία θέλει να μοιράσει τα δεδομένα ενός πίνακα στις υπόλοιπες διεργασίες ή να συγκεντρώσει τα επιμέρους αποτελέσματα των υπολοίπων διεργασιών σε έναν πίνακα. Παρόλα αυτά οι συλλογικές επικοινωνίες από τη στιγμή που για να ολοκληρωθούν θα πρέπει όλες οι διεργασίες να έχουν περατώσει την εκτέλεση της συγκεκριμένης λειτουργίας, ουσιαστικά συντελείται ένας ολικός συγχρονισμός στις διεργασίες της υλοποίησης, επηρεάζουν αρνητικά την απόδοση της εκτέλεσης. Όσες περισσότερες διεργασίες συμμετέχουν στην κλήση, τόσο πιο αργά ολοκληρώνεται η εκτέλεσή της.

Οι ομάδες διεργασιών MPI, αποκαλούνται Communicators και σε κάθε κλήση επικοινωνίας MPI θα πρέπει ο προγραμματιστής να δηλώνει και αυτήν την πληροφορία, είτε αυτή αφορά δύο διεργασίες, είτε συγκεντρωτικά ολόκληρη την ομάδα. Δεν είναι δεσμευτικό από το πρότυπο, οτι οι διεργασίες που θα επικοινωνήσουν θα πρέπει να ανήκουν στην ίδια ομάδα, δηλαδή στον ίδιο communicator. Το MPI δίνει τη δυνατότητα όχι μόνο να επικοινωνούν οι διεργασίες της ίδιας ομάδας μεταξύ τους μέσω των αντικειμένων MPI intracommunicators, αλλά και με διεργασίες άλλων ομάδων, μέσω intercommunicators. Σε μετέπειτα κεφάλαια της εργασίας θα γίνει αναλυτικότερη περιγραφή αυτών των χαρακτηριστικών.

Για τα χαρακτηριστικά που θα υιοθετήσει το πρότυπο καθώς και για τη γενικότερη εξέλιξή του, υπάρχει μια κοινότητα χρηστών και προγραμματιστών που αποφασίζει [5]. Μέσα από συζητήσεις διαμορφώνονται οι ανάγκες και οι ελλείψεις του προτύπου και ανατίθενται εργασίες για την επίλυση των ζητημάτων του. Η τρέχουσα έκδοση του προτύπου είναι η έκδοση 2 και ήδη έχει αρχίσει η συζήτηση για την επερχόμενη έκδοση 3.

2.1.1. MPI-1

Η πρώτη έκδοση του MPI, 1.0 δημοσιεύθηκε τον Ιούνιο του 1994. Η αναγκαιότητα προτυποποίησης μια διεπαφικής βιβλιοθήκης για την ανταλλαγή μηνυμάτων, οδήγησε μια ομάδα αρκετών ανθρώπων που συμμετείχαν σε διάφορους οργανισμούς και πανεπιστήμια στη σύσταση ενός forum για το MPIF (Message Passing Interface Forum) από τον Ιανουάριο του 1993. Λίγο καιρό αργότερα, τον Μάιο του 1995 η κοινότητα συνεδρίασε ξανά για να διορθώσει κάποιες ελλείψεις και κάποια λάθη του προτύπου και οδήγησε στην έκδοση 1.1, η οποία όμως εν τέλει, έχει ελάχιστες διαφορές. Στην μετέπειτα έκδοση 1.2, περιλαμβάνονται 128 συναρτήσεις.

2.1.2. MPI-2

Το MPI-2 [7] ουσιαστικά αποτελεί μια επέκταση της έκδοσης 1.2. Περιλαμβάνει νέες λειτουργίες και δυνατότητες που δεν προσέφερε το προηγούμενο. Μέσα σε αυτές είναι η δυναμική δημιουργία και διαχείριση διεργασιών [9], που καθιστούσε μη δεσμευτικό το γεγονός οτι ο αριθμός διεργασιών που θα εκκινήσουν την εκτέλεση της εφαρμογής MPI δε μπορεί να αλλάξει, δίνοντας τη δυνατότητα στον προγραμματιστή να εκκινήσει μια διεργασία MPI σε οποιοδήποτε μέρος της εφαρμογής και αν επιθυμεί. Συναρτήσεις επικοινωνίας μιας πλευράς, με την έννοια οτι μια διεργασία MPI μπορεί να επικοινωνήσει με κάποια άλλη στέλνοντας τα δεδομένα της χωρίς να είναι απαραίτητο η δεύτερη να συμμετέχει. Επίσης παράλληλες και κλιμακώσιμες κλήσεις Εισόδου/Εξόδου, περισσότερες συλλογικές κλήσεις καθώς και υποστήριξη για επιπλέον γλώσσες προγραμματισμού. Ουσιαστικά το MPI-2 καθορίζει πάνω από 500 κλήσεις MPI και υποστηρίζει τις γλώσσες προγραμματισμού Fortran90, C και C++.

2.1.3. MPI-3 (Μελλοντικά)

Αυτή τη στιγμή, βρίσκεται σε εξέλιξη η συζήτηση για την ανάπτυξη του MPI από τις εκδόσεις 2 στην έκδοση 3.0 [17]. Δεν έχει προτυποποιηθεί και άρα δημοσιευθεί ακόμα, αλλά έχουν επισημανθεί τα προβλήματα και οι ελλείψεις του προτύπου. Η μεγαλύτερη έλλειψη που παρουσιάζει αφορά τον ορισμό μιας ενιαίας στρατηγικής διαχείρισης και αντιμετώπισης των σφαλμάτων. Πιο συγκεκριμένα, αφορά το ποια θα πρέπει να είναι η ενδεικνυόμενη από το πρότυπο συμπεριφορά των υπόλοιπων διεργασιών σε περίπτωση που κάποια διεργασία MPI παρουσιάσει σφάλμα είτε αυτή ανήκει στην ίδια ομάδα, είτε όχι. Γίνεται επίσης αναφορά και στην αναγκαιότητα δημιουργίας και ορισμού νέων δυνατοτήτων χειρισμού λαθών [24]. Αξίζει να επισημανθεί οτι ο όρος "ανοχή σφαλμάτων" δεν υπάρχει ούτε καν σαν αναφορά σε καμία από τις προηγούμενες εκδόσεις.

2.2. Επισκόπηση του MPI

Το MPI προσφέρει μια γενικότερη διεπαφή που αναπαριστά μια εικονική τοπολογία των υπολογιστικών οντοτήτων που την αποτελούν, που ουσιαστικά πρόκειται για διεργασίες, χωρίς βέβαια αυτό να είναι δεσμευτικό. Υπάρχουν υλοποιήσεις του MPI και για αρχιτεκτονικές κοινόχρηστης μνήμης, όπου οι υπολογιστικές οντότητες είναι νήματα.. Προσφέρει λειτουργίες συγχρονισμού και επικοινωνίας ανάμεσα στις διεργασίες χωρίς η γλώσσα προγραμματισμού που θα χρησιμοποιηθεί για την ανάπτυξη της εφαρμογής να επηρεάζει τη χρήση του προτύπου.

2.2.1. Βασικές Έννοιες

Αρκετές δυνατότητες είναι διαθέσιμες από το MPI για την ανάπτυξη κατανεμημένων εφαρμογών. Θα παρουσιαστούν κάποιες βασικές έννοιες του προτύπου σύμφωνα με το πως έχουν διαμορφωθεί στην έκδοση MPI-2.

Communicator

Ο communicator, είναι βασικό συστατικό του MPI και η λειτουργία του είναι να συνδέει ομάδες διεργασιών μεταξύ τους. Στις διεργασίες που συμμετέχουν σε έναν communicator, ανατίθεται ένα μοναδικό αναγνωριστικό ως ταυτότητα. Κάθε κλήση επικοινωνίας του MPI πρέπει να περιλαμβάνει και τον communicator που συσχετίζει τις εμπλεκόμενες με αυτήν, διεργασίες. Ο communicator, που χρησιμοποιείτε για την κάλυψη των επικοινωνιακών αναγκών μιας ομάδας διεργασιών αποκαλείται και intracommunicator, ενώ αυτός που χρησιμοποιείτε για την επικοινωνία μεταξύ διεργασιών διαφορετικών ομάδων διεργασιών, ονομάζεται intercommunicator. Τις διαφορές και την αναλύτικότερη χρήση και περιγραφή των δύο αυτών communicator, θα περιγράψουμε σε επόμενο κεφάλαιο. Ο αρχικός communicator του προγράμματος στον οποίον ανήκουν όλες οι διεργασίες που εκκινούν την εφαρμογή, ονομάζεται MPI_COMM_WORLD.

Το MPI προσφέρει επίσης και τη δυνατότητα της επεξεργασίας των communicator μέσα από κλήσεις του ίδιου του προτύπου. Ο χρήστης μπορεί να διασπάσει ή να

ενώσει ομάδες διεργασιών εφαρμόζοντας τις αντίστοιχες λειτουργίες πάνω στους communicators.

Επικοινωνίες Point-to-point

Οι πιο βασικές επικοινωνιακές κλήσεις του MPI, αφορούν την επικοινωνία μεταξύ δύο διεργασιών οι οποίες και θα πρέπει να συμμετέχουν στην κλήση για να επιτευχθεί η ολοκλήρωσή της. Τυπικό παράδειγμα και πιο συνηθισμένο, είναι οι κλήσεις αποστολής και παραλαβής δεδομένων, MPI_Send και MPI_Recv.

Αν μια διεργασία επιθυμεί να στείλει δεδομένα σε κάποια άλλη συγκεκριμένη, θα πρέπει να ορίσει στην κλήση της συνάρτησης το ποιά θα είναι αυτή, σύμφωνα με το αναγνωριστικό που έχει η τελευταία στον communicator. Ο τελευταίος επίσης πρέπει να γίνει ρητός στην κλήση της συνάρτησης. Για να ολοκληρωθεί τυπικά η κλήση αποστολής, θα πρέπει και η έτερη διεργασία να παραλάβει τα δεδομένα, συμμετέχοντας και αυτή με την κλήση παραλαβής. Ο ορισμός της κλήσης αποστολής:

```
int MPI_Send(void *buf, int count, MPI_Datatype datatype,
             int dest, int tag, MPI_Comm comm)
```

Τα πρώτα τρία ορίσματα, αφορούν τα χαρακτηριστικά των δεδομένων προς αποστολή. Το πρώτο όρισμα αναφέρεται στη διεύθυνση των δεδομένων που πρόκειται να σταλθούν, το δεύτερο στην ποσότητα των δεδομένων και το τρίτο στον τύπο τους. Στη συνέχεια ακολουθεί η παράμετρος που ορίζει το ποιά διεργασία θα παραλάβει τα συγκεκριμένα δεδομένα και η παράμετρος tag, που δίνει τη δυνατότητα στον προγραμματιστή να δώσει ένα αναγνωριστικό στην κλήση αποστολής που θα τη συσχετίσει με μια αντίστοιχη παραλαβής που θα έχει ίδια τιμή σε αυτό το πεδίο. Τέλος ορίζεται και ο communicator στον οποίο συμμετέχουν και οι δύο εμπλεκόμενες διεργασίες. Η λογική των ορισμάτων είναι παρόμοια και στην κλήση παραλαβής, με τη διαφορά οτι υπάρχει ακόμα ένα πεδίο, status το οποίο περιλαμβάνει πληροφορίες (metadata) για το μήνυμα που έλαβε η διεργασία:

```
int MPI_Recv(void *buf, int count, MPI_Datatype datatype,
             int source, int tag, MPI_Comm comm,
             MPI_Status *status)
```

Οι κλήσεις σε αυτήν την κατηγορία μπορεί να είναι είτε εμποδιστικές, είτε και όχι. Ο ορισμός της κλήσης παραλαβής, προϋποθέτει οτι θα πρέπει πρώτα να ληφθούν τα δεδομένα για να συνεχίσει την εκτέλεση η διεργασία και έτσι η κλήση της θεωρείτε εμποδιστική. Το MPI προσφέρει και μια αντίστοιχη, μη-εμποδιστική κλήση MPI_Irecv, όπου αν τα δεδομένα δεν έχουν παραληφθεί στο σημείο που κλήθηκε η συνάρτηση, η διεργασία δε θα περιμένει να ληφθούν και έχει τη δυνατότητα να τα παραλάβει αργότερα. Από την άλλη, η κλήση αποστολής δεν απαιτεί τα δεδομένα να έχουν παραληφθεί από την έτερη διεργασία για να ολοκληρωθεί. Υπάρχει όμως η δυνατότητα από το MPI με την κλήση MPI_Ssend, η διεργασία να μπλοκάρει μέχρις ότου επιβεβαιωθεί από την υλοποίηση οτι τα δεδομένα έχουν παραληφθεί.

Συλλογικές Επικοινωνίες

Οι συλλογικές επικοινωνίες αφορούν την επικοινωνία που συντελείται μεταξύ όλων των διεργασιών που συμμετέχουν σε μια ομάδα. Η πιο χαρακτηριστική κλήση από αυτήν την κατηγορία, είναι η MPI_Bcast (συντομογραφία του broadcast) για μετάδοση των δεδομένων. Σε αυτή τη λειτουργία, ουσιαστικά μια διεργασία, η root, θα στείλει σε όλες τις υπόλοιπες της ομάδας, ένα δεδομένο. Η αντίθετη διαδικασία είναι η MPI_Reduce, όπου μια διεργασία συγκεντρώνει τα επιμέρους δεδομένα από κάθε άλλη που συμμετέχει στην ομάδα.

```
int MPI_Bcast(void *buf, int count, MPI_Datatype datatype
              int root, MPI_Comm comm)
```

Κάθε κλήση συλλογικής επικοινωνίας, πρέπει να κληθεί από κάθε διεργασία που συμμετέχει σε αυτή, όχι απαραίτητα στο ίδιο σημείο κώδικα. Για να ολοκληρωθεί, θα πρέπει εν τέλη να την έχουν καλέσει όλες οι διεργασίες. Επομένως, με τις συλλογικές επικοινωνίες επιτελείται επίσης και ένας συγχρονισμός μεταξύ της ομάδας διεργασιών. Εύκολα γίνεται κατανοητό οτι οι συλλογικές επικοινωνίες μπορούν να υλοποιηθούν και από σύνθεση απλών επικοινωνιών Point-to-point. Παρόλα αυτά είναι συνήθως υλοποιημένες με τέτοιο τρόπο ώστε να προσφέρουν βελτιστοποιημένες επιδόσεις, ανάλογα και με την οργάνωση του όλου συστήματος.

2.2.2. Παράδειγμα προγράμματος

Συνοψίζοντας τα όσα υπόθηκαν στις βασικές έννοιες του προτύπου, παρατίθεται στην Εικόνα 2.1 ένα απλό παράδειγμα ενός προγράμματος MPI. Το παράδειγμα δεν παρουσιάζει κάποιο λειτουργικό πρόγραμμα και παρουσιάζεται κυρίως για λόγους κατανόησης των παραπάνω.

```
#include "mpi.h"

int main(int argc, char** argv) {
  int Num, Rank, i, temp;

  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &Num);
  MPI_Comm_rank(MPI_COMM_WORLD, &Rank);

  MPI_Bcast(&Num, 1, MPI_INT, 0, MPI_COMM_WORLD);

  if(Rank == 0) {
    for(i = 1; i < Num; i++)
      MPI_Send(&Rank, 1, MPI_INT, i, 1, MPI_COMM_WORLD);
  }
  else {
    MPI_Recv(&temp,1,MPI_INT,0,MPI_ANY_TAG, MPI_COMM_WORLD, NULL);
  }

  MPI_Finalize();
  return 0;
}
```

Εικόνα 2.1: Παράδειγμα Εφαρμογής MPI

Κάθε πρόγραμμα MPI, ξεκινάει με την κλήση MPI_Init και ολοκληρώνεται με την κλήση MPI_Finalize. Οι κλήσεις αυτές εκτελούνται από όλες τις διεργασίες που συμμετέχουν στην εφαρμογή. Οι κλήσεις MPI_Comm_size και MPI_Comm_rank , που αφορούν τον εξορισμού communicator του MPI (MPI_COMM_WORLD), επιστρέφουν το συνολικό αριθμό των διεργασιών MPI που συμμετέχουν στο πρόγραμμα και το μοναδικό αναγνωριστικό της κάθε μιας, αντίστοιχα. Στη συνέχεια υπάρχει η κλήση MPI_Bcast, που όπως αναφέρθηκε προηγουμένως πρέπει να εκτελεστεί από όλες τις διεργασίες για να ολοκληρωθεί. Στη συγκεκριμένη κλήση η διεργασία 0, στέλνει σε όλες τις υπόλοιπες την τιμή της μεταβλητής Num. Ανάλογα με το αναγνωριστικό τις κάθε διεργασίας εκτελείται και ο κώδικας εν συνεχεία. Η

διεργασία 0, θα στείλει σε όλες τις υπόλοιπες διεργασίες ρητά, το αναγνωριστικό της. Από την άλλη, οι υπόλοιπες διεργασίες θα παραλάβουν τα δεδομένα από τη διεργασία 0. Αξίζει να σημειωθεί και η παρουσία της τιμή MPI_ANY_TAG, στο πεδίο tag της κλήσης MPI_Recv, με την οποία γίνεται ταύτιση με οποιαδήποτε τιμή tag και να έχει το μήνυμα.

2.3. Υλοποιήσεις MPI

Το πρότυπο του MPI, πέρα από μια διεπαφή οδηγιών για τη συγγραφή κατανεμημένων εφαρμογών, περιγράφει και τον τρόπο με τον οποίο οι διάφορες υλοποιήσεις του προτύπου θα πρέπει να συμπεριφέρονται σύμφωνα με αυτό. Στην ενότητα αυτή, θα αναφερθούν τρεις από τις πιο διαδεδομένες υλοποιήσεις του MPI. Κάθε μια με τα δικά της χαρακτηριστικά, επιδόσεις και ιδιαιτερότητες αλλά όλες στην υπηρεσία των οδηγιών του προτύπου.

2.3.1. MPICH

Το MPICH [25], είναι ίσως η πιο ευρέως διαδεδομένη υλοποίηση του MPI. Είναι διαθέσιμο ελεύθερα για χρήση από κάποιον προγραμματιστή και σύμφωνα με τα κριτήρια του MPI, μεταφέρσιμο για διάφορες αρχιτεκτονικές. Άλλωστε το CH στο όνομα της υλοποίησης προέρχεται από το "Chameleon", μια βιβλιοθήκη παράλληλου προγραμματισμού με υψηλή μεταφερσιμότητα που αναπτύχθηκε από τον William Gropp, έναν από τους εμπνευστές του MPI.

Το MPICH1, που ουσιαστικά είναι η έκδοση του MPICH που αναφέρεται στο πρότυπο MPI-1, αναπτύχθηκε από το Argonne National Laboratory. Για την υποστήριξη του προτύπου MPI-2 αναπτύχθηκε το MPICH2 [18], από την ίδια κοινότητα και με τα ίδια χαρακτηριστικά της μεταφερσιμότητας και της διανομής του ελεύθερα. Ο βασικός στόχος επίσης του MPICH2, ήταν να υποστηρίξει αρχιτεκτονικές πλατφόρμες όπως συστάδες υπολογιστών, SMPs και υπολογιστές με πολυεπεξεργαστές. Το MPICH2 αντικαθιστά το MPICH1, εκτός από τις περιπτώσεις

συστάδων ετερογενών υπολογιστών. Μια από τις εκκρεμής εργασίες του MPICH2 είναι και η υποστήριξη τέτοιων συστημάτων. Η τελευταία έκδοση του MPICH2 είναι η 1.4 και δημοσιεύτηκε στις 16 Ιουνίου του 2011.

2.3.2. LAM/MPI

Το LAM/MPI [26] (Local Area Multiprocessor) είναι ένα προγραμματιστικό και αναπτυξιακό περιβάλλον για MPI πάνω σε ένα δίκτυο με ετερογενείς υπολογιστές. Προσφέρει πλήρη υποστήριξη του MPI-1 και αρκετό βαθμό υποστήριξης στο MPI-2. Επίσης περιλαμβάνει και αρκετές δυνατότητες αποσφαλμάτωσης και παρακολούθησης του κώδικα και της εκτέλεσης του προγράμματος.

Αυτήν τη στιγμή, η κοινότητα του LAM δεν αναπτύσσει κάτι επιπλέον για το συνολικό project. Η δουλειά που επιτελείται πάνω σε αυτό έχει να κάνει κυρίως με τη διόρθωση κάποιων σφαλμάτων στον ήδη υπάρχοντα κώδικα. Ουσιαστικά η ομάδα του LAM έχει επικεντρωθεί στις απαραίτητες ενέργειες που χρειάζεται για τη μετάβαση από το LAM/MPI στο Open MPI, καθώς όπως αναφέρθηκε παραπάνω το πρώτο αποτελεί ένα επιμέρους project του τελευταίου. Βέβαια το Open MPI περιλαμβάνει αρκετές περισσότερες δυνατότητες και καλύτερη απόδοση από το LAM/MPI.

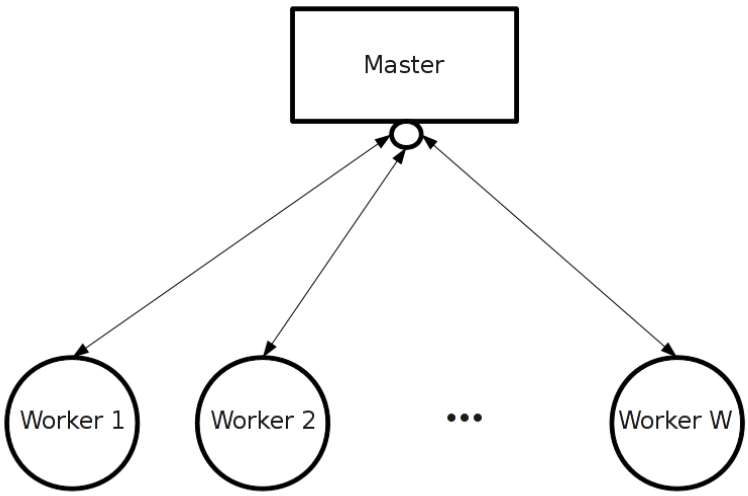
2.3.3. Open MPI

Το Open MPI [6], είναι μια αρκετά φιλόδοξη προσπάθεια που ξεκίνησε με στόχο να αποτελέσει την καλύτερη υλοποίηση του προτύπου MPI. Πρόκειται για ένα project που συνδυάζει πλεονεκτήματα και χαρακτηριστικά από διάφορες υλοποιήσεις όπως του FT-MPI, LA-MPI, LAM/MPI και PACX-MPI. Υποστηρίζει το πρότυπο MPI-2 και είναι πλήρως προσαρμοσμένο πάνω σε αυτό. Συνδυάζει λειτουργίες που πριν δεν ήταν διαθέσιμες σε επίπεδο ανοιχτού κώδικα για την υποστήριξη του MPI όντας το ίδιο ανοιχτού κώδικα και ελεύθερης διανομής.

Υπάρχει μια μεγάλη κοινότητα από οργανισμούς που συμμετέχουν στην ανάπτυξη του Open MPI. Ανάμεσα σε αυτούς είναι το πανεπιστήμιο της Indianna (LAM/MPI), το πανεπιστήμιο του Tenesse (FT-MPI) και το Los Alamos National Laboratory (LA-MPI). Άλλοι συνεργάτες είναι επίσης και τα Sandia National Laboratories και το High Computing Center της Στουτγκάρδης (PACX-MPI).

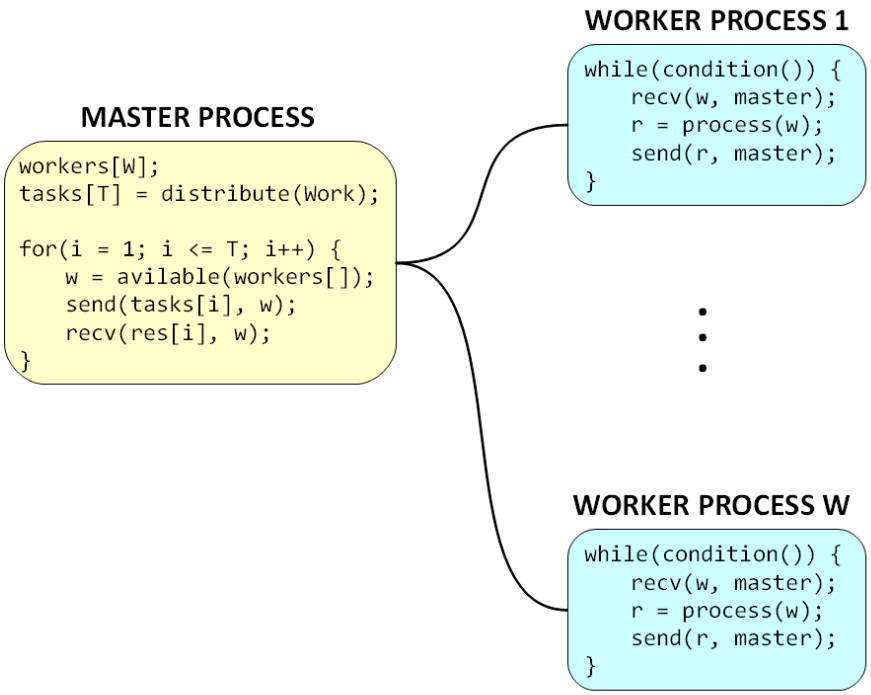
2.4. Το μοντέλο Master-Worker

Υπάρχουν μια σειρά από διαφορετικά μοντέλα προγραμματισμού για την ανάπτυξη μιας εφαρμογής που προσανατολίζεται σε μια κατανεμημένη παράλληλη αρχιτεκτονική [21]. Αυτό όμως που συγκεντρώνει το ενδιαφέρον στην υλοποίηση μας, είναι το μοντέλο Master-Worker, καθώς στην υλοποίηση μας υποστηρίζουμε εφαρμογές MPI που είναι γραμμένες αποκλειστικά σε αυτό το μοντέλο προγραμματισμού. Συνήθως μαθηματικές εφαρμογές βελτιστοποίησης αλλά και γενικότερα εφαρμογές οι οποίες είναι εύκολο να χωριστούν σε πολλές μικρότερες εργασίες, επιλέγουν να χρησιμοποιήσουν αυτό το μοντέλο. Το συγκεκριμένο μοντέλο είναι απλό στη γενική του ιδέα και προσανατολισμένο να διευκολύνει την προσαρμογή μιας στρατηγικής ανοχής σφαλμάτων.



Εικόνα 2.2: Το Μοντέλο Master-Worker

Σε μια εφαρμογή Master-Worker, δεν έχουν όλες οι διεργασίες που συμμετέχουν τις ίδιες αρμοδιότητες και δεν εκτελούν τον ίδιο κώδικα. Στη γενική μορφή του μοντέλου, υπάρχουν δύο διαφορετικές οντότητες διεργασιών. Η διεργασία Master και οι διεργασίες Worker, όπως φαίνεται και στην Εικόνα 2.2. Η διεργασία Master είναι αυτή που οργανώνει τη γενικότερη εκτέλεση και ροή της εφαρμογής, στέλνοντας εργασίες στις διεργασίες Worker. Από την άλλη, μια διεργασία Worker, αφού παραλάβει τη δουλειά, θα την επεξεργαστεί και θα στείλει τα εξαγόμενα αποτελέσματα πίσω στη διεργασία Master. Στην απλή μορφή του μοντέλου, η διεργασία Master μπορεί να επικοινωνήσει με κάθε διεργασία Worker, ενώ αντίθετα οι διεργασίες Worker δεν επικοινωνούν μεταξύ τους. Το παρακάτω σχήμα, αναπαριστά το μοντέλο Master-Worker.



Εικόνα 2.3: Παράδειγμα Τυπικής εφαρμογής Master-Worker

Η Εικόνα 2.3, παρουσιάζει τον ψευδοκώδικα μιας τυπικής εφαρμογής σε μοντέλο Master-Worker. Η διεργασία Master, είναι υπεύθυνη να διαχωρίσει τη συνολική δουλειά σε μικρότερες εργασίες, τις οποίες θα δρομολογήσει στις διαθέσιμες διεργασίες Worker. Συγκεντρώνει τα επί μέρους αποτελέσματα από κάθε Worker και σχηματίζει ένα νέο αποτέλεσμα. Αυτή η διαδικασία θα συνεχιστεί είτε στατικά μέχρι να τελειώσουν οι εργασίες που πρέπει να μοιραστούν και να εκτελεστούν, είτε δυναμικά όταν προσεγγιστεί το επιθυμητό τελικό αποτέλεσμα ή εκπληρωθεί μια συνθήκη. Από την άλλη, η διεργασία Worker είναι υπεύθυνη να επεξεργαστεί την εργασία και να στείλει το εξαγόμενο αποτέλεσμα πίσω στη διεργασία Master. Ένα πολύ απλό σχήμα στο οποίο υπάρχουν σημεία συγχρονισμού μεταξύ των διεργασιών μόνο στη αρχή και στο τέλος του κάθε υπολογισμού, όσο μακρόκοκκος ή μικρόκοκκος μπορεί να είναι αυτός, αλλά με τον περιορισμό οτι δεν είναι προσαρμόσιμο πλήρως σε όλα τα είδη των εφαρμογών.

ΚΕΦΑΛΑΙΟ 3. ΣΦΑΛΜΑΤΑ ΚΑΙ MPI

Στο κεφάλαιο αυτό θα ασχοληθούμε με την ιδιότητα της ανοχής σφαλμάτων και που αυτή αναφέρεται. Τι αναφέρει και τι προβλέπει το MPI για την περίπτωση σφάλματος. Γίνεται επίσης αναφορά στις πιο διαδεδομένες τεχνικές για την αντιμετώπιση του προβλήματος αλλά και στο μοντέλο προγραμματισμού Master-Worker και πως μπορεί να εφαρμοστεί σε αυτό κάποια τεχνική ανοχής σφαλμάτων. Πάνω στην εφαρμογή του μοντέλου αυτού σε εφαρμογές MPI βασίζεται και η εργασία.

3.1. Έννοιες Σφαλμάτων

Η εμφάνιση σφαλμάτων σε μια μικρή συστάδα υπολογιστών, δεν είναι ένα συχνό φαινόμενο. Η συνεχόμενη όμως αύξηση των μηχανημάτων, αυξάνει παράλληλα και την πιθανότητα εμφάνισης σφάλματος σε κάποιο από αυτά [27]. Η όσο το δυνατόν ανεπηρέαστη λειτουργία της συστάδας και πολύ περισσότερο η ανεπηρέαστη λειτουργία της εφαρμογής που εκτελείται σε αυτά, είναι ένα βασικό ζητούμενο.

Τα σφάλματα μπορούν να συμβούν σε διάφορα επίπεδα του συνολικού συστήματος. Σφάλμα μπορεί να εμφανιστεί σε κάποιο μηχάνημα της συστάδας, είτε με την διακοπή της σύνδεσης του από το υπόλοιπο σύστημα, είτε με την πτώση του λόγω

τροφοδοσίας. Σφάλμα επίσης μπορεί να εμφανιστεί και στο ίδιο το δίκτυο που συνδέει τα μηχανήματα της συστάδας. Εκτός από τα σφάλματα αυτά που αποτελούν σφάλματα στο υλικό του συστήματος, μπορούν να εμφανιστούν σφάλματα και στο λογισμικό. Παράδειγμα τέτοιων σφαλμάτων, είναι τα σφάλματα του λειτουργικού συστήματος που τρέχει στους κόμβους της συστάδας, αλλά και τα σφάλματα που μπορεί να προκληθούν από την εκτέλεση της ίδια της εφαρμογής. Επίσης τα σφάλματα είτε είναι μόνιμα στο σύστημα, είτε παροδικά.

3.1.1. Αντιμετώπιση σφαλμάτων

Αρκετές υλοποιήσεις συστημάτων επιχειρούν να αντιμετωπίσουν τις συνέπειες από την εμφάνιση σφαλμάτων στη λειτουργία της συστάδας. Βασική προϋπόθεση όμως για να μπορεί μια υλοποίηση να το επιτύχει, είναι η αξιόπιστη και έγκαιρη ανίχνευση του σφάλματος, από οποιοδήποτε επίπεδο του συστήματος και αν προέρχεται αυτό. Από πειραματισμό που κάναμε με υλοποιήσεις του MPI, δυστυχώς διαπιστώσαμε οτι παρουσιάζουν προβληματική συμπεριφορά στην ανίχνευση του λάθους. Μια δεύτερη προϋπόθεση είναι ο τρόπος με τον οποίο θα αναφέρεται στο επίπεδο της εφαρμογής του χρήστη. Η προσφερόμενη διεπαφή από την υλοποίηση, θα πρέπει να παρέχει τη δυνατότητα στο χρήστη να λαμβάνει επίγνωση του σφάλματος με κάποιο γεγονός στη διάρκεια εκτέλεσης και να είναι διακριτό επίσης, το ποιό επίπεδο αφορά το σφάλμα. Στην 3.1.2, γίνεται αναφορά για το πως το MPI, δίνει τη δυνατότητα στον προγραμματιστή της εφαρμογής να χειριστεί πιθανά σφάλματα.

Αφού εξασφαλιστούν αυτές οι προϋποθέσεις, ακολουθεί ο όρος της ανοχής σφαλμάτων. Με τον όρο αυτό εννοούμε την ικανότητα της υλοποίησης να αντιμετωπίζει τα σφάλματα που θα εμφανιστούν κατά τη διάρκεια της εκτέλεσης, αφήνοντας ανεπηρέαστη την εκτέλεση της εφαρμογής. Μια σειρά τενχικών και υλοποιήσεων έχουν προταθεί για να εξασφαλίσουν ανοχή σφαλμάτων. Η εργασία αυτή, προσανατολίζεται στο πως να προσφέρει ανοχή σφαλμάτων σε εφαρμογές MPI.

3.1.2. Χειρισμός σφαλμάτων στο MPI

Το MPI προσφέρει τη δυνατότητα χειρισμού των σφαλμάτων που είναι πιθανόν να εμφανιστούν κατά τη διάρκεια της εκτέλεσης, μέσω του ορισμού χειριστών λαθών (error handlers) οι οποίοι συνδέονται με τους communicators (βλέπε 2.2.1). Υπάρχουν δύο πρότυποι χειριστές λαθών στο πρότυπο, που ο προγραμματιστής μπορεί να ορίσει στην εφαρμογή του.

MPI_ERRORS_ARE_FATAL: Ο προκαθορισμένος χειριστής λαθών, ορίζει οτι στην περίπτωση ενός σφάλματος, όλες οι διεργασίες MPI (στον ίδιο communicator) τερματίζουν την εκτέλεσή τους. Μια προσέγγιση σαφώς πολύ αρνητική όταν έχουμε να κάνουμε με κρίσιμες εφαρμογές που εκτελούνται για μεγάλα χρονικά διαστήματα και των οποίων τα αποτελέσματα ή οι γενικότεροι πόροι που χρησιμοποιούν θέλουν μια επιπλέον μεταχείριση πριν τον τερματισμό τους.

MPI_ERRORS_RETURN: Στην περίπτωση εμφάνισης σφάλματος, επιστρέφεται η σχετική τιμή λάθους από τη συνάρτηση MPI, δίνοντας έτσι τον έλεγχο της εφαρμογής στον προγραμματιστή. Ο τελευταίος με τη σειρά του, είναι σε θέση λάβει τις απαραίτητες ενέργειες για τον ορθό και ομαλό τερματισμό της εφαρμογής.

Επίσης ένας προγραμματιστής μπορεί να ορίσει και τους δικούς του χειριστές λαθών που θα συνδέσει με τους MPI communicators της εφαρμογής του. Παραδείγματα και τρόποι χρήσης της τελευταίας δυνατότητας αναλύονται περισσότερο στο [10].

3.2. Ανοχή σφαλμάτων και MPI

Όπως αναφέρθηκε παραπάνω, σε καμία έκδοση του προτύπου δεν υπάρχει αναφορά στον όρο της ανοχής σφαλμάτων. Κατά συνέπεια, το πρότυπο δεν προσφέρει μια στρατηγική για μια ενιαία αντιμετώπιση του σφάλματος από τις διάφορες υλοποιήσεις. Αυτό έχει ως αποτέλεσμα να υπάρχει ένα χάσμα αξιοπιστίας στη χρήση του MPI για την ανάπτυξη σημαντικών εφαρμογών που εκτελούνται για μεγάλα χρονικά διαστήματα. Η επικείμενη έκδοση του προτύπου [17], που βρίσκεται ακόμα υπό συζήτηση, έχει ως άμεσο στόχο να καλύψει αυτό το κενό. Στο κεφάλαιο αυτό παρουσιάζονται μέθοδοι για την αξιοποίηση κάποιων λειτουργιών του MPI, για τη δημιουργία εφαρμογών ανεκτικών σε σφάλματα. Επίσης παρουσιάζονται τα

αποτελέσματα που προέκυψαν από πειραματισμό με συμβατικές υλοποιήσεις MPI, ώστε να διαπιστωθεί η συμπεριφορά που παρουσιάζουν, σε περίπτωση εμφάνισης σφαλμάτων. Και τέλος παρουσιάζονται και κάποιες εργασίες που προσφέρουν ανοχή σφαλμάτων σε προγράμματα MPI.

3.2.1. Η ιδιότητα της ανοχής σφαλμάτων

Η ανοχή σφαλμάτων είναι η ικανότητα ενός συστήματος να ανταποκρίνεται ορθά ή με μια λέξη, να "επιβιώνει" στην περίπτωση εμφάνισης ενός μη αναμενόμενου σφάλματος που μπορεί να προκύψει σε οποιοδήποτε επίπεδο του συστήματος, είτε μιλάμε για υλικό, είτε για λογισμικό, είτε για το δίκτυο.

Μεγάλη συζήτηση γίνεται για το ποιό επίπεδο αφορά η ανοχή σφαλμάτων σαν ιδιότητα. Δεν είναι θέμα της εφαρμογής του χρήστη, καθώς από μόνη της μπορεί να δομηθεί ώστε να μένει ανεπηρέαστη από σφάλματα, παρόλα αυτά χρειάζεται και την υποστήριξη ενός εξειδικευμένου κάτω επιπέδου, που θα προσφέρει και θα υποστηρίζει τις απαραίτητες λειτουργίες. Δεν είναι λοιπόν ιδιότητα του προτύπου MPI, καθώς το MPI περιγράφει μια διεπαφή για τη σωστή και εύκολη ανάπτυξη κατανεμημένων προγραμμάτων που θα εκτελούνται πάνω σε ένα αξιόπιστο υλικό [4]. Από την άλλη, δεν είναι ούτε ιδιότητα της υλοποίησης του MPI, καθώς δεν μπορεί το ίδιο να εξασφαλίσει οτι η εφαρμογή που τρέχει στο πάνω επίπεδο, είναι απρόσβλητη από σφάλματα. Καταλήγουμε λοιπόν στον ισχυρισμό οτι η ανοχή σφαλμάτων είναι μια ιδιότητα της εφαρμογής MPI σε συνδυασμό με την υλοποίηση MPI που υπάρχει από κάτω για να την υποστηρίζει να επιβιώσει από σφάλματα.

Η "επιβίωση" του συστήματος στο σφάλμα είναι μια ευρεία έννοια και περιλαμβάνει αρκετά διαφορετικά επίπεδα δράσης τα οποία μια υλοποίηση ή μια εφαρμογή μπορεί να υιοθετήσει για να υποστηρίξει ανεκτικότητα σε σφάλματα. Το πρώτο είναι οτι η υλοποίηση MPI αυτόματα αναρρώνει από σφάλματα και η εφαρμογή MPI συνεχίζει την εκτέλεση της ανεπηρέαστα. Ο χρήστης ενημερώνεται για την εμφάνιση του σφάλματος ή τουλάχιστον δεν καθίσταται αναγκαίο και άρα δε χρειάζεται να προβεί σε ενέργειες αντιμετώπισής του, καθώς την αποκατάσταση του σφάλματος την

αναλαμβάνει η υλοποίηση διαφανώς από το χρήστη. Το δεύτερο επίπεδο είναι οτι η υλοποίηση αναφέρει το σφάλμα στην εφαρμογή. Ο προγραμματιστής από εκεί και πέρα και με τη βοήθεια της υλοποίησης, η οποία του παρέχει σχετικές δυνατότητες, αναλαμβάνει να αποκαταστήσει το σφάλμα. Ένα άλλο επίπεδο είναι οτι συγκεκριμένες και όχι όλες οι λειτουργίες του MPI, ακυρώνονται. Για παράδειγμα, η υλοποίηση του MPI, μπορεί να επιλέξει να μην ακυρώσει συνολικά τον communicator και άρα τις επικοινωνίες με όλες τις διεργασίες που συμμετέχουν σε αυτόν, αλλά να καταστήσει δυνατή την επικοινωνία με τις υπόλοιπες που δεν έχουν εμφανίσει σφάλμα. Ένα τελευταίο επίπεδο είναι οτι η εφαρμογή διακόπτεται αλλά μπορεί να επανεκκινήσει από μια ενδιάμεση κατάσταση (checkpoint), στην οποία έχουν αποθηκευτεί οι απαραίτητες πληροφορίες. Τα checkpoints μπορούν να παρέχονται είτε από την ίδια την υλοποίηση διαφανώς από το χρήστη, είτε να τα υλοποιήσει ο χρήστης σε διάφορα σημεία της εφαρμογής του. Βέβαια και συνδυασμός των παραπάνω τεχνικών μπορεί να χρησιμοποιηθεί. Η παρούσα εργασία, αναφέρεται κυρίως στο δεύτερο επίπεδο, αλλά θα δούμε περισσότερα για αυτό στο 4ο κεφάλαιο.

3.2.2. Ανοχή σφαλμάτων στο μοντέλο Master-Worker

Στην 2.4, αναφέρθηκε και περιγράφτηκε το μοντέλο προγραμματισμού Master-Worker. Ένα μοντέλο που υιοθετείτε ευρέως για ανάπτυξη κατανεμημένων εφαρμογών, ακολουθώντας τη λογική της διάσπασης του αρχικού προβλήματος σε μικρότερα τα οποία, η διεργασία Master αναλαμβάνει να τα δρομολογήσει στις διεργασίες Worker. Αυτές με τη σειρά τους θα εκτελέσουν τους υπολογισμούς και θα στείλουν πίσω τα ενδιάμεσα αποτελέσματά τους.

Ένα σημαντικό χαρακτηριστικό του μοντέλου αυτού, είναι η ανεξαρτησία της εκτέλεσης που εξασφαλίζει ανάμεσα στις διεργασίες Worker, πράγμα που διευκολύνει σε μεγάλο βαθμό τη χάραξη μιας στρατηγικής ανοχής σφαλμάτων. Η δουλειά που εκτελεί ένας Worker είναι ανεξάρτητη από τη δουλειά που εκτελεί ένας άλλος, με αποτέλεσμα το σφάλμα του ενός να μην επηρεάζει επί της ουσίας την εκτέλεση του άλλου. Από την άλλη, η διεργασία Master είναι σε θέση να

αποκαταστήσει εύκολα το σφάλμα που θα συμβεί σε κάποιον Worker. Αυτό μπορεί να γίνει είτε δρομολογώντας την εργασία που του ανέθεσε και από την οποία δεν πήρε αποτελέσματα, σε κάποιον άλλον διαθέσιμο Worker, είτε επανεκκινώντας την πεσμένη διεργασία και αναθέτοντάς της εκ νέου την ίδια εργασία.

Εύκολα μπορούν να εφαρμοστούν και άλλες μέθοδοι που μπορούν να εξασφαλίσουν ανοχή σφαλμάτων στο συγκεκριμένο μοντέλο. Η διεργασία Master, η οποία επι της ουσίας είναι και αυτή που οργανώνει τη συνολική ροή της εκτέλεσης, μπορεί να δρομολογεί την ίδια εργασία σε περισσότερες από μια διεργασία Worker. Σε περίπτωση εμφάνισης σφάλματος της μιας εξ αυτών, τα αποτελέσματα της εργασίας μπορούν να είναι διαθέσιμα από άλλη διεργασία Worker, η οποία εκτελεί την ίδια εργασία. Ίδια τεχνική μπορεί να ακολουθηθεί και στην πλευρά της διεργασίας Master, για την εξασφάλιση της ομαλής ροής εκτέλεσης σε περίπτωση σφάλματος. Πολλές διεργασίες Master, που εκτελούνται παράλληλα αλλά πιθανότατα διαφανώς από το χρήστη, θα είναι σε θέση να λάβουν αυτές το ρόλο της βασικής διεργασίας Master σε περίπτωση που εμφανίσει η τελευταία σφάλμα και τερματίσει. Έτσι η ροή της εκτέλεσης της εφαρμογής μπορεί να κρατηθεί ανεπηρέαστη. Τη συγκεκριμένη λογική ακολουθούμε και εμείς στην υλοποίηση μας, δίνοντας τη δυνατότητα στο χρήστη να ορίσει κατά βούληση το αν και πόσες επιπλέον διεργασίες Master θα συμμετέχουν στην εκτέλεση της εφαρμογής (βλέπε 4.5).

3.2.3. Χρήση MPI Intercommunicators

Οι William Gropp και Ewing Lusk [11], περιγράφουν τεχνικές που μπορεί να χρησιμοποιήσει κάποιος προγραμματιστής στο επίπεδο της εφαρμογής, ώστε να μετατρέψει το πρόγραμμα του σε μια εφαρμογή ανεκτική σε σφάλματα που μπορεί να συμβούν.

Όπως προαναφέρθηκε ο communicator είναι ένα θεμελιώδες κατανεμημένο αντικείμενο μεταξύ κάποιων διεργασιών MPI που μαζί αποτελούν ομάδα, μέσω του οποίου μπορούν να επιτελεστούν είτε οι επικοινωνίες μεταξύ δύο διεργασιών, είτε οι συλλογικές επικοινωνίες. Παρόλα αυτά, το σφάλμα και κατά συνέπεια ο τερματισμός

μιας διεργασίας που συμμετέχει στον communicator προκαλεί και τον τερματισμό των υπολοίπων διεργασιών που συμμετέχουν σε αυτόν. Οπότε η χρήση ενός communicator στο πρόγραμμα MPI, το καθιστά ιδιαίτερα ευάλωτο σε εμφάνιση σφάλματος.

```
#include "mpi.h"

#define MAX_WORKERS 1000
#define IC_CREATE_TAG 100

int main(int argc, char *argv[]) {
     int i, myrank, origsize, currsize;
     MPI_Comm workerComm[MAX_WORKERS];
     MPI_Init(&argc, &argv);
     MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
     MPI_Comm_size(MPI_COMM_WORLD, &origsize);
     currsize = origsize;

     /* create intercommunicators and set error handlers */
     for(i = 1; i < currsize; i++) {
          MPI_Intercomm_create(MPI_COMM_SELF, 0,
                               MPI_COMM_WORLD, i,
                               IC_CREATE_TAG, &workerComm[i-1]);
          MPI_Comm_set_errhandler(workerComm[i-1],MPI_ERRORS_RETURN);
     }
     /* set up three lists of task descriptors:
     *  not done
     *  in progress
     *  done */
     /* manager part of manager worker algorithm:
     *  when send or receive fails, mark intercommunicator as dead,
     *  keep task in in-progress list, send to next free worker */
     /* work until both not done and in progress lists are empty */

     for(i = 1; i < currsize; i++)
          MPI_Comm_free(&worker_comm[i-1]);

     MPI_Finalize();

     return 1;
}
```

Εικόνα 3.1: Παράδειγμα Ορισμού και Χρήσης Intercommunicators

Εκτός όμως από τους communicators που προαναφέρθηκαν, οι οποίοι αποκαλούνται και intracommunicators, το πρότυπο δίνει τη δυνατότητα ορισμού και intercommunicators. Ο intercommunicator συσχετίζει δύο ομάδες διεργασιών που

ανήκουν σε διαφορετικούς communicators. Κάθε διεργασία από τη μια ομάδα μπορεί να επικοινωνήσει με κάθε μια από την άλλη μέσω του intercommunicator και στην περίπτωση που εμφανιστεί σφάλμα, μόνο οι διεργασίες που βρίσκονται στον ίδιο communicator θα τερματιστούν. Για να προχωρήσει ακόμα ένα βήμα παραπέρα ο συλλογισμός, θα μπορούσε ο προγραμματιστής να αναπτύξει το πρόγραμμα MPI του σε μορφή Master-Worker (βλέπε 2.4). Σε μια τέτοια περίπτωση μπορεί να οριστεί και να συνδεθεί ένας intercommunicator ανάμεσα σε κάθε ζευγάρι Master-Worker. Σε περίπτωση σφάλματος ενός Worker, ακυρώνεται ο intercommunicator, ο Master χάνει την επικοινωνία μαζί του, αλλά το σφάλμα δε διαδίδεται σε περισσότερες διεργασίες MPI και άρα δεν επηρεάζει τη συνολική εκτέλεση της εφαρμογής MPI. Τρόποι χρήσης και γενικότερης αξιοποίησης των intercommunicators, σε εφαρμογές MPI σύμφωνα με την παραπάνω λογική, παρουσιάζονται αναλυτικότερα εδώ [13].

Στην Εικόνα 3.1, μπορούμε να δούμε πως μπορεί να επιτευχθεί με κώδικα στο επίπεδο της εφαρμογής MPI, η παραπάνω στρατηγική χρήσης των intercommunicators από την πλευρά του Master. Αρχικά δημιουργούνται οι intercommunicators, ένας για κάθε διεργασία Worker και se κάθε ένα από αυτούς, ορίζεται ως χειριστής λαθών ο MPI_ERRORS_RETURN ώστε σε περίπτωση εμφάνισης σφάλματος ο έλεγχος να δοθεί πίσω στην εφαρμογή και να μην τερματιστεί. Στο τέλος χρησιμοποιείται η απαραίτητη κλήση MPI για την αποδέσμευση των πόρων. Όσον αφορά τους Workers δε χρειάζεται κάποια ιδιαίτερη στρατηγική, καθώς ο Master έχει την ευθύνη για το αν θα επανεκκινήσει την πεσμένη διεργασία Worker ή θα συνεχίσει με λιγότερους την υπόλοιπη εκτέλεση του προγράμματος.

3.2.4. Πειραματισμός με υλοποιήσεις MPI

Όσα αναφέρθηκαν παραπάνω, για το πως θα μπορούσε ο προγραμματιστής της εφαρμογής να οργανώσει και να αναπτύξει την εφαρμογή MPI, έτσι ώστε να είναι ανεκτική σε σφάλματα απαιτεί ορθή υποδομή MPI. Η υλοποίηση MPI, που υπάρχει κάτω από το επίπεδο της εφαρμογής MPI, θα πρέπει να είναι σε θέση πρώτον να

ανιχνεύσει και δεύτερον να αναφέρει την εμφάνιση του σφάλματος στο επίπεδο της εφαρμογής, μέσω ενός των συναρτήσεων της υλοποίησης.

Σκοπός μας είναι να ελέγξουμε κατά πόσο αυτό συμβαίνει όντως στην πράξη και για αυτό πειραματιστήκαμε με δύο από τις πιο διαδεδομένες υλοποιήσεις του MPI, το MPICH και το OpenMPI. Και οι δύο υλοποιήσεις, αποτυγχάνουν να υποστηρίξουν τις παραπάνω προϋποθέσεις. Αδυνατούν να μεταδώσουν το σφάλμα στο επίπεδο της εφαρμογής ορθά και πολύ περισσότερο δεν παρουσιάζουν μια συνεπή συμπεριφορά κατά την εκτέλεση της εφαρμογής σε περίπτωση λάθους.

Τα παραπάνω συμπεράσματα τα εξάγαμε με τη βοήθεια κάποιων δοκιμαστικών προγραμμάτων MPI, τα οποία εκτελέσαμε και με τις δύο υλοποιήσεις. Τα προγράμματα είχαν το σχήμα μιας τυπικής εφαρμογής Master-Worker, με τον τρόπο που παρουσιάστηκε στην Το μοντέλο Master-Worker2.4. Σκοπός ήταν να διαπιστώσουμε τη συμπεριφορά των υλοποιήσεων, περισσότερο στο σημείο της ανίχνευσης και της αναφοράς του σφάλματος. Ελέγξαμε μόνο τις περιπτώσεις σφάλματος στη διεργασία Worker, τερματίζοντας σε συγκεκριμένο σημείο της εκτέλεσης κάποια από αυτές. Τα προγράμματα αναφερόντουσαν τόσο στην κλήση MPI_Send, όσο και στη κλήση MPI_Recv. Ο κώδικας των προγραμμάτων αυτών για μια διεργασία Worker, φαίνεται στην Εικόνα 3.2. Η διεργασία Master, στέλνει/λαμβάνει δεδομένα στη διεργασία Worker η οποία έχει ήδη τερματίσει.

```
int main(int argc, char **argv) {
    int temp;
    ...
    MPI_Init(&argc, &argv);
    ...
    if(main_process()) {
        err_ret = MPI_Recv(&temp, 1, MPI_INT, 0, 0, interCom, MPI_STATUS_IGNORE);
        err_ret = MPI_Send(&temp, 1, MPI_INT, 0, 0, interCom);
        if(err_ret != MPI_SUCCESS) {
            fprintf(stdout, "Proc%d: Error Reported!\n", getpid());
        }
    }
    else {
        exit(1);
        MPI_Send(&temp, 1, MPI_INT, 0, 0, interCom);
        MPI_Recv(&temp, 1, MPI_INT, 0, 0, interCom, MPI_STATUS_IGNORE);
    }

    MPI_Finalize();
    return 0;
}
```

Test Recv

Test Send

Εικόνα 3.2: Κώδικας για Έλεγχο Αναφοράς με μια Διεργασία

Και οι δύο υλοποιήσεις αποτύγχαναν να αναφέρουν το σφάλμα στην εφαρμογή, τόσο με την κλήση MPI_Send όσο και με την κλήση MPI_Recv. Όσο για την κατάσταση της εκτέλεσης της εφαρμογής, αυτή για το MPICH2 εκτελούνταν επ άπειρον ενώ αντίθετα για το OpenMPI τερματιζόταν συνολικά. Τα αποτελέσματα φαίνονται στον Πίνακας 3.1.

Πίνακας 3.1: Αναφορά Εκτελέσεων MPICH2 και OpenMPI

Υλοποίηση	**Αναφορά**		**Κατάσταση**	
	Send	Recv	Send	Recv
MPICH2	Όχι	Όχι	Ατέρμονη εκτέλεση	Ατέρμονη εκτέλεση
OpenMPI	Όχι	Όχι	Τερματισμός	Τερματισμός

Εκτός από τη συμπεριφορά της επικοινωνίας μεταξύ Master και Worker σε περίπτωση σφάλματος στη δεύτερη, ένα επιπλέον χαρακτηριστικό που θέλαμε να

ελέγξουμε, είναι και η συμπεριφορά της επικοινωνίας μεταξύ διεργασίας Master και μιας διεργασίας Worker, η οποία ανήκει στον ίδιο communicator με μια διεργασία που έχει εμφανίσει σφάλμα. Με αυτό το πείραμα θέλαμε να διαπιστώσουμε τι προκαλείται στον communicator και πόσο μπορεί αυτό να επηρεάσει τις μη εσφαλμένες διεργασίες MPI. Ο κώδικας για το συγκεκριμένο πείραμα φαίνεται στην Εικόνα 3.3. Η διεργασία Master, στέλνει/λαμβάνει δεδομένα στη μη εσφαλμένη διεργασία του communicator, ενώ αντίθετα η έτερη διεργασία τερματίζει.

Τα αποτελέσματα, όπως φαίνονται και στους Πίνακες Πίνακας 3.2 και Πίνακας 3.3, δείχνουν και εδώ οτι οι υλοποιήσεις σε αρκετές περιπτώσεις δεν αναφέρουν το σφάλμα στην εφαρμογή και γενικότερα δεν παρουσιάζουν μια συνεπή συμπεριφορά. Σε κάποιες περιπτώσεις η εφαρμογή τερματίζει και σε άλλες τρέχει ατέρμονα. Το OpenMPI ειδικότερα, σε περίπτωση σφάλματος αποτυγχάνει και τερματίζει συνολικά την εφαρμογή, μια συμπεριφορά ιδιαίτερα ανεπιθύμητη, ειδικότερα όταν έχουμε να κάνουμε με κρίσιμες εφαρμογές που εκτελούντε για μεγάλα χρονικά διαστήματα.

```
int main(int argc, char **argv) {
    int temp;
    ...
    MPI_Init(&argc, &argv);
    ...
    if(main_process()) {
        err_ret = MPI_Recv(&temp, 1, MPI_INT, 0, 0, interCom, MPI_STATUS_IGNORE);
        err_ret = MPI_Send(&temp, 1, MPI_INT, 0, 0, interCom);
        if(err_ret != MPI_SUCCESS) {
            fprintf(stdout, "Proc%d: Error Reported!\n", getpid());
        }
    }
    else {
        if(rank == 0) {
            MPI_Send(&temp, 1, MPI_INT, 0, 0, interCom);
            MPI_Recv(&temp, 1, MPI_INT, 0, 0, interCom, MPI_STATUS_IGNORE);
        }
        else if(rank == 1) {
            exit(1);
        }
    }

    MPI_Finalize();
    return 0;
}
```

Test Recv

Test Send

Εικόνα 3.3: Κώδικας για Έλεγχο Αναφοράς με δύο Διεργασίες

Κατά συνέπεια και οι δύο υλοποιήσεις δεν προσφέρουν την κατάλληλη ανίχνευση και αναφορά σφαλμάτων στο επίπεδο της εφαρμογής. Πόσο μάλλον και η συμπεριφορά της εκτέλεσης του, δεν είναι συνεπής. Όπως αναφέρθηκε παραπάνω, απαραίτητη προϋπόθεση για να σχεδιάσεις και να αξιοποιήσεις λειτουργίες με προσανατολισμό την ανοχή σφαλμάτων, είναι η σωστή και αξιόπιστη ανίχνευση των σφαλμάτων. Στην υλοποίησή μας λοιπόν, για να υπερκεράσουμε αυτήν την παθογένεια, υλοποιήσαμε εκ νέου δικές μας εκδόσεις για τις βασικές επικοινωνίες του MPI, χρησιμοποιώντας το διαδικτυακό πρωτόκολλο TCP. Το τελευταίο μας δίνει τη δυνατότητα για έγκαιρη και αξιόπιστη ανίχνευση σφαλμάτων που μπορούν να εμφανιστούν σε μια διεργασία. Περισσότερα για αυτά στο ΚΕΦΑΛΑΙΟ 4.

Πίνακας 3.2: Αναφορά Εκτελέσεων για MPICH2 με δύο διεργασίες

Αναφερόμενη διεργασία	**Αναφορά**		**Κατάσταση**	
	Send	Recv	Send	Recv
Εσφαλμένη	Ναι	Όχι	OK!	Ατέρμονη εκτέλεση
Μη Εσφαλμένη	Ναι	Όχι	Μη επικοινωνία	Μη επικοινωνία

Πίνακας 3.3: Αναφορά Εκτελέσεων για OpenMPI με δύο διεργασίες

Αναφερόμενη διεργασία	**Αναφορά**		**Κατάσταση**	
	Send	Recv	Send	Recv
Εσφαλμένη	Όχι	Όχι	Τερματισμός εκτέλεσης	Τερματισμός εκτέλεσης
Μη Εσφαλμένη	Όχι	Όχι	Τερματισμός εκτέλεσης	Τερματισμός εκτέλεσης

3.2.5. Σχετικές εργασίες

Στη βιβλιογραφία, έχουν προταθεί αρκετές τεχνικές για την υποστήριξη της ιδιότητας ανοχής και αντιμετώπισης σφαλμάτων στο MPI. Μια πρώτη κατηγοριοποίηση αυτών των τεχνικών μπορεί να γίνει βάση του αν η διαδικασία ανοχής σφαλμάτων λαμβάνει χώρα διαφανώς από τον προγραμματιστή της εφαρμογής, χωρίς δηλαδή αυτός να χρειαστεί να γράψει κώδικα επιπλέον για αυτόν τον σκοπό ή αδιαφανώς, όπου θα πρέπει εκτός από τον κώδικα της εφαρμογής να λάβει και επιπλέον αποφάσεις. Γενικότερη κουβέντα πάνω σε αυτό το θέμα μπορεί κάποιος να βρει εδώ [11].

Η πιο κοινή λύση για την υποστήριξη ανοχής σφαλμάτων, είναι το checkpointing το οποίο τις περισσότερες φορές ακολουθείται και από έναν αλγόριθμο συνολικής επανεκκίνησης της εφαρμογής MPI. Συχνά μαζί με το checkpointing συνδυάζεται και η καταγραφή των επικοινωνιών. Μια από τις πρώτες υλοποιήσεις που χρησιμοποίησε αυτόν τον συνδυασμό για να υποστηρίξει την ανοχή σφαλμάτων, ήταν το CoCheck [23] το οποίο ουσιαστικά τοποθετείται πάνω από το επίπεδο της βιβλιοθήκης MPI. Επίσης το MPICH-V [1], που είναι η ανεκτική σε σφάλματα έκδοση της υλοποίησης MPICH, αποτελεί μια από τις πιο ολοκληρωμένες υλοποιήσεις για αυτόν το σκοπό. Στην ίδια κατεύθυνση βρίσκονται και οι υλοποιήσεις Egida [20] και MPI-FT [16]. Εκτός από το MPICH και το Open MPI διαθέτει έκδοση ανεκτική σε σφάλματα [15], [14] συνδυάζοντας τεχνικές για checkpoining, καταγραφη επικοινωνιών και τεχνικές που προσδίδουν μεγαλύτερη αξιοπιστία στο δίκτυο καθώς επίσης δίνει και στο χρήστη δυνατότητες διαχείρισης σφαλμάτων.

Παρόλα αυτά το checkpointing σαν τεχνική εισάγει σημαντικό επιπλέον φόρτο στη συνολική εκτέλεση της εφαρμογής καθώς και στο συνολικό χώρο αποθήκευσης. Υπάρχουν υλοποιήσεις που επιλέγουν άλλες τεχνικές για τον ίδιο σκοπό. Το FT-MPI [4], τροποποιεί τον ορισμό και την υλοποίηση των communicators του MPI επεκτείνοντας το πρότυπο ώστε να είναι δυνατή η αποφυγή τερματισμού της εφαρμογής σε περίπτωση λάθους. Δίνει τη δυνατότητα στον προγραμματιστή να επιλέξει μια σειρά από σχήματα διόρθωσης των σφαλμάτων και ουσιαστικά αποφεύγει να επανεκκινήσει τη συνολική εφαρμογή σε περίπτωση σφάλματος. Το LA-MPI [8], από την άλλη χρησιμοποιεί επιπλέον τεχνικές στο επίπεδο δικτύου, για

να εισάγει μεγαλύτερη αξιοπιστία στη μεταφορά των μηνυμάτων και στο συνολικό δίκτυο.

Στην υλοποίηση μας, υποστηρίζουμε μόνο εφαρμογές MPI που είναι ανεπτυγμένες με σχήμα Master-Worker και δίνουμε στον προγραμματιστή τη δυνατότητα να διαλέξει αν θέλει να επανεκκινήσει την πεσμένη διεργασία ή θα συνεχίσει την εκτέλεση της εφαρμογής με λιγότερους Worker. Είναι ένα ενδιάμεσο επίπεδο που τοποθετείται ανάμεσα στην υλοποίηση MPI και στην εφαρμογή MPI. Παρόμοια προσέγγιση υπάρχει και στο [3]. Για τα σφάλματα στη διεργασία Master, επιλέξαμε να έχουμε πολλαπλές διεργασίες Master. Διεργασίες MPI που ουσιαστικά τρέχουν τον ίδιο κώδικα αδιαφανώς από τον προγραμματιστή και σε περίπτωση πτώσης της μιας αναλαμβάνει το ρόλο της η επόμενη στη σειρά. Μελέτη για πολλαπλές διεργασίες MPI που εκτελούνται παράλληλα μπορεί κάποιος να βρει στο [2].

3.3. Τεχνικές Αντιμετώπισης

Εκτός από τη χάραξη μιας στρατηγικής ανάπτυξης της εφαρμογής ώστε να είναι ανεκτική σε σφάλματα, υπάρχουν μια σειρά μεθόδων και τεχνικών που υιοθετούν οι διάφορες υλοποιήσεις για να εξασφαλίσουν μια κάποια αξιοπιστία σε περίπτωση εμφάνισης σφάλματος. Αυτές οι μέθοδοι είτε είναι τελείως διαφανείς στο επίπεδο του χρήστη, χωρίς δηλαδή αυτός να χρειάζεται να επέμβει παραπάνω στον κώδικα της εφαρμογής του για να εξασφαλίσει ανοχή σφαλμάτων, είτε αδιαφανείς όπου επιβαρύνεται με το φόρτο να εισάγει επιπλέον κώδικα στην εφαρμογή του. Παράδειγμα της τελευταίας μεθόδου είναι η προαναφερθείσα τεχνική με χρήση των intercommunicators (βλέπε 3.2.3). Στο κεφάλαιο αυτό περιγράφονται μια σειρά τέτοιου τύπου τεχνικών και για περισσότερες λεπτομέρειες μπορεί κάποιος να ανατρέξει εδώ [27].

3.3.1. Checkpointing

Η πιο συνηθισμένη τεχνική που υιοθετούν οι περισσότερες υλοποιήσεις για να υποστηρίξουν ανοχή σφαλμάτων είναι η διατήρηση checkpoints κατά τη διάρκεια εκτέλεσης του κώδικα. Το checkpointing είναι η δυνατότητα που έχει η υλοποίηση να αποθηκεύει καταστάσεις των δομών της εφαρμογής, ουσιαστικά ενδιάμεσα δεδομένα που παράγονται κατά την εκτέλεση. Στη συνέχεια με τη βοήθεια αυτών των καταστάσεων, καθίσταται δυνατό να επανεκκινήσει η εφαρμογή σε περίπτωση που τερματιστεί ξαφνικά λόγω σφάλματος από το σημείο που πάρθηκε το τελευταίο checkpoint. Συνήθως οι υλοποιήσεις που επιλέγουν τη μέθοδο των checkpoints ([1], [14], [15], [16], [20], [23]), χρησιμοποιούν επιπλέον και έναν αλγόριθμο επανεκκίνησης της εφαρμογής από το τελευταίο checkpoint.

Τα checkpoints μπορούν να αποθηκεύονται είτε διαφανώς από το χρήστη, είτε κατά εντολή του στα σημεία του κώδικα που αυτός θα αποφασίσει. Επίσης ο προγραμματιστής μπορεί να αποθηκεύει με δική του πρωτοβουλία τις απαραίτητες δομές δεδομένων και δεδομένα γενικότερα, σε αρχείο όταν αυτός θεωρεί σκόπιμο και αναγκαίο χωρίς να εξαρτάται από λειτουργίες της υλοποίησης. Όταν η εφαρμογή επανεκκινήσει, αρκεί να διαβάσει τα δεδομένα από το τελευταίο checkpoint, είτε αυτό έχει αποθηκευτεί από την υλοποίηση, είτε βρίσκεται σε αρχείο και να συνεχίσει την εκτέλεση. Για την επίτευξη διαφανής μεθόδου checkpoint προς τον προγραμματιστή της εφαρμογής τα πράγματα από την πλευρά της υλοποίησης είναι πιο δύσκολα και περίπλοκα. Σε αυτήν την περίπτωση, η υλοποίηση θα πρέπει να αναλύσει τα δεδομένα που χρειάζεται ο χρήστης ώστε να έχει επίγνωση για το ποιά, το πότε και το πως θα τα αποθηκεύσει. Εκτός όμως από το θέμα της αναγκαιότητας για αυτόματη αποθήκευση των δομών δεδομένων που πρέπει να διαθέτει η υλοποίηση, σημαντική παράμετρος στην αδιαφανή μέθοδο checkpoint είναι η αυτόματη φόρτωση των απαραίτητων δεδομένων σε περίπτωση επανεκκίνησης της εφαρμογής.

Ένα άλλο ζήτημα που εγείρεται με τη λογική των checkpoints, είναι και το πόσες αλλά και το ποιές διεργασίες από αυτές που συμμετέχουν συνολικά στους υπολογισμούς, θα είναι υπεύθυνες για την αποθήκευση τους. Μια προσέγγιση είναι

οτι κάθε μια διεργασία, αποφασίζει για το ποιά δεδομένα θα αποθηκεύσει και το πότε. Αυτή η προσέγγιση αν και είναι απλή και παρουσιάζει μια κάποια κλιμακωσιμότητα, παρόλα αυτά υστερεί στο να δώσει μια συνεπή καθολική κατάσταση. Επίσης χρήζει μεγάλης αναγκαιότητας συνολικού αποθηκευτικού χώρου αλλά και εξειδικευμένων αλγορίθμων επανεκκίνησης της εφαρμογής. Μια δεύτερη προσέγγιση είναι οτι όλες οι διεργασίες συνεργάζονται για τη δημιουργία και αποθήκευση ενός καθολικού checkpoint. Τα σημεία συγχρονισμού μεταξύ των διεργασιών όμως που χρειάζονται για αυτήν την προσέγγιση εισάγουν μεγάλη καθυστέρηση στην εφαρμογή. Μια τελευταία προσέγγιση, είναι υβριδική και συνδυάζει τις δύο παραπάνω. Κάθε διεργασία μπορεί κρατάει ανεξάρτητα όποια δεδομένα και όποτε θέλει, αλλά είναι υποχρεωμένη να κρατάει και επιπλέον checkpoints, που εξαρτάται από πληροφορία που εισάγεται στα μηνύματα της εφαρμογής.

Παρόλα όσα αναφέρθηκαν παραπάνω, το checkpointing είναι μια τεχνική που επιβαρύνει πολύ την απόδοση της εφαρμογής ανεξάρτητα με το αν εμφανιστεί ή όχι, ένα σφάλμα κατά τη διάρκεια της εκτέλεσης. Η παρούσα εργασία, δεν υιοθετεί τη μέθοδο του checkpointing για να εξασφαλίσει ανοχή σφαλμάτων. Βέβαια ο προγραμματιστής της εφαρμογής μπορεί να επιλέξει να αποθηκεύει τα απαραίτητα δεδομένα σε ένα αρχείο μέσα στον κώδικα της εφαρμογής του. Η υλοποίηση της εργασίας, προσανατολίζεται στο να προσφέρει το υπόβαθρο και τις λειτουργίες ώστε να διατηρηθεί η σωστή και η ανεπηρέαστη εκτέλεση της εφαρμογής και όχι στο πως θα δημιουργηθούν οι απαραίτητες προϋποθέσεις ώστε αυτή να επανεκκινήσει σε περίπτωση εμφάνισης ενός σφάλματος.

3.3.2. Message Logging

Μια άλλη τεχνική των υλοποιήσεων, είναι και η καταγραφή των μηνυμάτων που έχουν ανταλλαγεί κατά τη διάρκεια της εκτέλεσης. Αυτή η τεχνική συνήθως χρησιμοποιείτε σε συνδυασμό με τη μέθοδο του checkpointing, για να χαλαρώσει λίγο την ποσότητα των δεδομένων που χρειάζεται να αποθηκευτούν και να μειώσει την επιβάρυνση του στη συνολική απόδοση. Με λίγα λόγια, αποθηκεύονται λιγότερα

checkpoints, αλλά στο ενδιάμεσο καταγράφονται τα μηνύματα που έχουν μεταδοθεί μεταξύ των διεργασιών.

Και σε αυτήν την τεχνική υπάρχουν διάφορες προσεγγίσεις. Η απαισιόδοξη προσέγγιση, εξασφαλίζει οτι όλες οι καταστάσεις που έχουν αποθηκευτεί μπορούν να προσφέρουν τη δυνατότητα επανεκκίνησης, αλλά εισάγει το επιπλέον κόστος οτι οι επικοινωνίες θα είναι εμποδιστικές (blocking). Αντίθετα, η αισιόδοξη προσέγγιση δεν απαιτεί εμποδιστικές επικοινωνίες αλλά δεν εξασφαλίζει οτι σημαντικά δεδομένα θα έχουν αποθηκευτεί, πριν την εμφάνιση κάποιου σφάλματος. Κατά συνέπεια δυσκολεύει και η διαδικασία ανάκτησης της ορθότητας της εφαρμογής ή καλύτερα η διαδικασία που χρειάζεται για την επανεκκίνησή της. Και εδώ υπάρχουν υβριδικές υλοποιήσεις της μεθόδου, που όπου ακολουθούν την απαισιόδοξη προσέγγιση και όπου είναι δυνατό, αφαιρούν την αναγκαιότητα για εμποδιστική επικοινωνία.

Στην παρούσα εργασία, δίνεται η δυνατότητα στο χρήστη να επιλέξει ποιά μηνύματα θέλει να καταγραφούν από την υλοποίηση, ώστε να είναι διαθέσιμη η πληροφορία σε περίπτωση σφάλματος. Περισσότερα όμως για το πως χρησιμοποιούμε αυτήν τη μέθοδο στην εργασία, στο επόμενο κεφάλαιο.

3.3.3. Replication

Μια τρίτη τεχνική για την ανοχή σε σφάλματα, είναι και το replication. Σε αυτήν τη μέθοδο, δύο ή περισσότερες διεργασίες που συμμετέχουν στους υπολογισμούς, εκτελούν την ίδια εργασία [3]. Εφαρμοσμένη αυτή η μέθοδος στο μοντέλο Master-Worker, οι διεργασίες εκτελούν την ίδια δουλειά πάνω στα ίδια δεδομένα και στο πέρας των υπολογισμών τους, στέλνουν τα αποτελέσματά τους στη διεργασία Master. Τα αποτελέσματά τους, ουσιαστικά είναι τα ίδια. Σε περίπτωση που μια διεργασία από αυτήν την ομάδα, παρουσιάσει σφάλμα και τερματίσει τη λειτουργία του, τότε τα αποτελέσματα είναι διαθέσιμα στη διεργασία Master, από τις υπόλοιπες διεργασίες που εκτελούν την ίδια δουλειά. Η διεργασία Master, μπορεί να συνεχίσει την εκτέλεση των υπολογισμών και τον διαμοιρασμό της δουλειάς χωρίς την πεσμένη διεργασία.

Οι διεργασίες που εκτελούν την ίδια δουλειά, θα ήταν δόκιμο να εκτελούνται σε διαφορετικούς κόμβους. Ο λόγος είναι οτι σε περίπτωση που το σφάλμα είναι μόνιμο, για παράδειγμα υπάρχει τεχνικό πρόβλημα στον κόμβο, δεν μπορεί να επανεκκινήσει σε αυτόν καμία διεργασία. Επεκτείνοντας τη λογική, οι κόμβοι θα πρέπει να τροφοδοτούνται και από διαφορετική πηγή ενέργειας και από διαφορετικό δρομολογητή δικτύου.

Η τεχνική αυτή όμως εισάγει και επιπλέον κόστη και πόρους. Η αναγκαιότητα περισσότερων υπολογιστών που συμμετέχουν στη συστάδα και στους υπολογισμούς και κατά συνέπεια αναγκαιότητα για μεγαλύτερο συνολικό αποθηκευτικό χώρο. Σε κάποιες περιπτώσεις επίσης, προσθέτει και μεγαλύτερο φόρτο στον προγραμματιστή για την ανάπτυξη της εφαρμογής του, σε περίπτωση που ζητηθεί από αυτόν να ορίσει τον αριθμό αλλά και την τοπολογία των επιπλέον μηχανημάτων ή διεργασιών.

Στην παρούσα εργασία, η συγκεκριμένη τεχνική του replication, χρησιμοποιείται στην πλευρά της διεργασίας Master. Δίνεται η δυνατότητα στον προγραμματιστή να ορίσει έναν αριθμό διεργασιών Master για τις περιπτώσεις που εμφανιστεί κάποιο σφάλματα στη διεργασία Master. Περισσότερα για το πως χρησιμοποιούμε αυτήν την τεχνική στο επόμενο κεφάλαιο.

ΚΕΦΑΛΑΙΟ 4. Ο ΜΗΧΑΝΙΣΜΟΣ ΤΟΥ FTMW

Στο κεφάλαιο αυτό γίνεται αναλυτική παρουσίαση της σχεδίασης και της υλοποίησης του προτεινόμενου μηχανισμού ανοχής σφαλμάτων. Πρόκειται για ένα ενδιάμεσο επίπεδο μεταξύ του επιπέδου εφαρμογής MPI και του επιπέδου υλοποίησης MPI για εφαρμογές ανεπτυγμένες με σχήμα Master-Worker. Αναλύεται η συμπεριφορά του μηχανισμού σε περίπτωση εμφάνισης σφάλματος, είτε στην περίπτωση που αυτό εμφανίζεται σε μια διεργασία Worker, είτε αυτό εμφανίζεται στη διεργασία Master. Επίσης γίνεται εκτενής περιγραφή για τη διεπαφή και τις λειτουργίες που προσφέρει.

4.1. Ανίχνευση και Χειρισμός Σφαλμάτων

Όπως είδαμε στην Ενότητα 3.2.4, διαπιστώσαμε πειραματικά ότι οι δύο πιο διαδεδομένες υλοποιήσεις του MPI, αποτυγχάνουν να αντιμετωπίσουν την εμφάνιση σφάλματος κατά τη διάρκεια εκτέλεσης της εφαρμογής, τόσο στο επίπεδο της ανίχνευσης και της αναφοράς του στο επίπεδο της εφαρμογής, όσο και στο επίπεδο της διατήρησης μιας συνεπούς συμπεριφοράς της εκτέλεσης.

Αναπτύξαμε ένα νέο μηχανισμό που θα υποστηρίζει πάνω από το επίπεδο της υλοποίησης MPI, αξιόπιστη και έγκαιρη ανίχνευση σφαλμάτων, καθώς και λειτουργίες που θα επιτρέπουν την αντιμετώπιση τους. Ο μηχανισμός βασίζεται στη λογική των William Gropp και Ewing Lusk [11] (βλέπε Ενότητα 3.2.3), ανάπτυξης της εφαρμογής MPI με σχήμα Master-Worker μέσω intercommunicators. Ανάμεσα σε κάθε ζευγάρι Master-Worker εγκαθίσταται ένας intercommunicator. Το σφάλμα που μπορεί να συμβεί σε κάποια διεργασία Worker, ακυρώνει μόνο τον intercommunicator που την συσχετίζει με τη διεργασία Master, αφήνοντας ανεπηρέαστες τις υπόλοιπες διεργασίες Worker. Με τη μέθοδο αυτή, μπορούμε μεν να εξασφαλίσουμε ανεξαρτησία μεταξύ των σφαλμάτων που εμφανίζονται στις διεργασίες Worker, χωρίς δηλαδή να επηρεάζουν την εκτέλεση των υπολοίπων και ανεκτικότητα, καθώς δεν τερματίζει η εκτέλεση της εφαρμογής, από την άλλη όμως, υπάρχει ακόμα το πρόβλημα της αξιόπιστης ανίχνευσής τους.

Πολύ πιθανό, σε μεταγενέστερες εκδόσεις τους, οι υλοποιήσεις του MPI να καταφέρουν να υποστηρίξουν τον τρόπο συμπεριφοράς που υποδεικνύει το πρότυπο του MPI για τις περιπτώσεις σφαλμάτων όσο αναφορά την ανίχνευση, την αναφορά και τη συμπεριφορά εκτέλεσης της εφαρμογής . Για να υποστηρίξουμε σε αυτό το στάδιο όμως την υλοποίηση μας και τις λειτουργίες που αυτή μπορεί να προσφέρει, έπρεπε να δημιουργήσουμε αρχικά το υπόβαθρο ώστε να μπορούμε να ανιχνεύσουμε τα σφάλματα έγκαιρα και αξιόπιστα. Για να ξεπεραστεί αυτή η αδυναμία ανίχνευσης, χρησιμοποιήσαμε TCP sockets για την πραγματοποίηση των επικοινωνιών μεταξύ των διεργασιών που συμμετέχουν στους υπολογισμούς, πάνω από την ήδη εγκατεστημένη υλοποίηση MPI. Το TCP [19], είναι ένα διαδικτυακό πρωτόκολλο βασισμένο στη διασύνδεση των κόμβων, πράγμα που σημαίνει οτι τα δεδομένα στέλνονται απευθείας στον παραλήπτη και όχι ως πακέτο στο δίκτυο. Η διαδικασία ώστε να επιτευχθεί η επικοινωνία μεταξύ δύο κόμβων μέσω TCP sockets περιλαμβάνει την εγκατάσταση και διατήρηση μιας σύνδεσης μεταξύ τους, από την οποία ανταλλάζουν δεδομένα. Πολύ περισσότερο όμως, για τις ανάγκες της ανίχνευσης πιθανού σφάλματος, μέσω των TCP sockets είναι δυνατή η έγκαιρη και αξιόπιστη ανίχνευση του. Το σφάλμα που θα εμφανιστεί σε μια από τις δύο διεργασίες που συμμετέχουν στην κλήση της υλοποίησης, θα προκαλέσει τον

τερματισμό της σύνδεσης μεταξύ τους και κατά συνέπεια την άμεση ανίχνευση του σφάλματος από την άλλη.

Τις βασικές επικοινωνίες λοιπόν στην υλοποίηση μας, οι διεργασίες τις επιτυγχάνουν μέσω μιας δικής μας υποδομής με TCP sockets και όχι μέσω των κλήσεων MPI. Για αυτόν το λόγο, κάθε διεργασία που συμετέχει στην υλοποίηση μας, εκτός από τις αρχικές συνδέσεις που εγκαθιστά με το MPI, εγκαθιστά διαφανώς και μια σύνδεση TCP με κάθε διεργασία που έχει την ευθύνη να ανιχνεύσει το σφάλμα. Η ανίχνευση σφάλματος είναι αμοιβαία και για τις δύο διεργασίες που συμμετέχουν στη σύνδεση. Με αυτόν τον τρόπο, εξασφαλίζουμε οτι πιθανό σφάλμα θα ανιχνευθεί από τη μια εκ των δύο διεργασιών που συμμετέχουν στην επικοινωνία και σε δεύτερο στάδιο θα αναφερθεί στο επίπεδο της εφαρμογής του χρήστη. Από εκείνο το σήμειο και μετά, είναι στην ευχέρεια του προγραμματιστή να αξιοποιήσει σε δεύτερο στάδιο, τις λειτουργίες που προσφέρει η υλοποίησή μας.

4.2. Διεπαφή

Η διεπαφή των συναρτήσεων της υλοποίησης (από εδώ και στο εξής στο κείμενο θα την αναφέρουμε ως FTMW), είναι ίδια με τις αντίστοιχες του MPI με ελάχιστες διαφορές που δεν αλλάζουν τη λογική του προτύπου και άρα τη λογική ανάπτυξης της εφαρμογής, αρκεί αυτή να είναι ανεπτυγμένη σε μορφή Master-Worker. Στην Εικόνα 4.1, φαίνεται ένα υποτυπώδες πρόγραμμα, στο οποίο η διεργασία Master στέλνει σε όλες τις διεργασίες Worker ένα δεδομένο και στη συνέχεια περιμένει να λάβει από αυτές ένα αποτέλεσμα. Αντιστοιχίζεται η υλοποίηση του προγράμματος με κλήσεις MPI και με τις αντίστοιχες κλήσεις του FTMW. Η διαφορά κυρίως εμφανίζεται στο γεγονός οτι υποστηρίζει μόνο συγκεκριμένες επικοινωνιακές λειτουργίες του προτύπου και μόνο ανάμεσα σε ζευγάρια διεργασιών Master-Worker, ανά δύο. Δύο διεργασίες Worker, δεν μπορούν να επικοινωνήσουν μεταξύ τους στο FTMW. Κατά επέκταση, δεν υποστηρίζονται και συλλογικές επικοινωνίες και άρα δεν υπάρχει η αναγκαιότητα χρήσης και ορισμού του τελευταίου ορίσματος των αντίστοιχων κλήσεων του MPI, που είναι ο communicator. Οι communicators έχουν οριστεί στο επίπεδο του FTMW και ο προγραμματιστής απαλλάσσεται από την

ευθύνη του χειρισμού τους. Στην περίπτωση που ο προγραμματιστής επιθυμεί να χρησιμοποιήσει συλλογικές επικοινωνίες, θα πρέπει να τις υλοποιήσει σε επίπεδο εφαρμογής, συνδυάζονται απλές επικοινωνιακές κλήσεις μεταξύ δύο διεργασιών.

Κάθε συνάρτηση του FTMW, επιστρέφει μια τιμή στο επίπεδο της εφαρμογής, που υποδεικνύει αν η κλήση έγινε επιτυχώς ή κατά τη διάρκεια της εμφανίστηκε κάποιο σφάλμα σε μια από τις δύο διεργασίες MPI (αποστολέας-παραλήπτης) που συμμετέχουν στην κλήση της. Η άλλη διεργασία, είτε έχει το ρόλο Master, είτε το ρόλο Worker, θα ανιχνεύσει το σφάλμα και θα το αναφέρει στο επίπεδο της εφαρμογής.

```
#include <stdio.h>
#inlcude <stdlib.h>
#include <mpi.h>
#include "ft_functions.h"

int main(int argc, char *argv[]) {
        int myrank, workers, temp, *info, i;
        MPI_Status status;
        FT_Status status;

        MPI_Init(&argc, &argv);
        MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
        MPI_Comm_size(MPI_COMM_WORLD, &workers);
        ft_init(&args, &argv, &myrank, &workers);

        info = (int *) malloc(workers * sizeof(int));

        if(myrank == 0) {
                for(i = 1; i <= workers; i++) {
                        MPI_Send(&temp, 1, MPI_INT, i, 0, MPI_COMM_WORLD);
                        ft_send(&temp, 1, FT_INT, i, 1);
                }
                for(i = 1; i <= workers; i++) {
                        MPI_Recv(&info[i-1], 1, MPI_INT, i, MPI_ANY_TAG, MPI_COMM_WORLD, &status);
                        ft_recv(&info[i-1], 1, FT_INT, i, FT_ANY_TAG, &status);
                }
        }
        else {
                printf("I am Worker - %d!\n", myrank);

                MPI_Recv(&temp, 1, MPI_INT, 0, MPI_ANY_TAG, MPI_COMM_WORLD, &status);
                ft_recv(&temp, 1, FT_INT, 0, 1, &status);
                i = myrank * temp;
                MPI_Send(&i, 1, MPI_INT, 0, 1, MPI_COMM_WORLD);
                ft_send(&i, 1, FT_INT, 0, 1);
        }

        MPI_Finalize();
        ft_finalize();
        return 1;
}
```

Ουδέτερος κώδικας
Κώδικας MPI
Κώδικας FTMW

Εικόνα 4.1: Αντιστοιχία Προγραμμάτων MPI και FTMW

Από εκεί και μετά είναι θέμα του προγραμματιστή το πως θα αντιμετωπίσει την κατάσταση του σφάλματος, είτε επιλέγοντας να συνεχίσει την εκτέλεση της εφαρμογής με λιγότερες διεργασίες, είτε επανεκκινώντας την πεσμένη διεργασία. Αξίζει να τονιστεί οτι το σφάλμα ανιχνεύεται έγκαιρα από την υλοποίηση του μηχανισμού μας, αλλά στο επίπεδο εφαρμογής αναφέρεται μόνο όταν κάποια διεργασία MPI προσπαθήσει να επικοινωνήσει με την πεσμένη διεργασία. Όλες οι άλλες διεργασίες συνεχίζουν ανεξάρτητα και ανεπηρέαστα την εκτέλεσή τους.

int ft_init(int *argc, char ***argv, int rank, int number)

Παράμετροι

argc	δείκτης στον αριθμό των παραμέτρων
argv	δείκτης στη διεύθυνση της λίστας παραμέτρων
rank	το μοναδικό αναγνωριστικό της διεργασίας
number	ο συνολικός αριθμός των διεργασιών

Περιγραφή

Επιστρέφει 1 όταν η διεργασία έχει δημιουργηθεί πρώτη φορά, 0 όταν έχει επανεκκινήσει από τη διεργασία Master.

---

int ft_finalize()

Περιγραφή

Δεν υπάρχει κάτι που αξίζει να σημειωθεί.

---

int ft_send(void *buf, int count, FT_Datatype datatype, int dest, int tag)

Παράμετροι

buf	αρχική διεύθυνση δομής προς αποστολή
count	αριθμός δεδομένων προς αποστολή
datatype	τύπος δεδομένων προς αποστολή
dest	αναγνωριστικό διεργασίας προορισμού
tag	αναγνωριστικό μηνύματος

Περιγραφή

Επιστρέφει τον αριθμό των δεδομένων που στάλθηκαν επιτυχώς. Σε περίπτωση σφάλματος επιστρέφει 0.

---

int ft_recv(void *buf, int count, FT_Datatype datatype, int source, int tag, FT_Status *status)

Παράμετροι

buf	αρχική διεύθυνση δομής προς παραλαβή
count	αριθμός δεδομένων προς παραλαβή
datatype	τύπος δεδομένων προς παραλαβή
source	αναγνωριστικό διεργασίας πηγής
tag	αναγνωριστικό μηνύματος
status	πληροφορίες κατάστασης μηνύματος

Περιγραφή

Επιστρέφει τον αριθμό των δεδομένων που λήφθηκαν επιτυχώς. Σε περίπτωση σφάλματος επιστρέφει 0.

---

int ft_waitany(int *rank)

Παράμετροι

rank	αναγνωριστικό διεργασίας

Περιγραφή

Επιστρέφει το αναγνωριστικό τις διεργασίας που έστειλε δεδομένα. Σε περίπτωση σφάλματος επιστρέφει -1.

Εικόνα 4.2: Επισκόπηση των Συναρτήσεων Επικοινωνίας του FTMW

Στις Εικόνες Εικόνα 4.2 και Εικόνα 4.3, περιγράφονται οι κλήσεις του API. Το FTMW προσφέρει τις κλήσεις ft_send και ft_recv για αποστολή και παραλαβή δεδομένων αντίστοιχα αλλά με το επιπλέον χαρακτηριστικό της ανίχνευσης του σφάλματος. Επίσης έχουν προστεθεί και οι κλήσεις ft_irecv για τη μη εμποδιστική παραλαβή δεδομένων καθώς και η ft_waitany για την παραλαβή δεδομένων από οποιονδήποτε κόμβο έχει στείλει δεδομένα στη διεργασία που την καλεί. Αυτές οι κλήσεις, είναι η βάση για την ανάπτυξη προγραμμάτων MPI με μια τυπική μορφή Master-Worker. Εκτός όμως από αυτές τις συναρτήσεις του προτύπου, έχουμε προσθέσει ακόμα κάποιες επιπλέον συναρτήσεις για ανάκτηση και ανίχνευση σφαλμάτων, οι οποίες περιγράφονται στην Εικόνα 4.3. Η συνάρτηση ft_restore δίνει τη δυνατότητα στον προγραμματιστή να επανεκκινήσει την πεσμένη διεργασία Worker σε ένα νέο κόμβο διαφανώς από τον προγραμματιστή. Αν δεν υπάρχουν άλλα διαθέσιμα μηχανήματα στο σύστημα, επιστρέφεται αντίστοιχο μήνυμα από τη συνάρτηση. Τέλος, οι συναρτήσεις ft_test και ft_testany χρησιμοποιούνται από μια διεργασία ώστε να διαπιστώσει στην πρώτη περίπτωση αν κάποια συγκεκριμένη διεργασία, το αναγνωριστικό της οποίας θέτεται ως παράμετρος της συνάρτησης, είναι πεσμένη ή για τη δεύτερη περίπτωση αν γενικότερα κάποια διεργασία έχει πέσει και σε περίπτωση που αυτό έχει συμβεί, επιστρέφει το αντίστοιχο αναγνωριστικό.

void ft_restore(int rank)

Παράμετροι
rank αναγνωριστικό διεργασίας

Περιγραφή
Επανεκκινεί την πεσμένη διεργασία (rank) σε κάποιο άλλο μηχάνημα.

==========

int ft_test(int rank)

Παράμετροι
rank αναγνωριστικό διεργασίας

Περιγραφή
Επιστρέφει 0 αν η διεργασία (rank) έχει παρουσιάσει σφάλμα, 0 σε αντίθετη περίπτωση.

==========

int ft_testany(void)

Περιγραφή
Επιστρέφει το αναγνωριστικό της διεργασίας που έχει πέσει, αν υπάρχει τέτοια και 0 σε αντίθετη περίπτωση

Εικόνα 4.3: Επισκόπηση των Συναρτήσεων για Ανάκτηση και Ανίχνευση Σφαλμάτων

Αξίζει να σημειωθεί, οτι οι τιμές (FT)_ANY_TAG και (FT)_ANY_SOURCE υποστηρίζονται από το FTMW και έχουν την ίδια σημασιολογία με τις αντίστοιχες σταθερές του MPI.

Λόγω του οτι το FTMW προσανατολίζεται για εφαρμογές που είναι ανεπτυγμένες σε σχήμα Master-Worker, εισάγονται κάποιοι περιορισμοί:

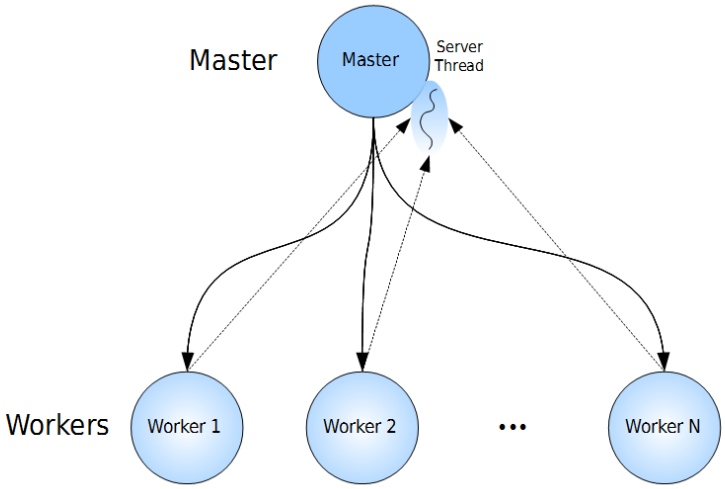
- Οι διεργασίες Workers, δεν μπορούν να επικοινωνήσουν μεταξύ τους και άρα στις κλήσεις αποστολής και παραλαβής, είναι λάθος για την υλοποίηση να οριστεί ως διεργασία προορισμού και πηγής αντίστοιχα, άλλη διεργασία εκτός από τη διεργασία Master (rank 0).
- Τη δυνατότητα επανεκκίνησης κάποιας εσφαλμένης και άρα τερματισμένης διεργασίας εκ νέου σε κάποιο κόμβο, την έχει μόνο η διεργασία Master.

4.3. Αρχιτεκτονική του μηχανισμού FTMW

Το FTMW είναι ένα ενδιάμεσο επίπεδο που τοποθετείται ανάμεσα στο επίπεδο της εφαρμογής MPI και στην υλοποίηση MPI που υπάρχει από κάτω. Για να μπορεί να εφαρμοστεί πλήρως το FTMW στην υλοποίηση MPI, θα πρέπει η τελευταία να υποστηρίζει την έκδοση 2 του προτύπου και ειδικότερα το χαρακτηριστικό της δυναμικής δημιουργίας διεργασιών MPI [9]. Το FTMW υλοποιεί και υποστηρίζει συγκεκριμένες επικοινωνιακές λειτουργίες του προτύπου MPI στις οποίες δεν συμπεριλαμβάνονται οι συλλογικές επικοινωνίες. Όλες οι λειτουργίες του FTMW έχουν τη δυνατότητα της ανίχνευσης και αναφοράς σφαλμάτων, στο επίπεδο της εφαρμογής. Προσφέρει τη δυνατότητα στον προγραμματιστή, από τη στιγμή που του έχει αναφερθεί το σφάλμα, να επιλέξει αν θα συνεχίσει την εκτέλεση με λιγότερες διεργασίες Worker ή θα επανεκκινήσει την πεσμένη διεργασία σε νέο μηχάνημα. Στη συνέχεια του κεφαλαίου γίνεται αναλυτικότερη περιγραφή της σχεδίασης και της υλοποίησης του FTMW.

4.3.1. Περιβάλλον Εκτέλεσης

Στην Εικόνα 4.4, φαίνεται σχηματικά η οργάνωση του περιβάλλοντος εκτέλεσης του FTMW. Φαίνεται η διάταξη των διεργασιών που συμμετέχουν στην εκτέλεση καθώς και των συνδέσεων που δημιουργούνται μεταξύ τους. Η αρχική διεργασία Master είναι αυτή που κατευθύνει τη γενικότερη εκτέλεση. Είναι αυτή που θα εκκινήσει όλες τις υπόλοιπες διεργασίες, θα κατανέμει σύμφωνα και με τον κώδικα της εφαρμογής του χρήστη τη συνολική δουλειά των υπολογισμών στους Workers και είναι αυτή που διατηρεί τις δομές για τις λειτουργίες αποκατάστασης και εξασφάλισης ορθής εκτέλεσης της εφαρμογής σε περίπτωση σφάλματος



Εικόνα 4.4: Διάταξη και Διασυνδέσεις Διεργασιών

Η διεργασία Master, διαθέτει επίσης και ένα νήμα server που επιτελεί ξεχωριστές λειτουργίες. Όλα τα μηνύματα που αποστέλλονται από κάποια διεργασία Worker στη διεργασία Master, παραλαμβάνονται από αυτό. Το νήμα εισάγει τα απεσταλμένα δεδομένα σε μια ουρά την οποία και θα ελέγξει η διεργασία Master, σε περίπτωση που θελήσει να παραλάβει δεδομένα. Σε περίπτωση που η συγκεκριμένη διεργασία Worker δεν έχει αποστείλει δεδομένα, η διεργασία Master, περιμένει μέχρι το νήμα server τα παραλάβει. Εκτός από τα δεδομένα, το νήμα server παραλαμβάνει και όλα τα μηνύματα σηματοδοσίας της υλοποίησης. Οι διεργασίες Worker δε διαθέτουν

νήμα server, καθώς όπως προαναφέρθηκε επικοινωνούν μόνο με τη διεργασία Master.

Το FTMW βασίζεται στην υποδομή των TCP sockets για την επίτευξη των επικοινωνιών προκειμένου, όπως αναφέραμε, να υπάρχει η δυνατότητα ανίχνευσης τυχόν σφαλμάτων. Για τη μη ετεροχρονισμένη αναφορά και ανίχνευση λαθών, επιλέξαμε να υλοποιήσουμε τις κλήσεις αποστολής και παραλαβής μηνυμάτων ως εμποδιστικές. Πιο συγκεκριμένα, στη λειτουργία της αποστολής, η μια διεργασία θα αναμένει την παραλαβή ενός μηνύματος επιβεβαίωσης (FT_ACK) από την έτερη διεργασία, ώστε να είναι βέβαιο οτι τα δεδομένα έχουν αποσταλεί επιτυχώς. Η λειτουργία της παραλαβής ούτως ή άλλως είναι εμποδιστική, καθώς θα πρέπει να παραλάβει τα δεδομένα για να ολοκληρωθεί. Η λειτουργία ft_irecv προστέθηκε για μη εμποδιστική παραλαβή δεδομένων και συνδυάζεται με τη χρήση της ft_wait ή της ft_waitany. Αν η διεργασία Master, καλέσει την ft_irecv και τα δεδομένα τα οποία επιθυμεί να λάβει δεν είναι διαθέσιμα σε εκείνο το χρονικό σημείο του κώδικα, εισάγει σε μια ουρά τις παραμέτρους της κλήσεωςς και συνεχίζει την εκτέλεση του προγράμματος. Στη συνέχεια με την κλήση της ft_wait δίνοντας ως όρισμα το αναγνωριστικό της συγκεκριμένης διεργασίας Worker από την οποία επιθυμεί να λάβει δεδομένα, ή με την κλήση της ft_waitany για οποιαδήποτε διεργασία Worker, γίνεται έλεγχος στην ουρά παραλαβής μηνυμάτων για να βρεθεί κάποιο μήνυμα που να αντιστοιχίζεται με κάποια προηγούμενη κλήση της ft_irecv. Αν όχι, περιμένει μέχρι αυτό να συμβεί.

Ο χρήστης έχει τη δυνατότητα, μέσω δύο μεταβλητών περιβάλλοντος να ορίσει το πόσες διεργασίες Worker (FT_NUM_SLAVES) αλλά και πόσες επιπλέον διεργασίες-αντίγραφα Master (FT_NUM_MASTERS) επιθυμεί να έχει. Για τη δεύτερη λειτουργία θα μιλήσουμε στην Ενότητα 4.7 και εξορισμού αν δεν οριστεί από το χρήστη, τότε στην εκτέλεση δε συμμετέχει κάποια επιπλέον διεργασία Master παρά μόνο η αρχική. Θα πρέπει επίσης να ορίσει και ένα αρχείο (hostfile) στο οποίο θα εισάγει τα ονόματα των υπολογιστών-κόμβων που θα συμμετέχουν στην εκτέλεση της εφαρμογής. Η αρχική διεργασία Master θα αναλάβει να διαβάσει το αρχείο και να εκκινήσει τις διεργασίες στα απαραίτητα μηχανήματα, ανάλογα και με τις μεταβλητές περιβάλλοντος που έχει ορίσει ο χρήστης. Τα πρώτα μηχανήματα του αρχείου

παίρνουν το ρόλο Worker και τα υπόλοιπα που αναλογούν στις τιμές των μεταβλητών, γίνονται οι επιπλέον διεργασίες Master.

4.3.2. Διαδικασία Αρχικοποίησης

Την εκτέλεση της εφαρμογής την εκκινεί μόνο η διεργασία Master, σε αντίθεση με τις γνωστές υλοποιήσεις του MPI όπου όλες οι διεργασίες εκκινούν εξ αρχής από την υλοποίηση. Όταν συναντήσει την κλήση ft_init (η αντίστοιχη της MPI_Init), που θα σηματοδοτεί την αρχή του κώδικα που θα εκτελεστεί από όλες τις διεργασίες που έχουν οριστεί, θα ξεκινήσει τη διαδικασία δημιουργίας όλων των απαραίτητων διεργασιών καθώς και την εγκατάσταση των συνδέσεων μεταξύ τους. Από το αρχείο hostfile του χρήστη, θα εκκινήσει όσες διεργασίες Worker έχουν οριστεί από την αντίστοιχη μεταβλητή περιβάλλοντος αρχικά (FT_NUM_SLAVES) και στη συνέχεια θα εκκινήσει και τις επιπλέον διεργασίες Master (FT_NUM_MASTERS, βλέπε Ενότητα 4.7). Η εκκίνηση και εγκατάσταση των αρχικών επικοινωνιακών συνδέσεων αλλά και οι πρώτες απαραίτητες ανταλλαγές δεδομένων αρχικοποίησης, πραγματοποιούνται από την υλοποίηση MPI που υπάρχει από κάτω. Πέρα από αυτό όμως, δημιουργείται και μια σύνδεση TCP socket, μεταξύ κάθε ζευγαριού Master-Worker. Συνδέσεις TCP δημιουργούνται επίσης και μεταξύ των διεργασιών Master, εφόσον είναι τουλάχιστον 2.

Όλες οι διεργασίες που έχουν εκκινηθεί από τη διεργασία Master, κατά τη διαδικασία της αρχικοποίησης λαμβάνουν κάποια απαραίτητα δεδομένα από αυτή. Ενημερώνονται για το πόσες είναι οι συνολικές διεργασίες Worker στο σύστημα, αν υπάρχουν και πόσες επιπλέον διεργασίες Master, ποιό είναι το μοναδικό αναγνωριστικό τους αλλά και κάποιες πληροφορίες απαραίτητες για την εγκατάσταση της μετέπειτα σύνδεσης TCP.

Εκτός όμως από τις συνδέσεις που είναι απαραίτητες, η διεργασία Master αρχικοποιεί και τις δομές δεδομένων που θα χρειαστούν για τη λειτουργία του FTMW. Διατηρεί την κατάσταση της κάθε διεργασίας, ουρές παραλαβής δεδομένων οι οποίες έχουν τη μορφή παραγωγού (νήμα server) και καταναλωτή (αρχική διεργασία).

4.3.3. Μηχανισμός Αντιμετώπισης Σφαλμάτων

Στην περίπτωση που έχουμε κάποιο σφάλμα σε κάποιον κόμβο, είναι δυνατή η ανίχνευση του από τη διεργασία Master, μέσω των TCP sockets καθώς αυτόματα θα τερματιστεί και η σύνδεση μεταξύ τους. Η αναφορά του σφάλματος στο επίπεδο της εφαρμογής, πραγματοποιείται μόνο όταν στην εφαρμογή κάποια κλήση σε συνάρτηση του FTMW, αναφερθεί στον πεσμένο κόμβο. Από εκεί και μετά, είναι στην ευχέρεια του προγραμματιστή και του κώδικα της εφαρμογής του, το αν θα επανεκκινήσει την πεσμένη διεργασία ή θα συνεχίσει την εκτέλεση της εφαρμογής με λιγότερες διεργασίες Worker. Σε περίπτωση που το σφάλμα συμβεί στη διεργασία Master, το ανιχνεύουν οι διεργασίες Worker και είτε θα τερματίσουν τη λειτουργία τους, αφού όμως έχουν αναφέρει πρώτα το σφάλμα στο επίπεδο της εφαρμογής, είτε θα συνεχίσουν την εκτέλεση έχοντας ως Master, την επόμενη διεργασία Master στη σειρά αν έχει οριστεί (βλέπε Ενότητα 4.7).

Αν ο προγραμματιστής επιλέξει να επανεκκινήσει την πεσμένη διεργασία με την κλήση ft_restore, η υλοποίηση θα αναλάβει τρεις βασικές ενέργειες. Πρώτα εισάγει το όνομα του μηχανήματος στο οποίο έτρεχε η διεργασία που εμφάνισε το σφάλμα, σε μια "μαύρη" λίστα μηχανημάτων, μια δομή που ουσιαστικά περιέχει όλα τα μηχανήματα από το αρχείο που έδωσε ως είσοδο στην υλοποίηση ο χρήστης (hostfile) που έχουν παρουσιάσει σφάλμα, ώστε να μην επανεκκινήσει σε αυτά κάποια άλλη διεργασία. Ουσιαστικά στην υλοποίηση του FTMW, έχουμε επιλέξει να θεωρούμε όλα τα σφάλματα ως μόνιμα και όχι ως παροδικά. Αυτό μπορεί να έχει ως αποτέλεσμα την ανάγκη ύπαρξης και καθορισμού πολλών διαφορετικών μηχανημάτων και τη μη επαναχρησιμοποίησή τους σε περίπτωση παροδικών σφαλμάτων, αλλά από την άλλη εξασφαλίζει μεγαλύτερη αξιοπιστία. Η δεύτερη ενέργεια, είναι να επιλέξει ένα μηχάνημα από το αρχείο του χρήστη και να επανεκκινήσει σε εκείνο τη διεργασία. Τέλος, ως τρίτο βήμα, επανεγκαθιστά τη σύνδεση TCP με τη νέα διεργασία. Το τι πρέπει να γίνει ώστε η κατάσταση της διεργασίας Worker (τιμές μεταβλητών, καταστάσεις δομών δεδομένων κ.α.) να επανέλθει στο ίδιο σημείο με το όταν εμφάνισε το σφάλμα, είναι ευθύνη του προγραμματιστή της εφαρμογής. Παρόλα αυτά η υλοποίηση προσφέρει μια μορφή

καταγραφής των επικοινωνιακών κλήσεων, ώστε να διευκολυνθεί η παραπάνω διαδικασία στα χέρια του χρήστη (βλέπε Ενότητα 4.4).

Στην περίπτωση που ο χρήστης επιλέξει να μην επανεκκινήσει τη διεργασία σε κάποιο άλλο μηχάνημα, τότε δεν αλλάζουν πολλά στην εκτέλεση. Όλες οι υπόλοιπες διεργασίες Worker θα συνεχίσουν την φυσιολογική ροή της εκτέλεσής τους καθώς ούτως ή άλλως δεν έχουν τη δυνατότητα να επικοινωνήσουν με την πεσμένη διεργασία Worker και αρα να επηρεαστούν από τη μη συμμετοχή της στους υπολογισμούς. Η διεργασία Master στο επίπεδο της εφαρμογής, κάθε φορά που θα αναφέρεται στην πεσμένη διεργασία Worker, θα λαμβάνει την αντίστοιχη τιμή σφάλματος. Βέβαια, δεν αναιρείται το γεγονός οτι η αποκατάσταση του σφάλματος με την επανεκκίνηση της διεργασίας, μπορεί να γίνει από το χρήστη και ετεροχρονισμένα σε οποιοδήποτε σημείο της εφαρμογής επιλέξει.

4.3.4. Διαδικασία Τερματισμού

Κατα τη διαδικασία τερματισμού, τα πράγματα είναι σχετικά πιο απλά. Όπως και σε ένα πρόγραμμα MPI, έτσι και στο FTMW, κάθε διεργασία που συμμετέχει στην εκτέλεση της εφαρμογής, είτε είναι Master, είτε είναι Worker, καλεί τη συνάρτηση τερματισμού ft_finalize. Για την επίτευξη συγχρονισμού, η διεργασία Master, στέλνει ένα μήνυμα σηματοδοσίας τερματισμού σε κάθε διεργασία Worker. Αφού τα μηνύματα σταλθούν, μπορεί πλέον η διεργασία Master να αποδεσμεύσει τη μνήμη των δομών δεδομένων που χρειάστηκε στην υλοποίηση. Επίσης τερματίζει και όλες τις συνδέσεις TCP με όλες τις διεργασίες Worker. Όταν λάβει το μήνυμα η διεργασία Worker, μπορεί να ξεκινήσει και αυτή με τη σειρά της την αποδέσμευση των δομών που χρειάστηκε. Για την περίπτωση που έχουμε περισσότερες διεργασίες Master, η διαδικασία περιλαμβάνει και κάποια ακόμα βήματα που θα δούμε αργότερα.

Στη διαδικασία τερματισμού, καλείται επίσης και η αντίστοιχη κλήση τερματισμού της υλοποίησης MPI (MPI_Finalize) για τον τερματισμό και του τμήματος του MPI. Σε περίπτωση που ο χρήστης έχει επιλέξει να συνεχίσει την εφαρμογή του με λιγότερες διεργασίες από αυτές που άρχισε την εκτέλεση, λόγω εμφάνισης

σφάλματος και επιλογής του να μην επανεκκινήσει την πεσμένη διεργασία ξανά, η κλήση της MPI_Finalize θα προκαλούσε ατέρμονη αναμονή, καθώς η χρήση της απαιτεί να κληθεί από όλες τις διεργασίες που συμμετέχουν στην εκτέλεση. Σε αυτήν την περίπτωση αναγκαστικά επιλέγεται η απότομη ολοκλήρωση της εκτέλεσης με την κλήση MPI_Abort.

4.4. Καταγραφή Συναρτήσεων

Όπως αναφέρθηκε προηγουμένως, η υλοποίηση του FTMW δεν προσφέρει στον προγραμματιστή της εφαρμογής τη λειτουργικότητα υποστήριξης checkpoints, καθώς ο βασικός στόχος της υλοποίησης δεν είναι το πως θα γίνει η σωστή επανεκκίνηση της εφαρμογής, αλλά η ανεκτικότητα της εκτέλεσης της εφαρμογής στο να μην τερματιστεί παρά τα σφάλματα των διεργασιών που θα εμφανιστούν. Αν το επιθυμεί ο προγραμματιστής, μπορεί με δικό του κώδικα να αποθηκεύει σε διάφορα σημεία της εφαρμογής, σε αρχείο τα απαραίτητα δεδομένα και τις καταστάσεις των δομών δεδομένων που χρειάζονται σε περίπτωση που δεν υπάρχουν οι απαραίτητες συνθήκες (μη αξιόπιστο σύστημα, λίγα διαθέσιμα μηχανήματα) να διατηρηθεί η εκτέλεση της εφαρμογής. Αργότερα διαβάζοντας το αρχείο, μπορεί να επανεκκινήσει η εφαρμογή από κάποιο σημείο της εκτέλεσης λίγο πριν από το συνολικό τερματισμό της.

Σε περίπτωση λοιπόν που ο προγραμματιστής επιλέξει να επανεκκινήσει κάποια εσφαλμένη διεργασία Worker, θα πρέπει ως δεύτερο βήμα, να την επαναφέρει στην κατάσταση που ήταν οι δομές της τη στιγμή που εμφανίστηκε το σφάλμα, στέλνοντας τα αντίστοιχα δεδομένα που χρειάζονται σύμφωνα και με την αρχική ροή της εκτέλεσης της εφαρμογής. Σύμφωνα με τη λογική των εφαρμογών Master-Worker, η διεργασία Master μπαίνει σε κύκλους επικοινωνιακών κλήσεων για να μοιράσει εργασίες στις διεργασίες Worker και να συγκεντρώσει αποτελέσματα. Ανάλογα σε ποιά κλήση του FTMW η διεργασία Master, ενημερώθηκε για το σφάλμα σε κάποιον Worker, διαφοροποιείται και το τι πρέπει να στείλει στην επανεκκινημένη διεργασία Worker.

Έκδοση χωρίς επαναφορά πεσμένης διεργασίας

```
do {
	for(i = 1; i <= workers; i++)
		ft_send(&a, 1, FT_INT, i, 1);

	for(i = 1; i <= workers; i++)
		ft_send(&b, 1, FT_INT, i, 1);

	for(i = 1; i <= workers; i++)
		ft_recv(&c, 1, FT_INT, i, 1);

} while(condition(c));
```

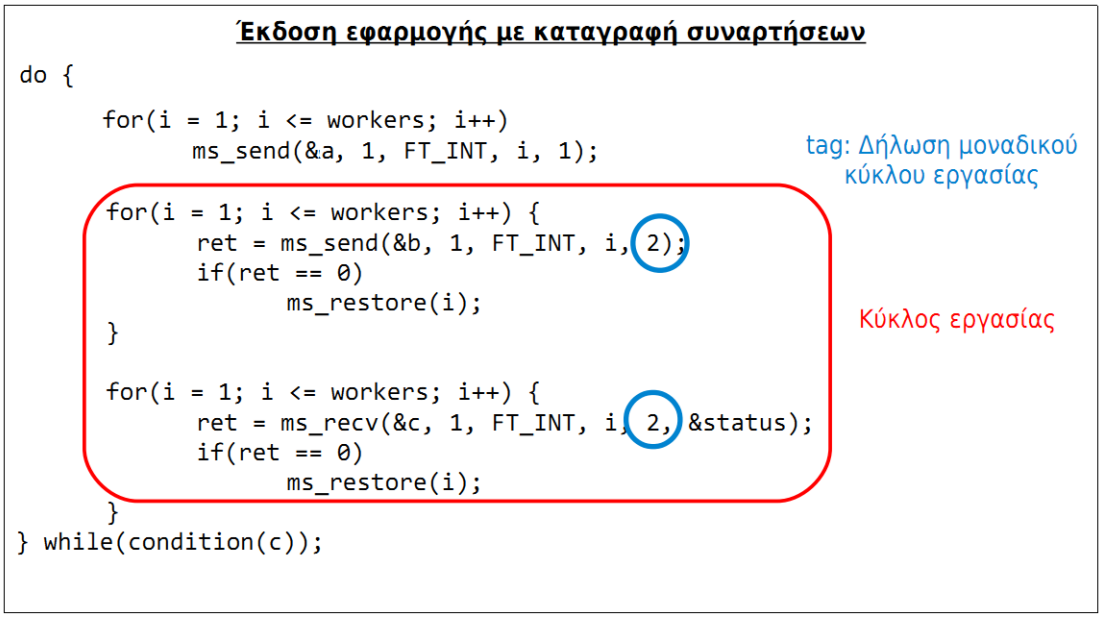
Έκδοση με επαναφορά πεσμένης διεργασίας

```
do {
	for(i = 1; i <= workers; i++)
		ret = ft_send(&a, 1, FT_INT, i, 1);

	for(i = 1; i <= workers; i++) {
		ret = ft_send(&b, 1, FT_INT, i, 1);
		if(ret == 0) {
			ft_restore(i);
			ret = ft_send(&a, 1, FT_INT, i, 1);
			ret = ft_send(&b, 1, FT_INT, i, 1);
		}
	}

	for(i = 1; i <= workers; i++) {
		ret = ft_recv(&c, 1, FT_INT, i, 1, &status);
		if(ret == 0) {
			ft_restore(i);
			ft_send(&a, 1, FT_INT, i, 1);
			ft_send(&b, 1, FT_INT, i, 1);
			ft_recv(&c, 1, FT_INT, i, 1, &status);
		}
	}
} while(condition(c));
```

Εικόνα 4.5: Αντιστοιχία Εφαρμογής Χωρίς Επαναφορά και με Επαναφορά Εσφαλμένης Διεργασίας

Στην Εικόνα 4.5, φαίνεται ακριβώς το πως θα έπρεπε να μετατραπεί μια απλή εφαρμογή Master-Worker με το FTMW, από την πλευρά της διεργασίας Master, που επιλέγει να μην επανεκκινήσει την πεσμένη διεργασία, σε μια αντίστοιχη, που σε κάθε κλήση στο FTMW ελέγχεται αν η διεργασία έχει πέσει, ώστε να την

επαναφέρει. Στην πρώτη εκδοσή μάλιστα, όχι μόνο δεν επιλέγει να αναφέρει το σφάλμα, αλλά πολύ περισσότερο σε περίπτωση εμφάνισης του, χάνεται και ο υπολογισμός, μια εκδοχή που ουσιαστικά δεν υπάρχει σε κανένα βαθμό ανοχή σφαλμάτων. Στη δεύτερη έκδοση ωστόσο, στην οποία υπάρχει η ιδιότητα της ανοχής σε σφάλματα, βλέπουμε οτι όσο προχωράμε στις κλήσεις του κώδικα, ο προγραμματιστής αφού αποκαταστήσει την πεσμένη διεργασία, θα πρέπει να έχει στο νου και ποιές συναρτήσεις θα πρέπει να καλέσει για να έρθει η πεσμένη διεργασία στο σωστό σημείο εκτέλεσης. Εισάγεται έτσι ένας επιπλέον φόρτος στην λογική ανάπτυξης της εφαρμογής για την επίτευξη ανοχής σφαλμάτων.



Εικόνα 4.6: Έκδοση με Συναρτήσεις ms για Καταγραφή Συναρτήσεων

Για τον παραπάνω λόγο, το FTMW, προσφέρει την επιπλέον λειτουργία της καταγραφής των συναρτήσεων του, που καλεί στο επίπεδο της εφαρμογής ο προγραμματιστής. Καλώντας τις συναρτήσεις ms_send, ms_recv και ms_irecv, οι οποίες επιτελούν τις ίδιες λειτουργίες με τις αντίστοιχες απλές συναρτήσεις του FTMW, αλλά προσφέρουν την επιπλέον δυνατότητα της καταγραφής και αποθήκευσης των δεδομένων τους. Με αυτόν τον τρόπο σε περίπτωση που ο προγραμματιστής επιθυμεί να επανεκκινήσει μια διεργασία, αρκεί να καλέσει την ms_restore και εκτός από τη διαδικασία επανεκκίνησης της πεσμένης διεργασίας που γίνεται σε πρώτο βήμα, η υλοποίηση θα αναλάβει να εκτελέσει αυτόματα και όλες τις

καταγεγραμμένες συναρτήσεις που όρισε ο προγραμματιστής, μειώνοντας έτσι τον επιπλέον φόρτο που επαφίεται στον προγραμματιστή, όσον αφορά στη λογική της ανοχής σφαλμάτων αλλά και στην ποσότητα κώδικα. Στην Εικόνα 4.6, παρουσιάζεται η εφαρμογή της Εικόνας 4.5, σε έκδοση όπου χρησιμοποιούνται οι συναρτήσεις ms_.

Σύμφωνα με τα παραπάνω, δημιουργούνται δύο ζητήματα. Τόσο για τον τρόπο που πρέπει να αποθηκεύονται οι συναρτήσεις, όσο και για το ποιες θα πρέπει να εκτελεστούν σε περίπτωση επανεκκίνησης της πεσμένης διεργασίας. Το πρώτο ζήτημα είναι, το πως η υλοποίηση του FTMW, θα μπορεί να ξεχωρίσει δύο συναρτήσεις που καλούνται με τις ίδιες τιμές στις παραμέτρους τους. Αν οι κλήσεις αναφέρονται στην ίδια συνάρτηση σε διαφορετικά σημεία του κώδικα υπάρχει ανάγκη ξεχωριστής αποθήκευσης τους στη σχετική δομή, ενώ αν οι κλήσεις αναφέρονται στην ίδια συνάρτηση που καλείται επαναληπτικά, όπου υπάρχει η ανάγκη αντικατάστασης της ήδη υπάρχουσας αποθηκευμένης. Ένα δεύτερο ζήτημα είναι το τι γίνεται με τους κύκλους εργασιών μεταξύ Master-Worker. Με τον όρο «κύκλος εργασίας» ορίζουμε μια ακολουθία επικοινωνιών που γίνεται επαναληπτικά, μια επαναληπτική ροή ανταλλαγής δουλειάς και αποτελέσματος για παράδειγμα μεταξύ ενός ζευγαριού Master-Worker μέχρι να επιτευχθεί το απαραίτητο αποτέλεσμα ή να εκπληρωθεί μια συνθήκη. Σε μια εφαρμογή Master-Worker, είναι πιθανό να υπάρχουν παραπάνω από ένας κύκλοι εργασιών. Σε περίπτωση σφάλματος, όταν κάποιος κύκλος έχει ολοκληρωθεί επιτυχώς δε θα πρέπει να ξαναεκτελεστούν οι καταγεγραμμένες συναρτήσεις του.

Για την επίλυση αυτών των ζητημάτων, επιλέξαμε να προσδώσουμε μια επιπλέον έννοια στην παράμετρο tag των συναρτήσεων της υλοποίησης. Το πεδίο tag, των συναρτήσεων που πρόκειται να καταγραφούν, στην υλοποίηση FTMW υποδεικνύει επιπλέον και σε ποιόν κύκλο εργασιών ανήκει η συνάρτηση, όπως φαίνεται και στο παράδειγμα της Εικόνα 4.6. Με αυτόν τον τρόπο, η υλοποίηση μπορεί να ξεχωρίσει οτι δύο συναρτήσεις βρίσκονται στο ίδιο ή σε διαφορετικά σημεία του κώδικα, ελέγχοντας το πεδίο tag και πράττοντας ανάλογα. Βέβαια είναι και μια επιπλέον ευθύνη του προγραμματιστή η μέριμνα, ώστε τα πεδία tag να έχουν τις κατάλληλες τιμές ώστε να αντιστοιχούν σε αυτό που θέλει να υλοποιήσει. Όσον αφορά το ζήτημα των κύκλων εργασιών, από τη στιγμή που η υλοποίηση πλέον έχει τη δυνατότητα να

ξεχωρίζει τους κύκλους, μπορεί επιπλέον να γνωρίζει και ποιός κύκλος έχει ολοκληρωθεί επιτυχώς, ώστε στη διαδικασία της αποκατάστασης του σφάλματος, να προσπεράσει τις καταγεγραμμένες συναρτήσεις που αναφέρονται σε αυτόν. Αξίζει να σημειωθεί οτι αν ο προγραμματιστής θέσει την τιμή FT_ANY_TAG στο πεδίο tag, τότε η συνάρτηση καταγράφεται ξεχωριστά χωρίς να ανήκει σε κάποιον κύκλο και άρα θα εκτελεστεί ούτως ή άλλως, κατά τη διαδικασία της αποκατάστασης.

Για την υποστήριξη της λειτουργίας της καταγραφής των συναρτήσεων, η διεργασία Master (και η κάθε διεργασία Master), διατηρεί κάποιες δομές δεδομένων επιπλέον. Για κάθε μια διεργασία Worker, δεσμεύεται μια ουρά που διατηρεί τις συναρτήσεις που έχουν επιλεγεί να καταγραφούν, με τη σειρά που έχουν κληθεί. Κάθε κόμβος της ουράς, είναι από μόνος του μια ουρά που αντιπροσωπεύει έναν κύκλο εργασιών που διαχωρίζεται από την τιμή της παραμέτρου tag. Η διαδικασία της αποκατάστασης, ξεκινάει εκτελώντας τις καταγεγραμμένες συναρτήσεις από την κεφαλή της δομής μέχρι την ουρά. Αν ο κόμβος περιλαμβάνει μόνο μια συνάρτηση, τότε αυτή εκτελείται. Αν ο κύκλος του κόμβου είχε ολοκληρωθεί επιτυχώς, τότε δεν εκτελείται καμία συνάρτηση του ενώ αν όχι, σημαίνει οτι το σφάλμα συναίβει κατα τη διάρκεια του κύκλου και άρα θα εκτελεστούν οι καταγεγραμμένες συναρτήσεις του, μέχρι τη συνάρτηση που ανέφερε το λάθος και άρα δεν εκτελέστηκε επιτυχώς.

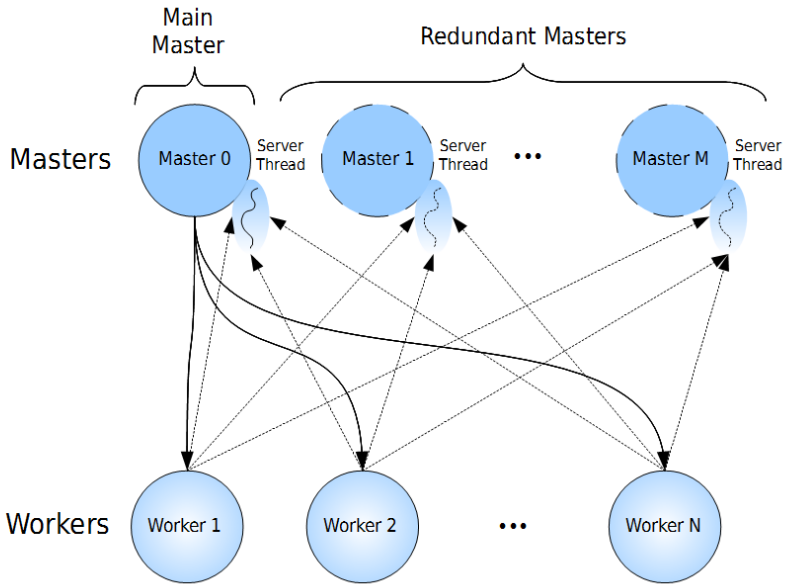
Η λειτουργία της καταγραφής συναρτήσεων, εισάγει κάποιους περιορισμούς. Η δυνατότητα καταγραφής συναρτήσεων μπορεί να χρησιμοποιηθεί μόνο από τη διεργασία Master. Επιπλέον δεν υποστηρίζει εμφωλευμένους κύκλους εργασιών, δηλαδή ένας κύκλος εργασίας που εξελίσσεται μέσα σε έναν άλλον που δεν έχει περατωθεί. Σε μια τέτοια περίπτωση, δεν μπορεί η υλοποίηση να εγγυηθεί για τη σωστή αποκατάσταση της εκτέλεσης της εφαρμογής σε περίπτωση σφάλματος και η συμπεριφορά της θα είναι απρόβλεπτη.

4.5. Πολλαπλές Διεργασίες Master (Master Redundancy)

Στην περίπτωση που το σφάλμα συμβεί σε κάποια διεργασία Worker, η επιλογή για το τι θα επακολουθήσει επαφίεται στο επίπεδο της εφαρμογής. Είναι δηλαδή στα

χέρια του προγραμματιστή, αν θα επιλέξει να επανεκκινήσει την πεσμένη διεργασία, είτε τη στιγμή που αναφέρθηκε το σφάλμα στην εφαρμογή, είτε σε κάποια άλλη στιγμή στον κώδικα, ή αν θα επιλέξει να συνεχιστεί η εκτέλεση με λιγότερες διεργασίες Worker. Εγείρεται όμως το ζήτημα του τι συμβαίνει, αν το σφάλμα συμβεί στη διεργασία που έχει το ρόλο του Master η οποία είναι υπεύθυνη να χειρίζεται τη γενικότερη ροή της εκτέλεσης και να ανιχνεύει πιθανά σφάλματα στους Workers.

Πολλές τεχνικές έχουν διατυπωθεί στη βιβλιογραφία. Μια επιλογή θα ήταν τα checkpoints (βλέπε Ενότητα 3.3.1), τα οποία αποθηκεύονται είτε αυτόματα από την υλοποίηση και διαφανώς από τον προγραμματιστή, είτε με υπευθυνότητα του ίδιου του χρήστη ο οποίος πρέπει να αποθηκεύει τα απαραίτητα δεδομένα σε διάφορα κρίσιμα σημεία του κώδικα. Την τελευταία τεχνική θα μπορούσε να ακολουθήσει κάποιος και στη δική μας υλοποίηση. Εκτός των checkpoints, χρειάζεται και ένας αλγόριθμος επανεκκίνησης. Από την άλλη όμως, το checkpointing είναι μια τεχνική που δε βοηθάει την κλιμακωσιμότητα, απαιτεί επιπλέον αποθηκευτικό χώρο και δε λύνει το πρόβλημα στη βάση του, όταν το σφάλμα στο μηχάνημα είναι μόνιμο. Επίσης η βασική αρχή που επιλέξαμε να ακολουθήσουμε στην υλοποίηση μας δεν είναι το πως θα διαμορφωθούν οι συνθήκες ώστε αν τερματιστεί βίαια η εφαρμογή να μπορεί να επανεκκινήσει, αλλά η διατήρηση της απρόσκοπτης εκτέλεσης ακόμα και υπό την παρουσία ενός αριθμού σφαλμάτων.



Εικόνα 4.7: Διάταξη και Διασυνδέσεις Διεργασιών με Πολλαπλούς Master

Προς αυτήν την κατεύθυνση, για την αντιμετώπιση σφαλμάτων στη διεργασία Master, επιτρέπουμε τον προγραμματιστή να ορίσει και έναν αριθμό επιπλέον διεργασιών-αντιγράφων του Master. Αυτές οι διεργασίες τρέχουν *παράλληλα* με την αρχική διεργασία Master και εκτελούν *τον ίδιο* κώδικα εφαρμογής, έχουν τις ίδιες λειτουργικότητες αλλά εκτελούνται διαφανώς από το χρήστη. Ο χρήστης πριν την εκτέλεση της εφαρμογής με τη μεταβλητή περιβάλλοντος FT_NUM_MASTERS ορίζει το πόσες επιπλέον διεργασίες Master επιθυμεί. Οι διεργασίες Master βρίσκονται σε διάταξη προτεραιότητας μεταξύ τους. Με αυτόν τον τρόπο δίνεται η δυνατότητα στην υλοποίηση, σε περίπτωση που τερματιστεί η αρχική διεργασία Master, αυτόματα τη θέση της να την πάρει η επόμενη στη διάταξη. Αυτή η διαδικασία μπορεί να συνεχιστεί για όλες τις επιπλέον διεργασίες Master. Στην Εικόνα 4.7, φαίνονται η διάταξη και οι διασυνδέσεις μεταξύ των διεργασιών της υλοποίησης του FTMW, όταν σε αυτή συμμετέχουν και επιπλέον διεργασίες Master.

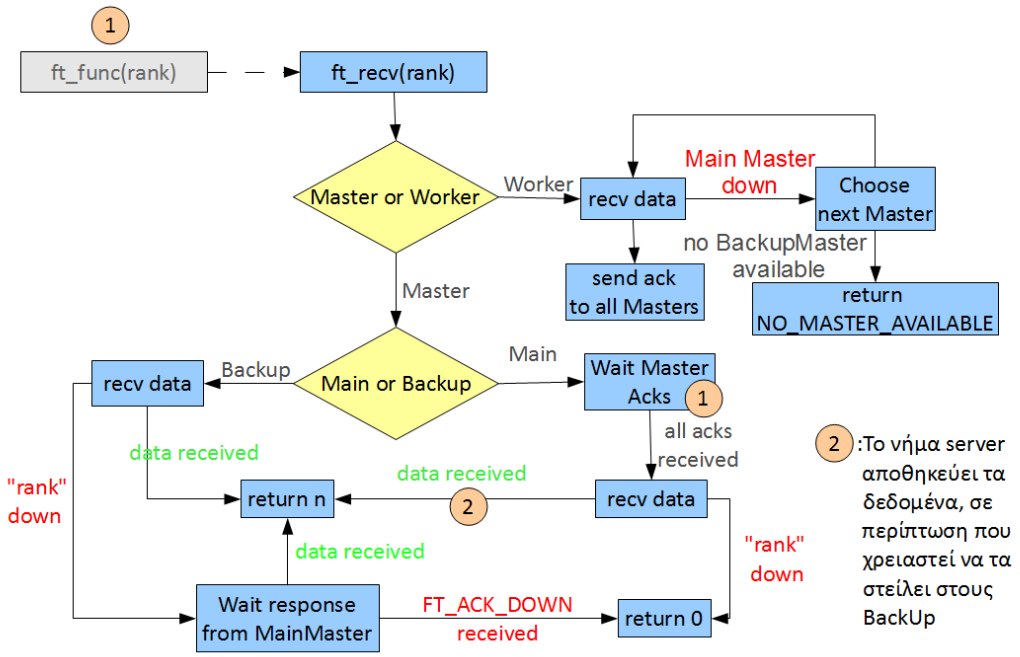
Κατά την αρχικοποίηση, η αρχική διεργασία Master εκτός από τις συνδέσεις TCP που διατηρεί με τις διεργασίες Worker, για να μπορεί να ανιχνεύει πότε εμφανίστηκε το σφάλμα, διατηρεί συνδέσεις και με κάθε επιπλέον διεργασία Master, για τον ίδιο λόγο. Επίσης και οι επιπλέον διεργασίες Master, διατηρούν μεταξύ τους συνδέσεις για να ελέγχουν η μια την κατάσταση της άλλης. Με αυτόν τον τρόπο μια διεργασία από τις δευτερεύουσες Master, μπορεί να ανιχνεύει τυχόν σφάλμα στην αρχική Master και ανάλογα με την κατάσταση των προηγούμενων διεργασιών Master στη διάταξη, να αναλάβει αυτή το ρόλο της βασικής διεργασίας Master στο σύστημα. Παράλληλα και μια διεργασία Worker που θα ανιχνεύσει το σφάλμα στη διεργασία Master, έχει τη δυνατότητα να γνωρίζει και άρα να ορίσει, το ποιά θα είναι η επόμενη διεργασία Master με την οποία θα επικοινωνεί.

Στον τερματισμό της υλοποίησης, κάθε δευτερεύουσα διεργασία Master, στέλνει το αντίστοιχο μήνυμα σηματοδοσίας τέλους στην εκείνη τη στιγμή βασική διεργασία Master, η οποία και περιμένει μέχρι όλα τα μηνύματα να ληφθούν από το νήμα της. Αφού ολοκληρωθεί αυτή η διαδικασία, είναι σε θέση να αρχίσει τη διαδικασία τερματισμού που αναφέραμε σε προηγούμενη ενότητα (βλέπε 4.3.4).

4.5.1. Ανταλλαγή Μηνυμάτων Πολλαπλών Master

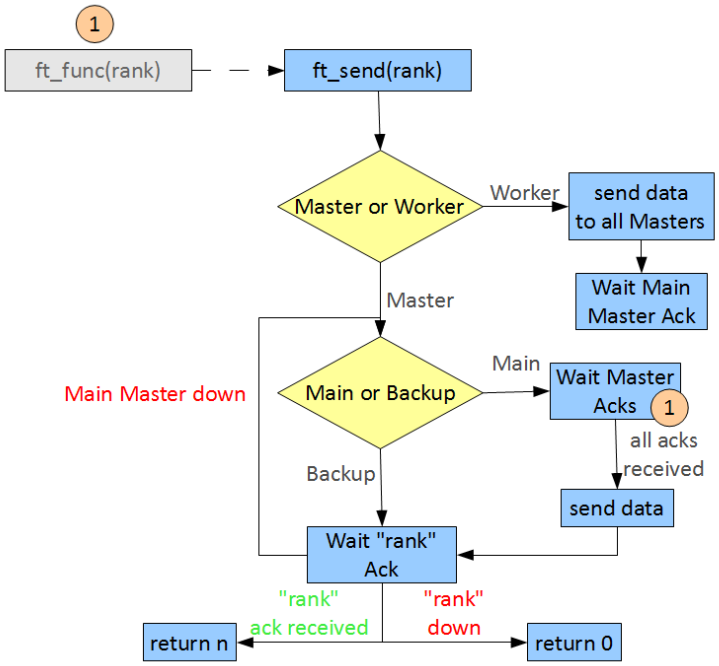
Για την υποστήριξη της σωστής λειτουργίας των πολλαπλών διεργασιών Master, χρειάζεται να υποστούμε κάποιον επιπλέον φόρτο επικοινωνιών. Τόσο στη διαδικασία αποστολής όσο και στη διαδικασία παραλαβής δεδομένων, αλλά και κατά τη διαδικασία ανίχνευσης του σφάλματος, για την οποία θα μιλήσουμε στην επόμενη ενότητα. Οι διεργασίες Master θα πρέπει να διατηρούν την ίδια κατάσταση στις μεταβλητές και στα δεδομένα της εφαρμογής άρα να λαμβάνουν όλες τα αποτελέσματα από τους υπολογισμούς των διεργασιών Worker. Έτσι σε περίπτωση που χρειαστεί να αναλάβουν αυτές ρόλο Master, λόγω σφάλματος στην βασική διεργασία Master, θα είναι σε θέση να συνεχίσουν από το σωστό σημείο την εκτέλεση.

Όταν μια διεργασία Worker θέλει να στείλει δεδομένα, επιβαρύνεται με επιπλέον δουλειά. Αφού στείλει πρώτα στη βασική διεργασία Master και λάβει και την επιβεβαίωση οτι τα δεδομένα ελήφθησαν επιτυχώς, θα πρέπει να στείλει και στις υπόλοιπες διεργασίες Master τα ίδια δεδομένα. Για τις υπόλοιπες παρόλα αυτά δε χρειάζεται να περιμένει σήμα επιβεβαίωσης, καθώς όπως θα δούμε αργότερα, αρκεί μια διεργασία να έχει παραλάβει τα δεδομένα για να μπορεί να γίνει επιτυχώς η αποκατάσταση του σφάλματος και στις πολλαπλές διεργασίες Master. Οι υπόλοιπες διεργασίες Master που θα λάβουν τα δεδομένα, θα στείλουν ένα σχετικό μήνυμα σηματοδοσίας στη βασική διεργασία Master, για να υποδείξουν οτι τα παρέλαβαν επιτυχώς. Τα μηνύματα αυτά θα τα παραλάβει το νήμα server και άρα η εκτέλεση της διεργασίας Master, δε θα επηρεαστεί ή θα εμποδιστεί ②. Στην επόμενη κλήση επικοινωνίας που θα έχει η διεργασία Master με τη συγκεκριμένη διεργασία Worker, πρώτα θα ελέγξει αν έχουν ληφθεί όλα τα μηνύματα σηματοδοσίας από όλες τις διεργασίες Master. Αν όχι, τότε θα περιμένει μέχρι αυτό να συμβεί. Σε αυτή τη διαδικασία της αναμονής των επιβεβαιώσεων της προηγούμενης κλήσης, αναφέρεται και ο αριθμός ①της εικόνας. Το διάγραμμα ροής της διαδικασίας παραλαβής δεδομένων φαίνεται στην Εικόνα 4.8.



Εικόνα 4.8: Διάγραμμα Ροής ft_recv, με Πολλαπλούς Master

Αντίθετα, κατά τη διαδικασία αποστολής δεδομένων από τη διεργασία Master προς κάποιον Worker, τα πράγματα διαφοροποιούνται ως έναν βαθμό. Δεδομένα θα στείλει μόνο η διεργασία Master προς τη διεργασία Worker, καθώς από τη στιγμή που δεν αλλάζει η κατάσταση των μεταβλητών της διεργασίας κατά την αποστολή, δεν είναι απαραίτητο να στείλουν όλες οι διεργασίες Master τα ίδια δεδομένα. Όταν παραλάβει τα δεδομένα η διεργασία Worker, επιβαρύνεται με τον επιπλέον φόρτο, να στείλει σε όλες τις επιπλέον διεργασίες Master, ένα μήνυμα σηματοδοσίας επιβεβαίωσης παρόλα αυτά. Για λόγους συγχρονισμού λοιπόν, όλες οι επιπλέον διεργασίες Master θα περιμένουν στην κλήση της αποστολή δεδομένων για την επιβεβαίωση από τη διεργασία Worker. Αυτές με τη σειρά τους, όπως και στην κλήση θα πρέπει να στείλουν τις επιβεβαιώσεις τους πίσω στη βασική διεργασία Master. Τις οποίες και θα λάβει το νήμα. Η βασική διεργασία θα περιμένει στην επόμενη κλήση της υλοποίησης μέχρι να ληφθούν όλες οι επιβεβαιώσεις ②Το διάγραμμα ροής της διαδικασίας αποστολής δεδομένων φαίνεται στην Εικόνα 4.9.



Εικόνα 4.9: Διάγραμμα Ροής ft_send, με Πολλαπλούς Master

4.5.2. Διαχείριση σφαλμάτων

Σε προηγούμενα κεφάλαια, αναλύθηκε ο τρόπος ανίχνευσης, αναφοράς και γενικότερης αντιμετώπισης των σφαλμάτων από τον μηχανισμό του FTMW, όταν αυτά εμφανίζονται στις διεργασίες Worker. Τα πράγματα γίνονται περισσότερο πολύπλοκα όμως, όταν το σφάλμα εμφανιστεί στη διεργασία Master. Σε αυτήν την περίπτωση θα πρέπει να υπάρξει μια γενικότερη μαζική επικοινωνία και συγχρονισμός μεταξύ των επιπλέον Master και των Worker.

Κάθε δευτερεύουσα διεργασία Master, σε κάθε επικοινωνία με μια διεργασία Worker, στέλνει επιπλέον και μια επιβεβαίωση στη βασική διεργασία Master. Η τελευταία θα ελέγξει αν έχει παραλάβει όλες τις επιβεβαιώσεις από τις δευτερεύουσες, στην επόμενη συνάρτηση που θα αναφερθεί στη συγκεκριμένη διεργασία Worker. Στο χρονικό διάστημα, μέχρι να συμβεί αυτό, είναι πολύ πιθανό οι επιβεβαιώσεις να έχουν παραληφθεί από το νήμα server της και έτσι να μη χρειαστεί να περιμένει καθόλου. Έτσι δεν εισάγουν και μεγάλο επιπλέον φόρτο, οι επιπλέον διεργασίες Master. Σε περίπτωση εμφάνισης σφάλματος, πρέπει να ληφθούν και περαιτέρω

ενέργειες. Παρακάτω περιγράφεται το πρωτόκολλο αντιμετώπισης σφαλμάτων με πολλαπλούς Masters για διάφορες περιπτώσεις σφαλμάτων. Το πρωτόκολλο θέτει μια βασική προϋπόθεση. Σε περίπτωση εμφάνισης σφάλματος σε κάποια διεργασία Worker, να μην εμφανιστεί σφάλμα στη βασική διεργασία Master, μέχρι την αποκατάσταση του σφάλματος.

Ανάλογα με το ποιό σημείο, έχει συμβεί το σφάλμα στη διεργασία Master, θα ακολουθηθεί και διαφορετική διαδικασία. Η πιο απλή περίπτωση, είναι όταν το σφάλμα της διεργασίας Master συμβεί σε κάποιο σημείο εκτέλεσης κώδικα της εφαρμογής και όχι κώδικα της υλοποίησης. Ουσιαστικά όταν δε βρίσκεται σε διαδικασία αποστολής ή παραλαβής και δεν εκκρεμεί κάποιο μήνυμα σηματοδοσίας από κάποια άλλη διεργασία Master. Σε αυτήν την περίπτωση, κάθε διεργασία Worker, θα ανακαλύψει το σφάλμα στην επόμενη κλήση της υλοποίησης, καθώς ούτως ή άλλως θα επικοινωνούσε με τη διεργασία Master. Στη συνέχεια θα αναθέσει στην επόμενη διεργασία Master της διάταξης που δεν έχει τερματιστεί από κάποιο σφάλμα, το ρόλο της βασικής διεργασίας Master. Όλες οι επικοινωνίες πλέον θα γίνονται με αυτή. Από την άλλη, μια δευτερεύουσα διεργασία Master με τη σειρά της θα ανιχνεύσει το σφάλμα της βασικής διεργασίας Master στην επόμενη κλήση της υλοποίησης. Ελέγχοντας τη θέση της αλλά και την κατάσταση των προηγούμενων από αυτή διεργασιών Master στη διάταξη, θα αποφασίσει αν θα είναι η επόμενη διεργασία βασική διεργασία Master. Σε περίπτωση σφάλματος σε κάποια δευτερεύουσα διεργασία Master, τα πράγματα είναι ακόμα πιο απλά, καθώς δε χρειάζεται να αλλάξει κάτι επί της ουσίας στην ροή της εκτέλεσης. Όταν και αν φτάσει η σειρά της πεσμένης διεργασίας να αναλάβει ρόλο Master, η επόμενη στη διάταξη διεργασία θα αναλάβει αυτόν τον ρόλο.

Για τα σφάλματα που μπορεί να συμβούν κατά τη διάρκεια κάποιας ανολοκλήρωτης διαδικασίας επικοινωνίας, ή καλύτερα κατά τη διάρκεια εκτέλεσης κώδικα της υλοποίησης, θα πρέπει να χωρίσουμε περιπτώσεις ανάλογα με το ποιό σημείο της επικοινωνίας ανιχνεύθηκε το σφάλμα, για να περιγράψουμε το πρωτόκολλο ανταλλαγής μηνυμάτων. Το πιο σημαντικό ζήτημα που εγείρεται με την παρουσία πολλών διεργασιών Master στην υλοποίηση, είναι οτι από τη στιγμή που όλες εκτελούν τον ίδιο κώδικα, το σφάλμα θα πρέπει να αναφερθεί στο ίδιο σημείο της

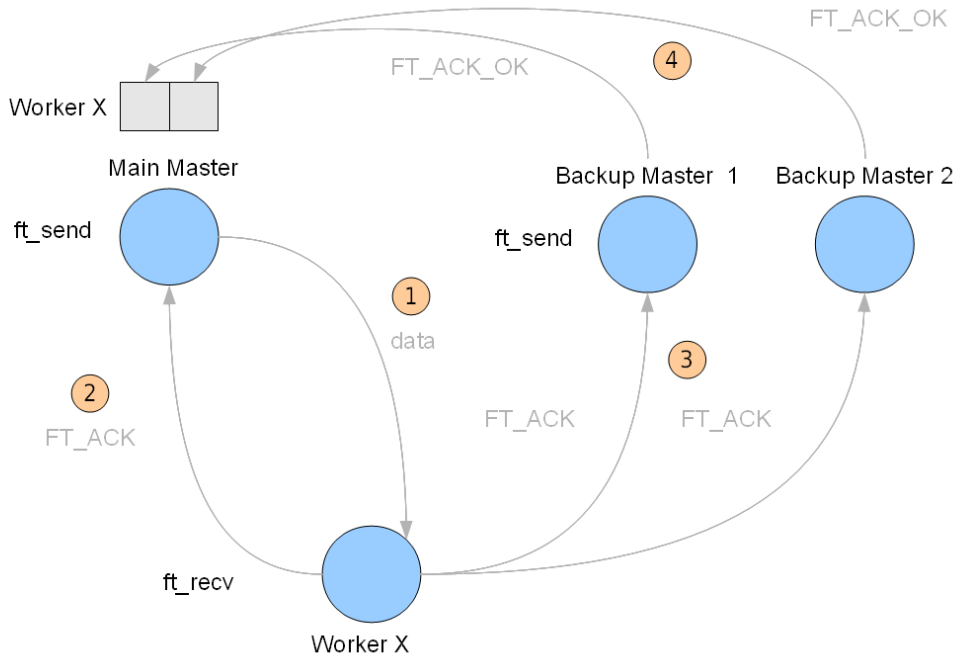
εφαρμογής για όλες. Για το λόγω αυτό, όταν το σφάλμα ανιχνευθεί από κάποια διεργασία Master, θα ξεκινήσει και ένας επιπλέον συγχρονισμός μεταξύ τους. Πιο συγκεκριμένα, δεν αρκεί από τη βασική διεργασία Master να ανιχνεύσει το σφάλμα, αλλά επωμίζεται και την επιπλέον αρμοδιότητα να ενημερώσει και τις άλλες διεργασίες Master οτι το ανίχνευσε. Κατά επέκταση, αν μια δευτερεύουσα διεργασία Master ανιχνεύσει το σφάλμα, θα πρέπει να αναμένει και το μήνυμα σηματοδοσίας που θα της έρθει από τη βασική διεργασία Master. Αν λάβει πρώτα το μήνυμα σηματοδοσίας για το σφάλμα, σημαίνει οτι ούτε η βασική διεργασία Master ολοκλήρωσε επιτυχώς την αντίστοιχη κλήση και άρα πρέπει να αναφερθεί από εκείνη την κλήση της υλοποίησης το σφάλμα. Παρακάτω αναλύονται οι δύο περιπτώσεις, αποστολής και παραλαβής δεδομένων, που μπορεί να εμφανιστεί σφάλμα στα μισά της κλήσης της συνάρτησης, δηλαδή κατά των κώδικα της υλοποίησης. Οι υπόλοιπες περιπτώσεις, είναι τετριμμένες και έχουν εξηγηθεί προηγουμένως, ή αποτελούν προέκταση των παρακάτω περιπτώσεων.

1η Περίπτωση: Η διεργασία Worker εμφανίζει σφάλμα, αφού έχει λάβει τα δεδομένα του από τη διεργασία Master

Το πρόβλημα σε αυτήν την περίπτωση είναι όπως αναφέρθηκε προηγουμένως, η ετεροχρονισμένη αναφορά του σφάλματος στην εφαρμογή από τις διεργασίες Master. Από τη στιγμή που η βασική διεργασία Master έχει ολοκληρώσει την κλήση Χ (σε αυτήν την περίπτωση είναι κλήση αποστολής δεδομένων) με τη συγκεκριμένη διεργασία Worker και στη συνέχεια η τελευταία παρουσιάσει σφάλμα, αυτό θα αναφερθεί ετεροχρονισμένα στις υπόλοιπες διεργασίας Master. Για τη βασική θα αναφερθεί στην κλήση συνάρτησης της υλοποίησης (Χ + 1) στην οποία θα γίνει η επόμενη αναφορά στη συγκεκριμένη διεργασία Worker, ενώ για τις υπόλοιπες διεργασίες Master στις οποίες η εσφαλμένη διεργασία Worker, δεν πρόλαβε να στείλει δεδομένα πριν εμφανίσει το σφάλμα, στην κλήση Χ.

Για να υπερκεραστεί αυτό το πρόβλημα, προσθέσαμε έναν επιπλέον επικοινωνιακό συγχρονισμό με μηνύματα σηματοδοσίας της υλοποίησης. Οι υπόλοιπες διεργασίες Master, μόλις ανακαλύψουν το σφάλμα της διεργασίας Worker, δε θα το αναφέρουν αμέσως στο επίπεδο της εφαρμογής. Σε πρώτο βήμα, θα ενημερώσουν τη βασική

διεργασία με το αντίστοιχο μήνυμα επιβεβαίωσης σφάλματος. Σε δεύτερο βήμα, θα περιμένουν την απόκριση από τη βασική διεργασία Master. Αν η επιβεβαίωση είναι θετική (FT_ACK_OK), τότε σημαίνει ότι η αποστολή δεδομένων έγινε επιτυχώς και αρα δεν πρέπει να αναφέρουν το σφάλμα. Σε αντίθετη περίπτωση (FT_ACK_DOWN), θα αναφέρουν το σφάλμα στο επίπεδο της εφαρμογής. Τα βήματα αριθμημένα με τη χρονική σειρά που συμβαίνουν φαίνονται στην Εικόνα 4.10.



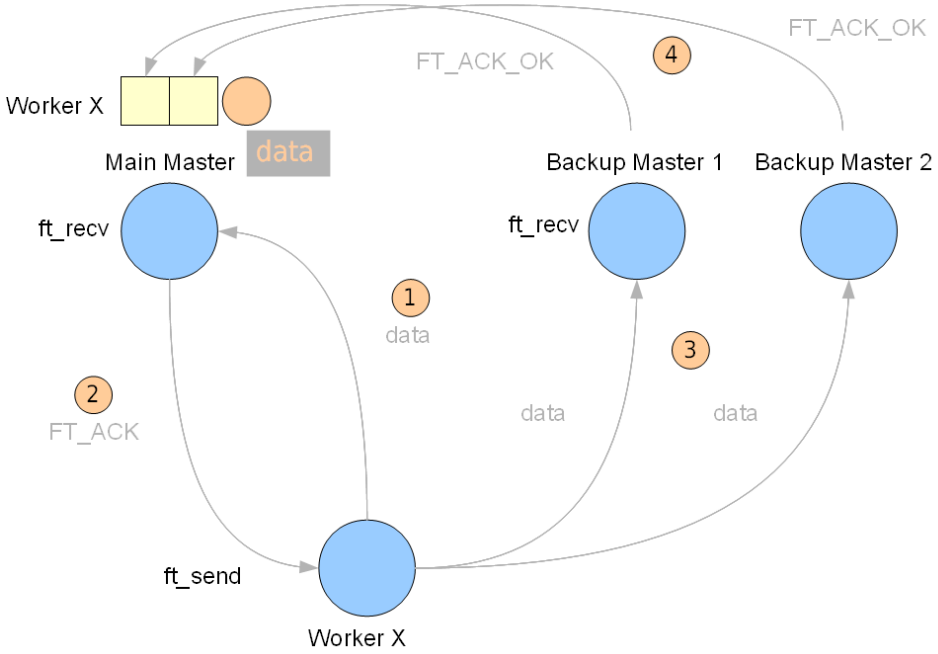
Εικόνα 4.10: Βήματα Διαδικασίας Αποστολής Δεδομένων, με Πολλαπλούς Master

2η Περίπτωση: Η διεργασία Worker εμφανίζει σφάλμα, αφού έχει στείλει τα δεδομένα του στη διεργασία Master

Σε αυτήν την περίπτωση, εκτός από το πρόβλημα της ετεροχρονισμένης αναφοράς του σφάλματος στο επίπεδο της εφαρμογής, εμφανίζεται και το πρόβλημα του οτι τα δεδομένα που έστειλε η διεργασία Worker που εμφάνισε το σφάλμα, μπορεί να είναι διαθέσιμα σε κάποιες διεργασίες Master και όχι σε κάποιες άλλες. Μπορεί για παράδειγμα να έχει στείλει τα δεδομένα στη βασική διεργασία Master και το σφάλμα να εμφανίστηκε κατά τη διάρκεια του κώδικα της υλοποίησης, όπου βρισκόταν σε διαδικασία αποστολής δεδομένων στις υπόλοιπες διεργασίες Master.

Στην υλοποίηση έχουμε εξασφαλίσει οτι αν τουλάχιστον η βασική διεργασία Master έχει λάβει τα δεδομένα από μια διεργασία Worker που δεν πρόλαβε να τα στείλει στις υπόλοιπες λόγω σφάλματος, είναι δυνατή η διάθεσή τους και στις υπόλοιπες. Για αυτόν τον σκοπό, η βασική διεργασία Master, με το που λάβει τα δεδομένα, τα αποθηκεύει σε μια δομή. Αν οι υπόλοιπες διεργασίες Master, ανιχνεύσουν το σφάλμα, δε θα το αναφέρουν κατευθείαν στην εφαρμογή, αλλά θα περιμένουν όπως είπαμε και προηγουμένως και το αντίστοιχο μήνυμα σφάλματος από την βασική διεργασία Master. Αν η τελευταία έχει λάβει τα δεδομένα από τη διεργασία Worker, πρώτα θα στείλει τα δεδομένα και στη συνέχεια το μήνυμα σφάλματος. Αυτό έχει ως αποτέλεσμα, η δευτερεύουσα διεργασία Master, αφού λάβει πρώτα τα δεδομένα, να ολοκληρώσει την κλήση παραλαβής. Στη συνέχεια της εκτέλεσης όταν ξανά αναφερθεί, στη εσφαλμένη διεργασία Worker, θα ανιχνεύσει πάλι το σφάλμα της αλλά θα μπορεί να λάβει υπόψιν της και το μήνυμα σηματοδοσίας αναφερόμενο στο σφάλμα της διεργασίας Worker, που έλαβε από τη βασική διεργασία Master στην προηγούμενη κλήση. Με αυτήν την συνθήκη, πλέον είναι σε θέση να αναφέρει το σφάλμα στο επίπεδο της εφαρμογής. Αξίζει να σημειωθεί οτι τα δεδομένα αποθηκεύονται από τη βασική διεργασία Master, μέχρι την επόμενη κλήση που θα αναφερθεί στην εσφαλμένη διεργασία Worker. Αυτό συμβαίνει, γιατί για να ολοκληρωθεί αυτή η επόμενη κλήση, θα πρέπει η βασική διεργασία Master, να έχει λάβει πρώτα όλα τα απαραίτητα μηνύματα σηματοδοσίας από τις υπόλοιπες διεργασίες Master, που σημαίνει οτι έχει ολοκληρωθεί η προηγούμενη διαδικασία και άρα τα παλιά αποθηκευμένα δεδομένα δε χρειάζονται. Τα βήματα αριθμημένα με τη χρονική σειρά που συμβαίνουν φαίνονται στην Εικόνα 4.11.

Αναλυτικότερη περιγραφή της λειτουργίας και των βημάτων της υλοποίησης του πρωτοκόλλου, γίνεται στο Παραρτημα. Εκεί περιγράφεται σε μεγαλύτερο βάθος η συμπεριφορά του πρωτοκόλλου για όλες τις δυνατές περιπτώσεις εμφάνισης σφάλματος σε οποιοδήποτε σημείο της εκτέλεσης.



Εικόνα 4.11: Βήματα Διαδικασίας Παραλαβής Δεδομένων, με Πολλαπλούς Master

ΚΕΦΑΛΑΙΟ 5. ΠΕΙΡΑΜΑΤΙΚΑ ΑΠΟΤΕΛΕΣΜΑΤΑ

Σε αυτό το κεφάλαιο παρουσιάζουμε πειραματικά αποτελέσματα της εκτέλεσης του συστήματος μας, χρησιμοποιώντας μια εφαρμογή αριθμητικής βελτιστοποίησης. Μια εφαρμογή που εύκολα μπορεί να διαχωριστεί σε μικρότερες κατανεμημένες και ανεξάρτητες εργασίες που δρομολογούνται στις διεργασίες Worker χρησιμοποιώντας τη μέθοδο Multistart. Επίσης γίνεται και περιγραφή των πειραμάτων που ακολουθήσαμε για να ελέγξουμε και να επιδείξουμε την ορθότητα της υλοποίησης, σε περιπτώσεις σφαλμάτων. Οι εκτελέσεις των πειραμάτων έλαβαν χώρα σε μια συστάδα υπολογιστών με 16 κόμβους Sun-Fire x4100 με Gigabit Ethernet, όπου κάθε κόμβος είχε δύο διπύρηνους επεξεργαστές AMD Opteron 275 CPUs. Το λειτουργικό σύστημα σε κάθε κόμβο ήταν Linux 2.6, GCC 4.3.

5.1. Πειράματα Ορθότητας

Αρχικά θα πρέπει να αναφέρουμε οτι την υλοποίηση του FTMW, την εξετάσαμε με μια σειρά τυπικών εφαρμογών Master-Worker, που αναπτύξαμε για να ελέγξουμε όλες τις πιθανές περιπτώσεις σφαλμάτων που μπορούν να εμφανιστούν, τόσο για διαφορετικές διεργασίες, όσο και για διαφορετικά σημεία του κώδικα. Η εκτέλεση με τα προγράμματα αυτά συμπεριλάμβανε και επιπλέον διεργασίες Master, ώστε να επιδείξουμε το πρωτόκολλο με το οποίο επικοινωνούν οι διεργασίες κυρίως όταν στην υλοποίηση συμμετέχουν και οι επιπλέον διεργασίες Master. Επιλέγαμε μια

διεργασία είτε ήταν Worker, είτε ήταν Δευτερεύουσα Master, είτε ακόμα και η βασική Master και την τερματίζαμε σε διάφορα σημεία τόσο του κώδικα της εφαρμογής, όσο και σε σημεία του κώδικα του ίδιου του μηχανισμού του FTMW.

Ακολουθούν αναλύτικά και διαχωρισμένα τα πειράματα ως προς το είδος της διεργασίας που επιλέγαμε να τερματίζουμε. Δεν παρουσιάζουμε κάτι μετρήσιμο, αλλά γίνεται περισσότερο μια ποιοτική περιγραφή των αποτελεσμάτων.

5.1.1. Τερματισμός Διεργασίας Worker

Σε αυτά τα πειράματα επιλέγαμε τυχαία μια ή περισσότερες διεργασίες Worker, τις οποίες και τερματίζαμε. Ανάλογα με το σημείο του κώδικα που τερματίζαμε την κάθε διεργασία, περιμέναμε και μια αντίστοιχη συνάρτηση της υλοποίησης να αναφέρει το σφάλμα στον κώδικα των διεργασιών Master. Σημεία που επιλέξαμε να τερματίζουμε κάποια από τις διεργασίες αυτές, είναι πριν η διεργασία στείλει αποτελέσματα στη διεργασία Master και πριν η διεργασία λάβει αποτελέσματα από τη διεργασία Master και σε διάφορα σημεία εντός του κώδικα του FTMW.

Σκοπός του πειράματος αυτού, ήταν να διαπιστώσουμε το κατά πόσο οι διεργασίες Master, αναφέρουν το σφάλμα στο ίδιο σημείο κώδικα, ή καλύτερα αναφέρουν το σφάλμα από την ίδια συνάρτηση της υλοποίησης, στον προγραμματιστή.

Σε όλες τις περιπτώσεις σφαλμάτων, είτε πρόκειται για σφάλματα εντός του κώδικα της εφαρμογής της διεργασίας Worker, είτε για σφάλματα εντός του κώδικα της βιβλιοθήκης, οι διεργασίες Master ανταποκρίνονταν ορθώς. Μπορεί ο χρόνος στον οποίο ανίχνευαν τον τερματισμό της διεργασίας να διέφερε, αλλά το σημαντικό ήταν ότι ανέφεραν το σφάλμα στο επίπεδο της εφαρμογής από την ίδια συνάρτηση βιβλιοθήκης (από το ίδιο σημείο του κώδικα).

5.1.2. Τερματισμός Βασικής Διεργασίας Master

Με την ίδια λογική που τερματίζαμε τις διεργασίες Worker, στο πείραμα της Ενότητας 5.1.1, επιλέγοντας διάφορα σημεία του κώδικα της εφαρμογής αλλά και του κώδικα της βιβλιοθήκης του FTMW, τερματίζαμε και τη βασική διεργασία Master.

Σκοπός του πειράματος, ήταν να διαπιστώσουμε τη συνολική συμπεριφορά των υπόλοιπων διεργασιών που συμμετέχουν. Όταν το σφάλμα συμβαίνει στην πλευρά της διεργασίας Master, όλες οι διεργασίες Worker, αφού ανιχνεύσουν το σφάλμα, επιλέγουν την επόμενη από τις δευτερεύουσες διεργασίες Master να πάρει τη θέση της. Η δευτερεύουσα διεργασία Master που θα λάβει το ρόλο της βασικής, είναι σε θέση να συνεχίσει από το σωστό σημείο της εκτέλεσης, καθώς σε όλη τη διάρκεια της εφαρμογής εκτελούσε τον ίδιο κώδικα με τη βασική και παραλάμβανε τα ίδια δεδομένα από τις διεργασίες Worker. Πολύ περισσότερο για τα σφάλματα που συνέβαιναν κατά τη διάρκεια εκτέλεσης κώδικα της υλοποίησης, η δευτερεύουσα διεργασία Master αναλαμβάνει το ρόλο της βασικής μόλις ολοκληρωθεί η συνάρτηση της υλοποίησης στην οποία συναίβει το σφάλμα. Όταν η διεργασία Worker, επικοινωνεί με τη βασική διεργασία Master, στέλνει επιπλέον και την πληροφορία του ποιά διεργασία Master ήταν σε εκείνο το χρονικό σημείο η βασική. Έτσι η επόμενη από τις δευτερευουσες διεργασίες Master, μπορεί να γνωρίζει αν στο σημείο της υλοποίησης που βρίσκεταιι, μπορεί να λάβει το ρόλο της βασικής διεργασίας Master, ή θα πρέπει να περιμένει στην επόμενη κλήση της υλοποίησης.

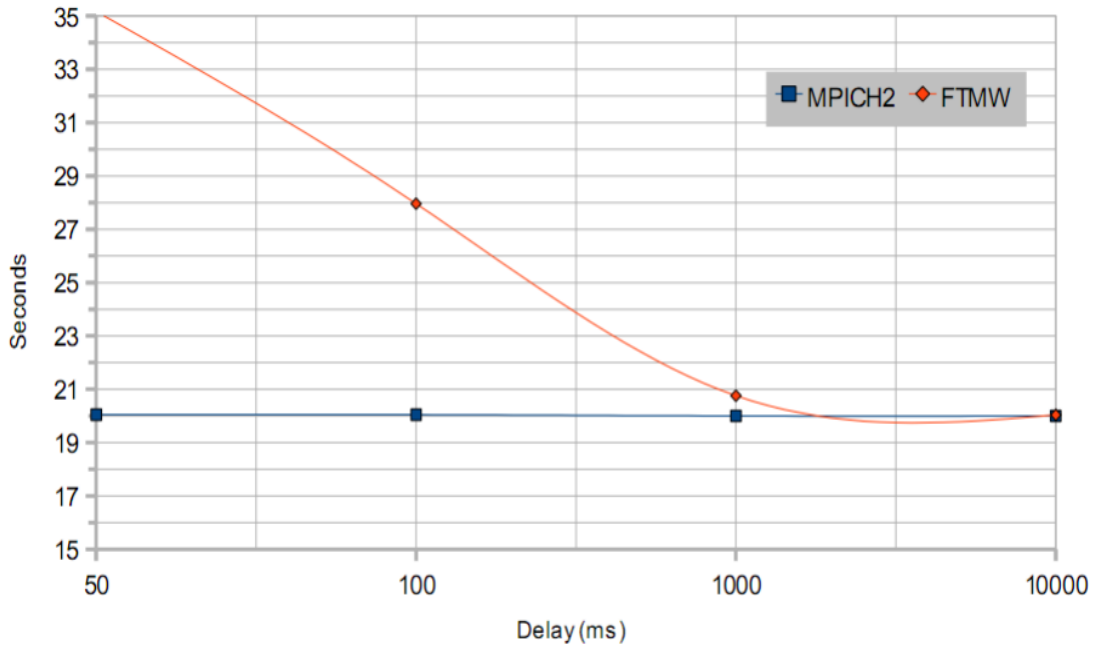
5.1.3. Τερματισμός Δευτερεύουσας Διεργασίας Master

Και εδώ τα πειράματα έγιναν με την ίδια λογική όπως και παραπάνω. Τα πράγματα όμως, είναι πιο απλά στην περίπτωση όπου τερματίζουμε μια δευτερεύουσα διεργασία Master. Το σφάλμα, ανιχνεύεται από όλες τις διεργασίες χωρίς να χρειάζονται περαιτέρω διαδικασίες. Αυτό που αλλάζει επί της ουσίας στη ροή της εκτέλεσης είναι οτι οι διεργασίες Worker, στέλνουν σε μια λιγότερη διεργασία Master, δεδομένα ή μηνύματα σηματοδοσίας της υλοποίησης.

5.2. Επιπλέον Φόρτος Εκτέλεσης

Το FTMW προσφέρει την ίδια λειτουργικότητα με το πρώτυπο του MPI. Προσφέρει όμως και επιπλέον λειτουργικότητες, είτε διαφανώς, είτε αδιαφανώς από τον προγραμματιστή, ώστε να εξασφαλίσει ανοχή σφαλμάτων κατά την εκτέλεση. Για να το πετύχει αυτό, αναπόφευκτα εισάγει ένα επιπλέον φόρτο εκτέλεσης στην εφαρμογή. Με το πείραμα αυτής της ενότητας, θέλαμε να μετρήσουμε το μέγεθος αυτού του επιπλέον φόρτου.

Αναπτύξαμε μια απλή εφαρμογή Master-Worker, όπου η διεργασία Master μοιράζει εργασίες στις διεργασίες Worker και αναμένει από αυτές αποτελέσματα. Την εργασία την προσομοιώνουμε με μια τεχνητή καθυστέρηση που εισάγαμε στον κώδικα. Αυξομειώναμε αυτήν την τεχνητή καθυστέρηση, ουσιαστικά το μέγεθος των εργασιών που θα εκτελούσαν οι διεργασίες Master, ώστε να διαπιστώσουμε τη συμπεριφορά της απόδοσης του FTMW για διάφορα επίπεδα παραλληλοποίησης.



Εικόνα 5.1: Σύγκριση Απόδοσης για Διάφορα Μεγέθη Εργασιών

Παρουσιάζουμε την εκτέλεση του πειράματος έχοντας σταθερό συνολικό χρόνο των υπολογισμών τα 80 secs, μεταβάλλοντας το μέγεθος των εργασιών και άρα και το συνολικό αριθμό των εργασιών. Ενδεικτικά χρησιμοποιήσαμε 4 διεργασίες Worker. Στον άξονα X φαίνεται το μέγεθος της καθυστέρησης σε ms και στον άξονα Y ο συνολικός χρόνος εκτέλεσης της εφαρμογής σε seconds.

Παρατηρούμε στην Εικόνα 5.1, ότι το MPICH εμφανίζει μια σταθερή απόδοση και μάλιστα κοντά στο βέλτιστο χρόνο εκτέλεσης της εφαρμογής (20 δευτερόλεπτα με συνολικό χρόνο δουλειάς 80 δευτερολέπτων διαμοιρασμένο σε 4 διεργασίες Worker). Αντίθετα το FTMW για μικρές εργασίες της τάξης των 50 ms, εισάγει μεγάλο επιπλέον φόρτο της τάξης του 90%. Αυτός ο φόρτος βέβαια όσο μεγαλώνουμε το μέγεθος των εργασιών και άρα μειώνουμε το συνολικό αριθμό τους, μειώνεται αναλόγως. Στο σημείο όπου έχουμε εργασίες της τάξεος των 10000ms, παρατηρούμε ότι εμφανίζει αναλόγως καλή συμπεριφορά όπως και το MPICH.

5.3. Εφαρμογή Βελτιστοποίησης

Πειραματιστήκαμε με μια εφαρμογή βελτιστοποίησης, απλή στη λογική και εύκολα διασπάσιμη σε μικρότερες εργασίες τις οποίες η διεργασία Master μπορεί να δρομολογήσει στις διάφορες διεργασίες Worker. Τέτοιες εφαρμογές βελτιστοποίησης συχνά χρήζουν αναγκαιότητας εκτέλεσης σε μεγάλο αριθμό μηχανημάτων λόγω της χρονοβόρας διαδικασίας για την ανεύρεση ενός βέλτιστου είτε τοπικού, είτε ολικού σημείου στο χώρο αναζήτησης.

Η αρχική εφαρμογή μας, είναι μια απλή μέθοδος τοπικής αριθμητικής βελτιστοποίησης που βασίζεται σε μια σειριακή αναζήτηση ενός βέλτιστου τοπικού σημείου, με δυναμικά μεταβαλλόμενα το βήμα και την κατεύθυνση αναζήτησης. Εύκολα αυτή η μέθοδος με την προσάρτηση της μεθόδου Multistart [29], μπορεί να μετατραπεί ως ένα βαθμό σε μέθοδο ολικής βελτιστοποίησης. Σύμφωνα με τη μέθοδο αυτή, η μέθοδος τοπικής βελτιστοποίησης εκτελείται ανεξάρτητα για κάθε ένα σημείο που έχει παραχθεί τυχαία από μια ομοιόμορφη κατανομή μέσα στο χώρο αναζήτησης.

Αποτέλεσμα αυτού είναι να παραχθούν πολλά τοπικά βέλτιστα σημεία, όχι απαραίτητα μοναδικά γιατί είναι πολύ πιθανό τυχαία αρχικά σημεία να καταλήγουν στο ίδιο τοπικό βέλτιστο, από τα οποία μπορεί να επιλεχθεί το βέλτιστο. Είναι μια μέθοδος που μπορεί εύκολα να παραληλοποιηθεί, παράγει καλά αποτελέσματα στη γενική περίπτωση και δεν εισάγει επιπλέον σημαντικό κόστος σε σχέση με το κόστος εκτέλεσης της μεθόδου τοπικής βελτιστοποίησης. Στην αντικειμενική συνάρτηση της εφαρμογής, εισάγαμε μια μεταβλητή τεχνητή καθυστέρηση που προσομειώνει το μέγεθος του προβλήματος βελτιστοποίησης. Πειραματιστήκαμε με διάφορα μεγέθη καθυστέρησης, προκειμένου να διαπιστωθεί η συμπεριφορά για διάφορες αντικειμενικές συναρτήσεις.

Στην εφαρμογή μας λοιπόν, η διεργασία Master αρχικά ενημερώνει όλες τις διεργασίες Worker με τις απαραίτητες παραμέτρους του αλγορίθμου. Στη συνέχεια παράγει τα τυχαία σημεία εκκίνησης τα οποία από μόνα τους αποτελούν μια εργασία η οποία μπορεί να δρομολογηθεί σε κάποια διαθέσιμη διεργασία Worker. Ο αλγόριθμος της τοπικής βελτιστοποίησης είναι σειριακός και άρα αποτελεί από μόνος του μια εργασία που θα εκτελεστεί από μόνο μια διεργασία. Στη συνέχεια τα αποτελέσματα στέλνονται πίσω στη διεργασία Master και όταν δεν υπάρχουν πλέον άλλες δουλειές (σημεία) να αποσταλούν, η διεργασία Master στέλνει την απαραίτητη πληροφορία για να τερματίσουν οι διεργασίες Worker. Ο αλγόριθμος Multistart έχει και ένα τελευταίο στάδιο, αυτό της επιλογής του βέλτιστου σημείου από όλα τα βέλτιστα τοπικά που έχουν παραχθεί. Το στάδιο αυτό αποτελεί μια σειριακή διαδικασία που δε συμπεριλαμβάνεται στο εύρος των μετρήσεων.

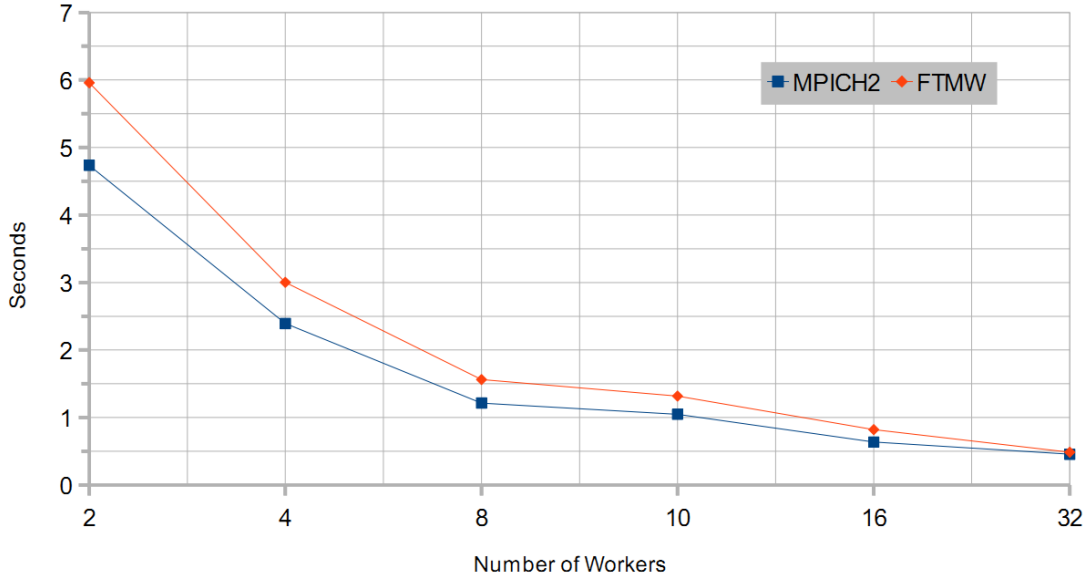
5.3.1. Σύγκριση με συμβατικές υλοποιήσεις MPI

Σε αυτό το πείραμα, θέλαμε να δείξουμε πόσο επιπλέον φόρτο εκτέλεσης εισάγει το FTMW σε σχέση με μια συμβατική υλοποίηση του MPI όπως είναι αυτή του MPICH2, που δεν προσφέρει ανοχή σφαλμάτων με ένα πιο ρεαλιστικό πρόβλημα. Και με τις δύο υλοποιήσεις εκτελέσαμε την ίδια εφαρμογή, αλλάζοντας μόνο τη διεπαφή των συναρτήσεων αντίστοιχα για κάθε υλοποίηση.

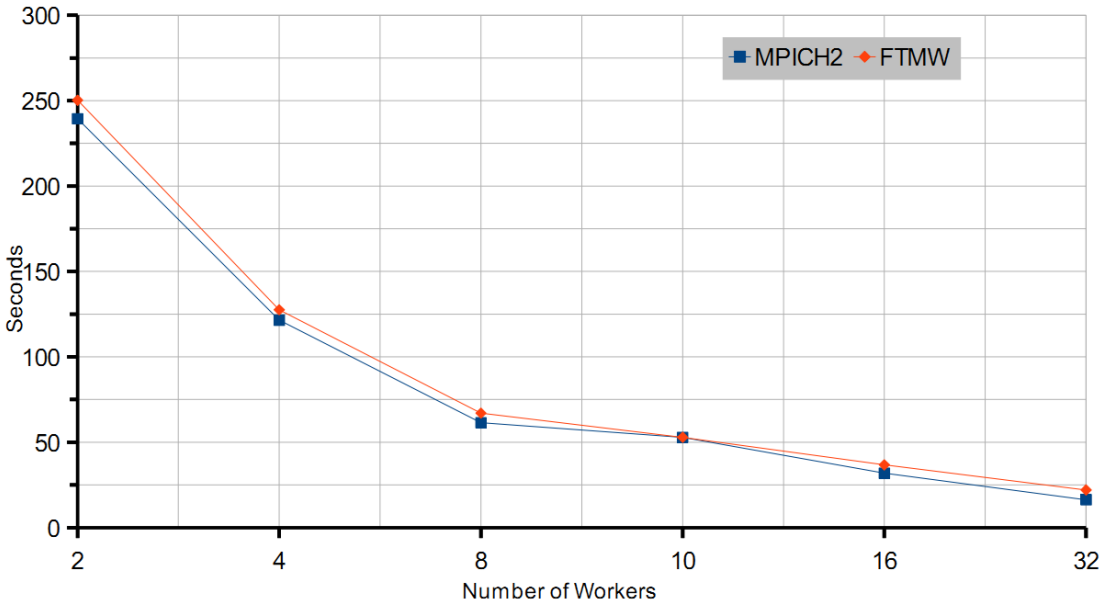
Στην εφαρμογή χρησιμοποιήσαμε 10 μεταβλητές και εισάγαμε και μια επιπλέον τεχνητή καθυστέρηση σε κάθε υπολογισμό της αντικειμενικής συνάρτησης. Πειραματιστήκαμε για διάφορες τιμές αυτής της καθυστέρησης, εξομειώνοντας με αυτόν τον τρόπο διάφορα μεγέθη προβλήμα, που κυμένονταν από 1ms μέχρι και 100ms. Εκτελέσαμε την εφαρμογή για διάφορα πλήθη διεργασιών Worker. Από το πλήθος των 10 Worker και κάτω, στο σύστημα που εκτελέσαμε τα πειράματα, μπορούσαμε να διασφαλίσουμε ότι η κάθε μια διεργασία Worker, θα τοποθετούνταν σε διαφορετικό κόμβο. Για μεγαλύτερο πλήθος, περισσότερες από μια διεργασίες Worker, θα τοποθετούνταν στον ίδιο κόμβο. Επίσης χρησιμοποιήσαμε και διαφορετικά πλήθη σημείων, για καθένα από τα οποία όπως αναφέραμε παραπάνω, εκκινεί και μια διαδικασία βελτιστοποίησης που δρομολογείται σε κάποια διεργασία Worker.

Στα πειράματα αυτά, δεν τερματίζουμε κάποια διεργασία Worker και δε χρησιμοποιούμε γενικότερα κάποια από τις λειτουργίες ανοχής σφαλμάτων της υλοποίησης. Η παρουσία επιπλέον διεργασιών Master, δεν προσθέτει σημαντικό επιπλέον φόρτο στην εκτέλεση, που θα επηρέαζε τη γενικότερη απόδοση και έτσι δεν συμπεριλήφθηκαν στα διαγράμματα. Αυτό συμβαίνει γιατί όπως είδαμε και από το πρωτόκολλο επικοινωνιών με τις πολλαπλές διεργασίες Master, κάθε μια από αυτές στέλνει σε κάθε επικοινωνία με τη διεργασία Worker, μια επιπλέον επιβεβαίωση στη βασική διεργασία. Η τελευταία, θα ελέγξει για την παραλαβή όλων των επιβεβαιώσεων στην επόμενη κλήση που θα αναφερθεί στη συγκεκριμένη διεργασία Worker. Στο χρονικό διάστημα μέχρι να συμβεί αυτό, είναι πιθανό όλες οι επιβεβαιώσεις να έχουν ληφθεί και αρα να μην σπαταλίσει χρόνο περιμένοντας.

Στα διαγράμματα της Εικόνα 5.3 και Εικόνα 5.4, παραθέτουμε τα αποτελέσματα με χρονική καθυστέρηση 100ms, επιπλέον όμως και μια ενδεικτική γραφική παράσταση με 64 βελτιστοποιήσεις στην οποία χρησιμοποιήσαμε μικρότερη καθυστέρηση της τάξης του 1ms (Εικόνα 5.2). Στον άξονα Χ υπάρχει ο αριθμός των διεργασιών Worker και στον άξονα Υ ο χρόνος εκτέλεσης σε δευτερόλεπτα.



Εικόνα 5.2: Χρόνος Εκτέλεσης για 64 Βελτιστοποιήσεις (1ms)

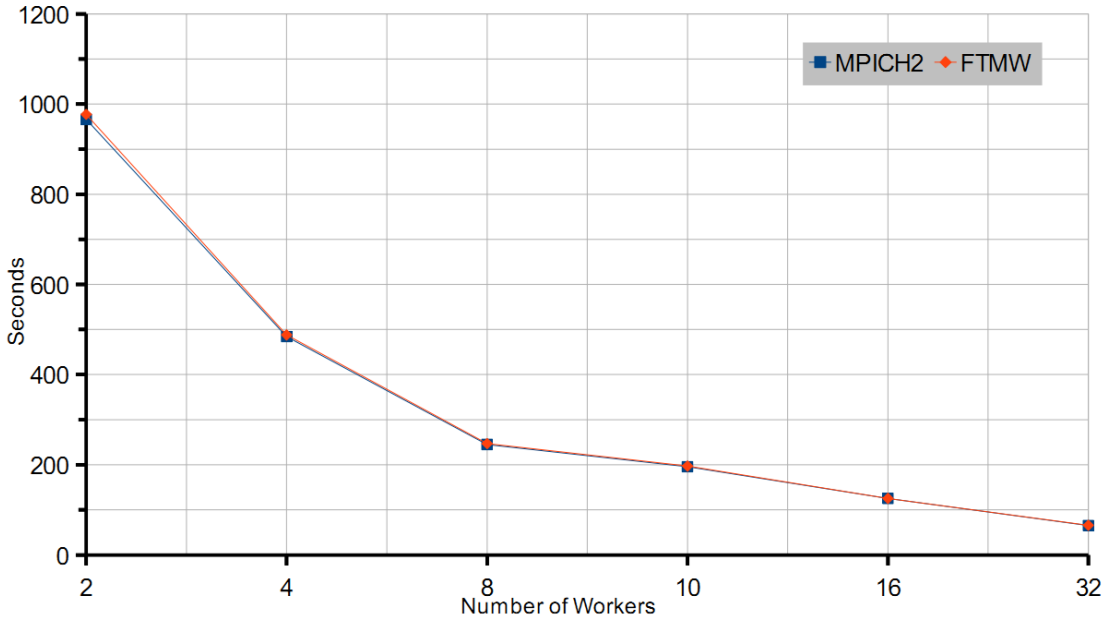


Εικόνα 5.3: Χρόνος Εκτέλεσης για 64 Βελτιστοποιήσεις (100ms)

Παρατηρούμε ότι για μικρότερο μέγεθος συνολικής δουλειάς Εικόνα 5.2: Χρόνος Εκτέλεσης για 64 Βελτιστοποιήσεις (1ms) η διαφορά μεταξύ των δύο υλοποιήσεων είναι μεγαλύτερη από ότι στην Εικόνα 5.3 όπου η καθυστέρηση αυξανόταν. Αυτό συμβαίνει γιατί η ποσότητα χρόνου υπολογισμών είναι πολύ μεγαλύτερη από την ποσότητα χρόνου των επικοινωνίων, η οποία είναι μεγαλύτερη για το FTMW.

Πιο συγκεκριμένα, για 2 διεργασίες Worker, η υλοποιήση του FTMW παρουσιάζει επιπλέον επιβάρυνση στο χρόνο εκτέλεσης της εφαρμογής της τάξης του 23% για μικρότερες εργασίες των 1ms (Εικόνα 5.2) ενώ για εργασίες των 100ms (Εικόνα 5.3), η επιβάρυνση φτάνει περίπου στο 5%. Παρόλα αυτά όσο αυξάνουμε τις διεργασίες Worker που συμμετέχουν στους υπολογισμούς, αυτή η επιβάρυνση, κυρίως για τις εργασίες του 1ms, μικραίνει σταδιακά.

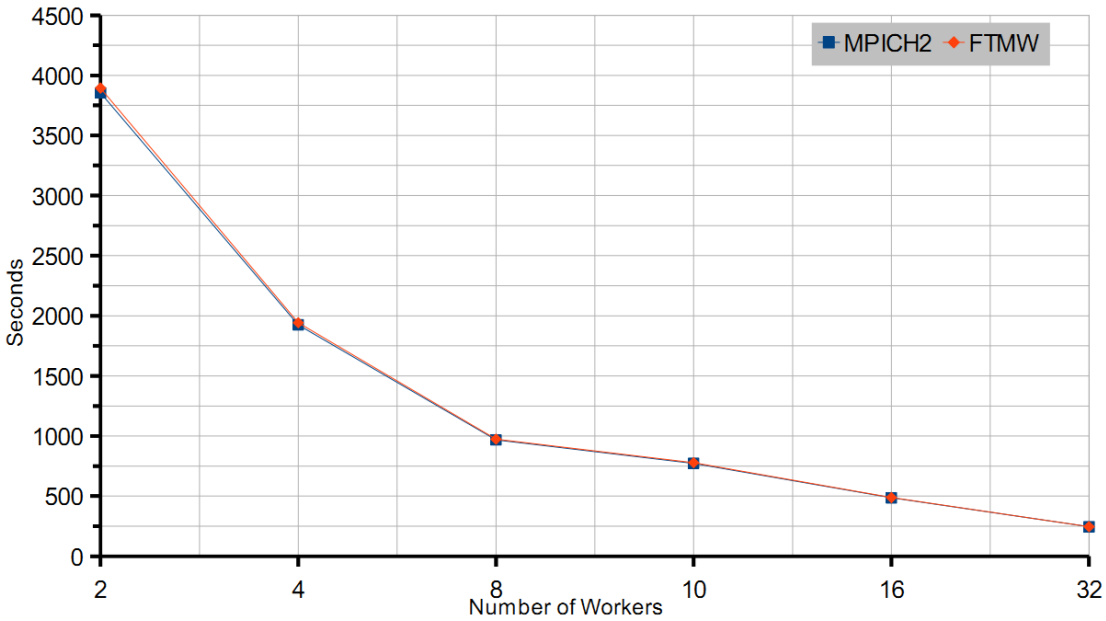
Στις επόμενες γραφικές παραστάσεις, αυξάνουμε και το συνολικό αριθμό των βελτιστοποιήσεων στο πρόβλημα.



Εικόνα 5.4: Χρόνος Εκτέλεσης για 256 Βελτιστοποιήσεις (100ms)

Παρατηρούμε στα διαγράμματα των Εικόνων Εικόνα 5.3, Εικόνα 5.4 και Εικόνα 5.5, οτι συγκριτικά η υλοποίηση του FTMW δεν ξεφέυγει από την απόδοση που επιτυγχάνει το MPICH2. Το FTMW, εισάγει επιπλέον χρόνο στις αντίστοιχες κλήσεις επικοινωνίας που προσφέρει, καθώς όπως είπαμε είναι όλες εμποδιστικές, που σημαίνει οτι ακόμα και όταν μια διεργασία που συμμετέχει στους υπολογισμούς θέλει να στείλει δεδομένα, θα πρέπει να περιμένει να λάβει επιβεβαίωση. Βέβαια αυτός ο επιπλέον χρόνος, είναι πολύ μικρότερος σε σχέση με το χρόνο που οι διεργασίες

Worker εκτελούν μια εργασία. Πολύ περισσότερο όσο αυξάνονται τα σημεία βελτιστοποίησης, δηλαδή οι εργασίες που πρέπει να δρομολογηθούν, αυτή η διαφορά μεταξύ των υλοποιήσεων μειώνεται περισσότερο αναλογικά με το συνολικό χρόνο εκτέλεσης.

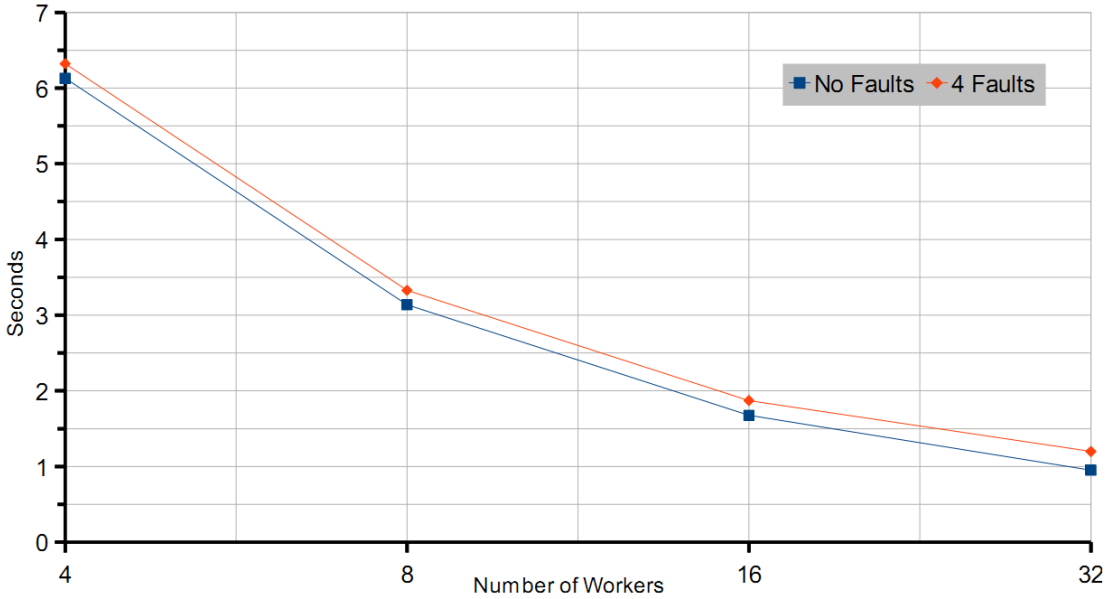


Εικόνα 5.5: Χρόνος Εκτέλεσης για 1024 Βελτιστοποιήσεις (100ms)

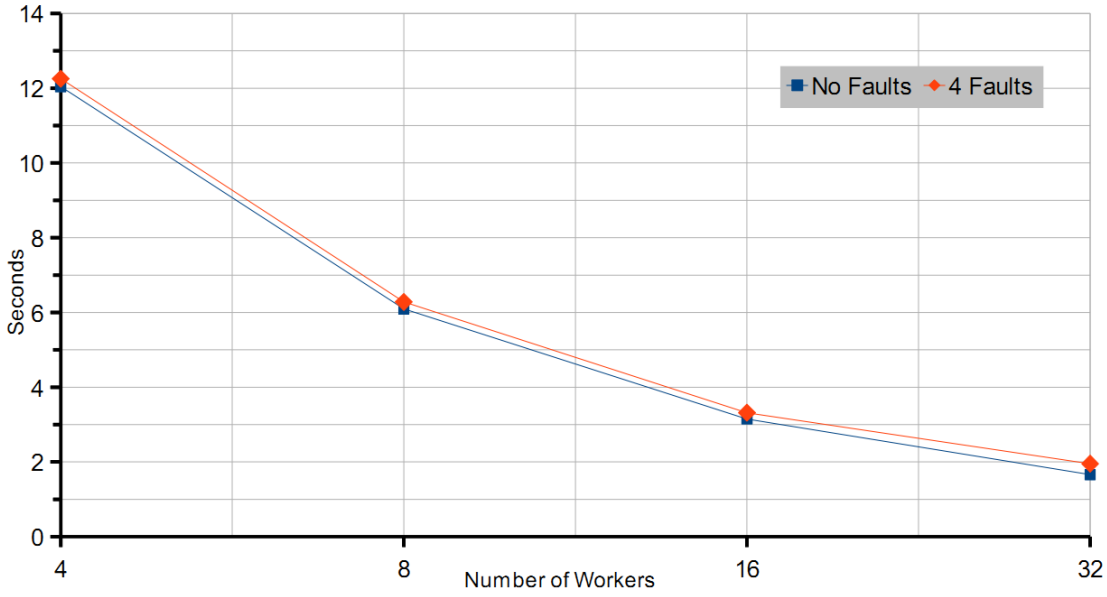
5.3.2. Εμφάνιση Σφαλμάτων κατά την Εκτέλεση

Με αυτό το πείραμα θέλαμε να μετρήσουμε το πόσο επιπλέον χρόνο εισάγει στο συνολικό χρόνο εκτέλεσης της εφαρμογής, η επανεκκίνηση διεργασιών Worker, που εμφανίζουν σφάλμα και τερματίζουν. Στο συγκεκριμένο πείραμα, επιλέξαμε 4 διεργασίες Worker, τις οποίες και τερματίζαμε σε διάφορα σημεία του κώδικα. Στη συνέχεια αφού η διεργασία Master ανίχνευε το σφάλμα, επανεκκινούσε την αντίστοιχη διεργασία Worker και έστελνε τα απαραίτητα δεδομένα σε αυτήν.

Και σε αυτήν την περίπτωση, στην εφαρμογή χρησιμοποιήσαμε 10 μεταβλητές και εισάγαμε και μια επιπλέον τεχνητή καθυστέρηση στην αντικειμενική συνάρτηση της μεθόδου της τάξης του 1ms. Εκτελέσαμε την εφαρμογή για διάφορα πλήθη διεργασιών Worker. Επίσης και για διαφορετικά πλήθη σημείων, για καθένα από τα οποία όπως αναφέραμε παραπάνω, εκκινεί και μια εργασία που δρομολογείται σε κάποια διεργασία Worker. Στον άξονα X υπάρχει ο αριθμός των διεργασιών Worker και στον άξονα Y ο χρόνος εκτέλεσης σε δευτερόλεπτα.



Εικόνα 5.6: Χρόνος Εκτέλεσης για 128 Βελτιστοποιήσεις



Εικόνα 5.7: Χρόνος Εκτέλεσης για 256 Βελτιστοποιήσεις

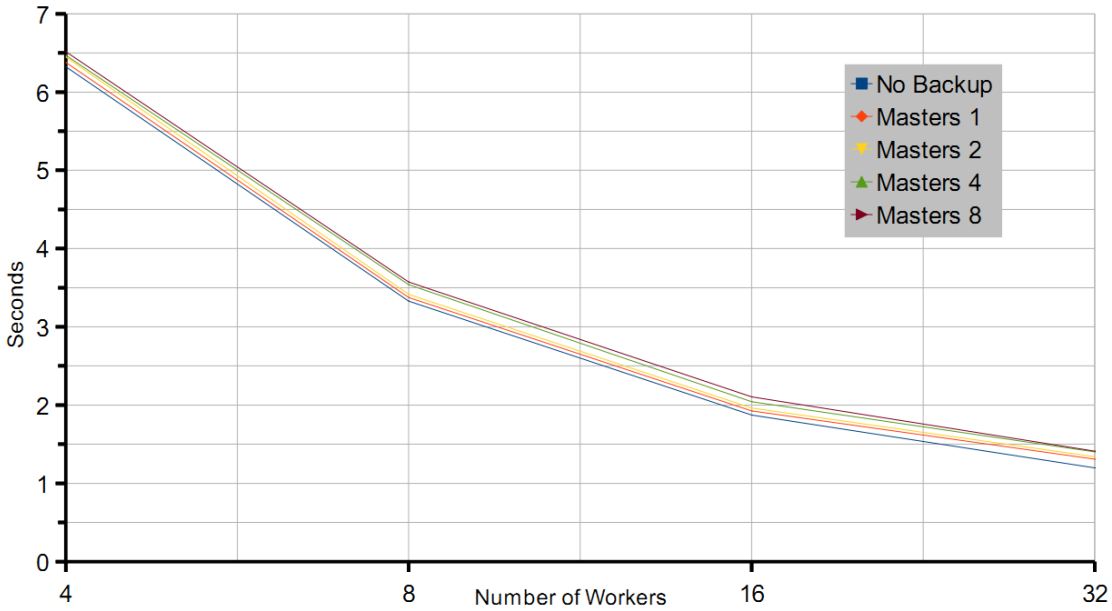
Από τα διαγράμματα της Εικόνα 5.6 και Εικόνα 5.7, φαίνεται οτι η συνολική διαδικασία της επαναφοράς της πεσμένης διεργασίας, δεν εισάγει σημαντικά περισσότερο χρόνο εκτέλεσης.. Η διαδικασία επαναφοράς της διεργασίας Worker, περιλαμβάνει το στάδιο της ανίχνευσης του σφάλματος από την υλοποίηση, της αναφοράς του στην εφαρμογή, της επανεκκίνησης της διεργασίας και τέλος της επαναφοράς της κατάστασης των δομών δεδομένων της όπως ήταν στο σημείο πριν εμφανιστεί το σφάλμα, στέλνοντας τα απαραίτητα δεδομένα.

Παρατηρούμε ότι η επιβάρυνση της επανεκκίνησης των τερματισμένων διεργασιών Worker δεν ξεπερνά το 20%. Μόνο για τις περιπτώσεις που έχουμε αρκετούς (32) Workers, όπου οι εργασίες που μοιράζονται σε αυτούς είναι λιγότερες και άρα ο συνολικός χρόνος των υπολογισμών μικραίνει, κάνοντας το χρόνο αποκατάστασης των τερματισμένων διεργασιών, πιο αισθητό στη συνολική εκτέλεση της εφαρμογής.

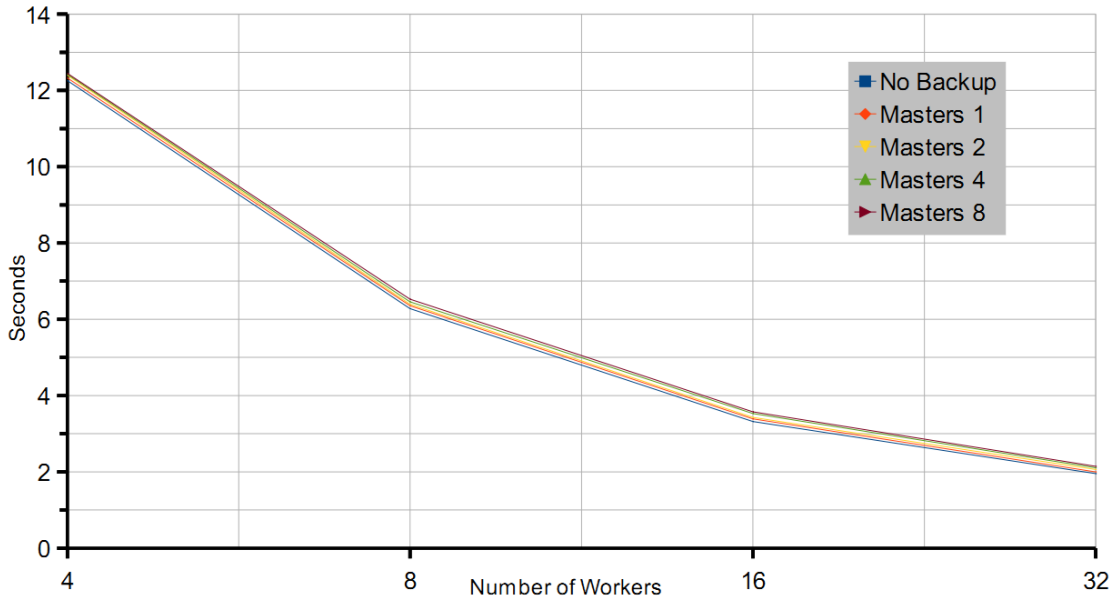
5.3.3. Εκτέλεση με Πολλαπλούς Masters και Σφάλματα

Στο συγκεκριμένο πείραμα, θέλαμε να διαπιστώσουμε τον επιπλέον χρόνο που εισάγουν οι δευτερεύουσες διεργασίες Master στο συνολικό χρόνο της εφαρμογής, όταν εμφανίζονται σφάλματα στις διεργασίες Worker και επιλέγουμε να τις επανεκκινήσουμε. Η λογική είναι ίδια με το πείραμα της Ενότητα 5.3.2, όπου τερματίζαμε 4 διεργασίες Worker, σε διάφορα σημεία του κώδικά τους. Αφού το σφάλμα ανιχνευόταν και αναφερόταν στο επίπεδο της εφαρμογής από όλες τις διεργασίες Master, τόσο δηλαδή από τη βασική, όσο και από τις δευτερεύουσες, στη συνέχεια η βασική διεργασία Master επανεκκινούσε την πεσμένη διεργασία Worker. Στη διαδικασία επανεκκίνησης, σε αυτό το πείραμα συμμετείχαν και οι επιπλέον διεργασίες Master, όπως και στη διαδικασία επαναφοράς της κατάστασης των δομών δεδομένων της πεσμένης διεργασίας.

Στην εφαρμογή χρησιμοποιήσαμε 10 μεταβλητές και εισάγαμε και μια επιπλέον τεχνητή καθυστέρηση σε κάθε αντικειμενική συνάρτηση της τάξης του 1ms. Τρέξαμε την εφαρμογή για διάφορα πλήθη διεργασιών Worker αλλά και για διαφορετικό αριθμό διεργασιών Master. Επίσης και για διαφορετικά πλήθη σημείων, για καθένα από τα οποία όπως αναφέραμε παραπάνω, εκκινεί και μια εργασία που δρομολογείται σε κάποια διεργασία Worker. Στον άξονα X υπάρχει ο αριθμός των διεργασιών Worker και στον άξονα Y ο χρόνος εκτέλεσης σε δευτερόλεπτα.



Εικόνα 5.8: Χρόνος Εκτέλεσης για 128 Βελτιστοποιήσεις



Εικόνα 5.9: Χρόνος Εκτέλεσης για 256 Βελτιστοποιήσεις

Στα διαγράμματα φαίνεται οτι οι επιπλέον διεργασίες Master, δεν εισάγουν μεγάλο επιπλέον φόρτο στην εκτέλεση. Οι δευτερεύουσες διεργασίες Master, έχουν την επιπλέον αρμοδιότητα να στείλουν μήνυμα σηματοδοσίας και στη βασική διεργασία Master, για κάθε επικοινωνία που έχουν με κάποια διεργασία Worker. Η βασική διεργασία Master, όταν χρειαστεί να ξαναεπικοινωνήσει με τη συγκεκριμένη διεργασία Worker, θα πρέπει πρώτα να ελέγξει αν έχει λάβει όλα τα μηνύματα σηματοδοσίας από τις δευτερεύουσες διεργασίες Master για την εν λόγω διεργασία Worker. Παρόλα αυτά, ο χρόνος αυτός υπερκαλύπτεται από το χρόνο που ξοδεύει η βασική διεργασία Master ενδιάμεσα για επικοινωνία με άλλες διεργασίες Worker. Αυτό εξηγεί γιατί οι δευτερεύουσες διεργασίες Master, δεν επηρεάζουν την απόδοση της εκτέλεσης. Ο επιπλέον χρόνος εκτέλεσης, που φαίνεται στα διαγράμματα του πειράματος, οφείλεται στο οτι οι δευτερεύουσες διεργασίες μόλις ανιχνεύσουν το σφάλμα της διεργασίας Worker, δεν το αναφέρουν κατευθείαν, αλλά περιμένουν την αναφορά του και από τη βασική διεργασία Master.

ΚΕΦΑΛΑΙΟ 6. ΣΥΝΟΨΗ ΚΑΙ ΜΕΛΛΟΝΤΙΚΗ ΕΡΓΑΣΙΑ

6.1. Σύνοψη

Η παρούσα εργασία, επικεντρώθηκε στη μελέτη της ιδιότητας της ανοχής σφαλμάτων πάνω σε εφαρμογές MPI. Οι πιο διαδεδομένες συμβατικές υλοποιήσεις του MPI, όπως το MPICH και το OepnMPI, αδυνατούν να αντιμετωπίσουν το ζήτημα της εμφάνισης σφαλμάτων με αξιοπιστία, κυρίως στα ζητήματα της ανίχνευσης και της αναφοράς τους στο επίπεδο της εφαρμογής MPI. Έχουν προταθεί μια σειρά από υλοποιήσεις MPI προσανατολισμένες για την επίλυση του προβλήματος της ανοχής σφαλμάτων, όπως το MPICH-V και το FT-MPI, χρησιμοποιώντας διάφορες τεχνικές για να καλύψουν το κενό. Κάποιες από αυτές είναι το checkpointing σε συνδυασμό με έναν αλγόριθμο επανεκκίνησης και κάποιες φορές και με την καταγραφή των επικοινωνιών. Τεχνικές όμως που εισάγουν σημαντικό επιπλέον φόρτο απόδοσης εκτέλεσης αλλά και χώρου αποθήκευσης.

Αναπτύξαμε ένα σύστημα υποστήριξης ανοχής σφαλμάτων, που τοποθετείται ανάμεσα στη συμβατική υλοποίηση MPI και στην εφαρμογή MPI του χρήστη. Η βασική και μοναδική απαίτηση που αξιώνει, είναι η υποστήριξη της έκδοσης του προτύπου MPI-2 από την υλοποίηση, κυρίως για τη λειτουργία της δυναμικής διαχείρισης των διεργασιών MPI. Το σύστημα μας αναφέρεται σε εφαρμογές MPI που είναι ανεπτυγμένες με βάση το μοντέλο προγραμματισμού Master-Worker, ένα

μοντέλο αρκετά ευέλικτο. Για το λόγο αυτό, υλοποιήσαμε όλες τις απαραίτητες συναρτήσεις του MPI που είναι αναγκαίες για την ανάπτυξη τέτοιων εφαρμογών. Όλες οι επικοινωνίες στο σύστημά μας, υλοποιούνται μέσω του διαδικτυακού πρωτοκόλλου TCP, με τέτοιον τρόπο ώστε να μας δίνει τη δυνατότητα να ανιχνεύσουμε πιθανά σφάλματα και να τα αναφέρουμε στο επίπεδο της εφαρμογής του χρήστη, αξιόπιστα.

Στις λειτουργίες που προσφέρει η υλοποίηση, εκτός από την αξιόπιστη ανίχνευση και αναφορά των σφαλμάτων στο επίπεδο της εφαρμογής και η δυνατότητα ανάκτησης της εσφαλμένης διεργασίας επανεκκινώντας την, η δυνατότητα καταγραφής των μηνυμάτων που ανταλλάσσονται μεταξύ των διεργασιών αλλά και η δυνατότητα χρήσης πολλαπλών διεργασιών Master. Πιο συγκεκριμένα, η υλοποίηση δίνει την επιλογή στον προγραμματιστή της εφαρμογής, για το αν θα επανεκκινήσει την εσφαλμένη και άρα τερματισμένη διεργασία MPI, ή θα συνεχίσει την υπόλοιπη εκτέλεση της εφαρμογής με λιγότερες διεργασίες Worker να συμμετέχουν στους υπολογισμούς. Η καταγραφή των επιτυχώς ανταλλασσόμενων μηνυμάτων, βοηθάει τον προγραμματιστή της εφαρμογής να κρατάει μια συνεπή κατάσταση στα μηνύματα που πρέπει να ανταλλαχθούν μεταξύ της διεργασίας Master και της επανεκκινημένης διεργασίας Worker, ώστε η τελευταία να επανέλθει σε μια συνεπή για τη συνολική εκτέλεση, κατάσταση. Τέλος, με το γεγονός οτι επιτρέπει την ορθή, κοινή παράλληλη λειτουργία πολλαπλών διεργασιών Master, μαζί με τη βασική διεργασία Master έχει ως στόχο την αντιμετώπιση των σφαλμάτων που μπορούν να εμφανιστούν στην πλευρά της διεργασίας Master.

Πραγματοποιήσαμε μια σειρά πειραμάτων για να ελέγξουμε διάφορα ζητήματα που αφορούν τη συμπεριφορά της υλοποίησης. Πρώτα, από τη σκοπιά της ορθότητας εκτέλεσης, κυρίως για τις περιπτώσεις όπου συνυπάρχουν στην υλοποίηση πολλαπλές διεργασίες Master. Σε αυτήν την περίπτωση το πρωτόκολλο επικοινωνιών πολυπλοκεύει, περισσότερο όταν στην εκτέλεση εμφανίζεται και κάποιο σφάλμα. Στα πειράματα αυτά, ρίχναμε κάποιες διεργασίες είτε Worker, είτε Master και ελέγχαμε την ορθότητα των τελικών αποτελεσμάτων. Πειραματιστήκαμε με μια εφαρμογή βελτιστοποίησης που χρησιμοποιεί τη μέθοδο Multistart. Σε αυτήν την περίπτωση ελέγξαμε αρχικά το πόσο επιπλέον φόρτο εκτέλεσης εισάγει η υλοποίησή

μας σε σχέση με μια συμβατική υλοποίηση MPI, σε εκτέλεση χωρίς σφάλματα. Επίσης τον επιπλέον φόρτο που εισάγει η λειτουργία της επανεκκίνησης της πεσμένης διεργασίας Worker καθώς και η συνύπαρξη πολλαπλών διεργασιών Master στην εκτέλεση.

6.2. Μελλοντική Εργασία

Το FTMW προσανατολίζεται αποκλειστικά, σε εφαρμογές MPI ανεπτυγμένες πάνω στο μοντέλο προγραμματισμού Master-Worker. Το μοντέλο αυτό, μπορεί μεν εύκολα να γενικεύεται για πολλές εφαρμογές, αλλά δεν καλύπτει όλο το εύρος εφαρμογών. Οι λειτουργίες που προσφέρει στην εφαρμογή MPI του προγραμματιστή, θα μπορούσαν να εφαρμοστούν και σε γενικότερα μοντέλα εφαρμογών. Ένα από αυτά θα μπορούσε να είναι και το μοντέλο Master-Worker πολλών επιπέδων. Σε αυτό το μοντέλο οι αρχικές διεργασίες Worker, έχουν με τη σειρά τους και ρόλο Master, για άλλες διεργασίες. Αυτό μπορεί να συνεχιστεί και για επόμενα επίπεδα. Οι λειτουργίες της υλοποίησης και οι αρμοδιότητες που αναθέτει στη διεργασία Master, κληρονομούνται και στα υπόλοιπα επίπεδα. Οι λειτουργίες της υλοποίησης όμως, μπορούν να εφαρμοστούν και σε άλλα μοντέλα πέρα από το Master-Worker, μη ιεραρχικά.

Ένα άλλο ζήτημα που θα μπορούσε να εξεταστεί είναι και η διαχείριση των εργασιών που έχουν ανατεθεί σε διεργασίες που έχουν εμφανίσει σφάλμα και δεν πρόλαβαν να εξάγουν ή να στείλουν αποτελέσματα στη διεργασία Master. Στην υλοποίηση μας ο χρήστης θα πρέπει να διαχειριστεί ρητά τον τρόπο με τον οποίο θα διαμοιραστούν στις διεργασίες Worker οι επιμέρους εργασίες του προβλήματος. Πολύ περισσότερο στην περίπτωση που κάποια διεργασία Master, θα πρέπει να επαναδρομολογήσει την εργασία που της είχε αναθέσει. Θα μπορούσε λοιπόν η υλοποίηση να προσφέρει περαιτέρω δυνατότητες διαχείρισης των εργασιών σε σχέση με τις διεργασίες Worker. Μια άλλη λύση επίσης, θα ήταν και η αυτόματη εκτέλεση των εργασιών για τις οποίες δεν έχουν ληφθεί αποτελέσματα, στο τέλος των υπολογισμών

Εκτός από το θέμα της ανολοκλήρωτης εργασίας, στην περίπτωση που εμφανιστεί σφάλμα σε μια διεργασία και ο προγραμματιστής επιθυμεί να την επανεκκινήσει εγείρεται και το ζήτημα της επαναφοράς της στη κατάσταση που βρισκόταν πριν το σφάλμα. Και σε αυτήν την περίπτωση στην υλοποίησή μας, είναι υπευθυνότητα του προγραμματιστή να την επαναφέρει στην εν λόγω κατάσταση μιας και όπως αναφέρθηκε σε προηγούμενο κεφάλαιο, η υλοποίηση δεν υποστηρίζει checkpoints. Παρόλα αυτά, η υλοποίηση εκτός από το να επανακκινήσει την εσφαλμένη διεργασία θα μπορούσε σε δεύτερο βήμα, αυτόματα να την επαναφέρει αν όχι στην κατάσταση ακριβώς πριν εμφανιστεί το σφάλμα, αλλά σε μια κατάσταση που θα της επιτρέπει να συμμετέχει ομαλά και πάλι στους υπολογισμούς, χωρίς ο προγραμματιστής να προβεί σε κάποια περαιτέρω ενέργεια.

ΑΝΑΦΟΡΕΣ

[1] G. Bosilca, A. Bouteiller, F. Cappello, S. Djilali, G. Fedak, C. Germain, T. Herault, P. Lemarinier, O. Lodygensky, F. Magniette, V. Neri, and A. Selikhov. Mpich-v: toward a scalable fault tolerant mpi for volatile nodes. In Supercomputing Conference, pages 1-18, Los Alamitos, CA, USA, 2002.

[2] R. Brightwell, K. Ferreira, and R. Riesen.Transparent redundant computing with mpi. In Recent Advances in the Message Passing Interface, volume 6305, pages 208-218. Springer Berlin/Heidelberg, 2010.

[3] J. Rodrigues de Souza, E. Argollo, A. Duarte, D. Rexachs, and E. Luque. Fault tolerant master-worker over a multi-cluster architecture. In Parallel Computing: Current & Future Issues of High-End Computing, Proceedings of the International Conference ParCo, pages 465-472, Department of Computer Architecture, University of Malaga, Spain, September 13-16, 2005.

[4] G. E. Fagg, E. Gabriel, Z. Chen, T. Angskun, G. Bosilca, J. Pjesivac-Grbovic, and J. J. Dongarra. Process fault tolerance: Semantics, design and applications for high performance computing. International Journal of High Performance Computing Applications, 19 (4): 465-477, Winter 2005.

[5] The MPI Forum. Mpi: A message passing interface, November 1993.

[6] E. Gabriel, G. E. Fagg, G. Bosilca, T. Angskun, J. Dongarra, J. M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine, R. H. Castain, D. J. Daniel, R. L. Graham, and T. S. Woodall. Openmpi: Goals, concept, and design of a next generation mpi implementation. In Recent Advances in Parallel Virtual Machine and Message Passing Interface, 11th European PVM/MPI Users' Group Meeting, Proceedings, pages 97-104, Budapest, Hungary, September 19-22, 2004.

[7] A. Geist, W. Gropp, S. Huss-Lederman, A. Lumsdaine, E. L. Lusk, W. Saphir, A. Skjellum, and M. Snir. Mpi-2: Extending the message-passing interface. In EuroPar '96 Parallel Processing, Second International EuroPar Conference, Proceedings, Volume I, pages 128-135, Lyon, France, August 26-29, 1996.

[8] R. L. Graham, S-E. Choi, D. J. Daniel, N. N. Desai, R. Minnich, C. E. Rasmussen, L. D. Risinger, and M. W. Sukalski. A network-failure-tolerant message-passing system for terascale clusters. International Journal of Parallel Programming, 31 (4): 285-303, 2003.

[9] W. Gropp and E. Lusk. Dynamic process management in an mpi setting. In Proceedings of the 7th IEEE Symposium on Parallel and Distributeed Processing, IEEE Computer Society, page 530. IEEE Computer Society, 1995.

[10] W. Gropp, E. Lusk, and A. Skjellum. Using MPI: Portable Parallel Programming with the Message Passing Interface. 1994.

[11] W. Gropp and E. L. Lusk. Fault tolerance in message passing interface programs. International Journal of High Performance Computing Applications, 18 (3): 363-372, 2004.

[12] Wikipedia. Non-uniform memory access, 2007. Available from World Wide Web: http://en.wikipedia.org/w/index.php?title=NonUniform_Memory_Access& oldid=101909975. This is an electronic document.

[13] K. I. Hagen. Fault-tolerance for mpi codes on computational clusters, Master–Thesis, Norwegian University of Science and Technology, June 2007.

[14] J. Hursey, T. Mattox, and A. Lumsdaine. Interconnect agnostic checkpoint/restart in openmpi. In Proceedings of the 18th ACM International Symposium on High Performance Distributed Computing, pages 49-58, Garching, Germany, June 11-13, 2009.

[15] J. Hursey, J. M. Squyres, T. Mattox, and A. Lumsdaine. The design and implementation of checkpoint/restart process fault tolerance for openmpi. In 21th International Parallel and Distributed Processing Symposium (IPDPS2007), Proceedings, pages 1-8, Long Beach, California, USA, March 26-30, 2007.

[16] S. Louca, N. Neophytou, A. Lachanas, and P. Evripidou. Mpi-ft: Portable fault tolerance scheme for mpi. Parallel Processing Letters, 10 (4): 371-382, 2000.

[17] Message Passing Interface Forum Meetings. Mpi 3.0 standardization effort.

[18] MPICH2. Mpich 2 project home page, 2008.

[19] J. Postel. Rfc 793: Transmission control protocol. Technical report, September 1981.

[20] S. Rao, L. Alvisi, and H. M. Vin. Egida: An extensible toolkit for low-overhead fault tolerance. In Proceedings of the 29th Fault-Tolerant Computing Symposium, pages 48-55, Los Alamitos, CA, USA, 1999.

[21] L. M. Silva and R. Buyya. Parallel programming models and paradigms. In High Performance Cluster Computing: Architectures and Systems: Volume 2, Prentice Hall PTR, NJ, USA, September/October 1999.

[22] M. Snir, S. W. Otto, S. Huss-Lederman, D. W. Walker, and J. Dongarra. MPI: The complete reference. MIT Press, Cambridge, MA, 1996.

[23] G. Stellner. Cocheck: Checkpointing and process migration for mpi. In Proceedings of IPPS '96, The 10th International Parallel Processing Symposium, 1996, pages 526-531, Honolulu, Hawaii, USA, April 15-19, 1996.

[24] A. Supalov. Mpi-3 fault handling = error handling, MPI Forum Meeting, Chicago Illinois, USA, 29 April 2008.

[25] W.Gropp, E.Lusk, N.Doss, and A.Skjellum. A high-performance portable implementation of the MPI message passing interface standard. Parallel Computing, 22(6):789–828, September 1996.

[26] G. Burns, R. Daoud, and J. Vaigl. LAM: An Open Cluster Environment for MPI. In Proceedings of Supercomputing Symposium, pages 379–386, 1994. Available from World Wide Web: http://www.lam-mpi.org/download/files/lam-papers.tar.gz.

[27] M.A. Heroux, P. Raghavan, and H.D. Simon. Parallel processing for scientific computing, Chapter 11, pages 203-220, Society for Industrial and Applied Mathematics, 2006.

[28] D.A. Reed, C. da Lu and C.L.Mendes, Reliability challenges in large systems, Future Generation Computer Systems, volume 22, no. 3, pages 293–302, 2006.

[29] L.C.W. Dixon and G.P Szego. The Global Optimization: An Introduction, Towards Global Optimization 2, North-Holland, Amsterdam, 1978, pages 1-15.

ΠΑΡΑΡΤΗΜΑ

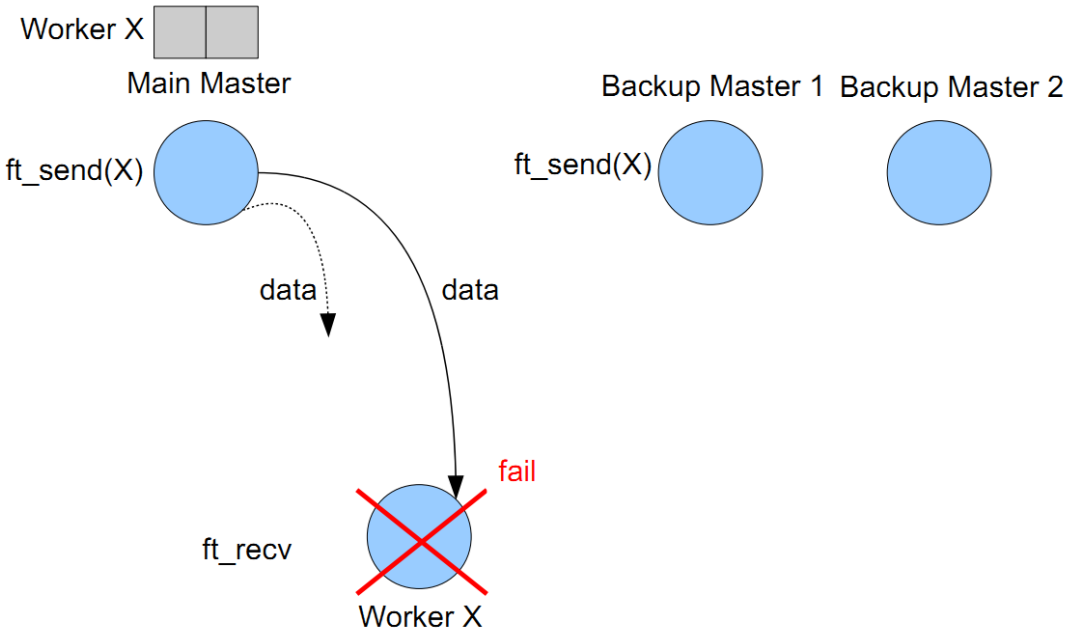
Στο Παράρτημα, παραθέτουμε σε εικόνες, τα βήματα που ακολουθεί το πρωτόκολλο αντιμετώπισης σφαλμάτων με πολλαπλούς Master, για διάφορες περιπτώσεις σφαλμάτων. Για τις περιπτώσεις που είναι τετριμένες, δεν παρατίθονται εικόνες.

Σφάλμα Διεργασίας Worker

Master: Αποστολή Δεδομένων

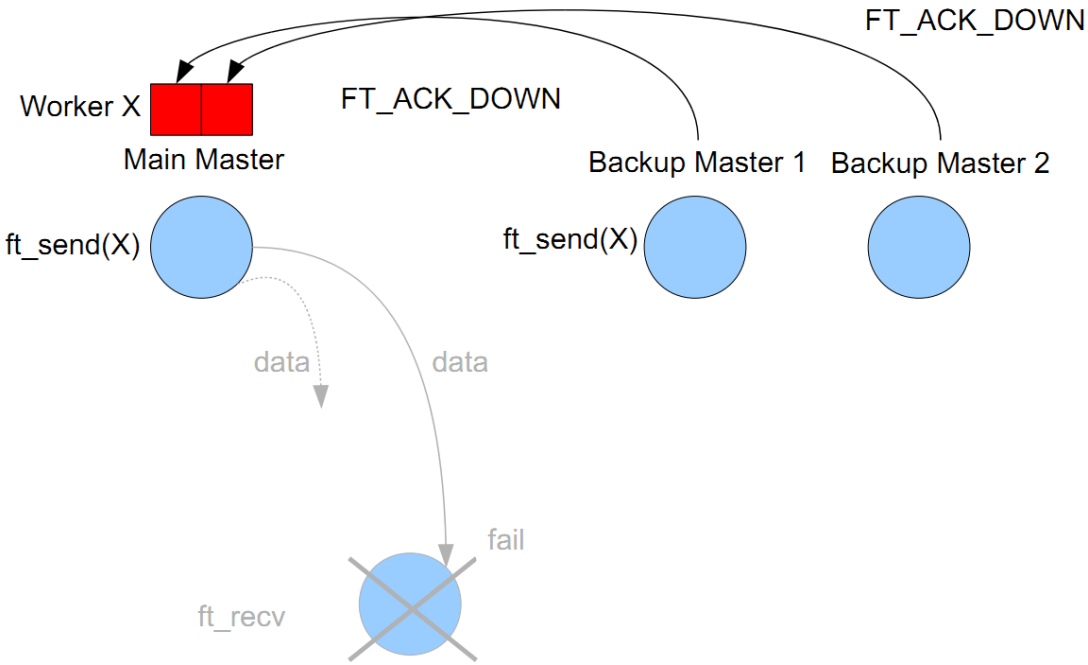
Περίπτωση 1:
Η διεργασία Worker εμφανίζει σφάλμα:
Πριν λάβει τα δεδομένα κατά τη διάρκεια εκτέλεσης κώδικα εφαρμογής
Πριν λάβει τα δεδομένα κατά τη διάρκεια εκτέλεσης κώδικα της υλοποίησης
Έχει λάβει τα δεδομένα και δεν έχει στείλει επιβεβαίωση στο Master

Αρχικά η διεργασία Master ανιχνεύει το σφάλμα και το αναφέρει.



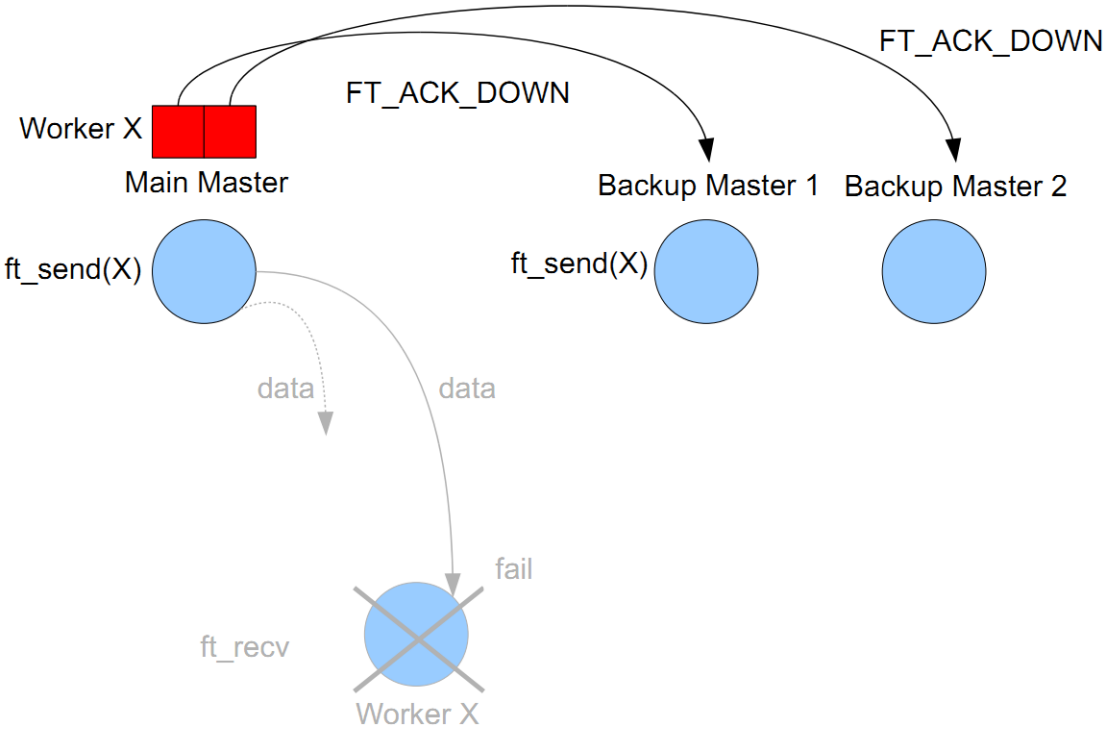
Εικόνα Π.6.1

Όταν το σφάλμα ανιχνευτεί από τις υπόλοιπες διεργασίες Master, δεν το αναφέρουν και σε δεύτερο βήμα στέλνουν το αντίστοιχο σήμα (FT_ACK_DOWN) στη βασική διεργασία Master.



Εικόνα Π.6.2

Όταν λάβει τα αντίστοιχα σήματα η διεγργασία Master, στέλνει πίσω στις δευτερεύουσες διεργασίες σήμα ότι δεν έχει γίνει επιτυχώς η κλήση ft_send (FT_ACK_DOWN) και αρα μπορούν να αναφέρουν το σφάλμα.



Εικόνα Π.6.3

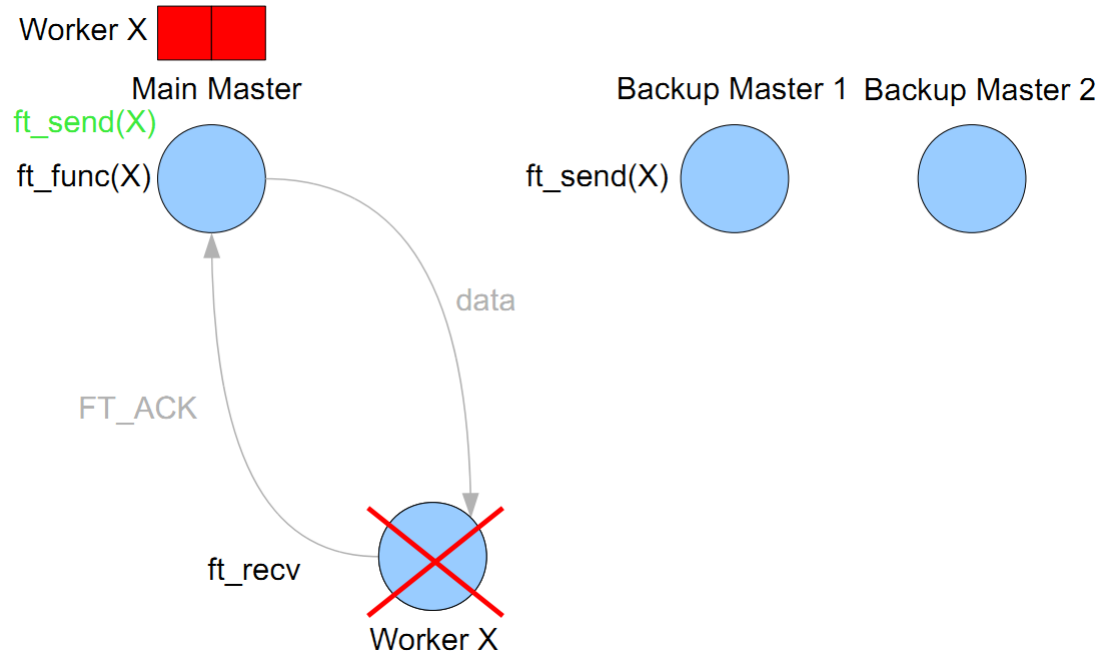
Περίπτωση 2:

Η διεργασία Worker εμφανίζει σφάλμα:

Αφού έχει στείλει επιβεβαίωση παραλαβής μόνο στη διεργασία Master (Εικόνες)

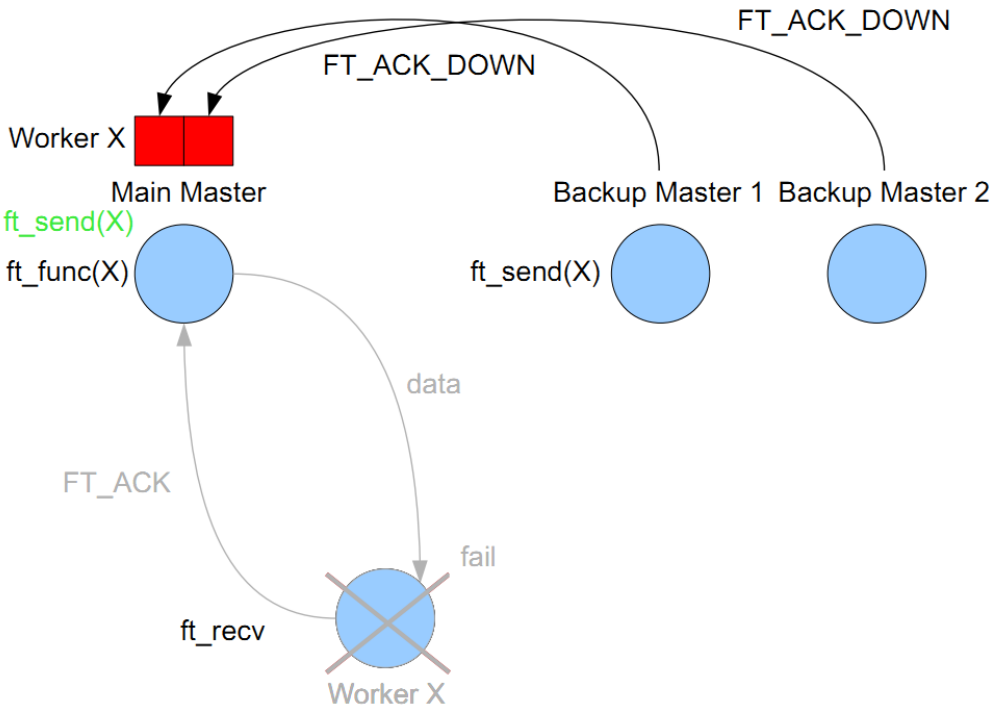
Αφού έχει στείλει επιβεβαίωση και σε κάποια από τις δευτερεύουσες διεργασίες αλλά όχι σε όλες

Αρχικά η διεργασία Master επικοινωνεί επιτυχώς με τη διεργασία Worker και ολοκληρώνει επιτυχώς την αποστολή.



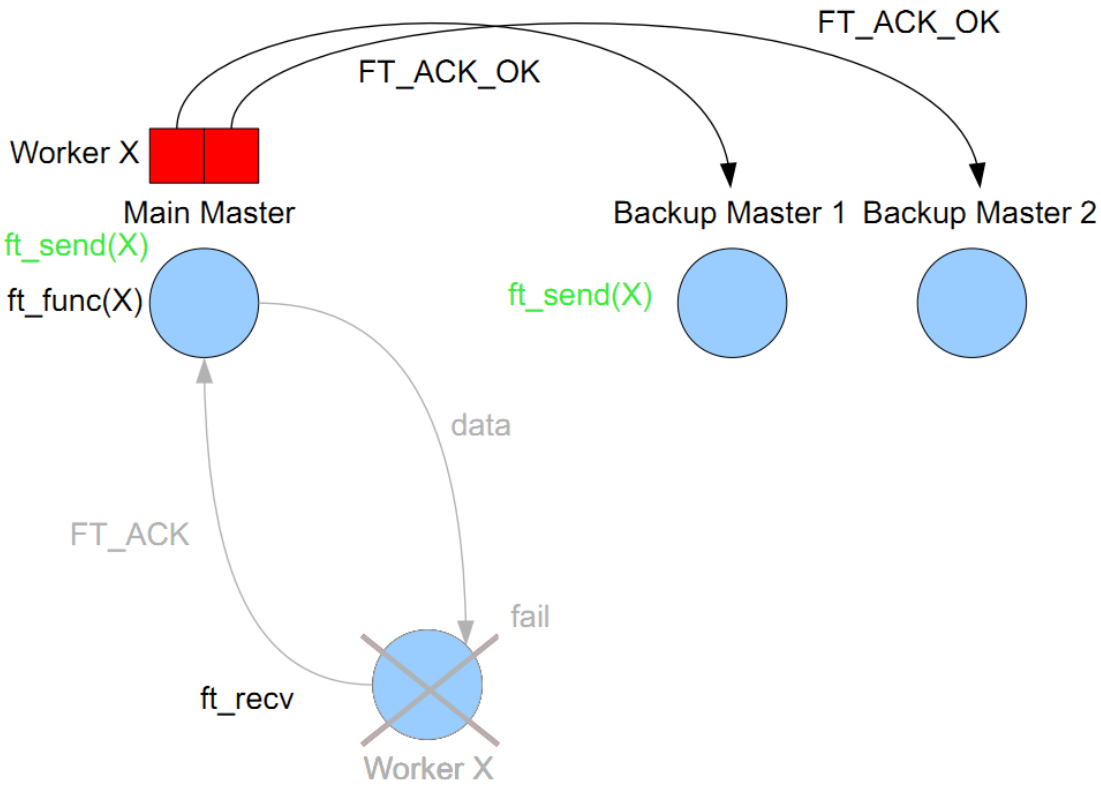
Εικόνα Π.6.4

Οι δευτερεύουσες διεργασίες Master, ανιχνεύουν το σφάλμα αλλά δεν το αναφέρουν ακόμα. Στέλνουν το αντίστοιχο μήνυμα σφάλματος στη βασική διεργασία Master.



Εικόνα Π.6.5

Η βασική διεργασία στέλνει πίσω σε αυτές το σχετικό σήμα επιτυχίας (FT_ACK_OK) που σημαίνει ότι η κληση έγινε επιτυχώς και άρα και αυτές θα πρέπει να την ολοκληρώσουν επιτυχώς.



Εικόνα Π.6.6

Master: Παραλαβή Δεδομένων

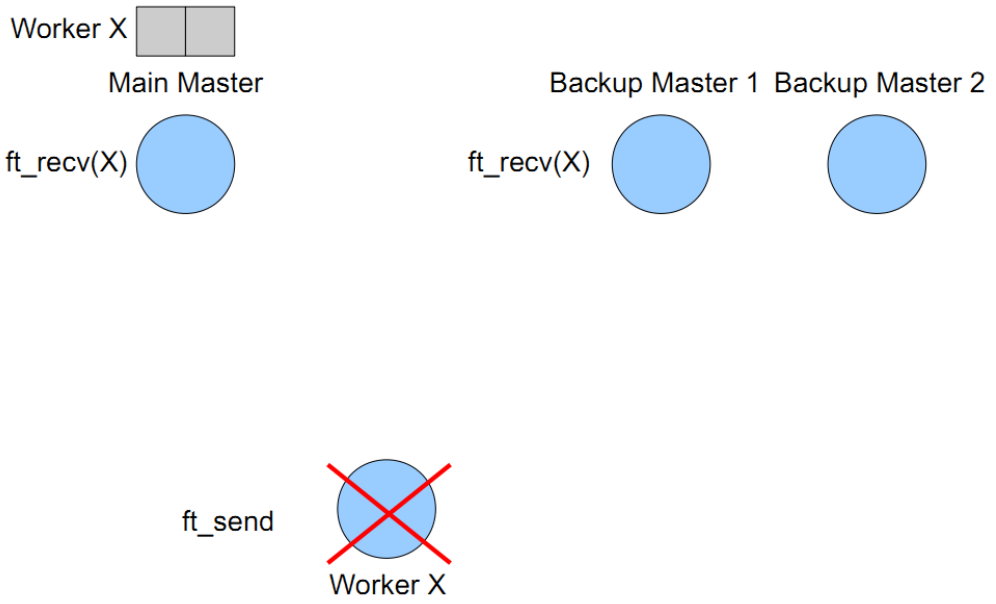
Περίπτωση 1:

Η διεργασία Worker εμφανίζει σφάλμα:

Πριν στείλει τα δεδομένα κατά τη διάρκεια εκτέλεσης κώδικα εφαρμογής

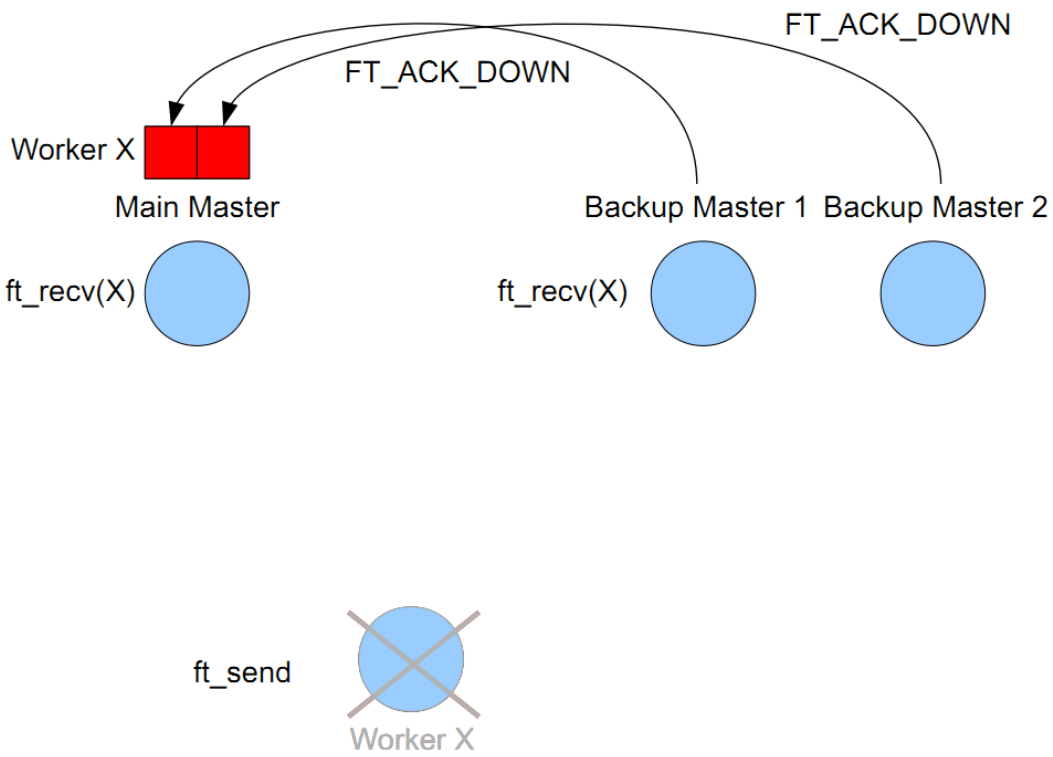
Πριν στείλει τα δεδομένα κατά τη διάρκεια εκτέλεσης κώδικα της υλοποίησης

Αρχικά η διεργασία Master ανιχνεύει το σφάλμα και το αναφέρει.



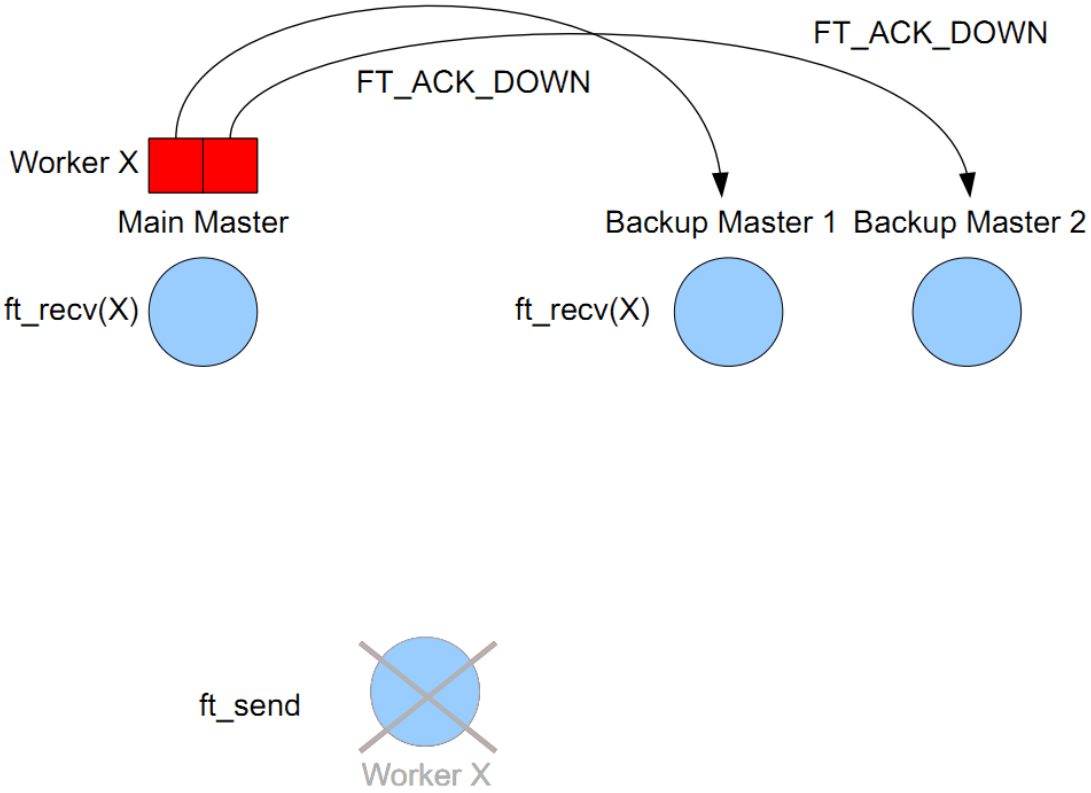
Εικόνα Π.6.7

Όταν το σφάλμα ανιχνευτεί από τις υπόλοιπες διεργασίες Master, στέλνουν το αντίστοιχο σήμα (FT_ACK_DOWN) στη βασική διεργασία Master.



Εικόνα Π.6.8

Όταν λάβει τα αντίστοιχα σήματα η διεργασία Master, στέλνει πίσω στις δευτερεύουσες διεργασίες Master σήμα ότι δεν έχει γίνει επιτυχώς η κλήση ft_recv (FT_ACK_DOWN) και αρα μπορούν να αναφέρουν το σφάλμα.



Εικόνα Π.6.9

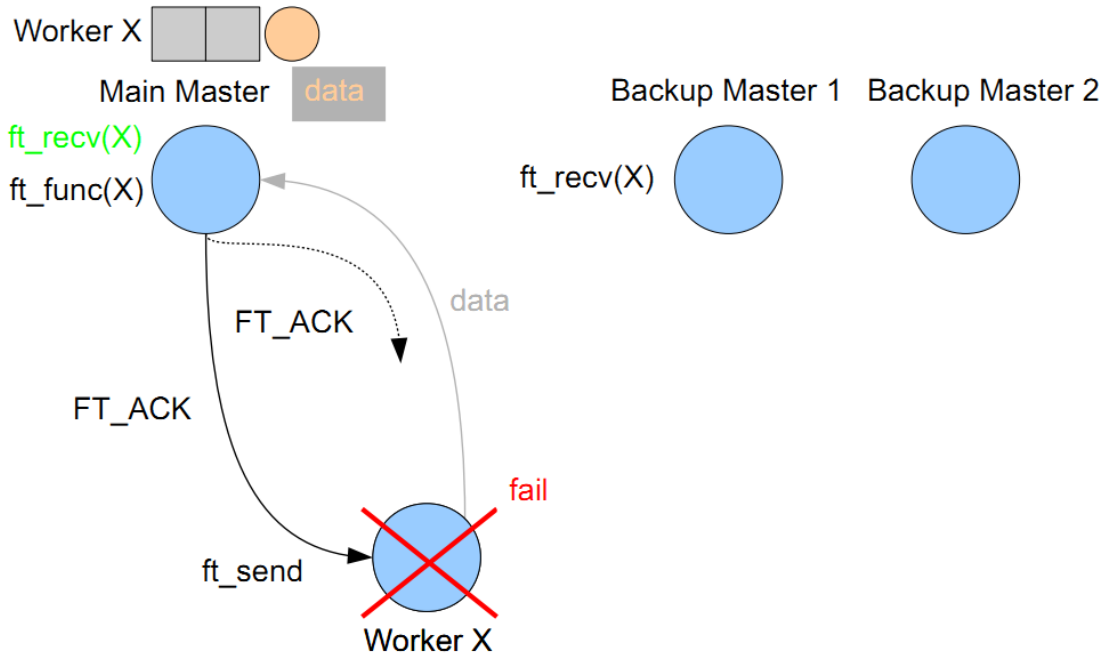
Περίπτωση 2:

Η διεργασία Worker εμφανίζει σφάλμα:

Αφού έχει στείλει αποτελέσματα στη βασική διεργασία Master και ενώ περίμενε επιβεβαίωση από αυτήν (Εικόνες)

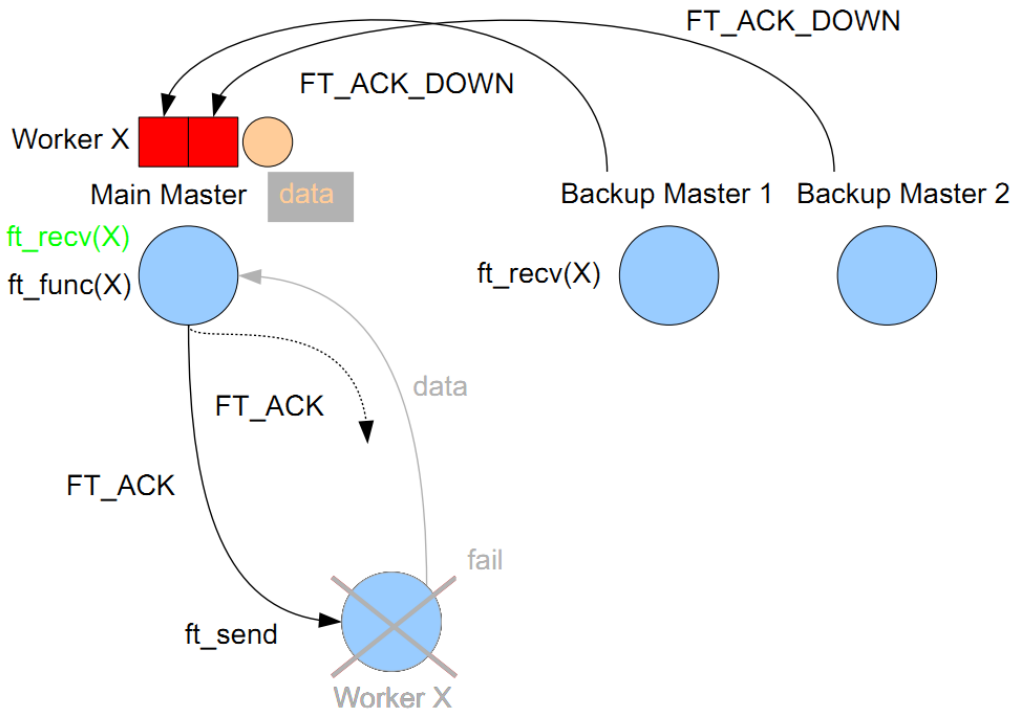
Αφού έχει στείλει αποτελέσματα και σε κάποια από τις δευτερεύουσες διεργασίες αλλά όχι σε όλες

Αρχικά η διεργασία Master επικοινωνεί επιτυχώς με τη διεργασία Worker και ολοκληρώνει την παραλαβή. Επιπλέον αποθηκεύει και τα δεδομένα που έλαβε από τη διεργασία Worker.



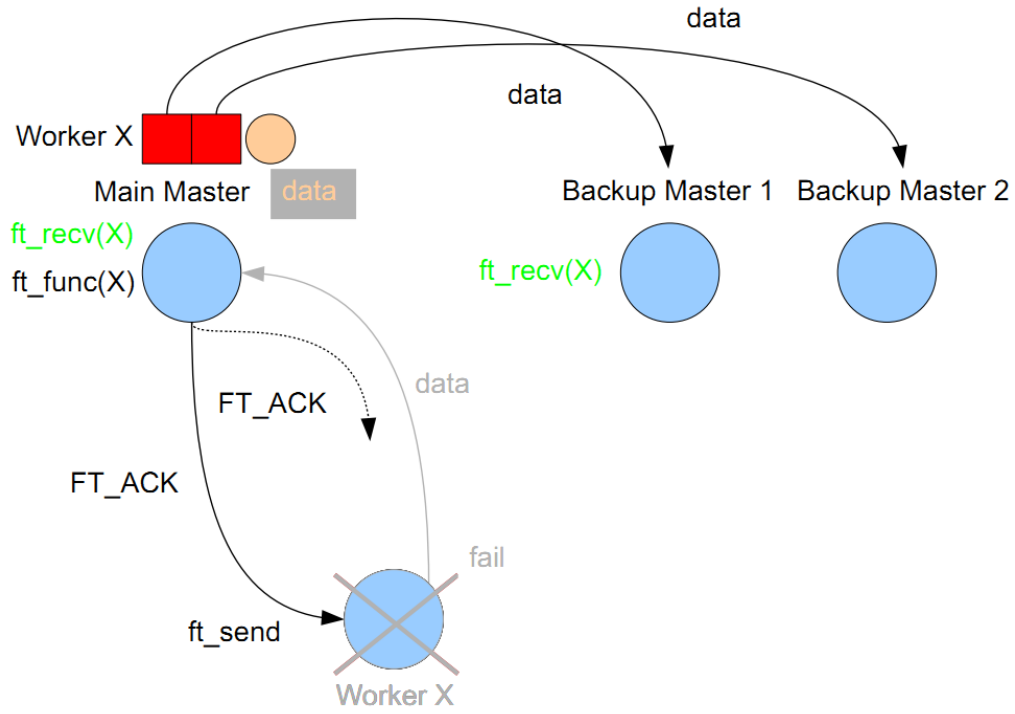
Εικόνα Π.6.10

Οι δευτερεύουσες διεργασίες Master, ανιχνεύουν το σφάλμα αλλά δεν το αναφέρουν ακόμα. Στέλνουν το αντίστοιχο μήνυμα σφάλματος στη βασική διεργασία Master.



Εικόνα Π.6.11

Η βασική διεργασία στέλνει πίσω σε αυτές μήνυμα με τα δεδομένα που έλαβε από τη διεργασία Worker και αφού τα λάβουν ολοκληρώνουν επιτυχώς την κλήση.



Εικόνα Π.6.12

Σφάλμα Βασικής Διεργασίας Master

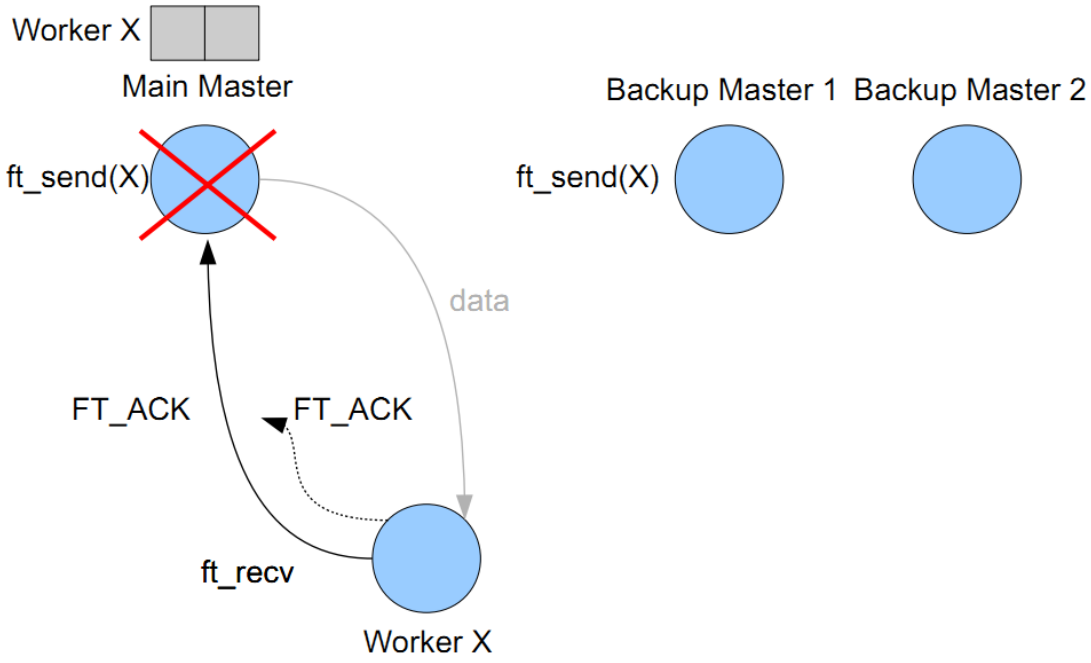
Master: Αποστολή Δεδομένων

Περίπτωση 1:
Η βασική διεργασία Master, εμφανίζει σφάλμα πριν ξεκινήσει οποιαδήποτε διαδικασία αποστολής δεδομένων.

Η διεργασία Worker, ανιχνεύει το σφάλμα όπως και οι δευτερεύουσες διεργασίες Master. Η διεργασία Worker, επιλέγει την επόμενη διεργασία Master, της δάταξης ενώ αντίστοιχα οι δευτερεύουσες διεργασίες Master, κοιτώντας τις καταστάσεις των προηγούμενων από αυτές στη διάταξη διεργασιών Master, διαπιστώνουν η κάθε μια ξεχωριστά αν είναι η ίδια η επόμενη βασκή διεργασία Master. Η εκτέλεση ξεκινάει γνωρίζοντας πλέον πάλι όλες οι διεργασίες, ποια είναι η θέση τους στην εκτέλεση.

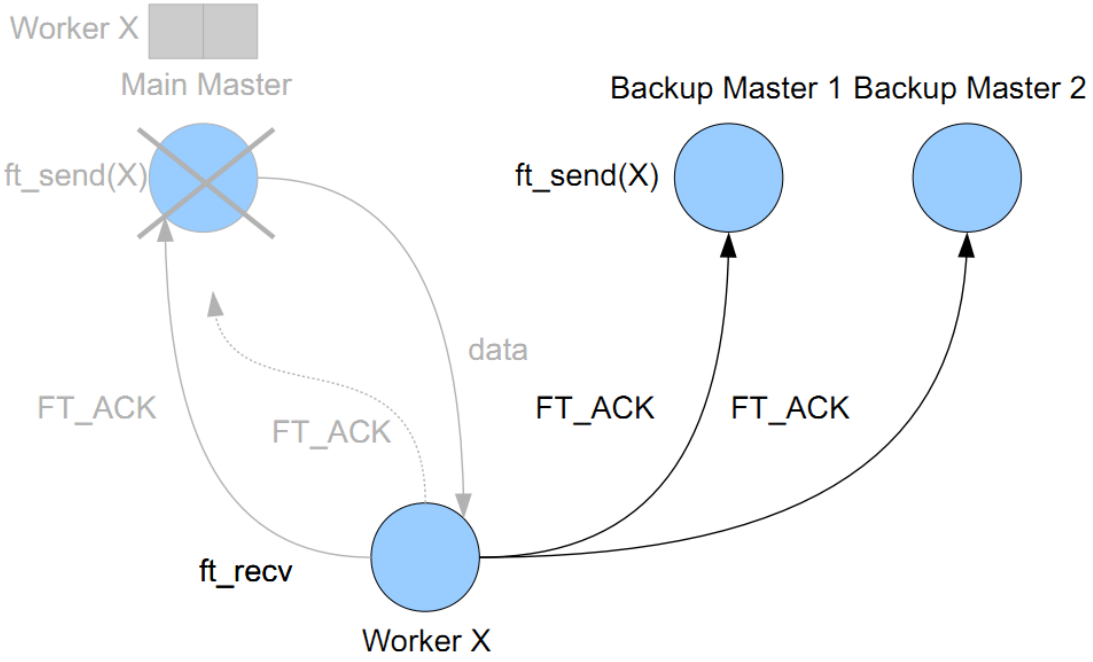
Περίπτωση 2:
Η βασική διεργασία Master, εμφανίζει σφάλμα αφού έχει στείλει τα δεδομένα και ενώ περίμενε επιβεβαίωση.

Αρχικά η διεργασία Worker λαμβάνει τα δεδομένα και αρχίζει τη διαδικασία αποστολής επιβεβαιώσεων. Αξίζει να σημειώσουμε ότι μαζί με το σήμα της επιβεβαίωσης, εισάγει και την πληροφορία για το ποια ήταν η διεργασία Master (αναγνωριστικό) όταν παρέλαβε τα δεδομένα.



Εικόνα Π.6.13

Όι δευτερεύουσες διεργασίες Master, λαμβάνουν την επιβεβαίωση και από την επόμενη κλήση της υλοποίησης, αναλαμβάνει μια από αυτές το ρόλο της βασικής.



Εικόνα Π.6.14

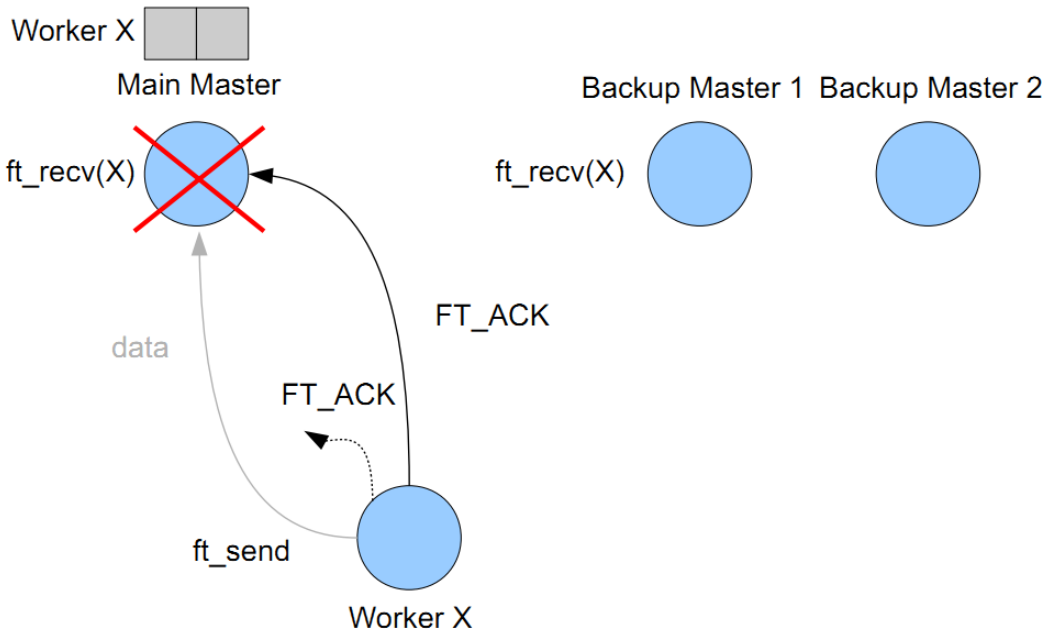
Master: Παραλαβή Δεδομένων

Περίπτωση 1:
Η βασική διεργασία Master, εμφανίζει σφάλμα πριν ξεκινήσει οποιαδήποτε διαδικασία παραλαβής δεδομένων.

Η διεργασία Worker, ανιχνεύει το σφάλμα όπως και οι δευτερεύουσες διεργασίες Master. Η διεργασία Worker, επιλέγει την επόμενη διεργασία Master, της δάταξης ενώ αντίστοιχα οι δευτερεύουσες διεργασίες Master, ελέγχοντας τις καταστάσεις των προηγούμενων από αυτές στη διάταξη διεργασιών Master, διαπιστώνουν η κάθε μια ξεχωριστά αν είναι η ίδια η επόμενη βασκή διεργασία Master. Η εκτέλεση ξεκινάει γνωρίζοντας πλέον όλες οι διεργασίες, ποια είναι η θέση τους στην εκτέλεση.

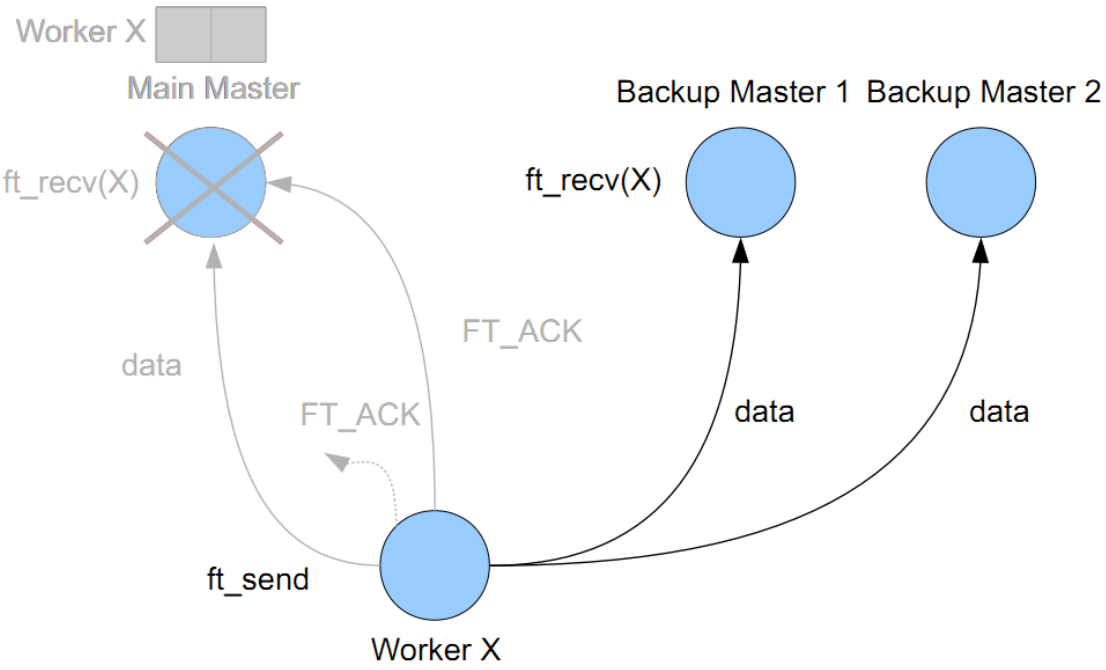
Περίπτωση 2:
Η βασική διεργασία Master, εμφανίζει σφάλμα αφού έχει παραλάβει τα δεδομένα και πριν στείλει επιβεβαίωση.

Αρχικά η διεργασία Worker στέλνει τα δεδομένα και ανιχνεύει το σφάλμα της βασικής διεργασίας Master ενώ περίμενε επιβεβαίωση.



Εικόνα Π.6.15

Στη συνέχεια αρχίζει τη διαδικασία αποστολής αποτελεσμάτων και στις υπόλοιπες διεργασίες Master. Αξίζει να σημειώσουμε ότι μαζί με τα αποτελέσματα, εισάγει και την πληροφορία για το ποια ήταν η διεργασία Master (αναγνωριστικό) όταν απέστειλε τα δεδομένα.



Εικόνα Π.6.16

Οι δευτερεύουσες διεργασίες Master, λαμβάνουν τα αποτελέσματα. Μια εξ αυτών που έχουν ανιχνεύσει το σφάλμα στη διεργασία Master, παίρνει εκ νέου το ρόλο της βασικής. Όταν λοιπόν λάβει τα αποτελέσματα θα ελέγξει και το ποια ήταν η βασική διεργασία Master, με την οποία ξεκίνησε η διεργασία Worker την επικοινωνία. Αν δεν είναι η ίδια, τότε δε χρειάζεται να κάνει κάποια παραπάνω διαδικασία. Από την επόμενη κλήση αρχίζει και παίρνει επί της ουσίας το ρόλο της βασικής Master.

Σφάλμα Δευτερεύουσας Διεργασίας Master

Όλες οι περιπτώσεις εδώ, είναι τετριμένες. Αν κάποια δευτερεύουσα διεργασία Master, εμφανίσει το σφάλμα, τότε η βασική διεργασία Master, δε χρειάζεται να ελέγχει για επιβεβαιώσεις από τη συγκεκριμένη διεργασία. Οι διεργασίες Worker από την άλλη δε χρειάζεται να στέλνουν επιβεβαιώσεις και αποτελέσματα στην τερματισμένη δευτερεύουσα διεργασία.